Казанский федеральный университет
*Институт вычислительной математики
и информационных технологий*

О.В. Пинягина

Основы работы с базами данных - СУБД mySQL

Учебное пособие

Казань – 2018

УДК 004.6

ББК 32.973.26 – 018.2

Печатается по решению учебно-методической комиссии
Института вычислительной математики и информационных технологий,
Протокол № 2 от 06.10.2017.

Рецензенты:

ведущий специалист по учебно-методической работе Отдела электронных технологий в образовании КНИТУ-КАИ **Устюгова В.Н.**

доцент кафедры системного анализа и информационных технологий КФУ, к.ф.-м.н. **Андрианова А.А.**

Пинягина О.В.

Основы работы с базами данных - СУБД mySQL: Учебное пособие / О.В. Пинягина – Казань: Казанский университет, 2018. – 92 с.

Данное учебное пособие разработано для поддержки компьютерных лабораторных занятий и самостоятельной работы по курсам «Управление базами данных», «Принципы организации и разработки баз данных» для студентов, обучающихся по направлениям «Хемоинформатика и молекулярное моделирование», «Биоинформатика».

В пособии рассматриваются основы проектирования и построения баз данных: **ER-модель** и **реляционная модель**, а также основы языка **SQL**. Каждая тема содержит задания для индивидуальной работы.

В качестве среды программирования используется **mySQL Workbench**.

Электронный ресурс по данному курсу располагается на сайте кафедры анализа данных и исследования операций КФУ по адресу: http://kek.ksu.ru/EOS/mysql/index.html .

© Казанский университет, 2018

© Пинягина О.В., 2018

Содержание

Проектирование баз данных ……………………………………… …. **4**

ER-модель (entity-relationship model)………………………... …. **4**

Пример ER-модели: Контора «Рога и копыта» …………….. …. **10**

Пример ER-модели: «Музыканты»………………………….. …. **11**

Установка MySQL……………………………………………… **13**

Создание EER-диаграммы………………………………………….. **16**

Создание базы данных из EER-диаграммы……………………... **23**

Заполнение базы данных, модификация данных……………….. **28**

Запросы к базе данных……………………………………………… **40**

Выборка из одной таблицы………………………………………. **41**

Использование условий отбора………………………………….. **43**

Использование агрегирующих функций………………………... **45**

Подзапросы …………………………………………………………. **47**

Группировка…………………………………………………………. **49**

Выборка из нескольких таблиц…………………………………….. **50**

Объединение запросов ……………………………………………... **52**

И еще несколько примеров…………………………………………. **53**

Представления ……………………………………………………… **56**

Хранимые процедуры……………………………………………….. **60**

Триггеры ……………………………………………………………….. **67**

Приложение 1. Сценарий создания базы данных………………… **72**

Приложение 2. Dbforge Studio от Devart…………...……………… **76**

Приложение 3. Реляционная алгебра и SQL……………………… **86**

Литература…………………………………………………………… **92**

Дистрибутивы ………………………………………………………… **92**

Проектирование баз данных

ER-модель (entity-relationship model)

Работа с базой данных начинается с построения модели предметной области. Наиболее распространенной является **ER-модель** (entity-relationship model) – модель «**Сущность-связь**».

Для построения ER-модели на практике будем использовать простую систему обозначений, предложенную Питером Ченом (обозначения, встречающиеся в разных источниках, могут несколько отличаться от нижеприведенных).

ER-модели можно просто рисовать на листочке бумаги, а также можно использовать программу **ERModeler**, разработанную автором специально для курсов по базам данных. В пособии приводятся примеры моделей, выполненные с помощью этой программы.

Базовые понятия:

Сущность (объект)	Сотрудник
Атрибут сущности (свойство, характеризующее объект)	ФИО
Ключевой атрибут (атрибут, входящий в первичный ключ)	ТабN
Связь	M : 1 Работает

Первичный ключ (primary key) – это атрибут или группа атрибутов, однозначно идентифицирующих объект.

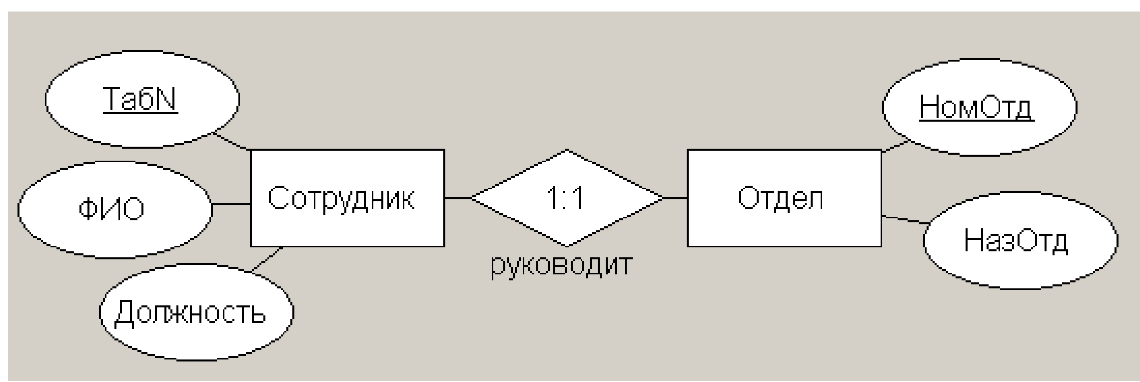
Первичный ключ может состоять из нескольких атрибутов, тогда подчеркивается каждый из них.

Объект и его атрибуты соединяются ненаправленными дугами.

Связи между объектами могут быть 3-х типов:

Один – к одному. Этот тип связи означает, что каждому объекту первого вида соответствует не более одного объекта второго вида, и наоборот.

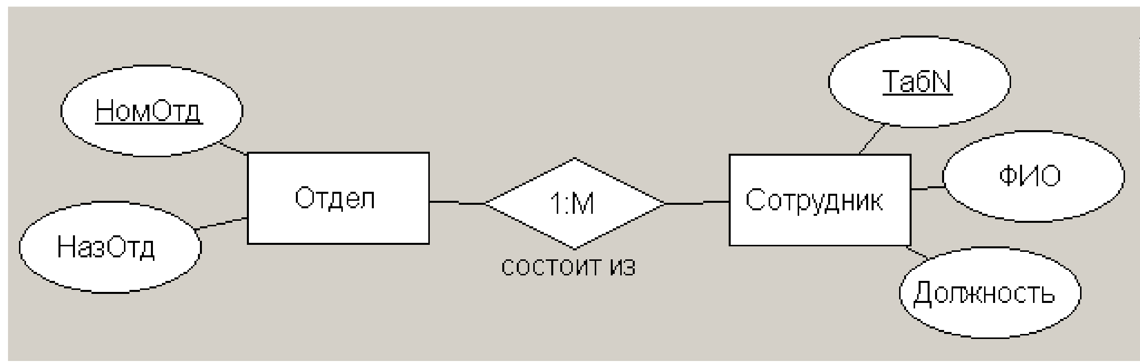
Например: сотрудник может руководить только одним отделом, и у каждого отдела есть только один руководитель.



Один – ко многим *(или в обратную сторону* ***Многие – к одному****)*. Этот тип связи означает, что каждому объекту первого вида может соответствовать более одного объекта второго вида, но

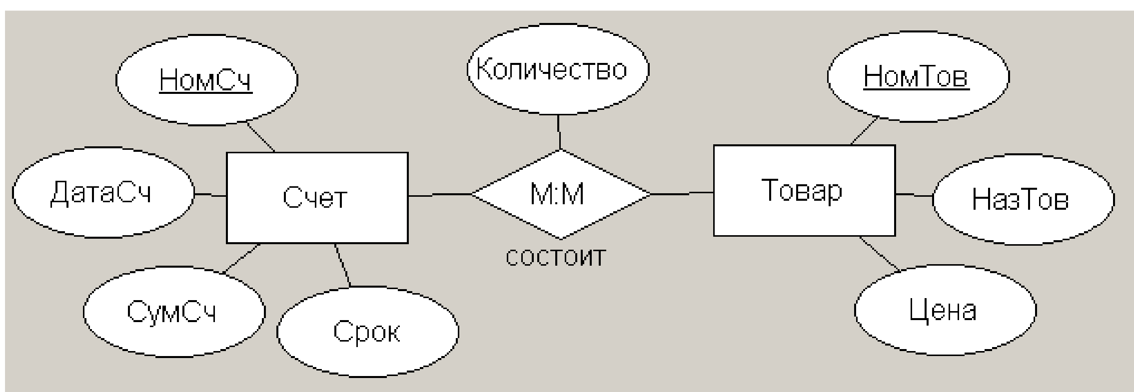
каждому объекту второго вида соответствует не более одного объекта первого вида.

Например: в каждом отделе может быть множество сотрудников, но каждый сотрудник работает только в одном отделе.

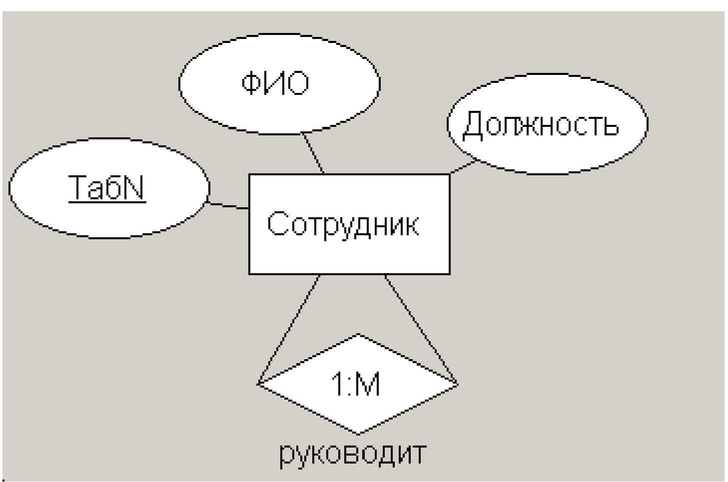


Многие – ко многим. Этот тип связи означает, что каждому объекту первого вида может соответствовать более одного объекта второго вида, и наоборот. У этого типа связи иногда бывают собственные атрибуты.

Например: каждый счет может включать множество товаров, и каждый товар может входить в разные счета.



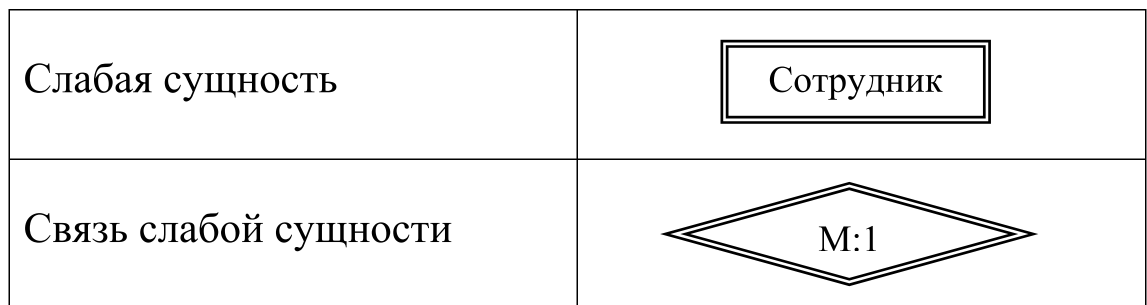
Связь может соединять сущность саму с собой, например:



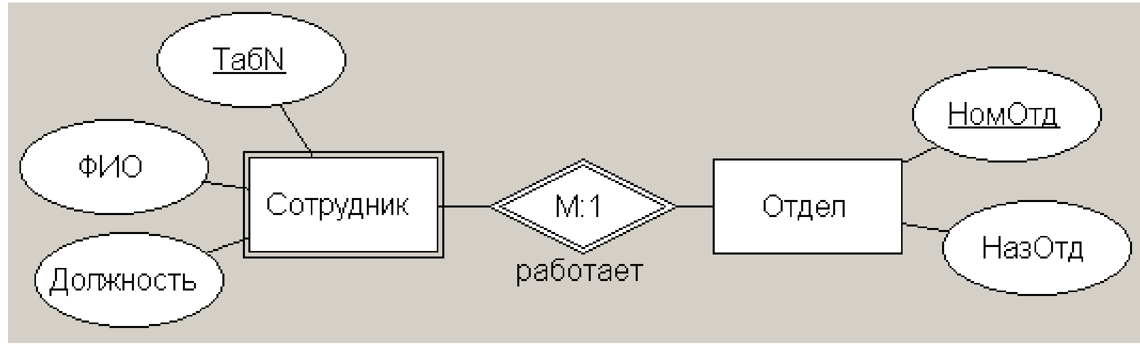
Иногда используют такое понятие, как ***слабая сущность***. Это сущность, которая не может быть однозначно идентифицирована с помощью собственных атрибутов, а только через связь с другой сущностью.

Пусть, например, номер сотрудника является уникальным только в пределах отдела, т.е. в разных отделах могут быть сотрудники с одинаковыми номерами. Уникальной в данном случае будет комбинация атрибутов «*НомерСотрудника, НомерОтдела*». Сущность «Сотрудник» является слабой.

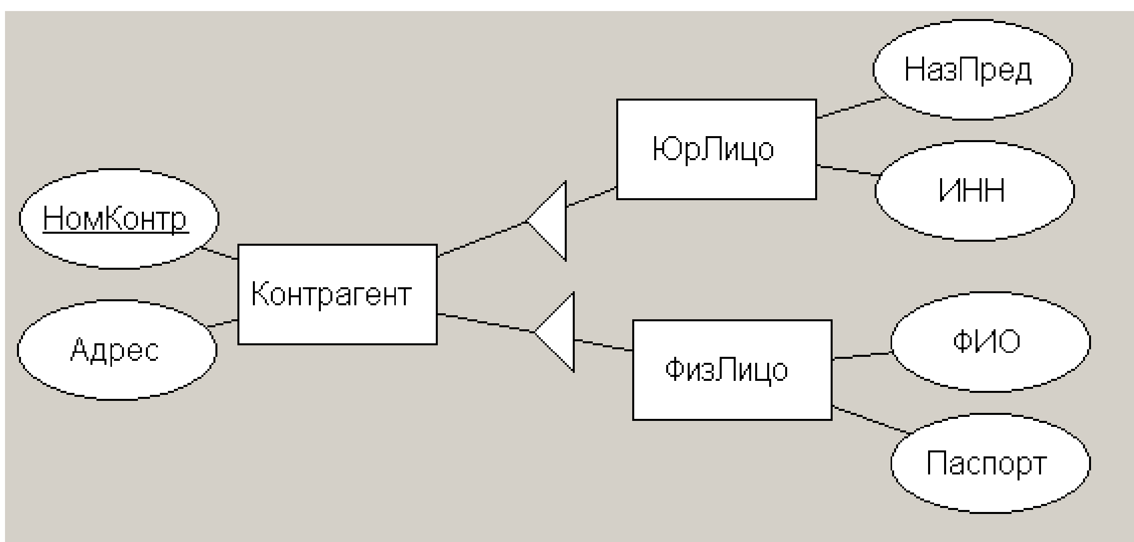
На схеме слабые сущности и их идентифицирующие связи обозначаются двойными линиями.



Слабая сущность	Сотрудник
Связь слабой сущности	M:1



Иногда для более удобной классификации используются так называемые ***подтипы сущностей.*** Их обозначают с помощью треугольника, направленного от подтипа к надтипу. Этот треугольник может сопровождаться надписью «есть» или «is a» (т.е., «является»).

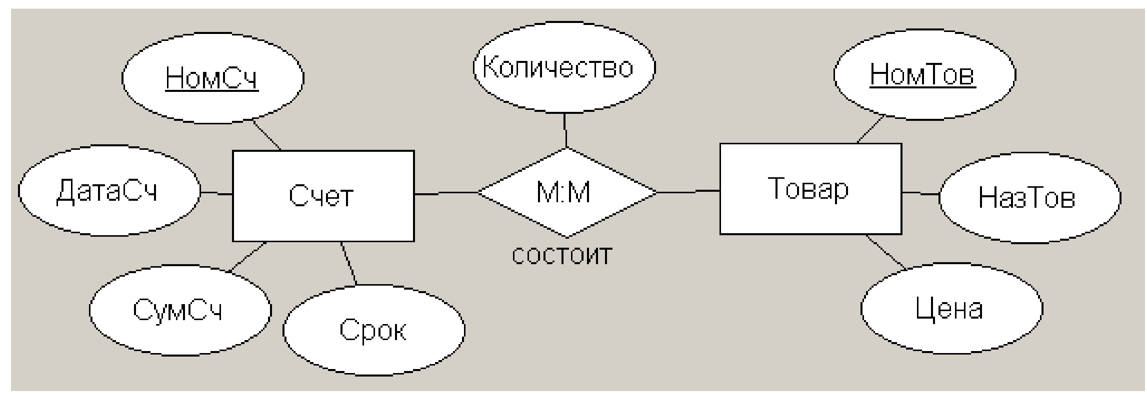


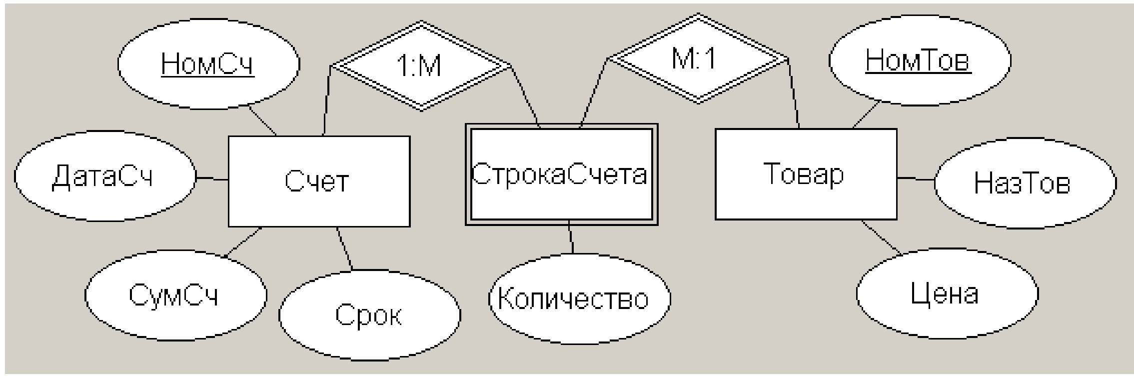
Пусть, например, среди контрагентов могут быть как физические, так и юридические лица. Поскольку они имеют разные атрибуты, то удобно создать для них подтипы.

Сущность «Контрагент» является ***надтипом*** для своих подтипов. Обратите внимание, что у подтипов обычно не бывает собственных первичных ключей.

Замечания по поводу связи М:М

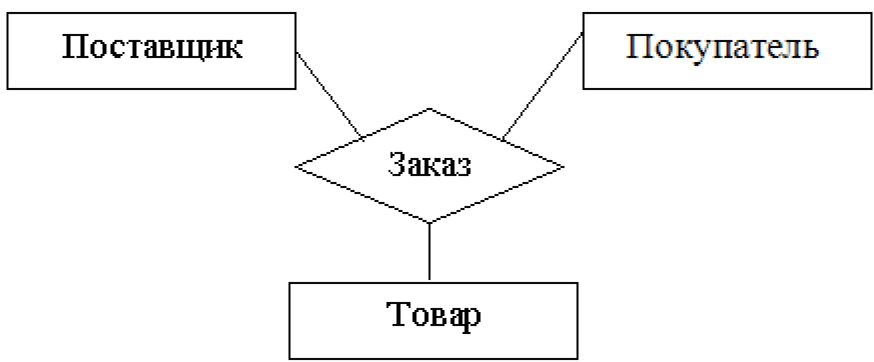
На самом деле этот тип связи представляет собой «замаскированную» слабую сущность, которая связана с другими двумя сущностями идентифицирующими связями многие – к одному:





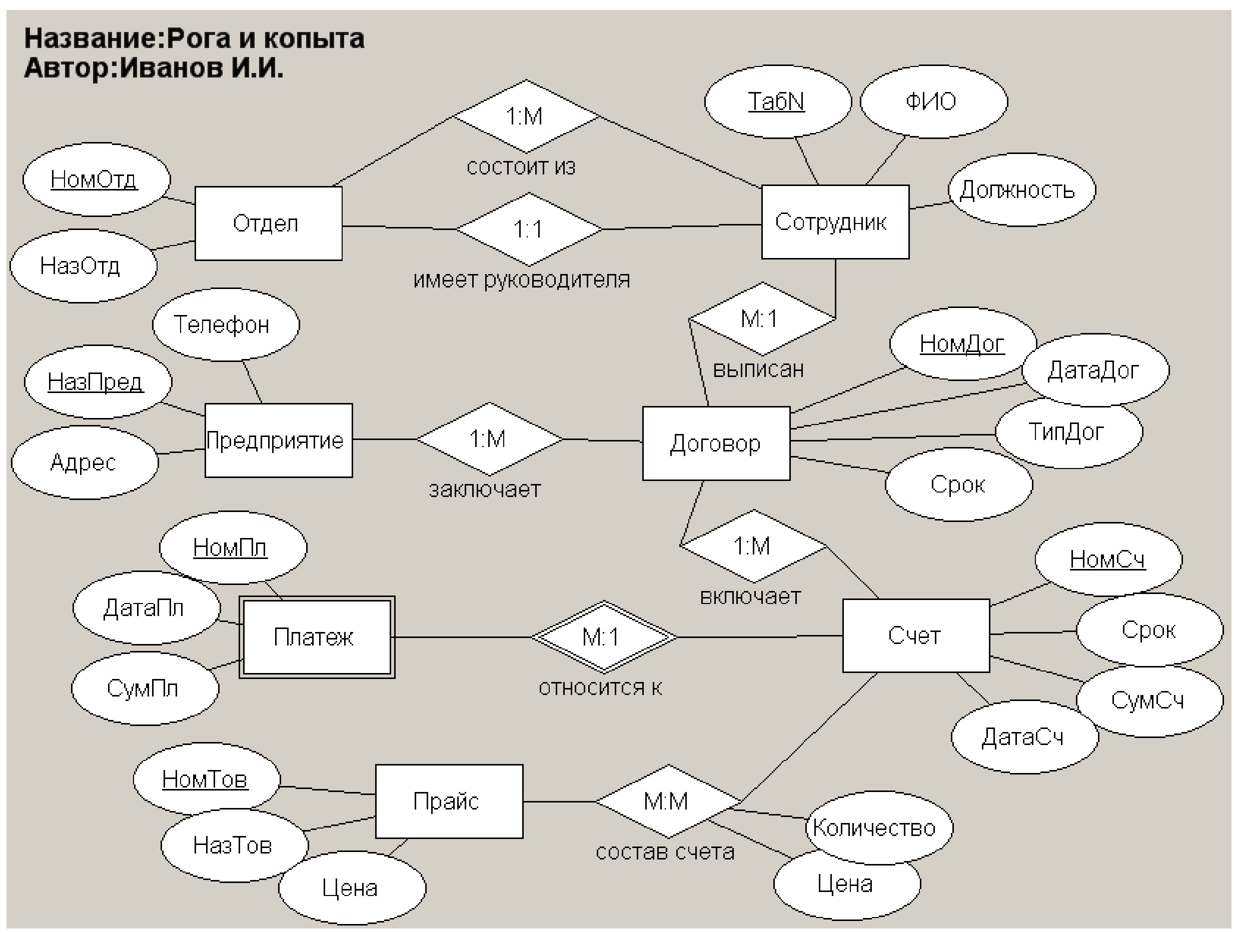
Если связь соединяет две сущности, она называется ***бинарной***.

Связь может соединять более двух сущностей, например, связь, соединяющая три сущности, называется ***тернарной***:



Связь с арностью более 2 обычно имеет тип **многие – ко многим** по отношению ко всем связанным сущностям.

Пример ER-модели: Контора «Рога и копыта»



Описание задачи

Контора «Рога и копыта» занимается коммерческой деятельностью по реализации продукции, произведенной из рогов и копыт, и предоставлению магических услуг.

Сотрудник организации имеет ФИО, табельный номер, должность. Сотрудники распределены по нескольким отделам. Каждый отдел имеет номер, название и руководителя. Сотрудник не может руководить более чем одним отделом.

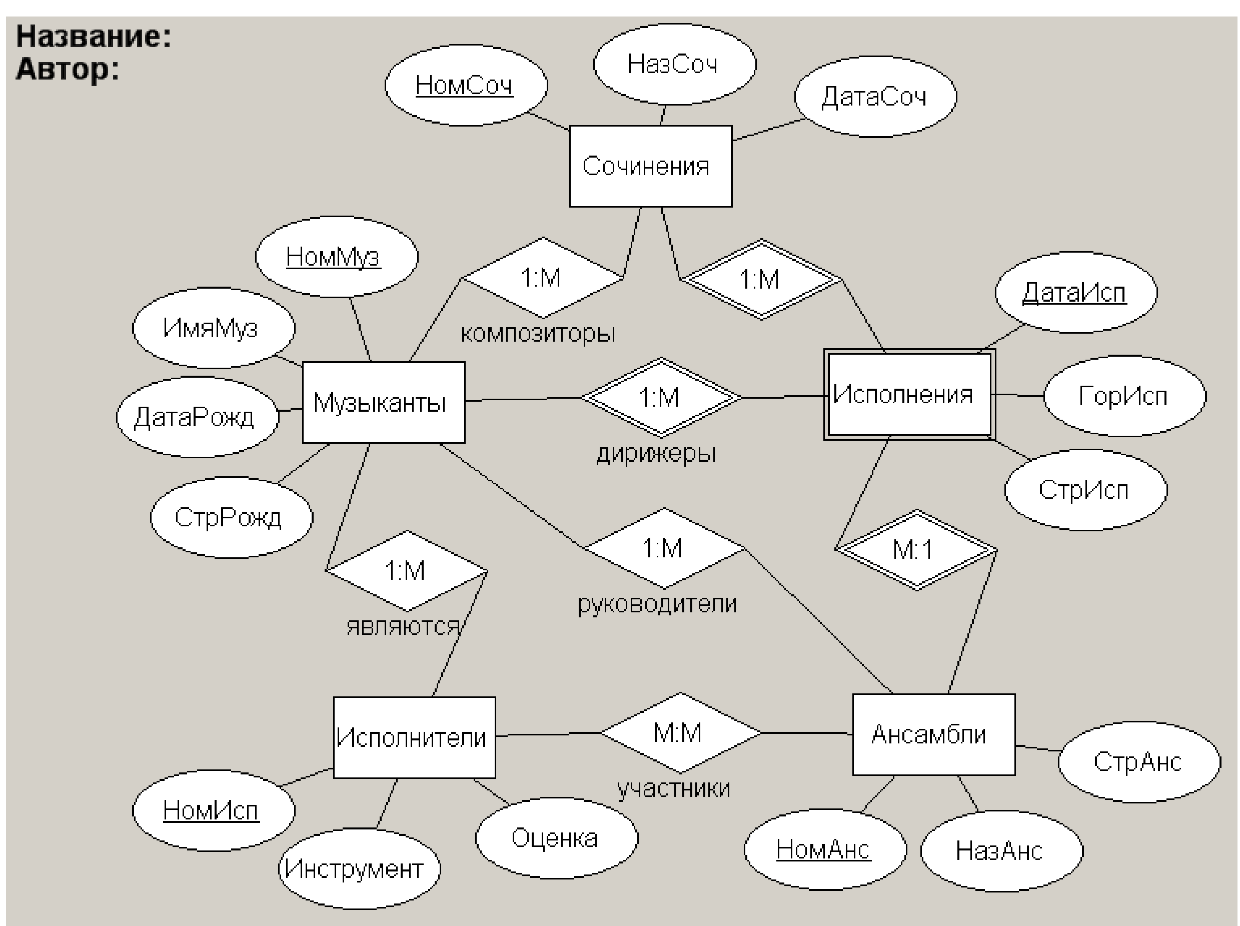
Организация работает с предприятиями-клиентами. Каждое предприятие имеет название и адрес. С предприятием может быть заключено несколько договоров.

Договор характеризуется уникальным номером, датой и типом. Каждый договор курирует некоторый сотрудник. По мере

реализации клиенту товаров и услуг по договору с некоторой периодичностью выставляются счета.

Счет характеризуется уникальным номером, датой выставления, сроком оплаты и суммой, а также списком реализованных товаров и услуг с указанием их количества. По неоплаченным счетам начисляются пени. Счет может быть оплачен в несколько приемов, каждый платеж характеризуется номером, датой и суммой. Номер платежа уникален в пределах его счета. Цены на товары и услуги могут изменяться со временем.

Пример ER-модели: «Музыканты»



Описание задачи

Необходимо разработать базу данных для хранения информации о музыкантах, сочинениях и концертах.

Музыкант характеризуется именем, датой рождения и страной рождения.

Сочинение включает информацию о названии, композиторе и дате первого исполнения.

Музыкант может играть на разных инструментах с разной степенью квалификации.

Из музыкантов-исполнителей формируются ансамбли. Каждый ансамбль, кроме своих участников, содержит информацию о названии, стране и руководителе.

Наконец, исполнения произведений характеризуются датой, страной, городом исполнения, а также ансамблем, дирижером и собственно исполняемым произведением.

Задание. Выберите предметную область (3-4 сущности и связи между ними). В произвольной форме опишите её. Нарисуйте ER-диаграмму базы данных.

Дополнительная информация. Подробнее о проектировании базы данных и построении ER-модели можно почитать в [**1**, Глава 1, параграфы 1.2, 1.3], [**4**, Глава 3].

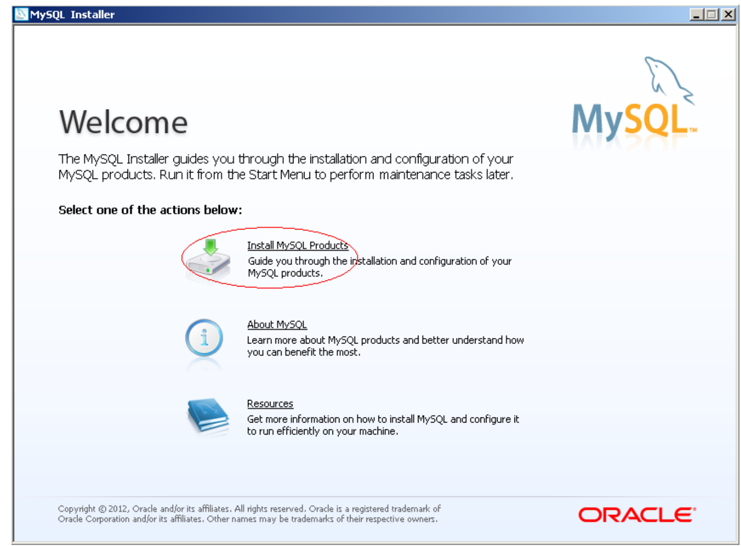
Установка MySQL

Рассмотрим установку пакета программ для Windows, содержащего **MySQL server**, среду для разработки и администрирования **MySQL Workbench** *и много других полезных компонентов*. (Источник - http://www.mysql.com/downloads/installer/)

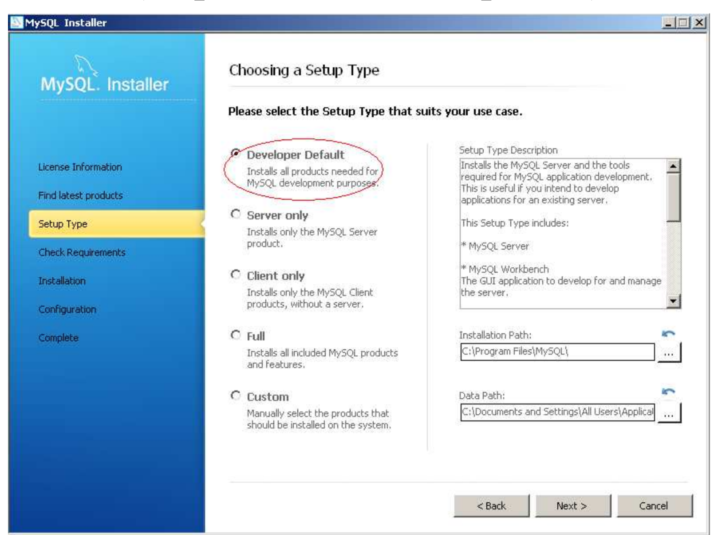
1. Для начала нужно установить Microsoft .NET 4.0 Framework, если его нет.

2. Установить Visual C++ Redistributable Packages for Visual Studio 2013.

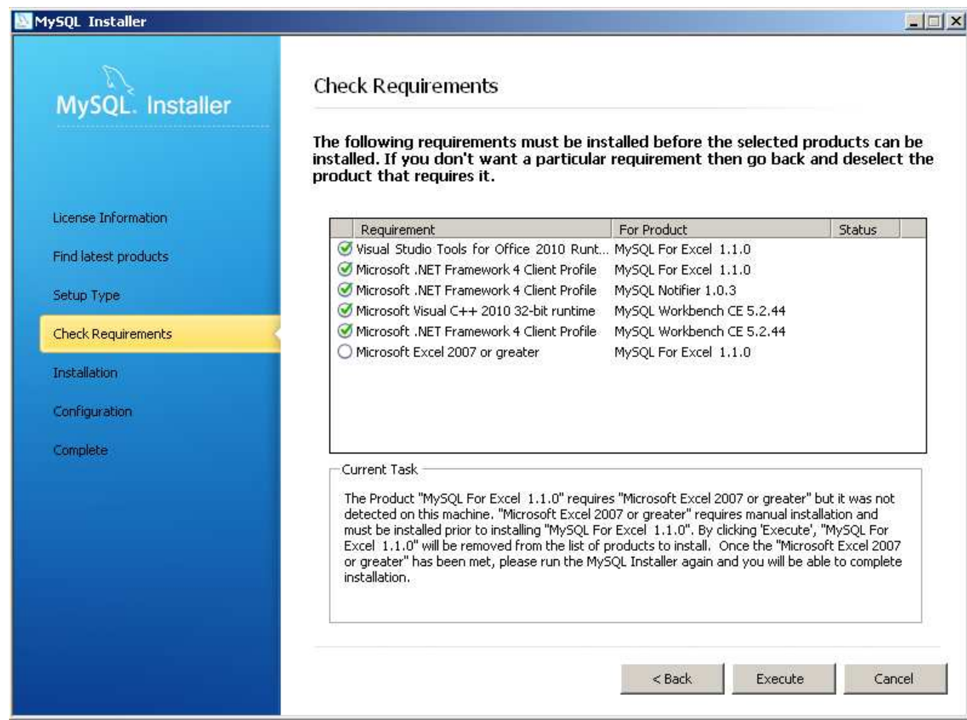
Теперь можно устанавливать **MySQL Workbench** - mysql-installer-community-5.5.28.3.msi или более новую версию:



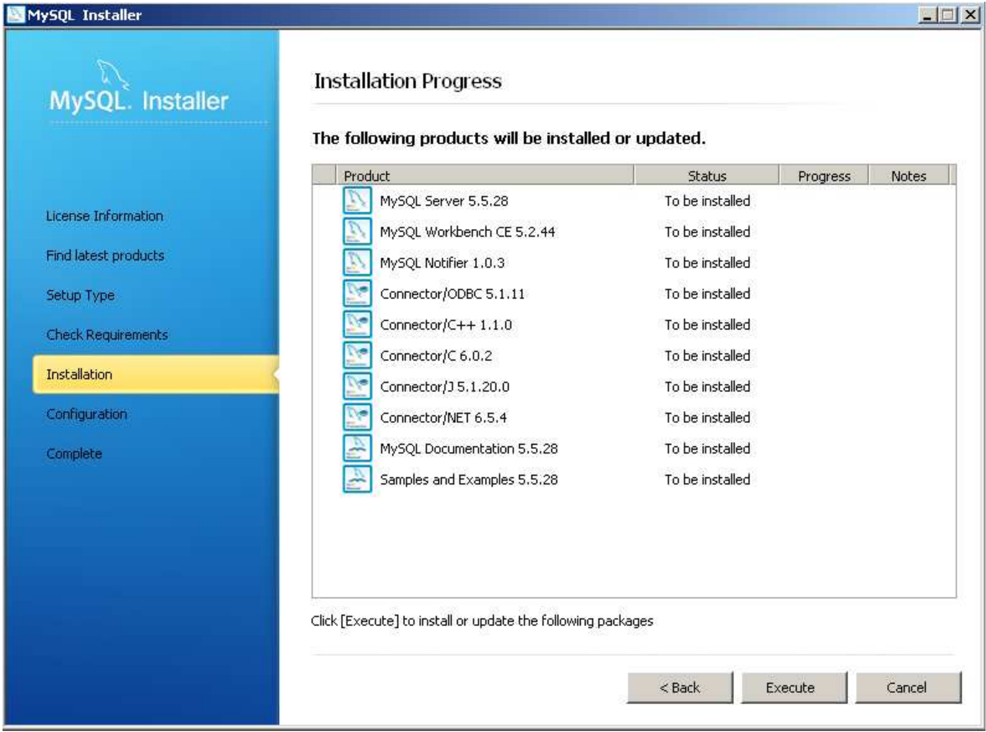
На 2 странице принимаем лицензию. На 3 странице **не проверяем** обновления (skip the check for updates).



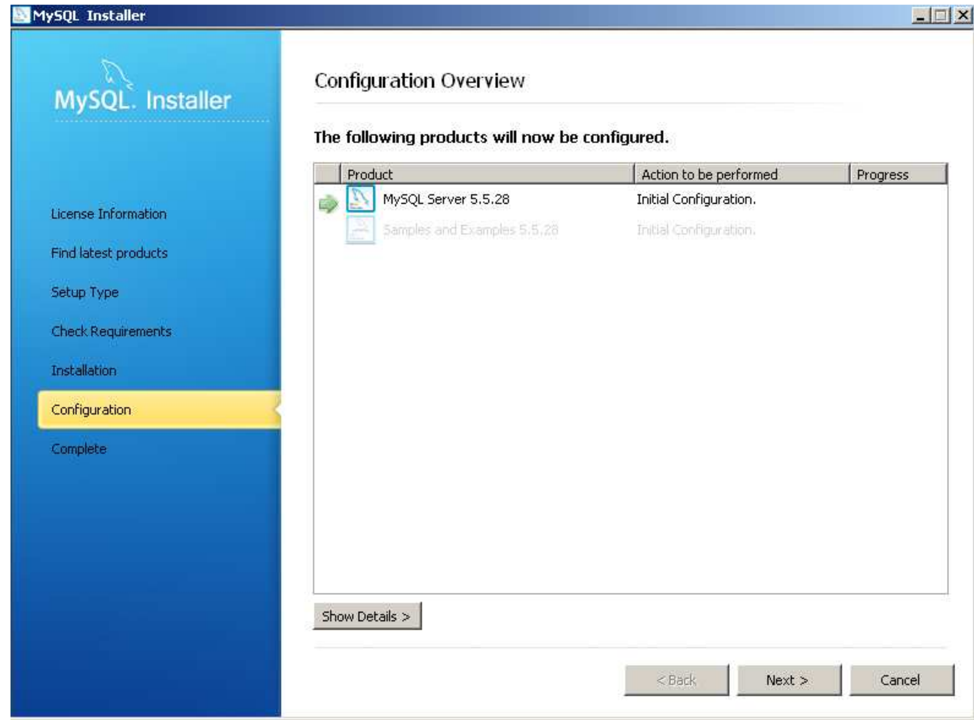
Проверяются необходимые условия для установки:



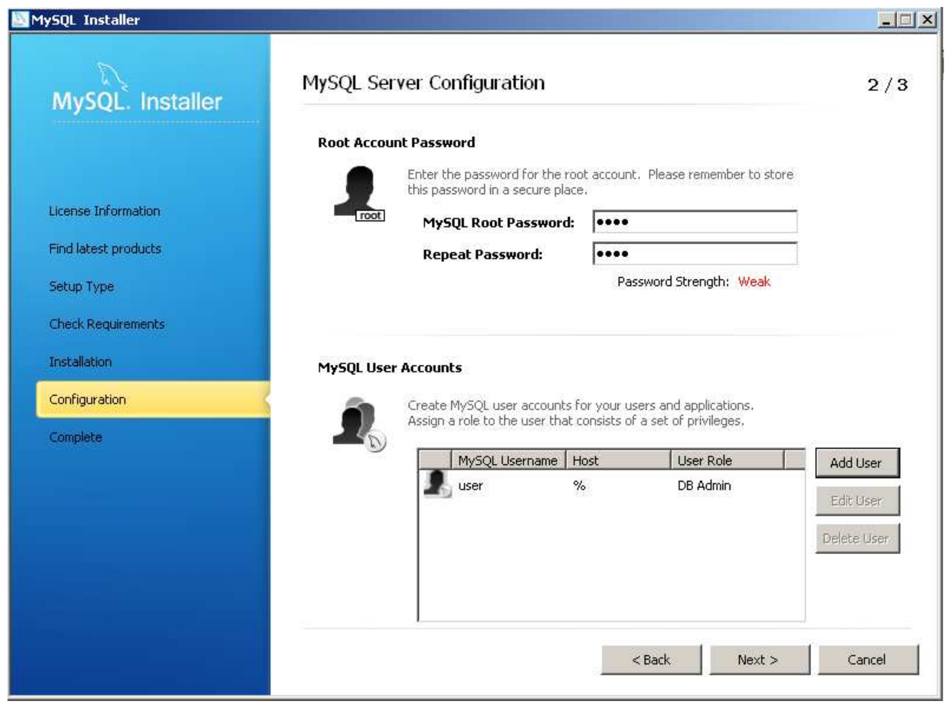
Устанавливаем следующие продукты: (первый пункт – это сам сервер, второй пункт – среда разработчика)



Конфигурирование сервера:



Для администратора по имени **root** зададим пароль. **Запомните его!** А также создадим пользователя по имени **user** с паролем. Т**оже запомните его!** Можно сделать пароли, совпадающие с именем пользователя: **root** и **user**, хотя в смысле безопасности это плохой вариант, но для учебных целей годится.



На следующих страницах ничего не изменяем. Имя сервера по умолчанию MySQL55. Установка успешно завершена.

Создание EER-диаграммы

Среда **mySQL Workbench** предназначена для визуального проектирования баз данных и управления сервером **mySQL**.

Для построения моделей предназначена секция Data Modeling:

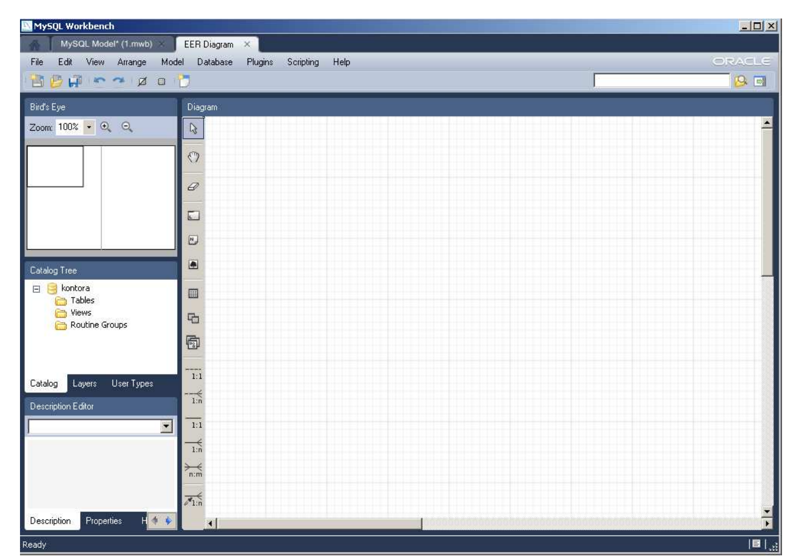
Выберем пункт **Create new EER Model**.

EER model расшифровывается как **Extended Entity-Relationship Model** и переводится как **Расширенная модель сущностей-связей**.

По умолчанию имя созданной модели **myDB**. Щелкните правой кнопкой мыши по имени модели и выберите в появившемся меню пункт **Edit schema.** В появившемся окне **можно** изменить имя модели. Назовем ее, например, **kontora**. В именах таблиц и столбцов нельзя использовать русские буквы.

В этом окне также **нужно** настроить так называемую «кодовую страницу» для корректного отображения русских букв **внутри** таблиц. Для этого выберите из списка пункт «**cp1251-cp1251_general_ci**». Окно свойств можно закрыть.

Диаграмму будем строить с помощью визуальных средств. Щелкнем по пункту **Add diagram**, загрузится пустое окно диаграммы:



Создать новую таблицу можно с помощью пиктограммы . Нужно щелкнуть по этой пиктограмме, а потом щелкнуть в рабочей области диаграммы. На этом месте появится таблица с названием по умолчанию **table1**. Двойной щелчок по этой таблице открывает окно редактирования, в котором можно изменить имя таблицы и настроить её структуру.

Будем создавать таблицу **Отделы** со следующими столбцами: номер_отдела, полное_название_отдела, короткое_название_отдела.

Переименуем **table1** в **k_dept** и начнем создавать столбцы.

Каждый столбец имеет:

- имя (не используйте русские буквы в имени!),
- тип данных. Самые распространенные типы данных:
    - INT – целое число;
    - VARCHAR(размер) – символьные данные переменной длины, в скобках указывается максимальный размер;
    - DECIMAL(размер, десятичные_знаки) – десятичное число;
    - DATE – дата:
    - DATETIME – дата и время.

Далее располагаются столбцы, в которых можно настроить дополнительные свойства поля, включив соответствующий флажок:

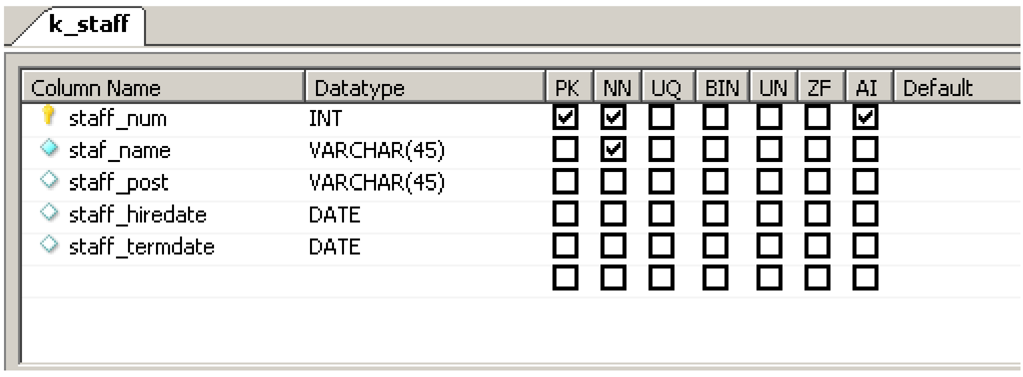
- PK (primary key) – первичный ключ;
- NN (not null) – ячейка не допускает пустые значения;
- UQ (unique) – значение должно быть уникальным в пределах столбца;
- AI (auto incremental) – это свойство полезно для простого первичного ключа, оно означает, что первичный ключ будет автоматически заполняться натуральными числами: 1, 2, 3, и т.п.;
- DEFAULT – значение по умолчанию, т.е., значение, которое при добавлении новой строки в таблицу автоматически вставляется в ячейку сервером, если пользователь оставил ячейку пустой.

Таблица **Отделы** имеет следующий вид:

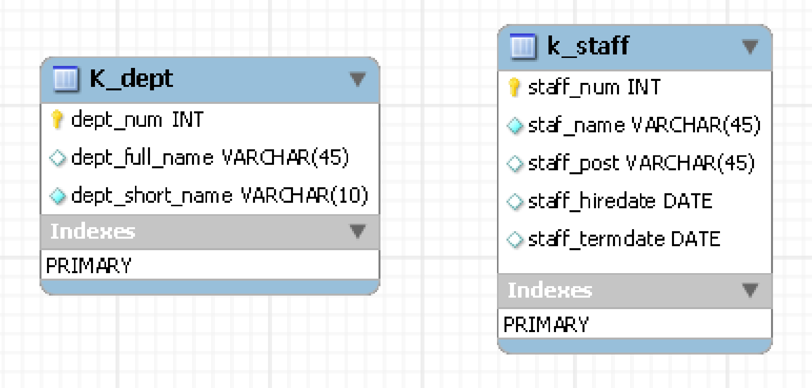
k_dept

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
dept_num	INT	☑	☑	☐	☐	☐	☐	☑	
dept_full_name	VARCHAR(45)	☐	☐	☐	☐	☐	☐	☐	
dept_short_name	VARCHAR(10)	☐	☑	☐	☐	☐	☐	☐	
		☐	☐	☐	☐	☐	☐	☐	

Далее создадим таблицу **Сотрудники** со следующими столбцами: номер_сотрудника, имя_сотрудника, должность, дата_начала_контракта, дата_окончания_контракта



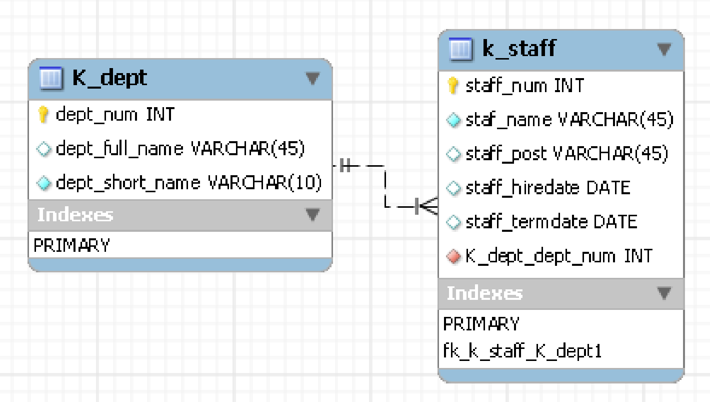
На диаграмме созданные таблицы выглядят следующим образом:



Обратите внимание, что при создании первичного ключа автоматически создается **индекс** по этому первичному ключу. **Индекс** представляет собой вспомогательную структуру, которая служит, прежде всего, для **ускорения поиска** и **быстрого доступа** к данным.

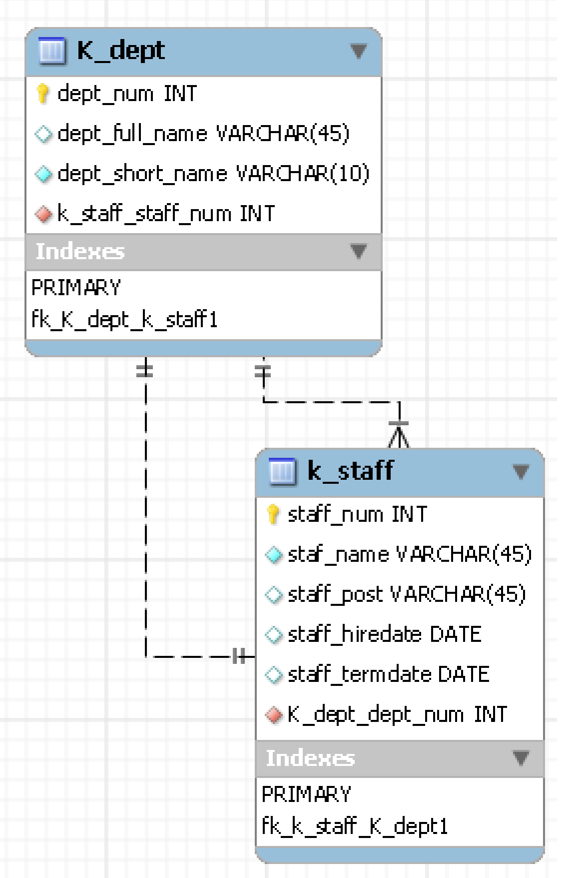
Теперь свяжем эти таблицы. Сначала создадим связь «Работает» между **Сотрудником** (дочерняя таблица) и **Отделом** (родительская таблица), степень связи М:1. Для создания связей М:1 служит пиктограмма на панели инструментов (с пунктирной линией). С ее помощью создается так называемая «неидентифицирующая связь», т.е. обыкновенный внешний ключ, при этом первичный ключ родительской таблицы добавляется в список столбцов дочерней таблицы.

Итак, щелкнем на пиктограмме, затем щелкнем на дочерней таблице **Сотрудники**, затем на родительской таблице **Отделы**:



Обратите внимание, что при этом произошло. Между таблицами образовалась пунктирная линия; в сторону «к одному» она отмечена двумя черточками, в сторону «ко многим» - «куриной лапкой». Кроме того, в таблице **Сотрудники** образовался дополнительный столбец, которому автоматически присвоено имя *k_dept_dept_num* (т.е., имя родительской таблицы плюс имя первичного ключа родительской таблицы). А в группе **Индексы** создан индекс по внешнему ключу.

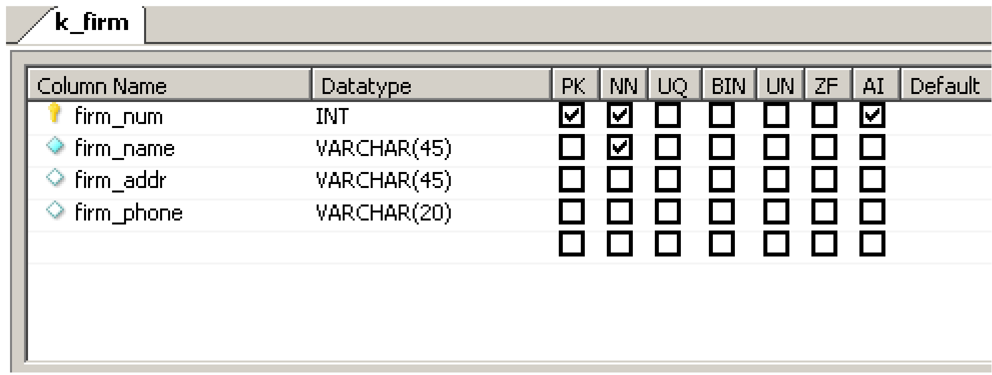
Теперь добавим связь между этими же таблицами «Руководит» 1:1. Выберем пиктограмму 1:1, затем щелкнем по **Отделам**, затем по **Сотрудникам**.



Чтобы 2 связи на картинке не «завязывались узлом», мы их разместили друг под другом.

Обратите внимание, что в таблицу Отделы был автоматически добавлен столбец *k_staff_staff_num*, а также индекс по внешнему ключу.

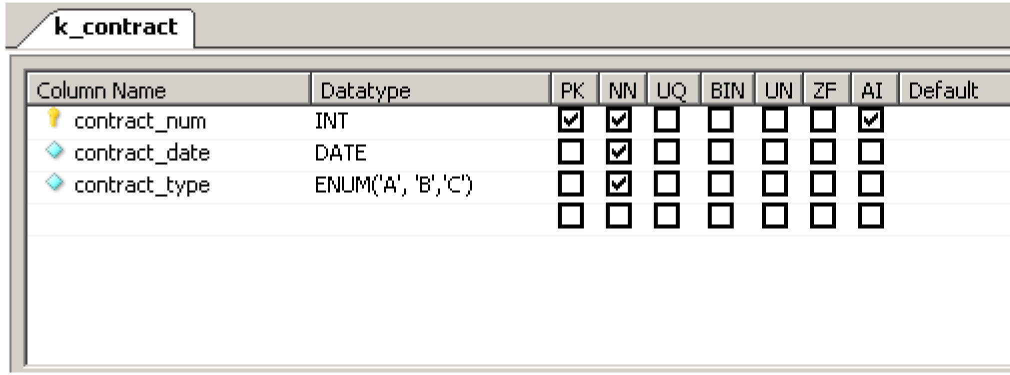
Создадим таблицу **Предприятия**:



k_firm

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
firm_num	INT	✓	✓					✓	
firm_name	VARCHAR(45)		✓						
firm_addr	VARCHAR(45)								
firm_phone	VARCHAR(20)								

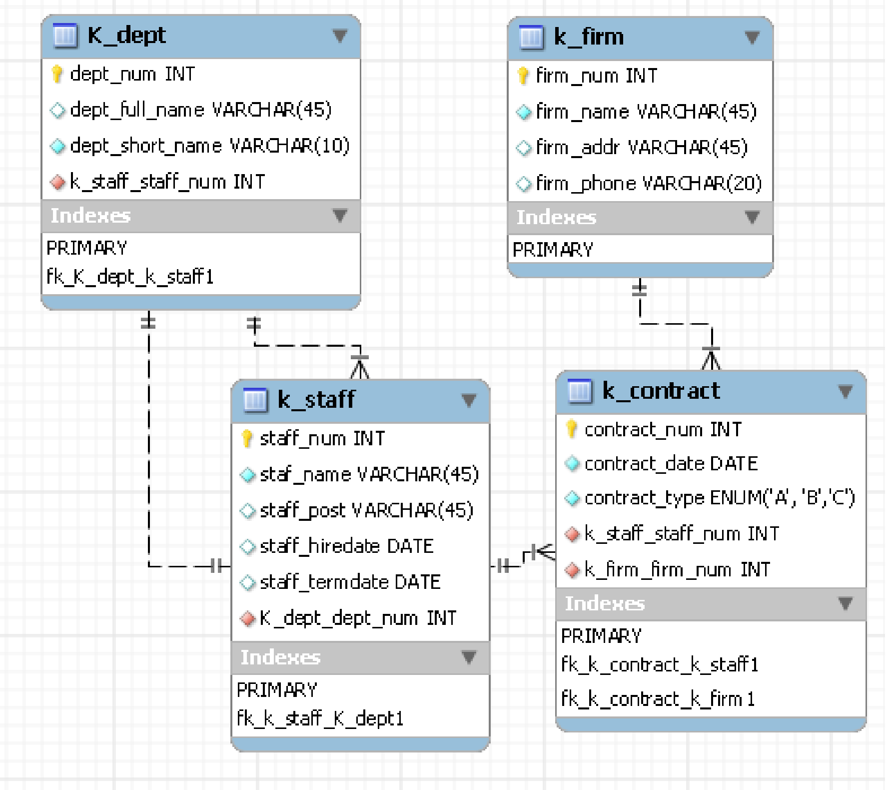
Создадим таблицу **Договоры**. У столбца Тип_договора зададим следующий формат: это буква из списка ‘A’, ‘B’, ‘C’.



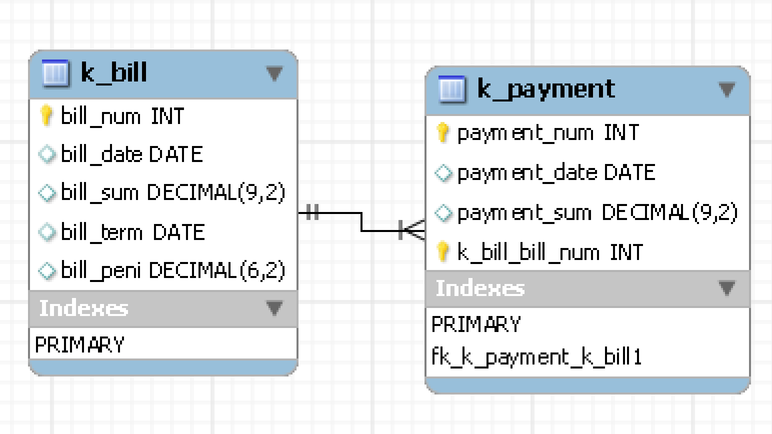
k_contract

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
contract_num	INT	✓	✓					✓	
contract_date	DATE		✓						
contract_type	ENUM('A', 'B','C')		✓						

Свяжем **Договоры** с **Сотрудниками** и **Предприятиями** связями М:1.

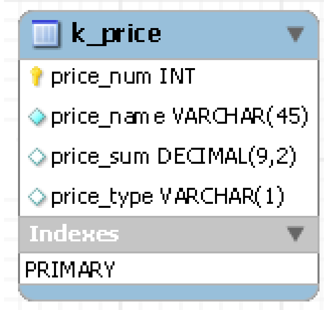


Затем создадим **Счета** и **Платежи**:



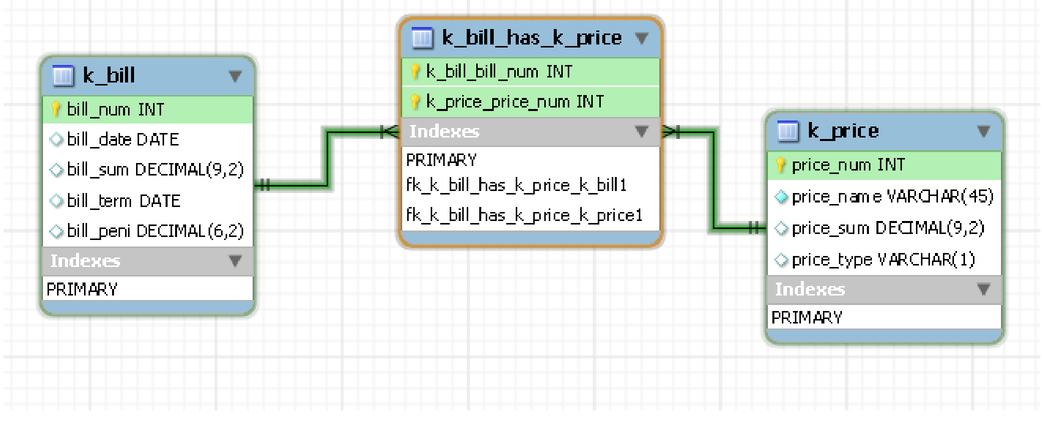
Поскольку сущность **Платеж** была «слабой», у нее нет полноценного первичного ключа, и каждый платеж однозначно идентифицируется группой атрибутов (номер_счета, номер платежа). Отметим в качестве ключевого поля *payment_num*, а затем создадим **идентифицирующую** связь между **Счетом** и **Платежом**.

Идентифицирующая связь создается с помощью пиктограммы (со сплошной линией). При этом новый столбец *k_bill_bill_num* становится не только внешним ключом в таблице **Платеж**, но и частью первичного ключа.

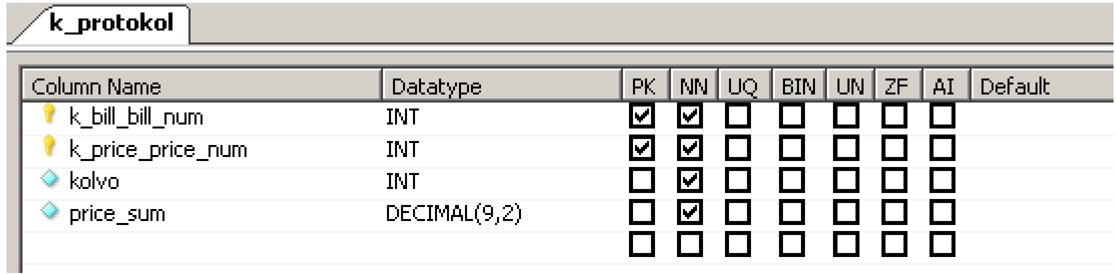


Далее создадим таблицу **Прайс-лист** со столбцами (номер_товара, название_товара, цена_товара и тип_товара).

Между объектами **Счет** и **Прайс-лист** имеется связь «многие - ко многим». Для создания этой связи нужно использовать пиктограмму . Следует щелкнуть мышью по этой пиктограмме, а затем последовательно щелкнуть по связываемым таблицам. Между ними появится новая таблица, обратите внимание на ее столбцы, первичный ключ и внешние ключи:



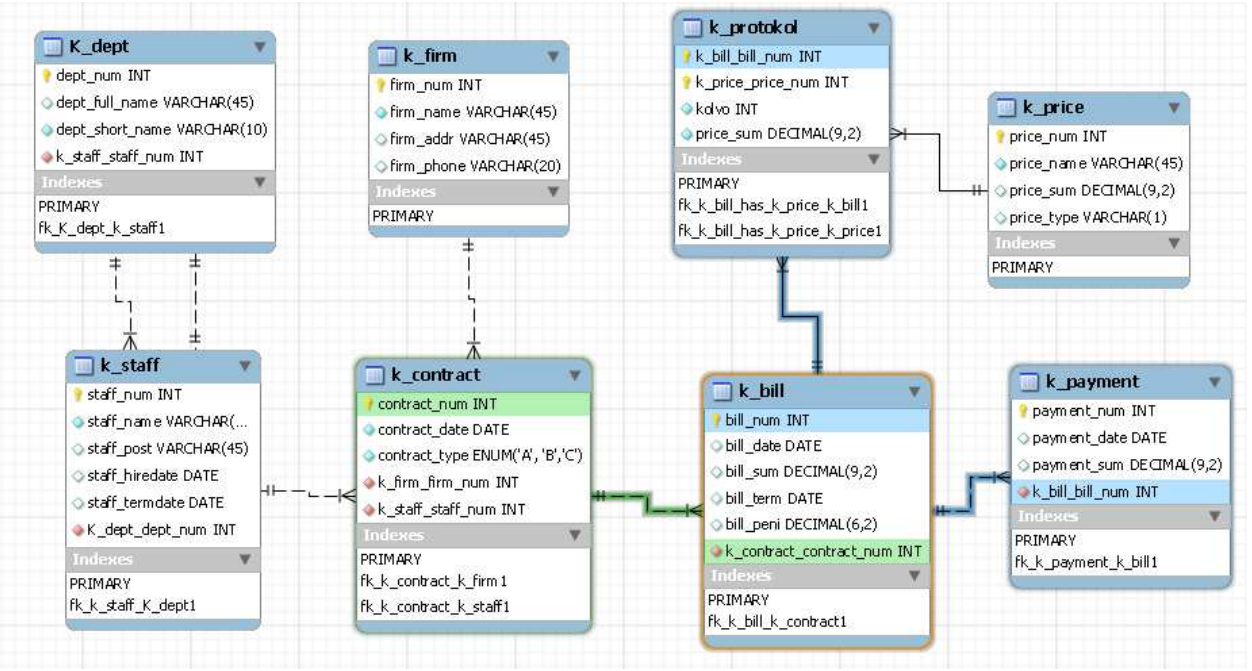
Для удобства переименуем эту таблицу в *k_protokol* (**ПротоколСчета),** добавим столбцы *kolvo* и *price_sum.*



k_protokol

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
k_bill_bill_num	INT	✓	✓						
k_price_price_num	INT	✓	✓						
kolvo	INT		✓						
price_sum	DECIMAL(9,2)		✓						

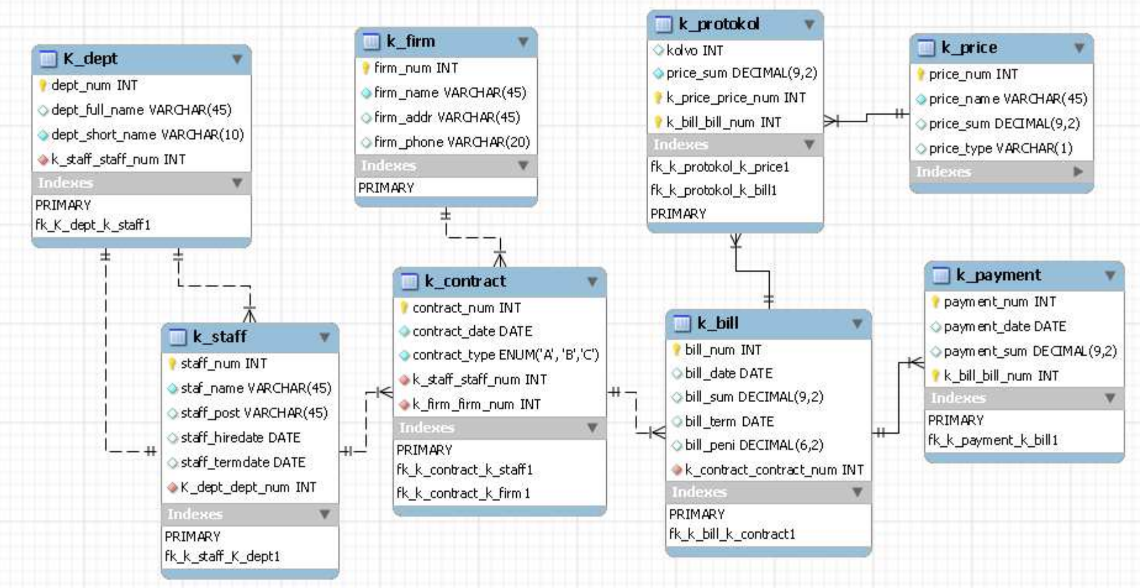
Теперь EER-диаграмма имеет такой вид:



Задание. Создайте в MySQL WORKBENCH EER-диаграмму для своей задачи.

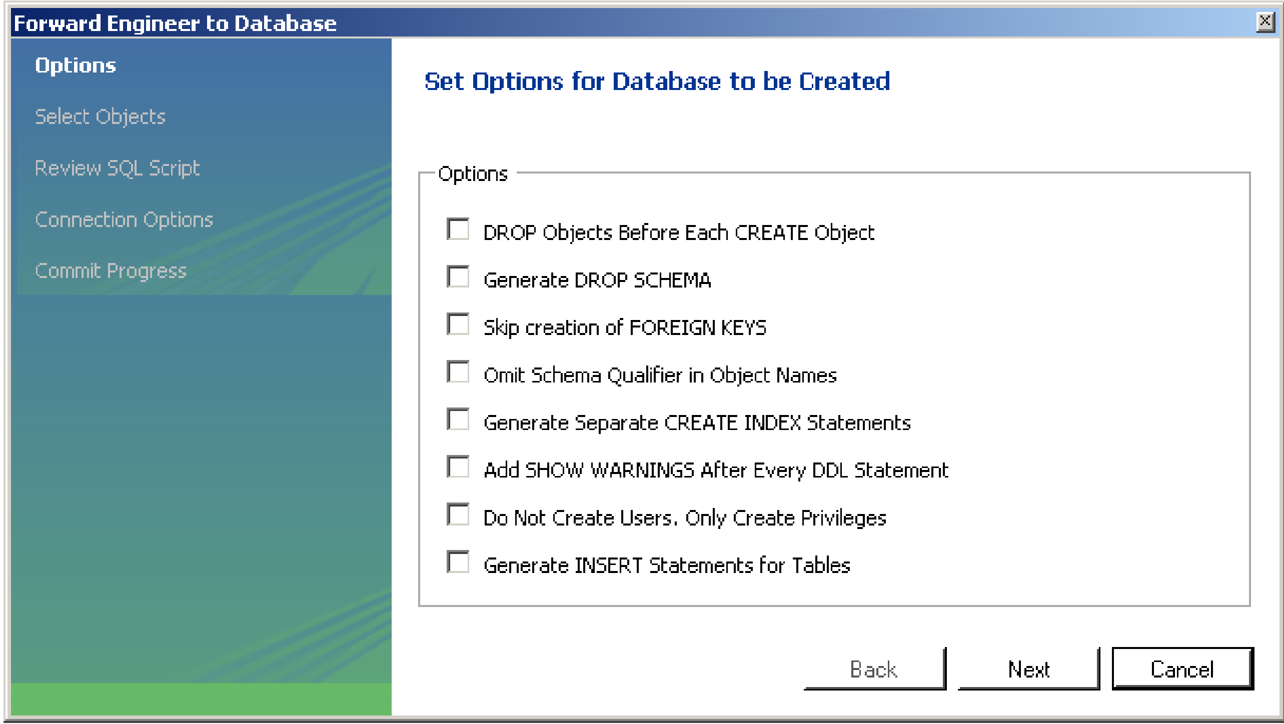
Создание базы данных из EER-диаграммы

На предыдущем этапе мы разработали EER-диаграмму для нашей предметной области:



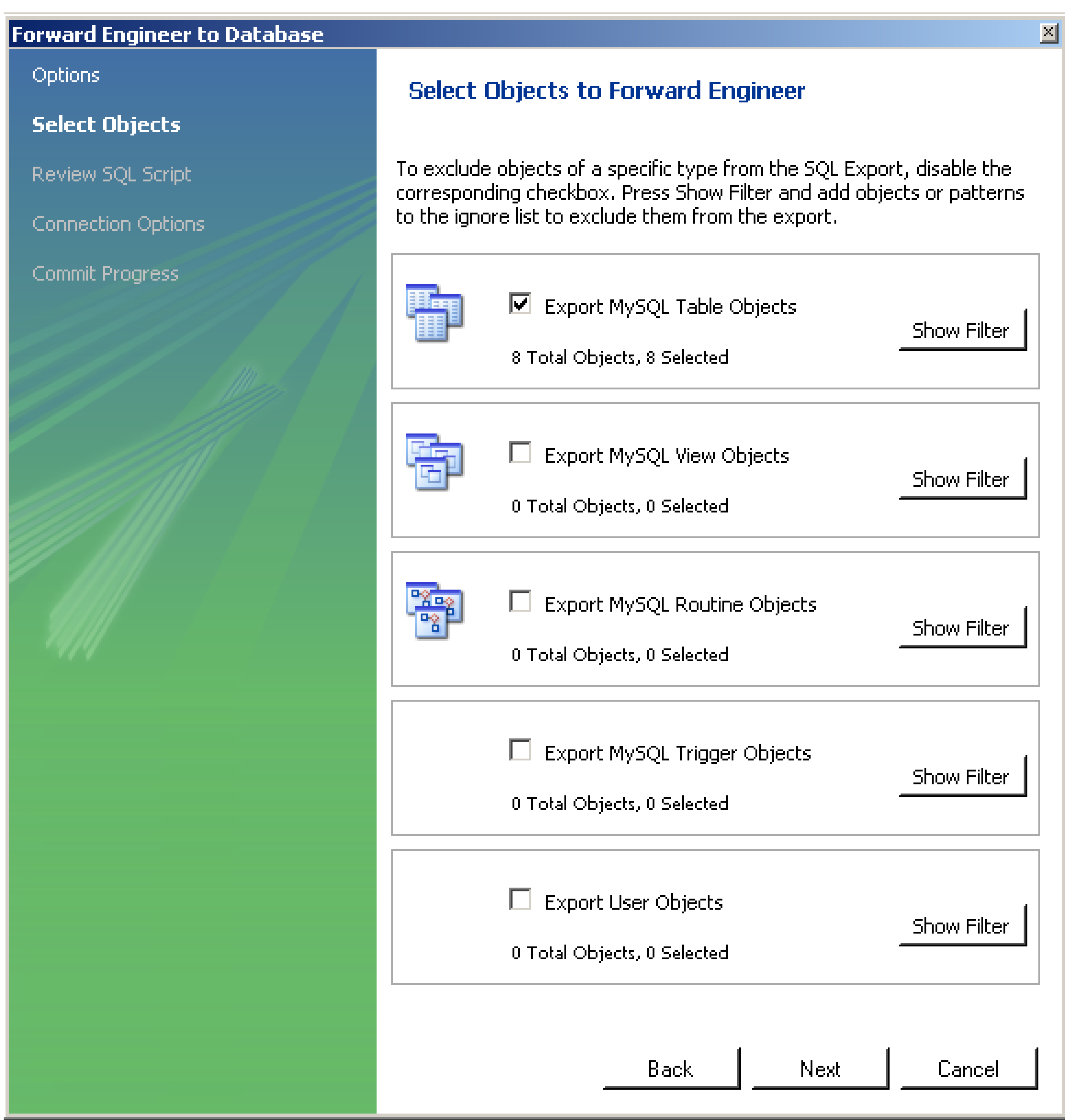
Теперь на основе этой диаграммы создадим физическую базу данных. Выберем пункт меню **Database - Forward Engeneer**. Запустится мастер построения базы данных.

На первом шаге можно задать некоторые дополнительные действия. Для начала ничего на этой странице не выбираем, нажимаем **Next.**

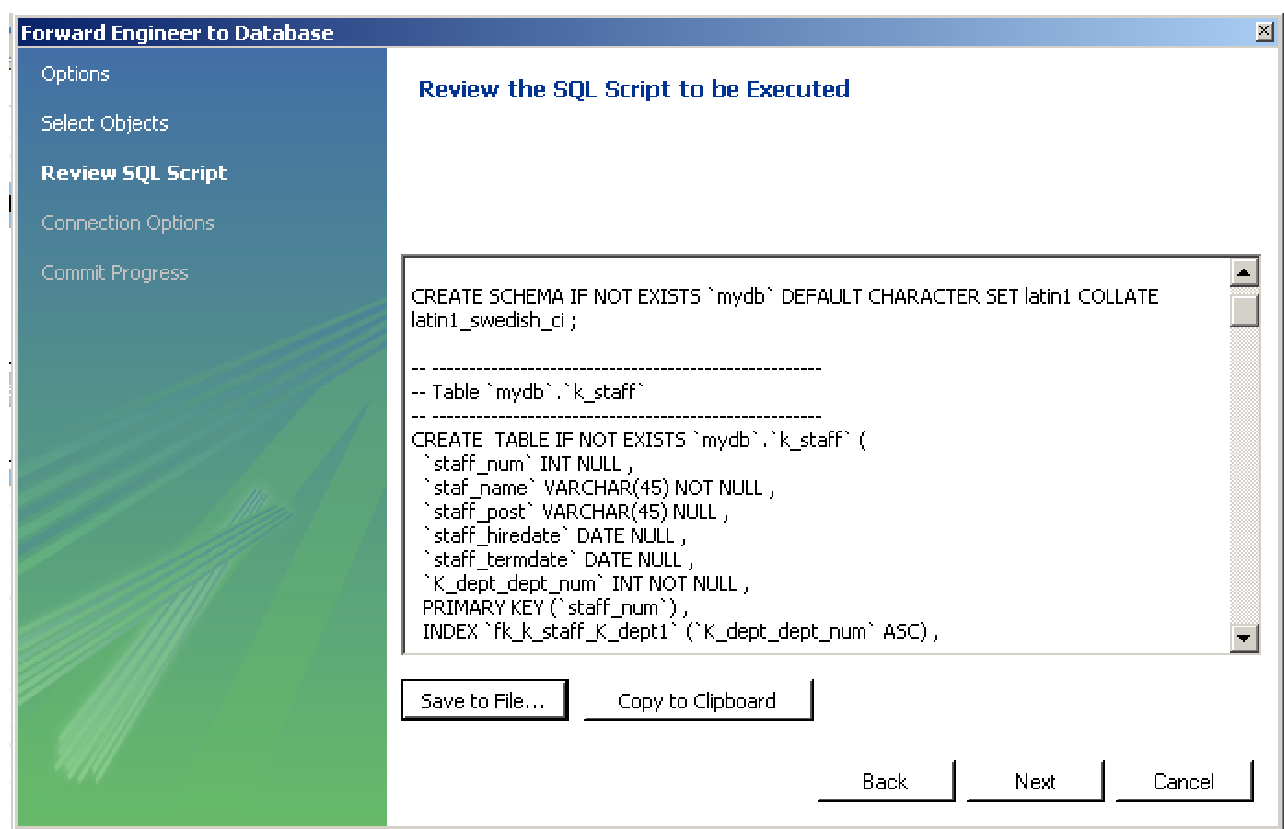


Примечание: при повторном создании базы данных нужно будет включить первые два флажка для удаления старых таблиц.

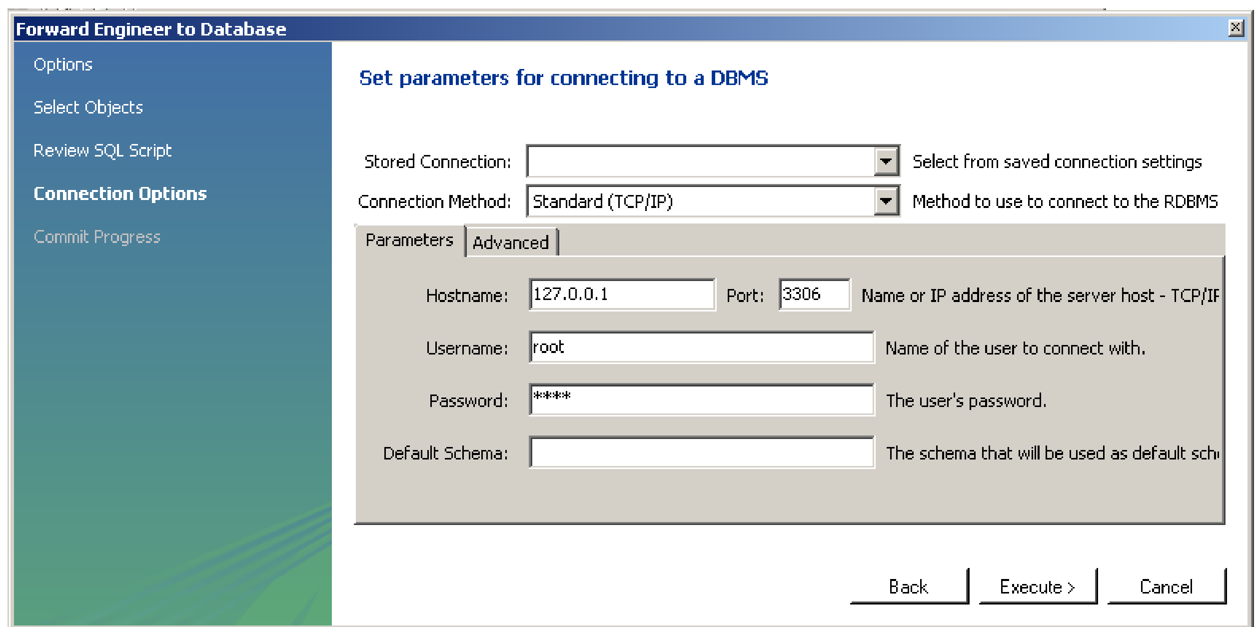
На второй странице включен флажок, указывающий, что мы создаем таблицы (всего 8 шт.). Других объектов пока у нас нет.



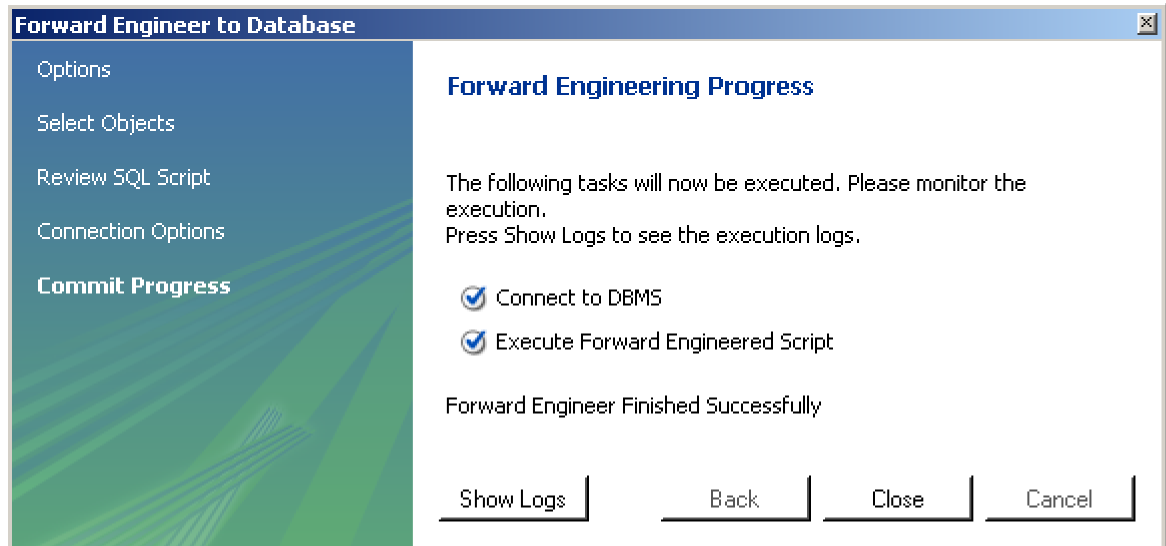
На следующей странице показывается текст сценария для создания базы данных. Его можно просмотреть прямо на месте, а также сохранить в файл (кнопка **Save to file**…). Полностью текст сценария приведен в **Приложении 1.**



Далее запрашивается логин и пароль для подключения к серверу:

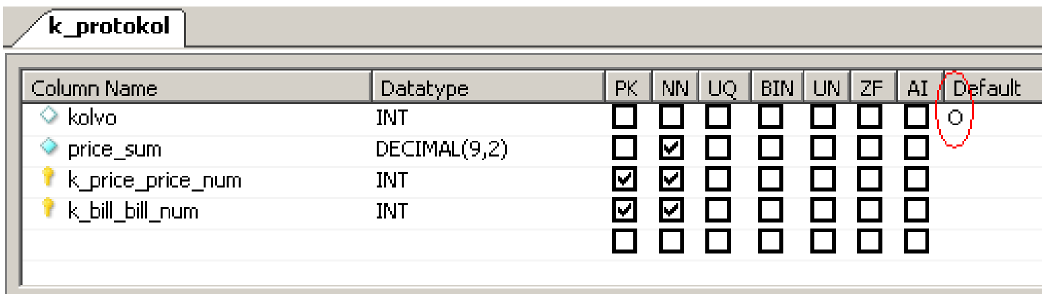


Если нет никаких ошибок, то получим окно с сообщением об успешном результате:

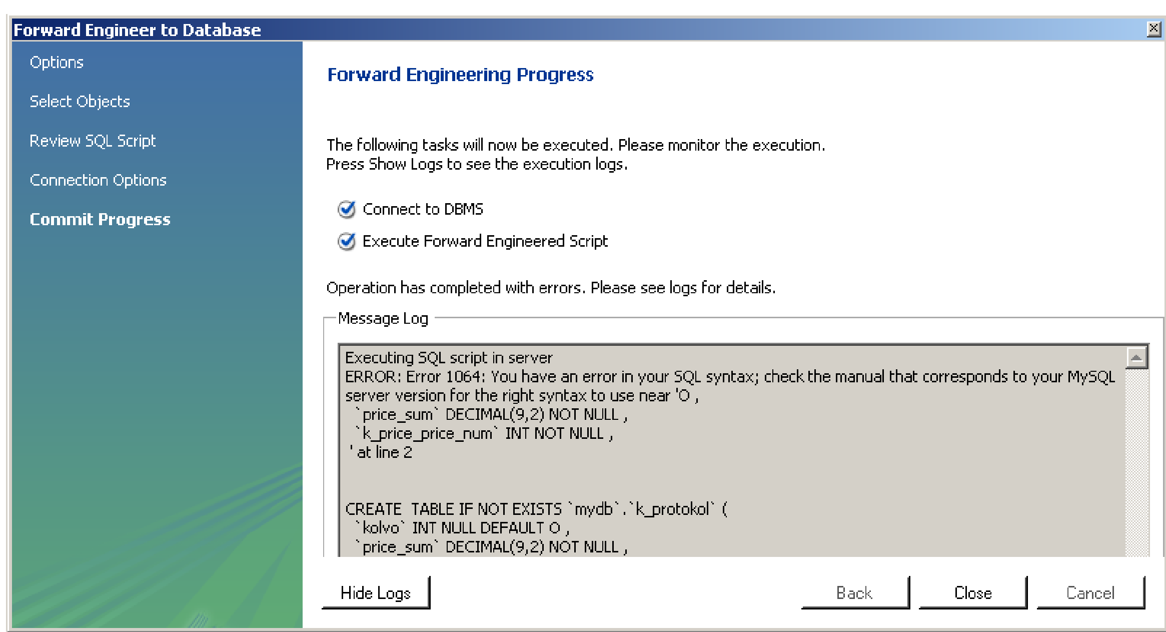


В противном случае можно нажать на кнопку **Show logs** и посмотреть протокол ошибок.

Какие могут быть ошибки? Например, в таблице ПротоколСчета мы захотели указать для количества значение по умолчанию «ноль», а вместо этого набрали букву О:



Получим ошибку:



Задание. Создайте в MySQL WORKBENCH базу данных из EER-диаграммы.

Дополнительная информация. При создании таблиц используется команда CREATE TABLE. Подробнее об этой команде можно прочитать в **[1**, Глава 2, параграфы 2.2, 2.3**]**, **[3**, Глава 4**]**.

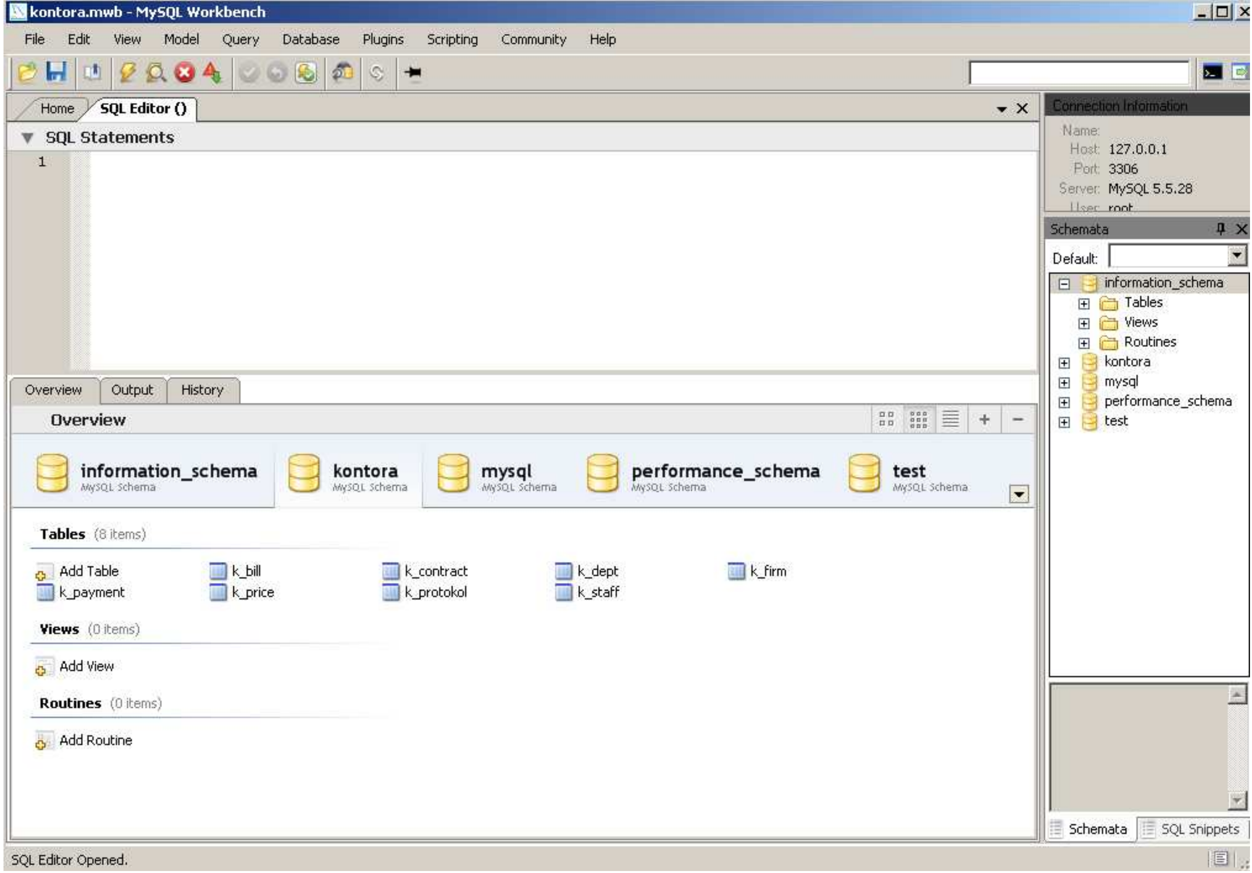
Заполнение базы данных, модификация данных

На предыдущем этапе мы создали базу данных. Теперь будем ее заполнять данными. Подключимся к серверу, в секции

щелкнем по ссылке Open Connection to Start Querying. В открывшемся окне нужно задать **username** и **password**, и нажать на кнопку «ОК».

Мы подключились к **MySQL server**.

В этом режиме работы рабочая область **MySQL Workbench** разделена на 3 окна:



- Верхнее окно **SQL Statements** предназначено для ввода и выполнения команд SQL. Внимание! В OS **Windows XP** текст, набранный в этом окне, автоматически не сохраняется. Если вы переместились из этого окна в какой-то другой режим работы, текст может быть потерян.
- Нижнее окно с несколькими вкладками показывает структуру имеющихся баз данных и позволяет ими управлять. Например,

если “дважды щелкнуть” по какой-либо таблице, то откроется дополнительное окно со структурой этой таблицы в нижней части рабочей области.

- Правое окно содержит иерархию объектов сервера.

Для заполнения базы данных в **MySQL Workbench** есть несколько возможностей. Рассмотрим три из них.

1-ый способ заполнения базы данных – используем команду INSERT

Самый универсальный и гибкий способ создания данных состоит в использовании SQL-команды INSERT. Формат у нее такой:

```
INSERT INTO ИмяТаблицы [(СписокСтолбцовТаблицы)]
VALUES (СписокЗначений);
```

В квадратных скобках указываются необязательные элементы команды. Если в этой команде пропустить СписокСтолбцовТаблицы, то имеются в виду ВСЕ столбцы, и именно в таком порядке, в каком они были определены при создании таблицы.

SQL-команды нужно набирать в окне **SQL statement**. Для выполнения команд нужно выбрать меню **Query – Execute** или кнопку на панели инструментов или нажать Ctrl+Enter.

Можно набрать несколько команд и выполнить их **все вместе**, или выделить **отдельную** команду (как для копирования) и выполнить только ее.

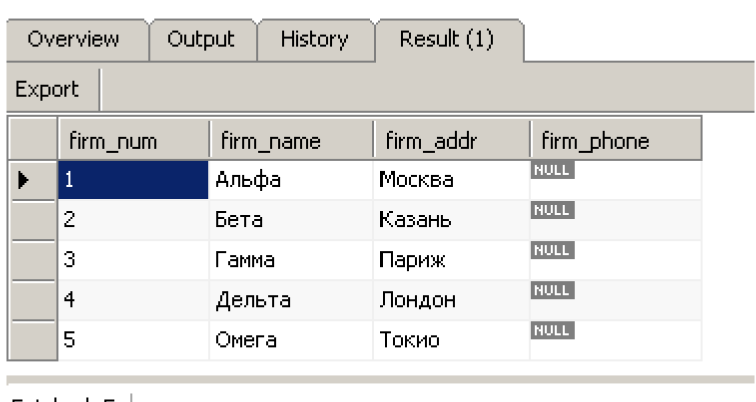
Текст SQL-команд, который также называют SQL-сценарием, можно (и нужно!) сохранять в файл. По умолчанию тип файла **sql**.

Заполним таблицу **Предприятия**:

```
# выберем базу данных
USE kontora;
# добавим строки
INSERT INTO k_firm (firm_name, firm_addr)
     VALUES('Альфа', 'Москва');
INSERT INTO k_firm (firm_name, firm_addr)
     VALUES('Бета', 'Казань');
```

```
INSERT INTO k_firm (firm_name, firm_addr)
     VALUES('Гамма', 'Париж');

INSERT INTO k_firm (firm_name, firm_addr)
     VALUES('Дельта', 'Лондон');

INSERT INTO k_firm (firm_name, firm_addr)
     VALUES('Омега', 'Токио');


# посмотрим результат

SELECT * FROM k_firm;
```

Обратите внимание, что мы не задавали значения для столбца *firm_num*, поскольку этот столбец имеет свойство **Auto increment**, и сервер его заполняет сам, натуральными числами.



firm_num	firm_name	firm_addr	firm_phone
1	Альфа	Москва	NULL
2	Бета	Казань	NULL
3	Гамма	Париж	NULL
4	Дельта	Лондон	NULL
5	Омега	Токио	NULL

Fetched: 5

Заполним **Отдел**

```
INSERT INTO k_dept (dept_short_name, dept_full_name)
     VALUES('Sales', 'Отдел продаж');

INSERT INTO k_dept (dept_short_name, dept_full_name)
     VALUES('Mart', 'Отдел маркетинга');

INSERT INTO k_dept (dept_short_name, dept_full_name)
     VALUES('Cust', 'Отдел гарантийного обслуживания');

SELECT * FROM k_dept;
```

Overview | Output | History | Result (1)

Export

dept_num	dept_full_name	dept_short_name	k_staff_staff_num
1	Отдел продаж	Sales	NULL
2	Отдел маркетинга	Mart	NULL
3	Отдел гарантийного обслуживания	Cust	NULL

Fetched: 3

Заполним таблицу **Сотрудник**. Обратите внимание, что в этой таблице можно указывать **только** такой номер отдела, который **существует** в таблице **Отдел**! (Оставить это поле пустым тоже можно.)

```
INSERT INTO k_staff
(staff_name, K_dept_dept_num, staff_hiredate, staff_post)
     VALUES('Иванов', 1, '1999-01-01', 'Менеджер');

INSERT INTO k_staff
(staff_name, K_dept_dept_num, staff_hiredate, staff_post)
     VALUES('Петров', 2, '2010-10-13','Менеджер');

INSERT INTO k_staff
(staff_name, K_dept_dept_num, staff_hiredate, staff_post)
     VALUES('Сидоров', 3, '2005-12-01','Менеджер');

INSERT INTO k_staff
(staff_name, staff_hiredate, staff_post)
     VALUES('Семенов', '1990-01-01','Директор');

INSERT INTO k_staff
(staff_name, K_dept_dept_num, staff_hiredate, staff_post)
     VALUES('Григорьев', 3, '2008-12-19','Программист');

SELECT * FROM k_staff;
```

Overview | Output | History | Result (1)

Export

staff_num	staff_name	staff_post	staff_hiredate	staff_termdate	K_dept_dept_num
1	Иванов	Менеджер	1999-01-01	NULL	1
2	Петров	Менеджер	2010-10-13	NULL	2
3	Сидоров	Менеджер	2005-12-01	NULL	3
4	Семенов	Директор	1990-01-01	NULL	NULL
5	Григорьев	Программист	2008-12-19	NULL	3

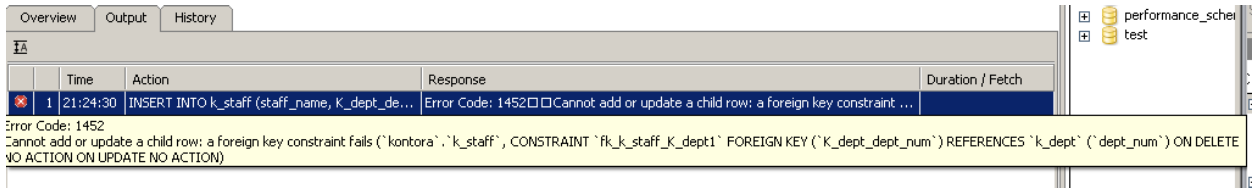
Fetched: 5

Что же будет, если указать несуществующий номер отдела?

```
INSERT INTO k_staff
(staff_name, K_dept_dept_num, staff_hiredate, staff_post)
     VALUES('Смит', 10, '2013-01-01','Консультант');
```

Будет получена следующая ошибка:

```
Error code: 1452

Cannot add or update a child row: a foreign key constraint fails
(`kontora`,`k_staff`, CONSTRAINT `fk_staff_k_dept` FOREIGN KEY
(`k_dept_dept_num`) REFERENCES `k_dept` (`dept_num`) …
```



Заполним таблицу **Договор**

```
INSERT INTO k_contract
(contract_type, k_firm_firm_num, k_staff_staff_num, contract_date)
     VALUES('A', 1, 1,'2011-11-01');

INSERT INTO k_contract
(contract_type, k_firm_firm_num, k_staff_staff_num, contract_date)
     VALUES('B', 1, 2,'2011-10-01');

INSERT INTO k_contract
(contract_type, k_firm_firm_num, k_staff_staff_num, contract_date)
     VALUES('C', 1, 1,'2011-09-01');

INSERT INTO k_contract
(contract_type, k_firm_firm_num, k_staff_staff_num, contract_date)
     VALUES('A', 2, 2,'2011-11-15');

INSERT INTO k_contract
(contract_type, k_firm_firm_num, k_staff_staff_num, contract_date)
     VALUES('B', 2, 2,'2011-08-01');

INSERT INTO k_contract
(contract_type, k_firm_firm_num, k_staff_staff_num, contract_date)
     VALUES('C', 3, 1,'2011-07-15');

INSERT INTO k_contract
(contract_type, k_firm_firm_num, k_staff_staff_num, contract_date)
     VALUES('A', 4, 1,'2011-11-12');


SELECT * FROM k_contract;
```

Overview | Output | History | Result (1)

Export

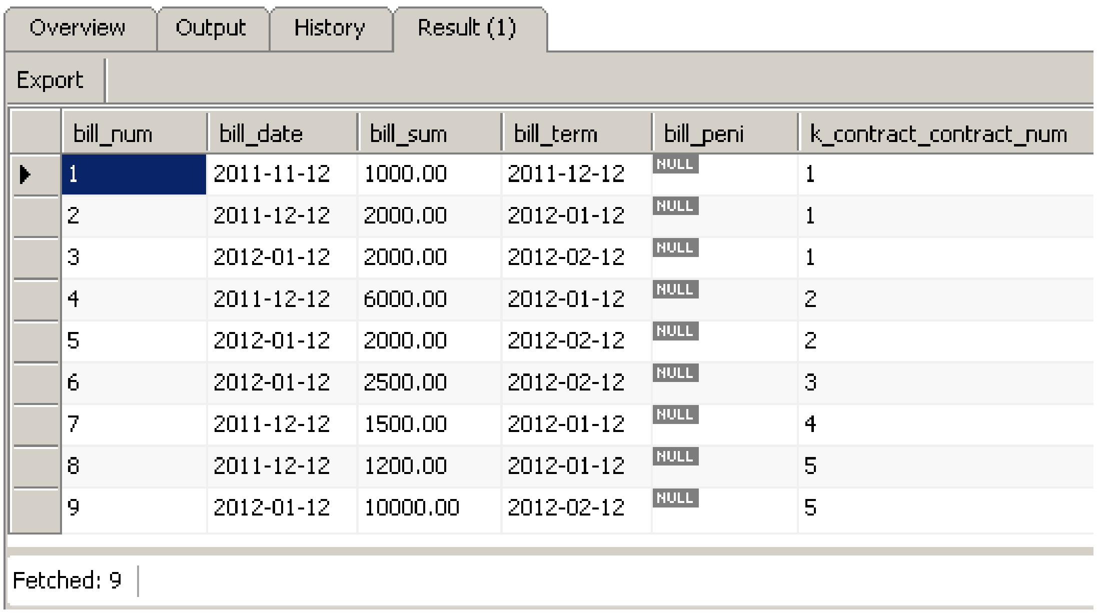
contract_num	contract_date	contract_type	k_staff_staff_num	k_firm_firm_num
1	2011-11-01	A	1	1
2	2011-10-01	B	2	1
3	2011-09-01	C	1	1
4	2011-11-15	A	2	2
5	2011-08-01	B	2	2
6	2011-07-15	C	1	3
7	2011-11-12	A	1	4

Fetched: 7

Заполним таблицу **Счет**

```
INSERT INTO k_bill
(k_contract_contract_num, bill_date, bill_term, bill_sum)
     VALUES(1, '2011-11-12', '2011-12-12', 1000);
INSERT INTO k_bill
(k_contract_contract_num, bill_date, bill_term, bill_sum)
     VALUES(1, '2011-12-12', '2012-01-12', 2000);
INSERT INTO k_bill
(k_contract_contract_num, bill_date, bill_term, bill_sum)
     VALUES(1, '2012-01-12', '2012-02-12',2000);
INSERT INTO k_bill
(k_contract_contract_num, bill_date, bill_term, bill_sum)
     VALUES(2, '2011-12-12', '2012-01-12', 6000);
INSERT INTO k_bill
(k_contract_contract_num, bill_date, bill_term, bill_sum)
     VALUES(2, '2012-01-12', '2012-02-12', 2000);
INSERT INTO k_bill
(k_contract_contract_num, bill_date, bill_term, bill_sum)
     VALUES(3, '2012-01-12', '2012-02-12', 2500);
INSERT INTO k_bill
(k_contract_contract_num, bill_date, bill_term, bill_sum)
     VALUES(4, '2011-12-12', '2012-01-12', 1500);
INSERT INTO k_bill
(k_contract_contract_num, bill_date, bill_term, bill_sum)
     VALUES(5, '2011-12-12', '2012-01-12', 1200);
INSERT INTO k_bill
(k_contract_contract_num, bill_date, bill_term, bill_sum)
     VALUES(5, '2012-01-12', '2012-02-12', 10000);
```

```
SELECT * FROM k_bill;
```

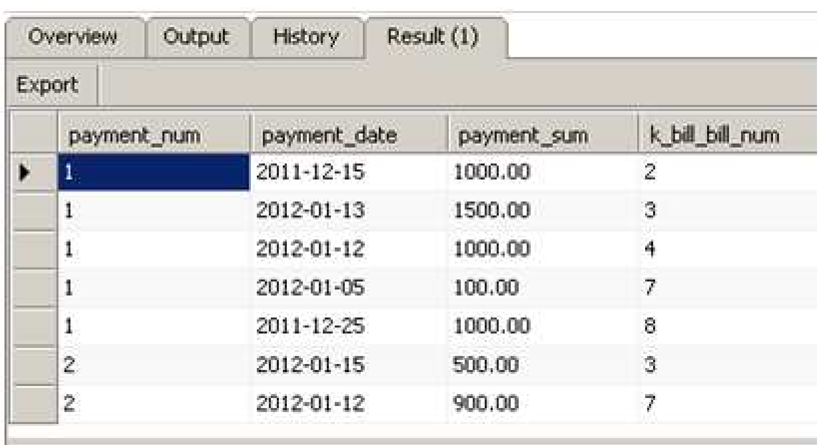


bill_num	bill_date	bill_sum	bill_term	bill_peni	k_contract_contract_num
1	2011-11-12	1000.00	2011-12-12	NULL	1
2	2011-12-12	2000.00	2012-01-12	NULL	1
3	2012-01-12	2000.00	2012-02-12	NULL	1
4	2011-12-12	6000.00	2012-01-12	NULL	2
5	2012-01-12	2000.00	2012-02-12	NULL	2
6	2012-01-12	2500.00	2012-02-12	NULL	3
7	2011-12-12	1500.00	2012-01-12	NULL	4
8	2011-12-12	1200.00	2012-01-12	NULL	5
9	2012-01-12	10000.00	2012-02-12	NULL	5

Fetched: 9

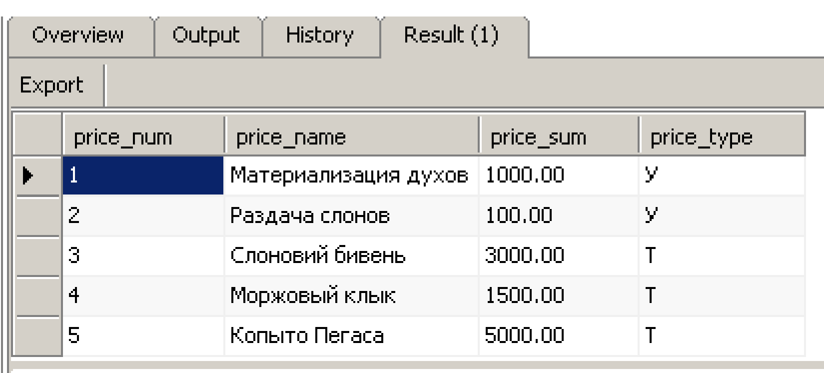
И остальные таблицы:

```
SELECT * FROM k_payment;
```



payment_num	payment_date	payment_sum	k_bill_bill_num
1	2011-12-15	1000.00	2
1	2012-01-13	1500.00	3
1	2012-01-12	1000.00	4
1	2012-01-05	100.00	7
1	2011-12-25	1000.00	8
2	2012-01-15	500.00	3
2	2012-01-12	900.00	7

```
SELECT * FROM k_price;
```



price_num	price_name	price_sum	price_type
1	Материализация духов	1000.00	У
2	Раздача слонов	100.00	У
3	Слоновий бивень	3000.00	Т
4	Моржовый клык	1500.00	Т
5	Копыто Пегаса	5000.00	Т

'У' означает услугу, 'Т' – товар.

```
SELECT * FROM k_protokol;
```

Overview | Output | History | Result (1)

Export

kolvo	price_sum	k_price_price_num	k_bill_bill_num
1	1000.00	1	1
2	1000.00	1	2
1	1000.00	1	5
2	1000.00	1	6
1	1000.00	1	8
20	100.00	2	3
10	100.00	2	5
5	100.00	2	6
2	100.00	2	8
2	3000.00	3	4
1	1500.00	4	7
2	5000.00	5	9

Кроме команды добавления данных INSERT, есть полезные команды изменения данных UPDATE и удаления данных DELETE.

Формат команды UPDATE:

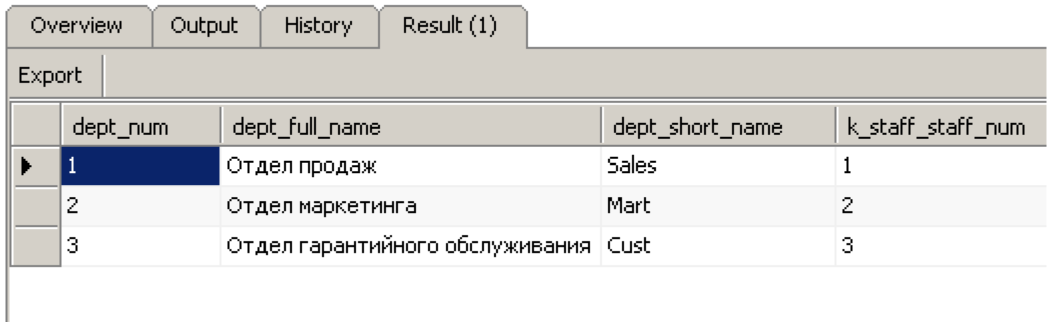
```
UPDATE [INTO] ИмяТаблицы SET ИмяСтолбца=НовоеЗначение
[WHERE Условие];
```

Квадратные скобки означают необязательную часть команды. Если условия нет, то изменяются ВСЕ строки заданной таблицы.

Применим эту команду на практике. Если вы обратили внимание, в таблице **Отдел** остался незаполненным столбец *k_staff_staff_num,* означающий номер сотрудника – руководителя отдела.

```
UPDATE k_dept SET k_staff_staff_num=2
     WHERE dept_short_name='Mart';

UPDATE k_dept SET k_staff_staff_num=3
     WHERE dept_short_name='Cust';

UPDATE k_dept SET k_staff_staff_num=1
     WHERE dept_short_name='Sales';
```

Результат:



dept_num	dept_full_name	dept_short_name	k_staff_staff_num
1	Отдел продаж	Sales	1
2	Отдел маркетинга	Mart	2
3	Отдел гарантийного обслуживания	Cust	3

Формат команды DELETE:

```
DELETE [FROM] имя_таблицы [WHERE условие];
```

Квадратные скобки означают необязательную часть команды. Если условия нет, то удаляются ВСЕ строки заданной таблицы.

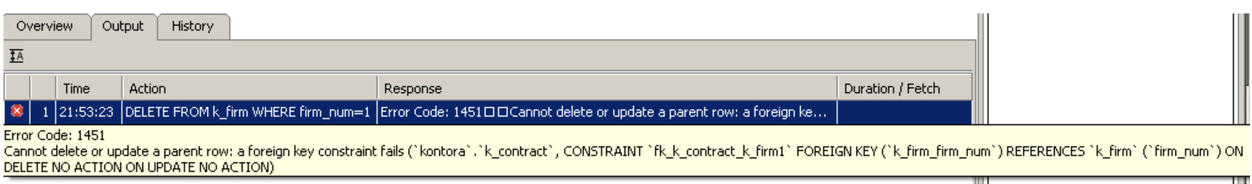
Пример: удаляем фирму с номером 5:

```
DELETE FROM k_firm WHERE firm_num=5;
```

Результат успешный. А что будет, если попробовать удалить фирму с номером 1? У этой фирмы есть подчиненные строки в таблице Договор.

Ошибка:

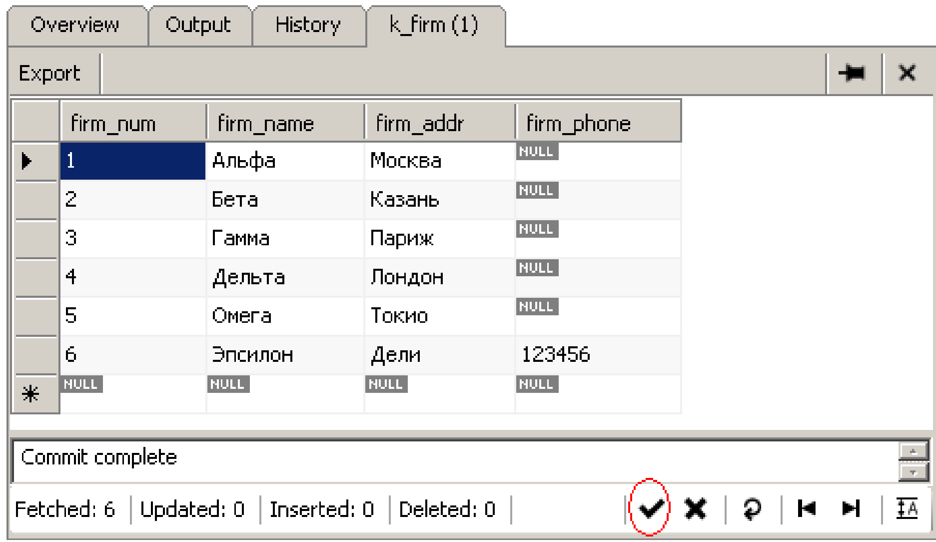
```
Error Code 1451

Cannot delete or update a parent row: a foreign key constraint
fails (`kontora`, `k_contract`, CONSTRAINT `fk_contract_k_firm`
FOREIGN KEY (`k_firm_firm_num`) REFERENCES `k_firm` (`firm_num`) …
```



2-ой способ заполнения базы данных – используем визуальные средства

Чтобы заполнять базу данных с помощью визуальных средств, в окне сервера нужно “дваждыщелкнуть” по нужной таблице (или выполнить команду EDIT ИмяТаблицы). Откроется окно

редактирования, в котором можно изменять и добавлять данные. Не забывайте сохранять изменения нажатием на кнопку «галочка»!



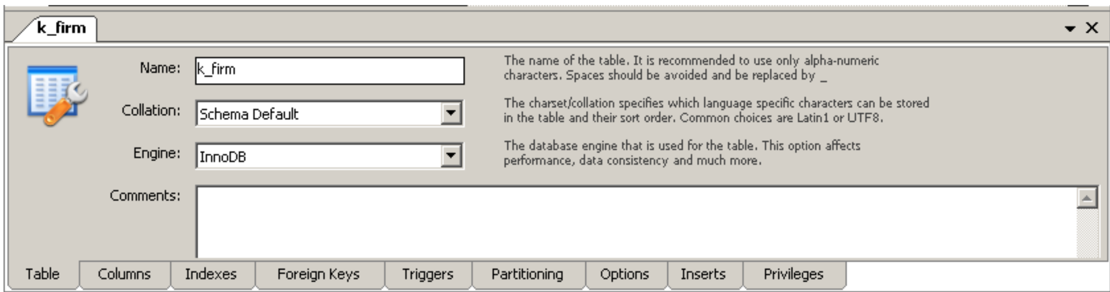
Этот способ добавления данных очень легкий – проблема возникает только при необходимости переноса данных на другой компьютер. Простых путей для копирования данных **нет**. Можно использовать выгрузку в текстовые файлы.

3-ий способ заполнения базы данных – данные хранятся в EER-модели

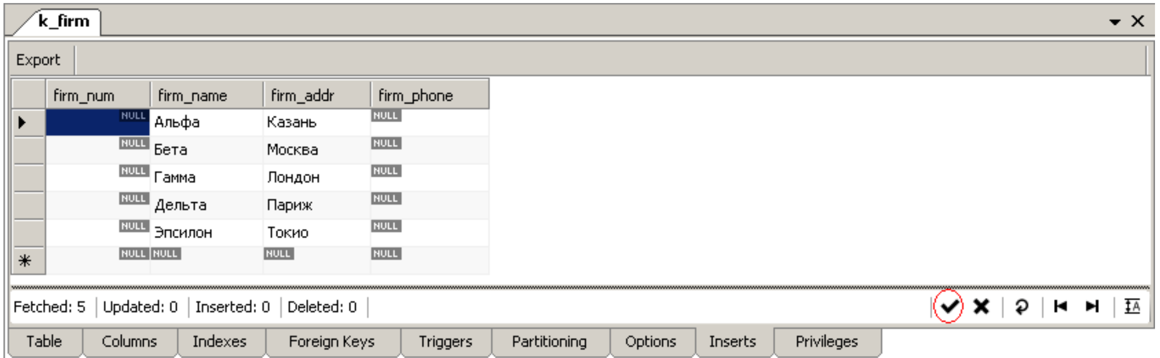
Без этого способа заполнения можно вполне обойтись, но для полноты картины расскажем о нем тоже.

Для применения этого способа придётся вернуться на шаг назад и открыть EER-диаграмму.

Как вы помните, если в диаграмме дважды щелкнуть по имени таблицы, то открывается окно ее свойств:



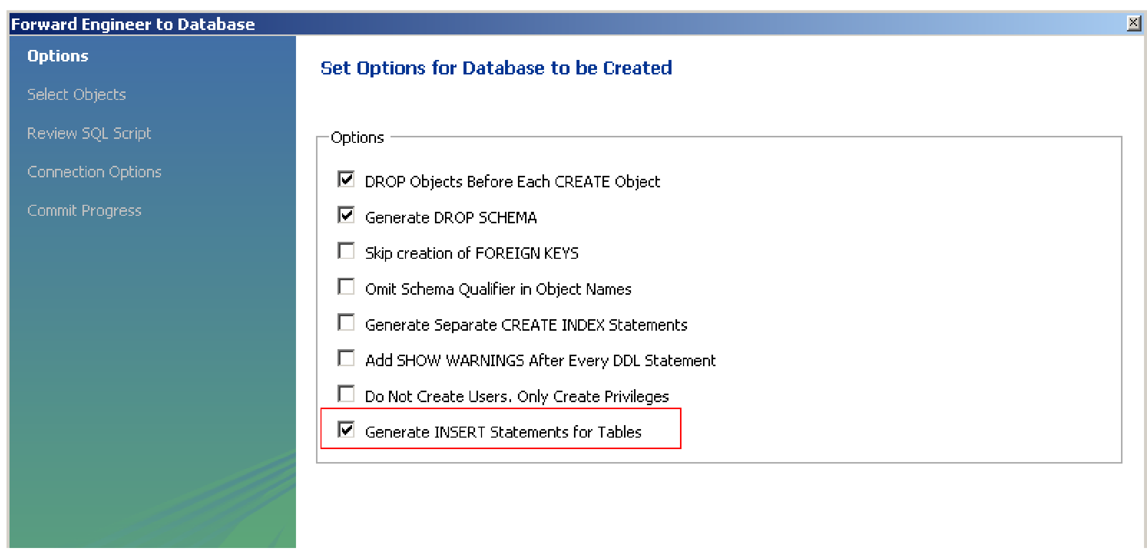
При разработке структуры таблицы мы использовали вкладку **Columns.** Теперь переключимся на вкладку **Inserts** и заполним данные в таблице**.**



НомерФирмы можем не заполнять, у него есть свойство Auto Increment. Телефон мы тоже не заполнили, он необязательный. После заполнения данных не забудьте нажать на кнопку «галочка», чтобы сохранились изменения в модели.

При использовании этого способа есть очень существенная проблема. При заполнении таблиц не проверяются никакие ограничения, ни на типы полей, ни на внешние ключи. Поэтому очень легко сделать ошибку.

Этот режим работы похож на предыдущий способ заполнения базы данных, но у него есть **очень важное отличие!** Заполненные таким образом данные хранятся **только в EER-модели**, на стороне сервера их **нет**. Для того чтобы данные появились на стороне сервера, нужно заново выполнить генерацию базы данных из EER-модели, как во втором задании. При этом обязательно нужно отметить флажок **Generate INSERT Statements for Tables:**



В этот момент проявят себя все ранее сделанные ошибки при вводе данных.

Разумеется, при этом старая база данных **удаляется**, вместе со **всеми** ранее введенными данными.

Задание. Заполните вашу базу данных. В каждой таблице создайте по несколько строк.

Дополнительная информация. Подробнее о SQL-командах модификации данных можно почитать в [**3**, Глава 5], [**1**, Глава 2, параграф 2.4] .

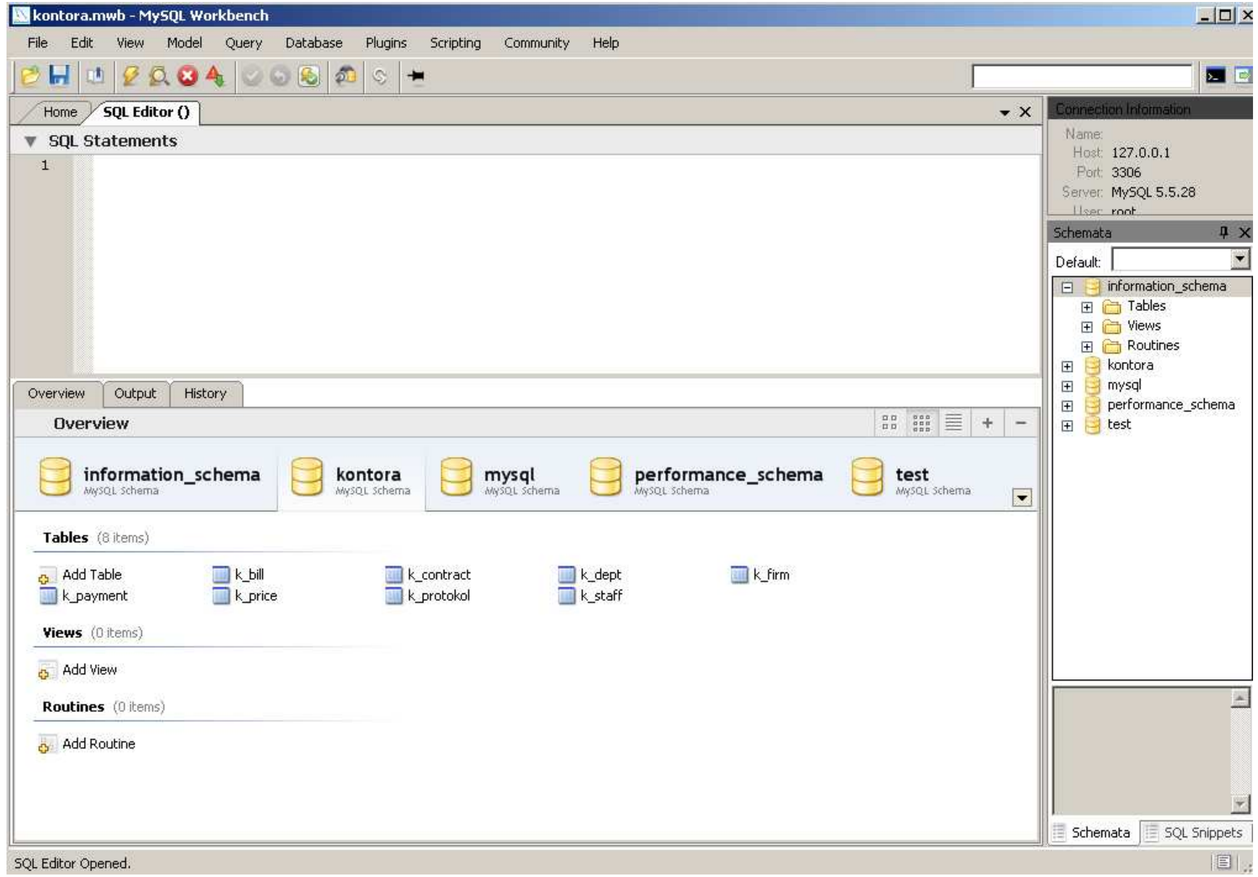
Запросы к базе данных

На предыдущем этапе мы заполнили таблицы базы данных. Рассмотрим теперь команду SELECT, позволяющую выполнять запросы к базе данных. По поводу этой команды написаны целые книги, здесь мы кратко на примерах рассмотрим ее основные возможности.

Как и в предыдущем задании, подключимся к серверу. Для этого в секции

щелкнем по ссылке Open Connection to Start Querying. В открывшемся окне нужно задать **username** и **password**, и нажать на кнопку «ОК».

Команды SELECT нужно набирать в окне **SQL Statements:**



Самой первой командой в сеансе данных должна быть команда выбора текущей базы данных:

```
USE kontora;
```

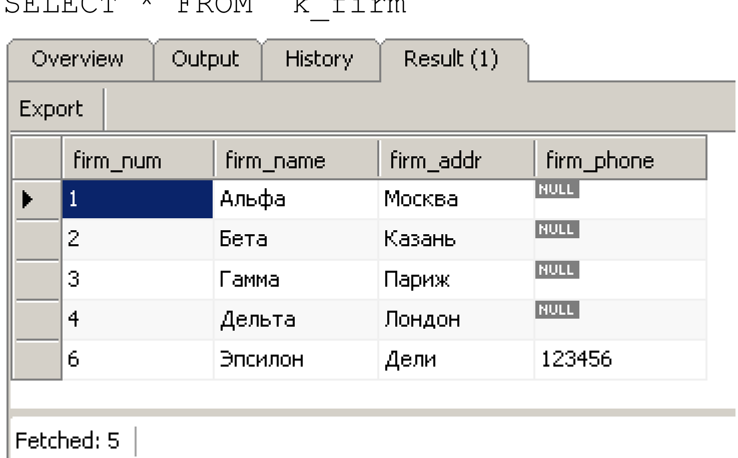
Выборка из одной таблицы

Обязательные ключевые слова команды – SELECT и FROM.

Тривиальная выборка всех полей и всех строк одной таблицы выполняется следующим образом.

Получить полную информацию обо всех предприятиях:

```
SELECT * FROM  k_firm
```

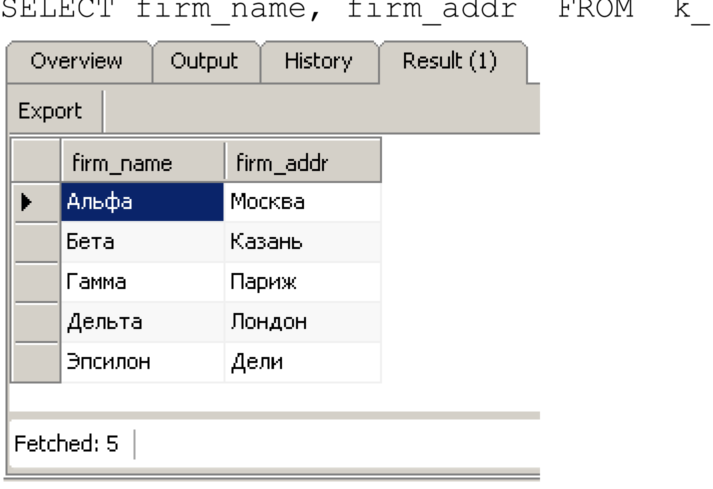


firm_num	firm_name	firm_addr	firm_phone
1	Альфа	Москва	NULL
2	Бета	Казань	NULL
3	Гамма	Париж	NULL
4	Дельта	Лондон	NULL
6	Эпсилон	Дели	123456

Выбор отдельных полей таблицы.

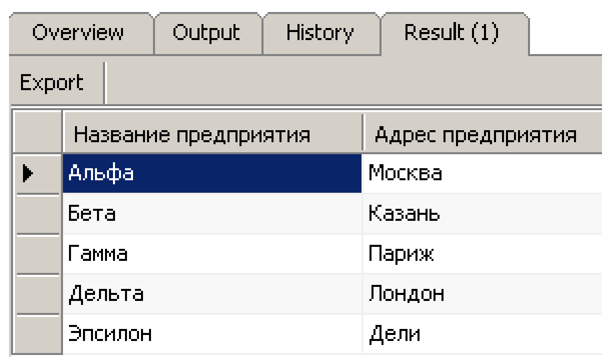
Получить названия и адреса всех предприятий:

```
SELECT firm_name, firm_addr  FROM  k_firm
```



firm_name	firm_addr
Альфа	Москва
Бета	Казань
Гамма	Париж
Дельта	Лондон
Эпсилон	Дели

Поля выборки можно переименовывать. Если новое название состоит из нескольких слов, помещайте его в кавычки.



Overview | Output | History | Result (1)

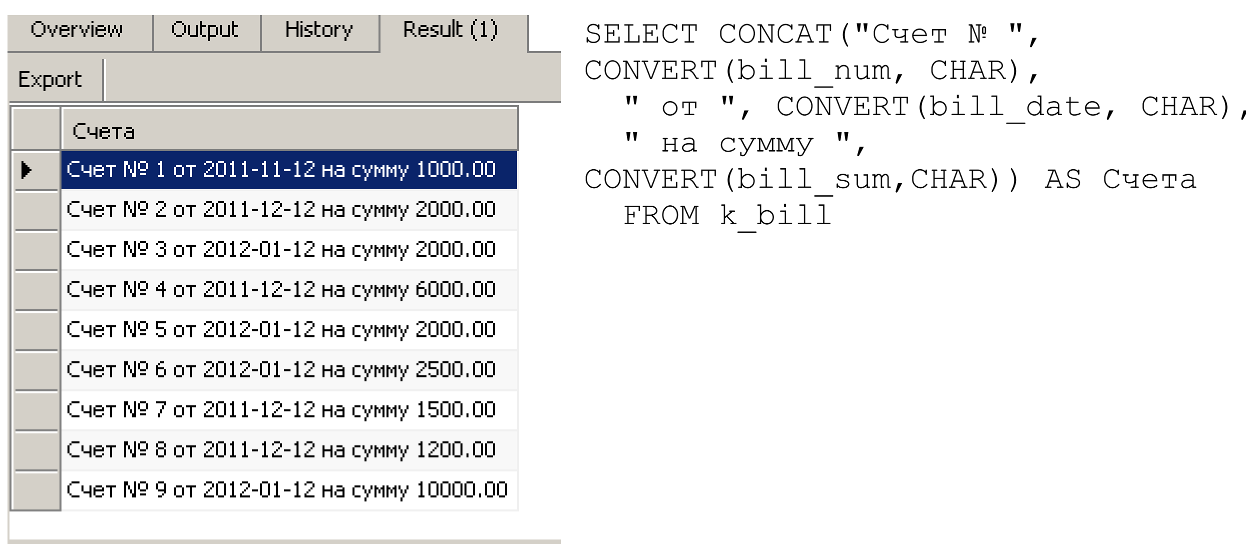
Export

Название предприятия	Адрес предприятия
Альфа	Москва
Бета	Казань
Гамма	Париж
Дельта	Лондон
Эпсилон	Дели

```
SELECT firm_name
  AS "Название предприятия",
    firm_addr
  AS "Адрес предприятия"
FROM  k_firm
```

В списке полей выборки можно использовать выражения. В этом случае часто приходится преобразовывать данные из одного типа в другой с помощью функции CONVERT. Строковые константы следует помещать в одинарные кавычки. Функция CONCAT служит для сцепления строк.

Распечатать информацию о счетах:



Overview | Output | History | Result (1)

Export

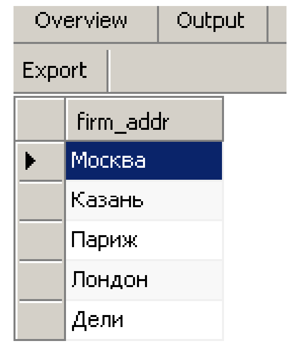
Счета
Счет № 1 от 2011-11-12 на сумму 1000.00
Счет № 2 от 2011-12-12 на сумму 2000.00
Счет № 3 от 2012-01-12 на сумму 2000.00
Счет № 4 от 2011-12-12 на сумму 6000.00
Счет № 5 от 2012-01-12 на сумму 2000.00
Счет № 6 от 2012-01-12 на сумму 2500.00
Счет № 7 от 2011-12-12 на сумму 1500.00
Счет № 8 от 2011-12-12 на сумму 1200.00
Счет № 9 от 2012-01-12 на сумму 10000.00

```
SELECT CONCAT("Счет № ",
CONVERT(bill_num, CHAR),
  " от ", CONVERT(bill_date, CHAR),
  " на сумму ",
CONVERT(bill_sum,CHAR)) AS Счета
  FROM k_bill
```

Для того чтобы исключить дубликаты строк, нужно использовать ключевое слово DISTINCT.

Напечатать список городов, в которых находятся предприятия-клиенты:

```
SELECT DISTINCT firm_addr FROM k_firm
```



Overview | Output

Export

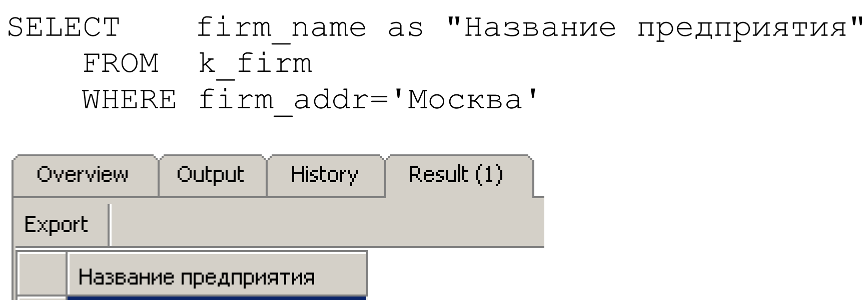
firm_addr
Москва
Казань
Париж
Лондон
Дели

Использование условий отбора

Для выбора отдельных строк по некоторому критерию используется ключевое слово WHERE

Получить список предприятий, расположенных в Москве:

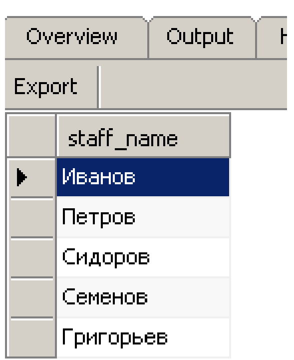
```
SELECT    firm_name as "Название предприятия"
    FROM  k_firm
    WHERE firm_addr='Москва'
```



Для сравнения поля со значением NULL нельзя использовать операции = и !=, вместо них нужно использовать выражения IS NULL и IS NOT NULL.

Получить список постоянно работающих сотрудников, т.е., таких, у которых staff_termdate равно NULL:

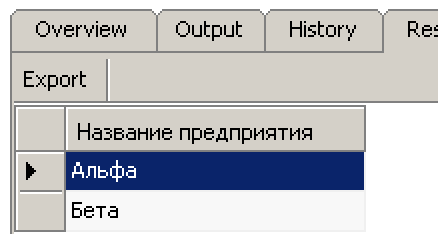
```
SELECT staff_name FROM k_staff
    WHERE staff_termdate IS NULL
```



Условия могут быть сложные, представляющие собой комбинацию нескольких операций сравнения. В них можно использовать логические связки AND и OR, а также отрицание NOT.

Получить список предприятий, расположенных в Москве или Казани:

```
SELECT  firm_name as "Название предприятия" FROM  k_firm
    WHERE firm_addr='Москва' OR firm_addr='Казань'
```



Название предприятия
Альфа
Бета

Если условие заключается в сравнении поля со **списком** значений, удобно использовать ключевое слово IN.

Получить список предприятий, расположенных в Москве или Казани:

```
SELECT    firm_name as "Название предприятия"
     FROM  k_firm
WHERE firm_addr IN ('Москва','Казань')
```

Если условие заключается в сравнении поля с **диапазоном** значений, удобно использовать ключевое слово BETWEEN.

Получить список договоров, заключенных в ноябре 2011 г.:

```
SELECT  * FROM  k_contract
WHERE contract_date BETWEEN '2011-11-01' AND '2011-11-30'
```

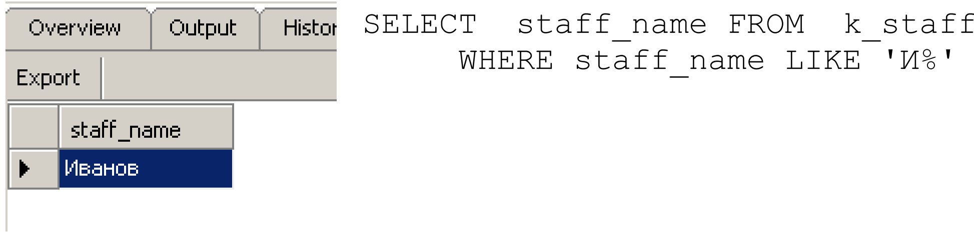
Overview | Output | History | Result (1)

Export

contract_num	contract_date	contract_type	k_staff_staff_num	k_firm_firm_num
1	2011-11-01	A	1	1
4	2011-11-15	A	2	2
7	2011-11-12	A	1	4

Для полей строкового типа можно применять сравнение с подстрокой.

Получить список сотрудников, фамилия которых начинается на И:



staff_name
Иванов

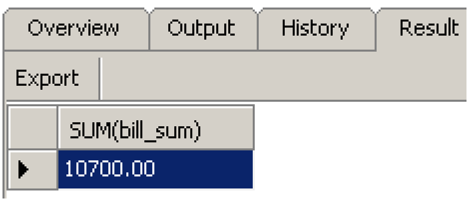
```
SELECT  staff_name FROM  k_staff
     WHERE staff_name LIKE 'И%'
```

Использование агрегирующих функций

Для подсчета **итоговых** значений служат функции SUM, COUNT, MAX, MIN, AVG. Если не используется группировка строк, запрос с применением итоговой функции вернет *ровно* одну строку.

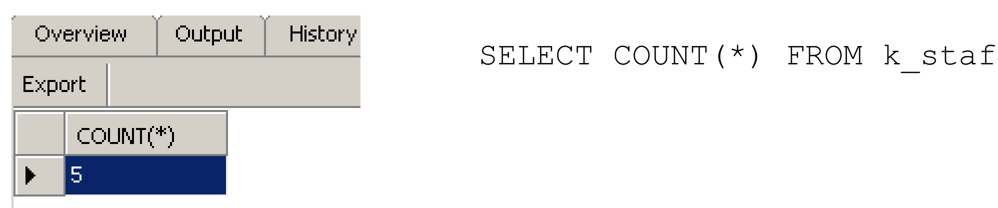
Подсчитать, на какую сумму выставлены счета в декабре 2011 года.

```
SELECT  SUM(bill_sum) FROM  k_bill
     WHERE bill_date
          BETWEEN '2011-12-01' AND '2011-12-31'
```



Функция COUNT позволяет подсчитать, сколько строк в таблице имеется вообще.

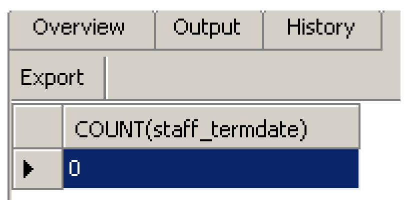
Подсчитать количество сотрудников.




```
SELECT COUNT(*) FROM k_staff
```

А также эта функция позволяет подсчитать, сколько строк с не-NULL-значениями в определенном поле.

Подсчитать количество временно работающих сотрудников (у них заполнен срок окончания трудового договора – поле staff_termdate). Предполагается, что даты все разные (точнее говоря, здесь подсчитывается количество различных не-null значений).



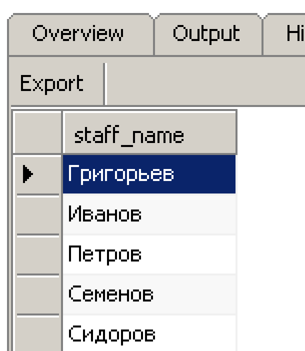

```
SELECT COUNT(staff_termdate)
FROM k_staff
```

Сортировка

Для сортировки используется ключевое слово ORDER BY и имя поля или его номер в списке полей выборки.

Напечатать список сотрудников, отсортированный по алфавиту:

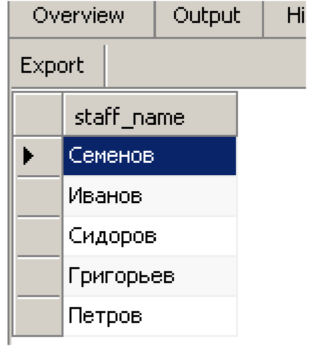
```
SELECT staff_name FROM k_staff ORDER BY 1
```



Можно сортировать строки даже по такому полю, которое не входит в список полей выборки.

Напечатать список сотрудников, отсортированный по дате поступления на работу:

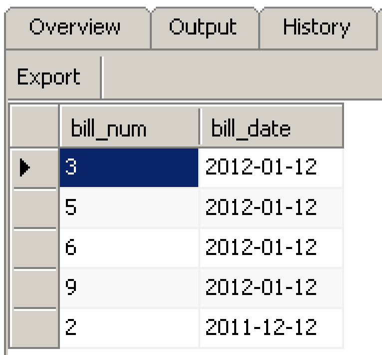
```
SELECT staff_name FROM k_staff ORDER BY staff_hiredate
```



Сортировать данные можно и по убыванию. Кроме того, можно ограничить количество строк в результате.

Напечатать информацию о 5 последних выписанных счетах в порядке убывания даты счета:

```
SELECT bill_num, bill_date
FROM k_bill ORDER BY bill_date DESC LIMIT 5
```



bill_num	bill_date
3	2012-01-12
5	2012-01-12
6	2012-01-12
9	2012-01-12
2	2011-12-12

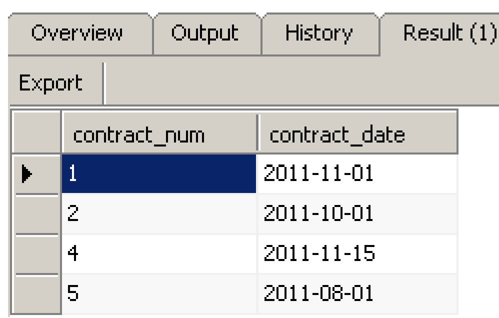
Подзапросы

Для более сложных формулировок иногда удобно использовать подзапросы. Подзапрос всегда указывается в скобках.

Подзапрос может быть **несвязанным**, т.е. в формулировке подзапроса нет ссылки на главный запрос. В этом случае подзапрос выполняется один раз при выполнении главного запроса. В данном примере используется ключевое слово IN, так как подзапрос может возвращать несколько значений.

Получить список договоров, по которым в декабре 2011 года выписаны счета:

```
SELECT contract_num, contract_date FROM k_contract
     WHERE contract_num IN
          (SELECT k_contract_contract_num FROM k_bill
          WHERE bill_date
          BETWEEN '2011-12-01' AND '2011-12-31')
```



contract_num	contract_date
1	2011-11-01
2	2011-10-01
4	2011-11-15
5	2011-08-01

Тот же самый запрос с использованием ключевого слова ANY:

```
SELECT contract_num, contract_date FROM k_contract
     WHERE contract_num =ANY
          (SELECT k_contract_contract_num FROM k_bill
          WHERE bill_date
          BETWEEN '2011-12-01' AND '2011-12-31')
```

Тот же самый запрос можно выполнить и с помощью **связанного** подзапроса, т.е., подзапроса, в котором есть ссылка на главный запрос. Для ссылки на таблицу главного запроса нужно указать псевдоним. Такой подзапрос будет выполняться **заново** для **каждой** строки главного запроса.

Кроме того, в данном примере иллюстрируется использование ключевого слова EXISTS:

```
SELECT contract_num, contract_date FROM k_contract c
     WHERE EXISTS
          (SELECT * FROM k_bill b
               WHERE bill_date
          BETWEEN '2011-12-01' AND '2011-12-31'
          AND c.contract_num=b.k_contract_contract_num)
```

Пример использования ключевого слова ALL.

Напечатать информацию о товаре (товарах) с наименьшей ценой.

```
SELECT price_name, price_sum FROM k_price
     WHERE price_sum <= ALL
          (SELECT price_sum FROM k_price)
```

Overview | Output | History | Result (1)

Export

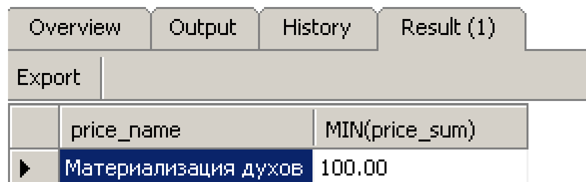
price_name	price_sum
Раздача слонов	100.00

Этот запрос можно сформулировать и по-другому. В этом примере мы можем использовать операцию сравнения =, т.к. подзапрос возвращает ровно одну строку и один столбец.

```
SELECT price_name, price_sum FROM k_price
     WHERE price_sum =
          (SELECT MIN(price_sum) FROM k_price)
```

А так, как в следующем примере, запрос формулировать нельзя. При запуске ошибок **не будет**, просто получится неверный результат:

```
SELECT price_name, MIN(price_sum) FROM k_price
```



Overview | Output | History | Result (1)

Export

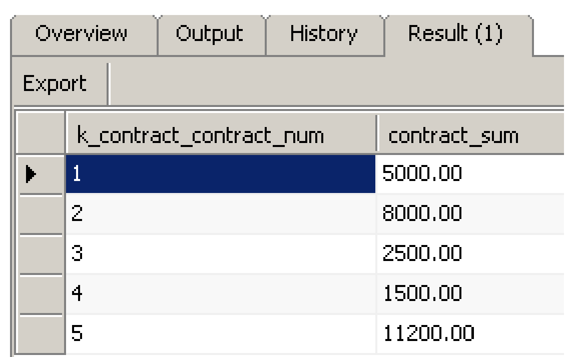
	price_name	MIN(price_sum)
▶	Материализация духов	100.00

Как видите, значение столбца price_name просто было взято из первой строки таблицы.

Группировка

Для подведения итога по группе данных используется комбинация ключевого слова GROUP BY и агрегирующих функций. Причем в списке полей для выборки обычно присутствуют *только* поля группировки и агрегирующие функции. При необходимости можно добавить дополнительные поля, которые функционально зависят от «ключа группировки».

Получить список договоров и общую сумму счетов по каждому договору:



Overview | Output | History | Result (1)

Export

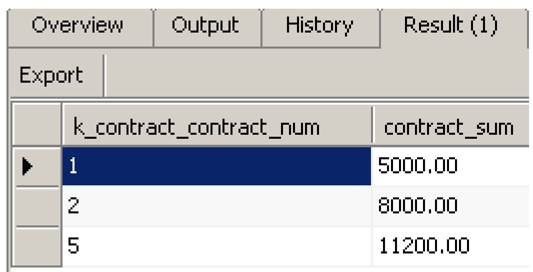
	k_contract_contract_num	contract_sum
▶	1	5000.00
	2	8000.00
	3	2500.00
	4	1500.00
	5	11200.00

```
SELECT contract_num,
SUM(bill_sum) AS contract_sum
    FROM k_bill
GROUP BY contract_num
```

В том случае, когда нужно выбрать не все группы, а только некоторые из них, используется ключевое слово HAVING:

Получить список договоров, имеющих 2 или более счетов, и общую сумму счетов по каждому договору:

```
SELECT k_contract_contract_num, SUM(bill_sum) AS contract_sum
FROM k_bill
     GROUP BY k_contract_contract_num
HAVING COUNT(bill_num)>=2;
```



k_contract_contract_num	contract_sum
1	5000.00
2	8000.00
5	11200.00

Выборка из нескольких таблиц

Для связи таблиц можно использовать то же ключевое слово WHERE, как и для условий отбора. При выборке из нескольких таблиц рекомендуется всегда использовать псевдонимы таблиц. Дело в том, что если в разных таблицах имеются одинаковые поля, то всегда нужно уточнять, к какой таблице они относятся, т.е., использовать синтаксис имя_таблицы.имя_поля. А так как имена таблиц обычно длинные, удобно заменять их псевдонимами.

Напечатать список договоров с указанием названия предприятия.

```
SELECT firm_name, contract_num, contract_date
     FROM k_firm f, k_contract c
     WHERE f.firm_num=c.k_firm_firm_num
```

firm_name	contract_num	contract_date
Альфа	1	2011-11-01
Альфа	2	2011-10-01
Альфа	3	2011-09-01
Бета	4	2011-11-15
Бета	5	2011-08-01
Гамма	6	2011-07-15
Дельта	7	2011-11-12

То же самое можно получить, если использовать синтаксис JOIN...ON. Это так называемое *внутреннее* (INNER) соединение. Строки соединяются, если совпадают значения полей в условии ON.

```
SELECT firm_name, contract_num, contract_date
FROM k_firm f JOIN k_contract c ON f.firm_num=c.k_firm_firm_num
```

Для соединения трех и более таблиц синтаксис в этом формате следующий:

Напечатать список сотрудников, номера и даты договоров, которые они заключили, с указанием названия предприятия.

```
SELECT staff_name, contract_num, contract_date, firm_name
FROM k_firm f JOIN k_contract c ON f.firm_num=c.k_firm_firm_num
              JOIN k_staff s ON  s.staff_num=c.k_staff_staff_num
```

Overview | Output | History | Result (1)

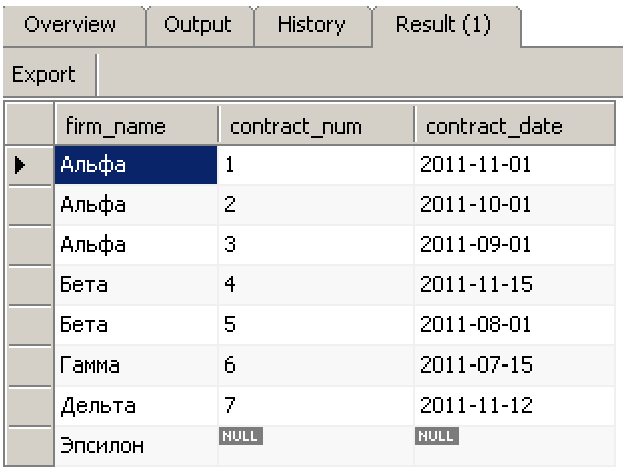
Export

staff_name	contract_num	contract_date	firm_name
Иванов	1	2011-11-01	Альфа
Иванов	3	2011-09-01	Альфа
Иванов	6	2011-07-15	Гамма
Иванов	7	2011-11-12	Дельта
Петров	2	2011-10-01	Альфа
Петров	4	2011-11-15	Бета
Петров	5	2011-08-01	Бета

Кроме внутреннего, бывают еще левое (LEFT), правое (RIGHT) и полное (FULL) соединения.

Рассмотрим, например, левое соединение. В результат попадут строки, в которых совпадают значения полей в условии ON, и те строки из левой таблицы, для которых не нашлось соответствующих строк в правой таблице. Поля из правой таблицы будут заполнены значениями NULL.

Напечатать список договоров с указанием названия предприятия плюс список предприятий, у которых нет договоров.



firm_name	contract_num	contract_date
Альфа	1	2011-11-01
Альфа	2	2011-10-01
Альфа	3	2011-09-01
Бета	4	2011-11-15
Бета	5	2011-08-01
Гамма	6	2011-07-15
Дельта	7	2011-11-12
Эпсилон	NULL	NULL

```
SELECT firm_name,
       contract_num,
       contract_date
FROM k_firm f
LEFT JOIN k_contract c
ON f.firm_num=c.k_firm_firm_num
```

А что будет в том случае, если условие связи вообще не указывать? Получится так называемое *декартово произведение* таблиц, в котором *каждая* строка первой таблицы будет сцеплена с *каждой* строкой второй таблицы. Результат получается обычно очень большим и не имеющим смысла.

```
SELECT firm_name, contract_num, contract_date
     FROM k_firm f, k_contract c

# Предыдущий запрос вернул 35 строк,
# т.е. 5 предприятий умножить на 7 договоров.
```

Разумеется, в одном и том же запросе можно связывать не только две, а три и более таблицы, использовать в этих запросах подзапросы, группировки и т.п. Например, запрос к 4 таблицам:

Напечатать информацию о платежах с указанием названия предприятия.

```
SELECT firm_name, payment_date, payment_sum
 FROM k_firm f, k_contract c, k_bill b, k_payment p
 WHERE f.firm_num=c.k_firm_firm_num  AND
       c.contract_num=b.k_contract_contract_num AND
       b.bill_num=p.k_bill_bill_num
```

Overview | Output | History | Result (1)

Export

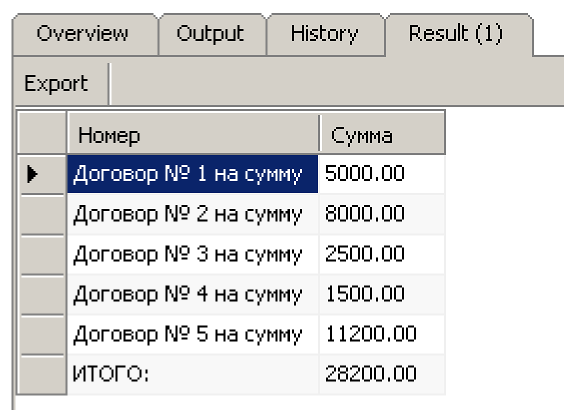
firm_name	payment_date	payment_sum
Альфа	2011-12-15	1000.00
Альфа	2012-01-13	1500.00
Альфа	2012-01-15	500.00
Альфа	2012-01-12	1000.00
Бета	2012-01-05	100.00
Бета	2012-01-12	900.00
Бета	2011-12-25	1000.00

Объединение запросов

Для объединения результатов двух и более запросов нужно использовать ключевое слово UNION. Объединяемые запросы должны иметь **одинаковое** количество и тип полей. Параметр ORDER BY , если он нужен, следует указывать только в **последнем** запросе.

Получить список договоров и общую сумму счетов по каждому договору, а также строку с итоговой суммой.

```
SELECT CONCAT('Договор № ',
               CONVERT(k_contract_contract_num, CHAR),
               ' на сумму ') AS "Номер",
       SUM(bill_sum) AS "Сумма" FROM k_bill
GROUP BY k_contract_contract_num
UNION
SELECT 'ИТОГО: ', SUM(bill_sum) FROM k_bill ORDER BY 1
```

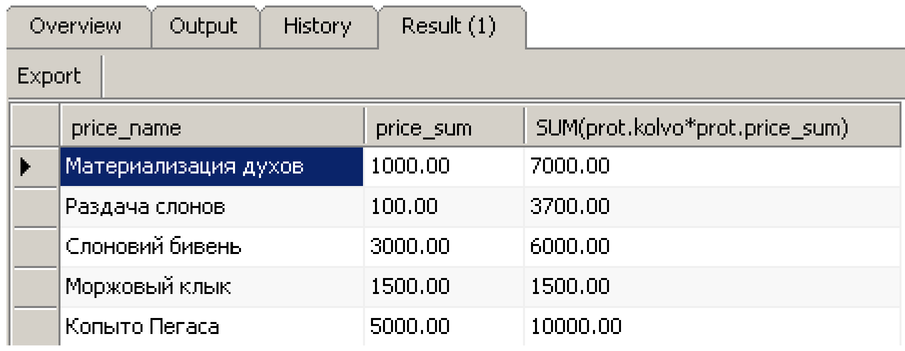


Номер	Сумма
Договор № 1 на сумму	5000.00
Договор № 2 на сумму	8000.00
Договор № 3 на сумму	2500.00
Договор № 4 на сумму	1500.00
Договор № 5 на сумму	11200.00
ИТОГО:	28200.00

И еще несколько примеров

Получить прайс-лист с суммой заказов по каждому товару. Обратите внимание, что название и цена товара могут использоваться в списке полей для выбора, поскольку они функционально (однозначно) зависят от номера товара, по которому проводится группировка.

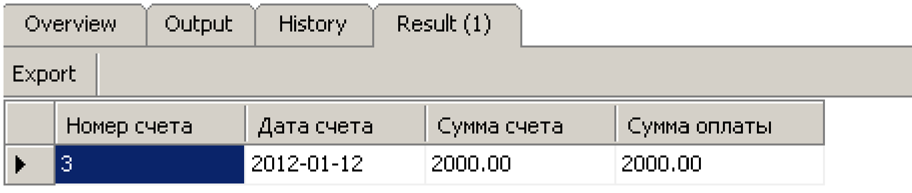
```
SELECT pr.price_name, pr.price_sum,
       SUM(prot.kolvo*prot.price_sum)
FROM k_price pr, k_protokol prot
WHERE pr.price_num=prot.k_price_price_num
GROUP BY pr.price_num ORDER BY 1
```



price_name	price_sum	SUM(prot.kolvo*prot.price_sum)
Материализация духов	1000.00	7000.00
Раздача слонов	100.00	3700.00
Слоновий бивень	3000.00	6000.00
Моржовый клык	1500.00	1500.00
Копыто Пегаса	5000.00	10000.00

Полностью оплаченные счета, т.е., счета, сумма платежей по которым больше или равна сумме счета. Обратите внимание на применение подзапроса.

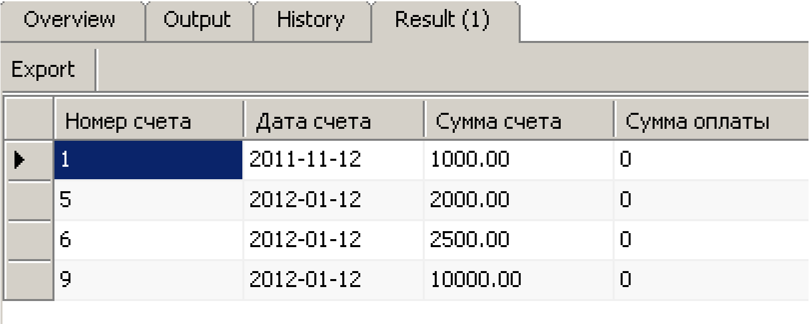
```
SELECT    b.bill_num AS "Номер счета",
     b.bill_date AS "Дата счета",
     b.bill_sum AS "Сумма счета",
     SUM(p.payment_sum) AS "Сумма оплаты"
FROM k_bill b, k_payment p
WHERE b.bill_num=p.k_bill_bill_num AND
     b.bill_sum<=
   (SELECT SUM(payment_sum) FROM k_payment p2
    WHERE b.bill_num=p2.k_bill_bill_num)
GROUP BY b.bill_num
```



Номер счета	Дата счета	Сумма счета	Сумма оплаты
3	2012-01-12	2000.00	2000.00

Полностью неоплаченные счета, по которым вообще нет платежей.

```
SELECT    b.bill_num AS "Номер счета",
     b.bill_date AS "Дата счета",
     b.bill_sum AS "Сумма счета",
     0 AS "Сумма оплаты"
FROM k_bill b
WHERE b.bill_num NOT IN (SELECT k_bill_bill_num FROM k_payment)
```



Номер счета	Дата счета	Сумма счета	Сумма оплаты
1	2011-11-12	1000.00	0
5	2012-01-12	2000.00	0
6	2012-01-12	2500.00	0
9	2012-01-12	10000.00	0

Частично оплаченные счета. Обратите внимание, что в этом примере в параметре FROM вместо второй таблицы используется вложенный SELECT

```
SELECT    b.bill_num AS "Номер счета",
     b.bill_date AS "Дата счета",
     b.bill_sum AS "Сумма счета",
```

```
    p.pay_sum AS "Сумма оплаты"
FROM k_bill b,
(SELECT k_bill_bill_num, SUM(payment_sum) as pay_sum
FROM k_payment
GROUP BY k_bill_bill_num) p
WHERE b.bill_sum >p.pay_sum AND b.bill_num=p.k_bill_bill_num
```

Overview | Output | History | Result (1)

Export

Номер счета	Дата счета	Сумма счета	Сумма оплаты
2	2011-12-12	2000.00	1000.00
4	2011-12-12	6000.00	1000.00
7	2011-12-12	1500.00	1000.00
8	2011-12-12	1200.00	1000.00

Задание. Напишите несколько (не менее 10) интересных запросов к вашей базе данных. Используйте вложенные подзапросы, группировки, итоговые значения, выборки из нескольких таблиц. Если ваш запрос требует ввода параметра, замените его пока на константу, запросы с параметрами можно будет в дальнейшем реализовать с помощью хранимых процедур.

Дополнительная информация. Подробнее о команде SELECT, используемых в ней операциях и функциях можно прочитать в **[1**, Глава 2, параграф 2.5, Глава 3**]**, **[3**, Главы 5,6,7**]**.

Представления

На предыдущем этапе мы выполняли запросы к базе данных. Существенным недостатком запросов является то, что их формулировки не сохраняются в базе данных.

Для того чтобы преодолеть этот недостаток, придумали **представления** (view). *Представления* – это объекты базы данных, которые можно рассматривать как виртуальные таблицы. На самом деле хранится только формулировка команды SELECT, с помощью которой производится выборка данных из реальных таблиц.

Необходимость в использовании представлений возникает, например, в том случае, когда нужно запретить доступ пользователя к отдельным столбцам или строкам таблицы – тогда можно просто написать представление, в котором эти столбцы или строки не будут присутствовать, и предоставить доступ пользователю именно к этому представлению, а не к реальной таблице.

Другой полезной возможностью является вычисление значений, которые не хранятся непосредственно в таблице, но всегда могут быть рассчитаны.

Представление, как и запрос, может содержать информацию из разных таблиц.

Представления могут быть **обновляемыми** (т.е., предоставлять возможность не только чтения, но и изменения данных в исходных таблицах) и **необновляемыми**. Представление будет обновляемым только в том случае, если его структура такова, что SQL server может точно определить, в какие строки каких таблиц нужно поместить измененные данные. Необновляемыми будут, например, представления, содержащие итоговые данные и группировки.

Для создания представлений используется команда CREATE VIEW.

Краткий формат этой команды:

```
CREATE VIEW имя_представления AS
     Команда_SELECT
```

Например, создадим представление, содержащее список договоров и их кураторов для отдела с номером 1. *Будет ли это представление обновляемым?*

```
CREATE VIEW k_contract1
AS
SELECT k_contract.contract_num, k_contract.contract_date,
    k_contract.contract_type, k_contract.k_firm_firm_num,
    k_staff.staff_name
FROM k_contract INNER JOIN
    k_staff ON k_contract.k_staff_staff_num = k_staff.staff_num
WHERE k_dept_dept_num = 1
```

Для просмотра представления следует выполнить команду

```
SELECT * FROM k_contract1
```

Overview | Output | History | Result (1)

Export

contract_num	contract_date	contract_type	k_firm_firm_num	staff_name
1	2011-11-01	A	1	Иванов
3	2011-09-01	C	1	Иванов
6	2011-07-15	C	3	Иванов
7	2011-11-12	A	4	Иванов

Проверим, является ли это представление обновляемым. Попробуем изменить, например, дату счета 1:

```
UPDATE k_contract1 SET contract_date='2011-11-02'
    WHERE contract_num=1
```

Команда выполнилась успешно, и повторный запрос на выборку дает следующий результат:

```
SELECT * FROM k_contract1
```

Overview | Output | History | Result (1)

Export

contract_num	contract_date	contract_type	k_firm_firm_num	staff_name
1	2011-11-02	A	1	Иванов
3	2011-09-01	C	1	Иванов
6	2011-07-15	C	3	Иванов
7	2011-11-12	A	4	Иванов

Создадим вспомогательное представление для запросов о полностью оплаченных и частично оплаченных счетах (см. предыдущее занятие). Это представление для каждого счета содержит его номер и сумму оплаты.

```
CREATE VIEW k_pay_sum
AS
SELECT k_bill_bill_num, SUM(payment_sum) AS pay_sum
FROM k_payment
GROUP BY k_bill_bill_num
```

Для просмотра представления следует выполнить команду

```
SELECT * FROM k_pay_sum.
```

Overview | Output | History | Result (1)

Export

k_bill_bill_num	pay_sum
2	1000.00
3	2000.00
4	1000.00
7	1000.00
8	1000.00

Это представление не будет обновляемым. Проверим:

```
UPDATE k_pay_sum SET pay_sum=1500 WHERE k_bill_bill_num=1
```

Получим ошибку:
ERROR 1288: The target table k_pay_sum of the UPDATE is not updatable.
Действительно, **невозможно** изменить значение поля, в котором находится **сумма** чисел из разных строк.

Но нам и не требовалось обновлять данные в этом представлении. Теперь с помощью этого представления можно переформулировать запрос на выборку полностью оплаченных счетов, этот запрос станет проще:

Полностью оплаченные счета

```
SELECT    b.bill_num AS "Номер счета",
     b.bill_date AS "Дата счета",
     b.bill_sum AS "Сумма счета",
     p.pay_sum AS "Сумма оплаты"
```

```
FROM k_bill b, k_pay_sum p
WHERE b.bill_num=p.k_bill_bill_num AND
     b.bill_sum<=p.pay_sum
```

Задание. Создайте несколько (не менее 3) представлений для вашей базы данных. Будут ли они обновляемыми или нет? Проверьте.

Дополнительная информация. Более подробно о представлениях в MySQL можно прочитать в [2, Глава 35].

Хранимые процедуры

Хранимые процедуры – это объекты базы данных, которые представляют собой программы, манипулирующие данными и **выполняемые на сервере**. Эти программы, кроме команд языка SQL, могут использовать немногочисленные управляющие команды.

Структура хранимой процедуры следующая:

```
DELIMITER //

CREATE PROCEDURE имя_процедуры [(параметры)]
    #Код процедуры

//
```

Объявление переменных имеет вид

```
DECLARE имя_переменной тип_переменной [(длина)];
```

Блок операторов заключается в команды BEGIN … END

Оператор присвоения выглядит так:

```
SET переменная=значение;
```

Если нужно присвоить переменной результат команды SELECT, то используется следующий формат (многоточие означает стандартное продолжение команды):

```
SELECT имя_столбца INTO переменная FROM ...;
```

Условный оператор имеет вид:

```
IF условие THEN
     Оператор1 или Группа операторов1
[ELSE
     Оператор2 или Группа операторов2]
END IF;
```

Есть несколько операторов цикла, самый распространенный из них:

```
WHILE условие DO
     Оператор или Группа операторов
END WHILE;
```

Выражение CASE применяется для выбора на основании нескольких опций:

```
CASE выражение
     WHEN вариант1 THEN выражение1
     WHEN вариант2 THEN выражение2
     …
     ELSE выражениеN
END CASE;
```

Для удаления процедур используется команда:

```
DROP PROCEDURE IF EXISTS Имя_процедуры;
```

Создадим процедуру, которая в качестве параметра получает фамилию сотрудника и печатает список всех договоров, которые он курирует.

```
DELIMITER //

CREATE PROCEDURE show_contracts
    (v_staff_name CHAR(50))
BEGIN

SELECT contract_num, contract_date, contract_type
   FROM k_contract c JOIN k_staff s ON
         c.k_staff_staff_num=s.staff_num
   WHERE s.staff_name=v_staff_name;

END//
```

Для запуска этой процедуры нужно выполнить, например, команду

```
CALL show_contracts('Иванов');
```

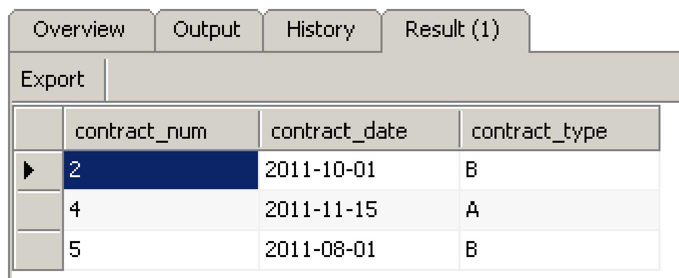
Overview | Output | History | Result (1)

Export

contract_num	contract_date	contract_type
1	2011-11-02	A
3	2011-09-01	C
6	2011-07-15	C
7	2011-11-12	A

или

```
CALL show_contracts('Петров');
```



Overview | Output | History | Result (1)

Export

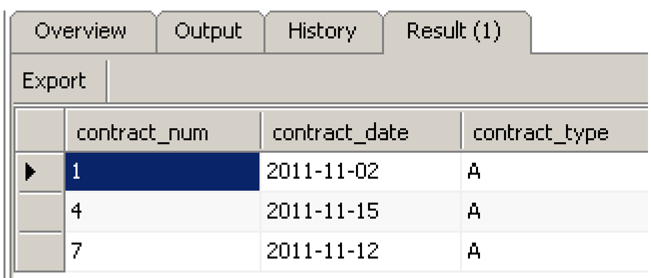
contract_num	contract_date	contract_type
2	2011-10-01	B
4	2011-11-15	A
5	2011-08-01	B

Создадим процедуру, которая получает номер месяца и номер года и печатает договоры за указанный период.

```
DELIMITER //
CREATE PROCEDURE find_contracts_by_month_and_year
    (v_month INT, v_year INT)
BEGIN

SELECT contract_num, contract_date, contract_type
    FROM k_contract
    WHERE MONTH(contract_date)=v_month AND
          YEAR(contract_date)=v_year;
END//
```

Выполним процедуру:

```
CALL find_contracts_by_month_and_year(11,2011);
```



Overview | Output | History | Result (1)

Export

contract_num	contract_date	contract_type
1	2011-11-02	A
4	2011-11-15	A
7	2011-11-12	A

Создадим процедуру «Распродажа», которая находит самый непродаваемый (по количеству) товар и уценивает его на заданный процент.

```
DELIMITER //
CREATE PROCEDURE clearance (percent INT)
BEGIN
DECLARE p INT;
  IF percent > 0 AND percent < 100 THEN
          SELECT k_price_price_num INTO p FROM k_protokol
             GROUP BY k_price_price_num
             HAVING SUM(kolvo)<=ALL
```

```
                    (SELECT SUM(kolvo) FROM k_protokol
                         GROUP BY k_price_price_num);
          UPDATE k_price
           SET price_sum=price_sum*(100-percent)/100
             WHERE price_num=p;
     END IF;
END//
```

Содержимое таблицы "Прайс-лист" до выполнения процедуры:

Overview | Output | History | Result (1)

Export

price_num	price_name	price_sum	price_type
1	Материализация духов	1000.00	У
2	Раздача слонов	100.00	У
3	Слоновий бивень	3000.00	Т
4	Моржовый клык	1500.00	Т
5	Копыто Пегаса	5000.00	Т

Для запуска этой процедуры нужно выполнить, например, команду

```
CALL clearance(10);
```

Содержимое таблицы "Прайс-лист" после выполнения процедуры:

Overview | Output | History | Result (1)

Export

price_num	price_name	price_sum	price_type
1	Материализация духов	1000.00	У
2	Раздача слонов	100.00	У
3	Слоновий бивень	3000.00	Т
4	Моржовый клык	1350.00	Т
5	Копыто Пегаса	5000.00	Т

Как видим, товар с номером 4 в прайс-листе уценен на 10%.

А что произойдет, если в нашей базе данных есть **несколько** товаров, количество продаж которых минимально? К сожалению, в нашем случае при выполнении команды SELECT процедура выдаст ошибку: Error Code: 1172, Result consisted of more than one row. Когда

в команде SELECT выбирается сразу **несколько** значений поля *k_price* из таблицы *k_protokol,* невозможно присвоить эти несколько значений **одной** переменной *р.* Данную ситуацию можно обработать с помощью так называемых курсоров.

Курсор (**cur**rent **s**et of **r**ecord) – это временный набор строк, которые можно перебирать последовательно, с первой до последней.

Для работы с курсорами существуют следующие команды.

Объявление курсора:

```
DECLARE имя_курсора CURSOR FOR SELECT текст_запроса;
```

Таким образом, любой курсор создается на основе некоторого оператора SELECT.

Открытие курсора:

```
OPEN имя_курсора;
```

Только после открытия курсора он становится активным, и из него можно читать строки.

Чтение значений из текущей строки курсора в набор переменных и перемещение указателя на следующую строку:

```
FETCH имя_курсора INTO список_переменных;
```

Переменные в списке должны иметь **то же количество и тип**, что и столбцы курсора.

Закрытие курсора:

```
CLOSE имя_курсора;
```

При переборе строк курсора возникает необходимость проверки, добрались ли мы до конца курсора или еще нет. В разных СУБД для этого могут быть предусмотрены разные средства. В СУБД MySQL

назначается обработчик состояния "NOT FOUND". Определять его нужно сразу же после описания структуры курсора:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND оператор;
```

Например, этот обработчик может выглядеть так:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;
```

Теперь попробуем модифицировать процедуру «Распродажа» с учетом того, что у нас может быть **несколько** товаров с минимальным количеством продаж.

```
DELIMITER //

CREATE PROCEDURE clearance2 (percent INT)
BEGIN
DECLARE p INT;
DECLARE finished NUMERIC(1);
-- объявляем курсор на основе некоторого оператора SELECT
DECLARE my_cursor CURSOR
   FOR SELECT k_price_price_num FROM k_protokol
         GROUP BY k_price_price_num
         HAVING SUM(kolvo)<=ALL
          (SELECT SUM(kolvo) FROM k_protokol
               GROUP BY k_price_price_num);
-- объявляем обработчик состояния NOT FOUND
DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;

IF percent > 0 AND percent < 100 THEN
    SET finished = 0;
    OPEN my_cursor;           -- открываем курсор
    FETCH my_cursor INTO p;   -- читаем первую строку
    WHILE( finished != 1) DO
        UPDATE k_price
           SET price_sum=price_sum*(100-percent)/100
           WHERE price_num=p;
        FETCH my_cursor INTO p; -- читаем очередную строку
    END WHILE;
    CLOSE my_cursor;  -- закрываем курсор
END IF;
END//
```

Задание. Создайте несколько (не менее 2) хранимых процедур для вашей базы данных. Желательно использовать параметры. Запустите процедуры на выполнение.

Дополнительная информация. Более подробно о хранимых процедурах в MySQL можно прочитать в [2, Глава 33].

Триггеры

Триггеры – это хранимые процедуры специального вида, которые **автоматически** выполняются при изменении данных с помощью операторов INSERT, UPDATE и DELETE. Триггер создается для определенной таблицы, но может использовать данные других таблиц и объекты других баз данных.

Оператор CREATE TRIGGER позволяет создать новый триггер и имеет следующий синтаксис:

```
CREATE TRIGGER имя_триггера время_триггера событие_триггера
   ON имя_таблицы FOR EACH ROW
    тело_триггера
```

Конструкция *время_триггера* указывает момент выполнения триггера и может принимать два значения:

BEFORE - действия триггера производятся до выполнения операции изменения таблицы;

AFTER - действия триггера производятся после выполнения операции изменения таблицы.

Конструкция *событие_триггера* может принимать значения INSERT, UPDATE и DELETE.

Идентификаторы OLD и NEW означают старое и новое значение изменяемых данных.

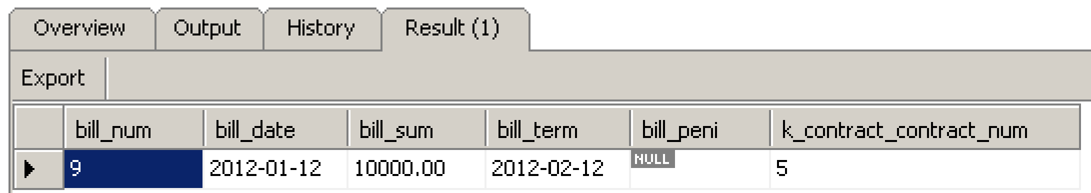
Рассмотрим пример триггера вставки, который вызывается при выполнении команды INSERT в таблице протоколов счетов. При добавлении новой позиции в счете нам нужно заново пересчитать его общую сумму:

```
DELIMITER //
-- триггер запускается перед добавлением строки в протокол
--счетов
CREATE TRIGGER ins_prot BEFORE INSERT ON k_protokol
FOR EACH ROW
BEGIN
    DECLARE v_kolvo NUMERIC(6);          --количество
    DECLARE v_bill_num NUMERIC(6);       --номер счета
    DECLARE v_price_num NUMERIC(6);      --номер товара
```

```
    DECLARE v_price_sum NUMERIC(9,2);   --цена товара
    SET v_kolvo=New.kolvo;
    SET v_bill_num=New.k_bill_bill_num;
    SET v_price_num=New.k_price_price_num;

    IF v_kolvo>0 THEN  -- только если количество >0
       --из прайс-листа получаем цену товара
       SELECT p.price_sum INTO v_price_sum FROM k_price p
            WHERE p.price_num=v_price_num;
       -- обновляем общую сумму счета
       UPDATE k_bill
            SET bill_sum=bill_sum+v_kolvo*v_price_sum
                WHERE k_bill.bill_num=v_bill_num;
       -- цену товара продублируем в протоколе счета
       SET New.price_sum=v_price_sum;
    END IF;
END//
```

Протестируем триггер. Предварительно посмотрим информацию о счете №9:

```
SELECT * FROM k_bill WHERE bill_num=9;
```



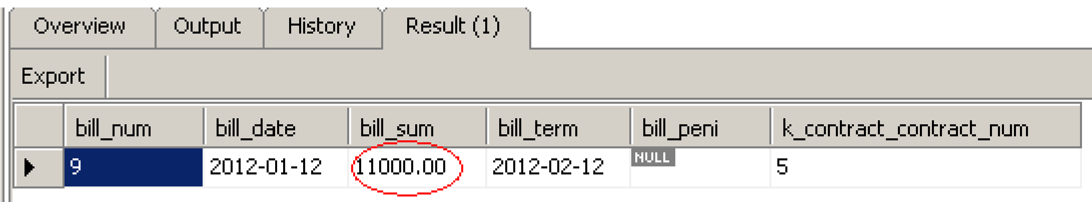
bill_num	bill_date	bill_sum	bill_term	bill_peni	k_contract_contract_num
9	2012-01-12	10000.00	2012-02-12	NULL	5

Теперь добавим новую строку в протокол этого счета: добавляем 1 штуку товара с номером 1 и ценой 1000 р. (цену берем из таблицы k_price).

```
INSERT INTO k_protokol
  (kolvo, price_sum, k_price_price_num, k_bill_bill_num)
   VALUES(1, 0, 1, 9);
```

Посмотрим, как изменился счет №9:

```
SELECT * FROM k_bill WHERE bill_num=9;
```

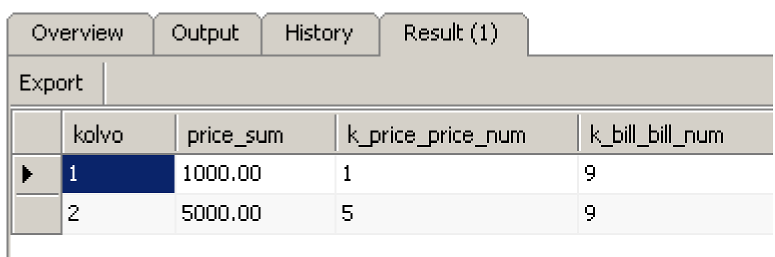


bill_num	bill_date	bill_sum	bill_term	bill_peni	k_contract_contract_num
9	2012-01-12	11000.00	2012-02-12	NULL	5

Общая сумма счета увеличилась на 1000 р.

Посмотрим таблицу протокола счетов:

```
SELECT * FROM k_protokol WHERE k_bill_bill_num=9;
```



kolvo	price_sum	k_price_price_num	k_bill_bill_num
1	1000.00	1	9
2	5000.00	5	9

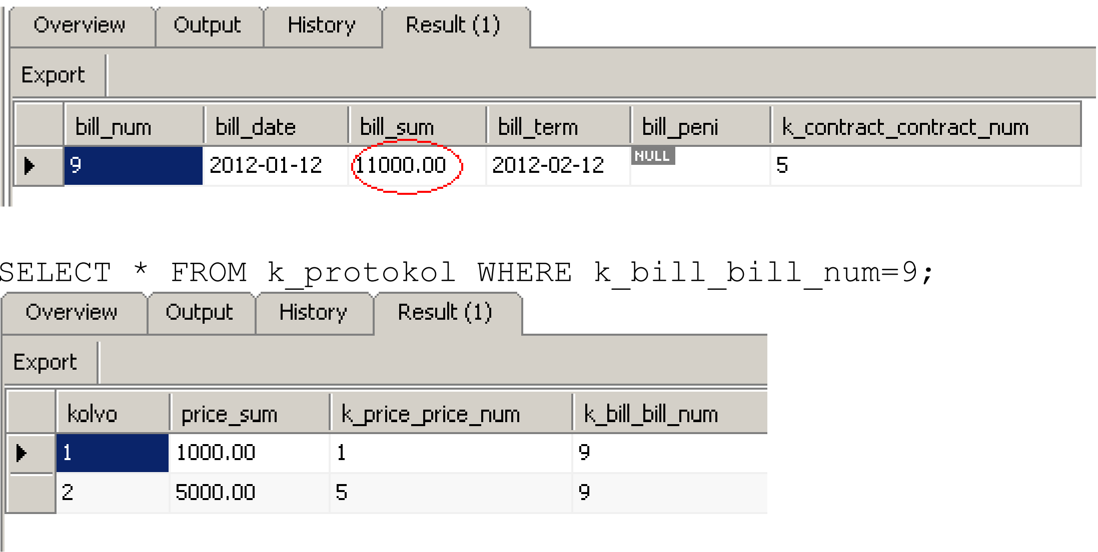
В добавленной строке с номером товара 1 цена заполнилась автоматически, из прайс-листа.

Теперь создадим триггер для операции удаления из той же таблицы. При удалении строки из протокола счета должна уменьшаться общая сумма счета.

```
-- триггер запускается перед удалением строки из протокола
--счетов
DELIMITER //
CREATE TRIGGER del_prot BEFORE DELETE ON k_protokol
FOR EACH ROW
BEGIN
    DECLARE v_kolvo NUMERIC(6);          -- количество
    DECLARE v_bill_num NUMERIC(6);       -- номер счета
    DECLARE v_price_sum NUMERIC(9,2);    -- цена товара
    SET v_kolvo=Old.kolvo;
    SET v_bill_num=Old.k_bill_bill_num;
    SET v_price_sum=Old.price_sum;

    IF v_kolvo>0 THEN     -- только если количество >0
       -- обновляем общую сумму счета
       UPDATE k_bill
            SET bill_sum=bill_sum-v_kolvo*v_price_sum
                WHERE k_bill.bill_num=v_bill_num;
    END IF;
END//
```

Протестируем триггер. Сначала посмотрим содержание таблиц до выполнения операции удаления:

```
SELECT * FROM k_bill WHERE bill_num=9;
```



bill_num	bill_date	bill_sum	bill_term	bill_peni	k_contract_contract_num
9	2012-01-12	11000.00	2012-02-12	NULL	5

```
SELECT * FROM k_protokol WHERE k_bill_bill_num=9;
```

kolvo	price_sum	k_price_price_num	k_bill_bill_num
1	1000.00	1	9
2	5000.00	5	9

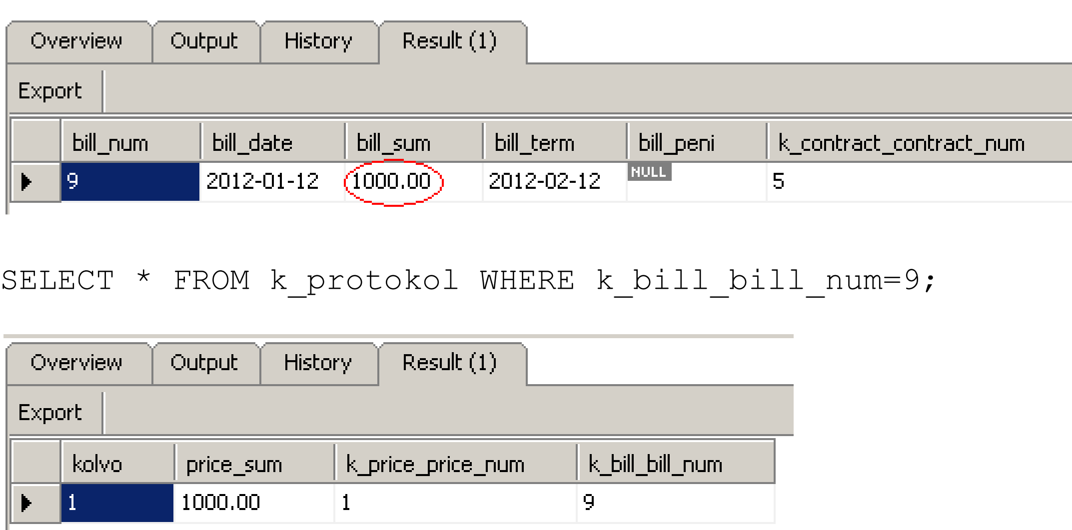
Теперь удалим из протокола счетов информацию о товаре с номером 5:

```
DELETE FROM k_protokol
      WHERE k_bill_bill_num=9 AND k_price_price_num=5;
```

Снова посмотрим содержимое таблицы k_bill:

```
SELECT * FROM k_bill WHERE bill_num=9;
```

Общая сумма счета уменьшилась до 1000 р.



bill_num	bill_date	bill_sum	bill_term	bill_peni	k_contract_contract_num
9	2012-01-12	1000.00	2012-02-12	NULL	5

```
SELECT * FROM k_protokol WHERE k_bill_bill_num=9;
```

kolvo	price_sum	k_price_price_num	k_bill_bill_num
1	1000.00	1	9

Задание. Создайте и протестируйте по крайней мере 1 триггер для вашей базы данных.

Дополнительная информация. Более подробно о триггерах MySQL можно прочитать в **[2**, Глава 34**]**.

Приложение 1. Сценарий создания базы данных

В данном приложении приведен сценарий, автоматически сформированный при создании базы данных из EER-модели.

```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';

CREATE SCHEMA IF NOT EXISTS `kontora` DEFAULT CHARACTER SET cp1251
COLLATE cp1251_general_ci ;

-- -----------------------------------------------------
-- Table `kontora`.`k_staff`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `kontora`.`k_staff` (
  `staff_num` INT NULL AUTO_INCREMENT ,
  `staff_name` VARCHAR(45) NOT NULL ,
  `staff_post` VARCHAR(45) NULL ,
  `staff_hiredate` DATE NULL ,
  `staff_termdate` DATE NULL ,
  `K_dept_dept_num` INT NOT NULL ,
  PRIMARY KEY (`staff_num`) ,
  INDEX `fk_k_staff_K_dept1` (`K_dept_dept_num` ASC) ,
  CONSTRAINT `fk_k_staff_K_dept1`
    FOREIGN KEY (`K_dept_dept_num` )
    REFERENCES `kontora`.`K_dept` (`dept_num` )

    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `kontora`.`K_dept`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `kontora`.`K_dept` (
  `dept_num` INT NULL AUTO_INCREMENT ,
  `dept_full_name` VARCHAR(45) NULL ,
  `dept_short_name` VARCHAR(10) NOT NULL ,
  `k_staff_staff_num` INT NOT NULL ,
  PRIMARY KEY (`dept_num`) ,
```

```
  INDEX `fk_K_dept_k_staff1` (`k_staff_staff_num` ASC) ,
  CONSTRAINT `fk_K_dept_k_staff1`
    FOREIGN KEY (`k_staff_staff_num` )
    REFERENCES `kontora`.`k_staff` (`staff_num` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `kontora`.`k_firm`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `kontora`.`k_firm` (
  `firm_num` INT NULL AUTO_INCREMENT ,
  `firm_name` VARCHAR(45) NOT NULL ,
  `firm_addr` VARCHAR(45) NULL ,
  `firm_phone` VARCHAR(20) NULL ,
  PRIMARY KEY (`firm_num`) )
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `kontora`.`k_contract`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `kontora`.`k_contract` (
  `contract_num` INT NULL AUTO_INCREMENT ,
  `contract_date` DATE NOT NULL ,
  `contract_type` ENUM('A', 'B','C') NOT NULL ,
  `k_firm_firm_num` INT NOT NULL ,
  `k_staff_staff_num` INT NOT NULL ,
  PRIMARY KEY (`contract_num`) ,
  INDEX `fk_k_contract_k_firm1` (`k_firm_firm_num` ASC) ,
  INDEX `fk_k_contract_k_staff1` (`k_staff_staff_num` ASC) ,
  CONSTRAINT `fk_k_contract_k_firm1`
    FOREIGN KEY (`k_firm_firm_num` )
    REFERENCES `kontora`.`k_firm` (`firm_num` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_k_contract_k_staff1`
    FOREIGN KEY (`k_staff_staff_num` )
    REFERENCES `kontora`.`k_staff` (`staff_num` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
```

```
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `kontora`.`k_bill`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `kontora`.`k_bill` (
  `bill_num` INT NULL AUTO_INCREMENT ,
  `bill_date` DATE NULL ,
  `bill_sum` DECIMAL(9,2) NULL ,
  `bill_term` DATE NULL ,
  `bill_peni` DECIMAL(6,2) NULL ,
  `k_contract_contract_num` INT NOT NULL ,
  PRIMARY KEY (`bill_num`) ,
  INDEX `fk_k_bill_k_contract1` (`k_contract_contract_num` ASC) ,
  CONSTRAINT `fk_k_bill_k_contract1`
    FOREIGN KEY (`k_contract_contract_num` )
    REFERENCES `kontora`.`k_contract` (`contract_num` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB
AUTO_INCREMENT = 1;

-- -----------------------------------------------------
-- Table `kontora`.`k_payment`
-- -----------------------------------------------------

CREATE  TABLE IF NOT EXISTS `kontora`.`k_payment` (
  `payment_num` INT NULL ,
  `payment_date` DATE NULL ,
  `payment_sum` DECIMAL(9,2) NULL ,
  `k_bill_bill_num` INT NOT NULL ,
  PRIMARY KEY (`payment_num`) ,
  INDEX `fk_k_payment_k_bill1` (`k_bill_bill_num` ASC) ,
  CONSTRAINT `fk_k_payment_k_bill1`
    FOREIGN KEY (`k_bill_bill_num` )
    REFERENCES `kontora`.`k_bill` (`bill_num` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `kontora`.`k_price`
```

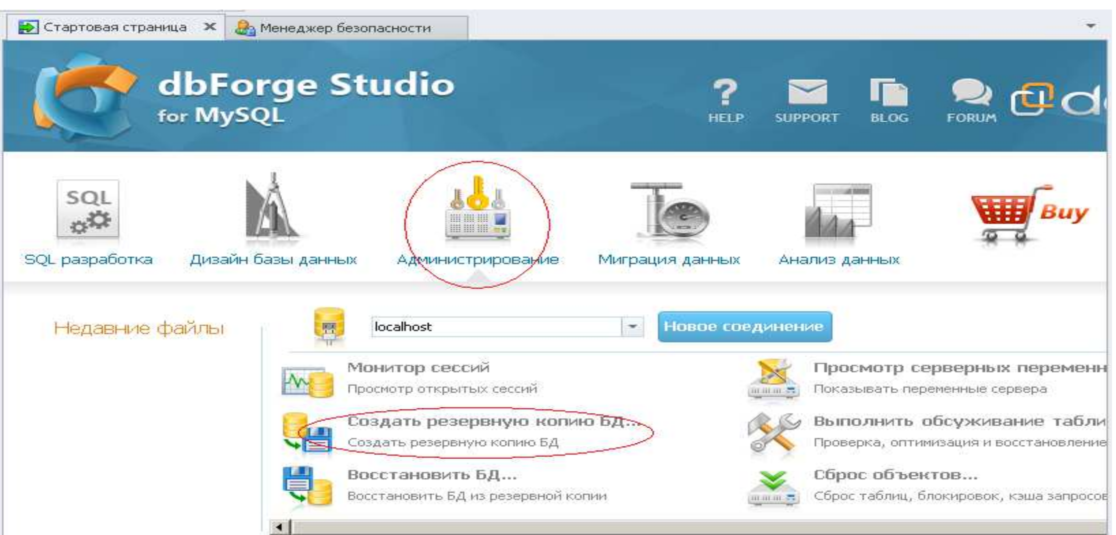
```
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `kontora`.`k_price` (
  `price_num` INT NULL AUTO_INCREMENT ,
  `price_name` VARCHAR(45) NOT NULL ,
  `price_sum` DECIMAL(9,2) NULL ,
  `price_type` VARCHAR(1) NULL ,
  PRIMARY KEY (`price_num`) )
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `kontora`.`k_protokol`
-- -----------------------------------------------------
CREATE  TABLE IF NOT EXISTS `kontora`.`k_protokol` (
  `k_bill_bill_num` INT NOT NULL ,
  `k_price_price_num` INT NOT NULL ,
  `kolvo` INT NOT NULL ,
  `price_sum` DECIMAL(9,2) NOT NULL ,
  PRIMARY KEY (`k_bill_bill_num`, `k_price_price_num`) ,
  INDEX `fk_k_bill_has_k_price_k_bill1` (`k_bill_bill_num` ASC) ,
  INDEX `fk_k_bill_has_k_price_k_price1` (`k_price_price_num` ASC) ,
  CONSTRAINT `fk_k_bill_has_k_price_k_bill1`
    FOREIGN KEY (`k_bill_bill_num` )
    REFERENCES `kontora`.`k_bill` (`bill_num` )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_k_bill_has_k_price_k_price1`
    FOREIGN KEY (`k_price_price_num` )
    REFERENCES `kontora`.`k_price` (`price_num` )
    ON DELETE NO ACTION

    ON UPDATE NO ACTION)
ENGINE = InnoDB;

SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

Приложение 2. Dbforge Studio от Devart

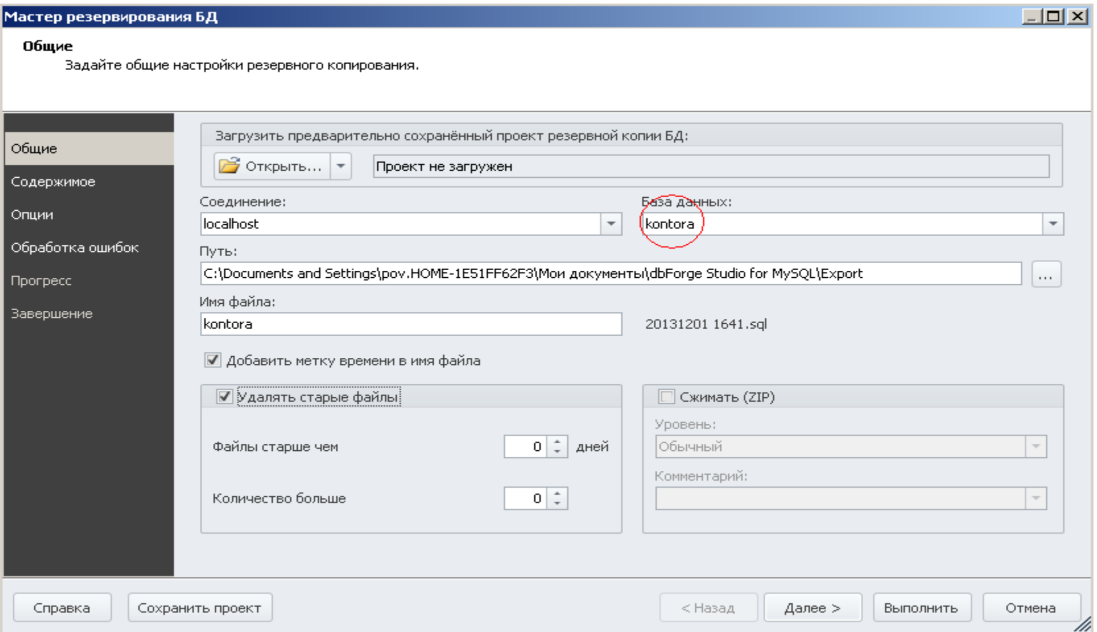
Для работы с сервером MySQL существуют альтернативные интерфейсы, позволяющие осуществлять разработку баз данных и администрирование сервера. Один из наиболее интересных вариантов – **dbForge Studio** от компании **Devart**. Приведем несколько иллюстраций для наиболее типичных административных функций.

Резервное копирование базы данных в dbforge Studio for mySQL

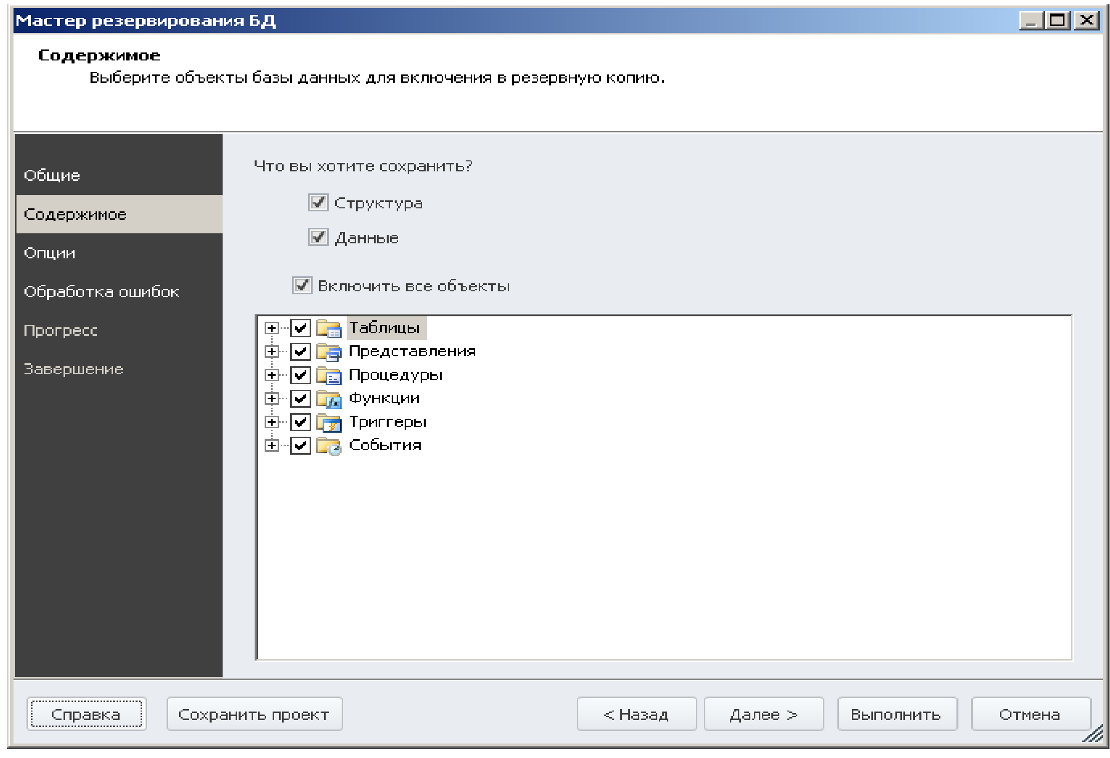
В секции **Администрирование в**ыбираем «**Создать резервную копию БД**»:



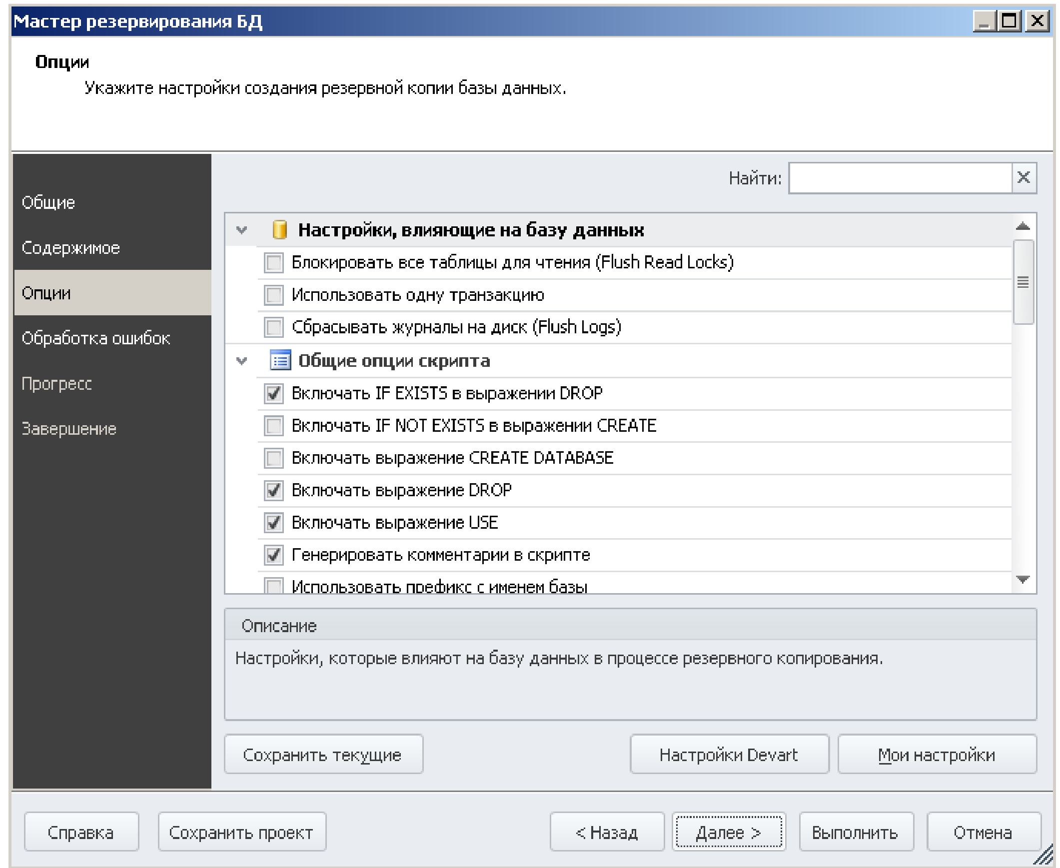
В следующем окне следует выбрать нужную базу данных:



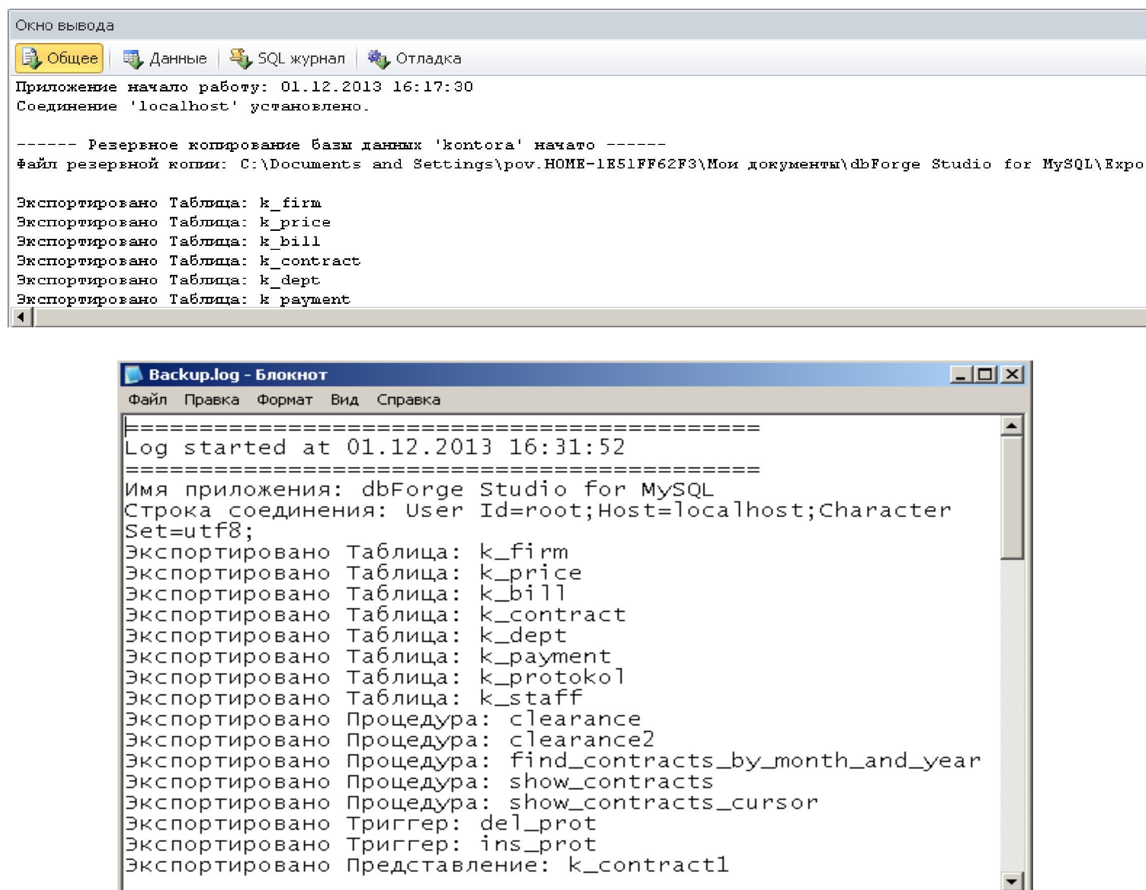
Далее можно указать, какие именно типы объектов базы данных следует архивировать:



В следующем окне можно уточнить настройки резервного копирования:

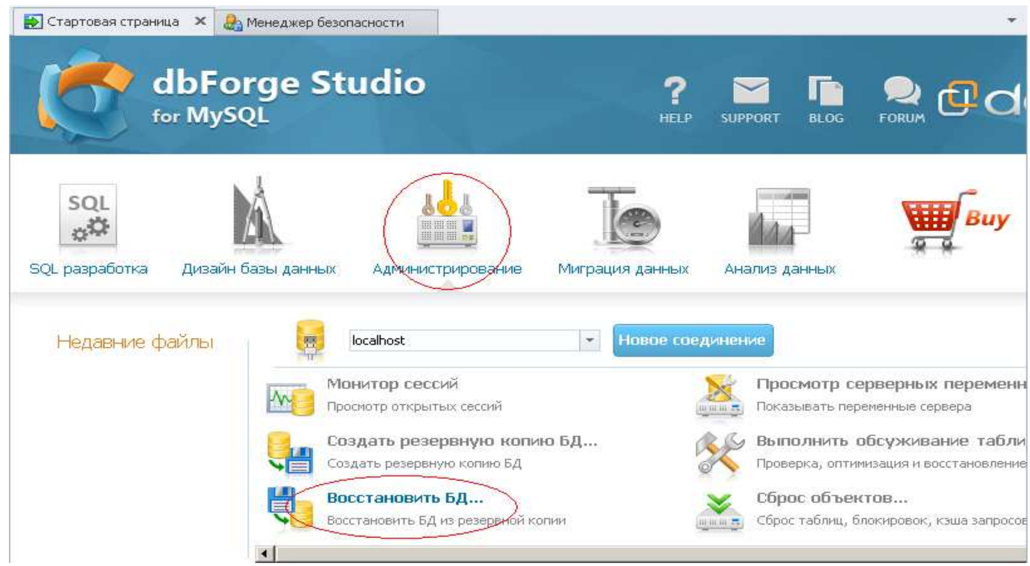


После нажатия на кнопку «**Выполнить»** происходит архивирование, и протокол работы может быть выведен на экран:

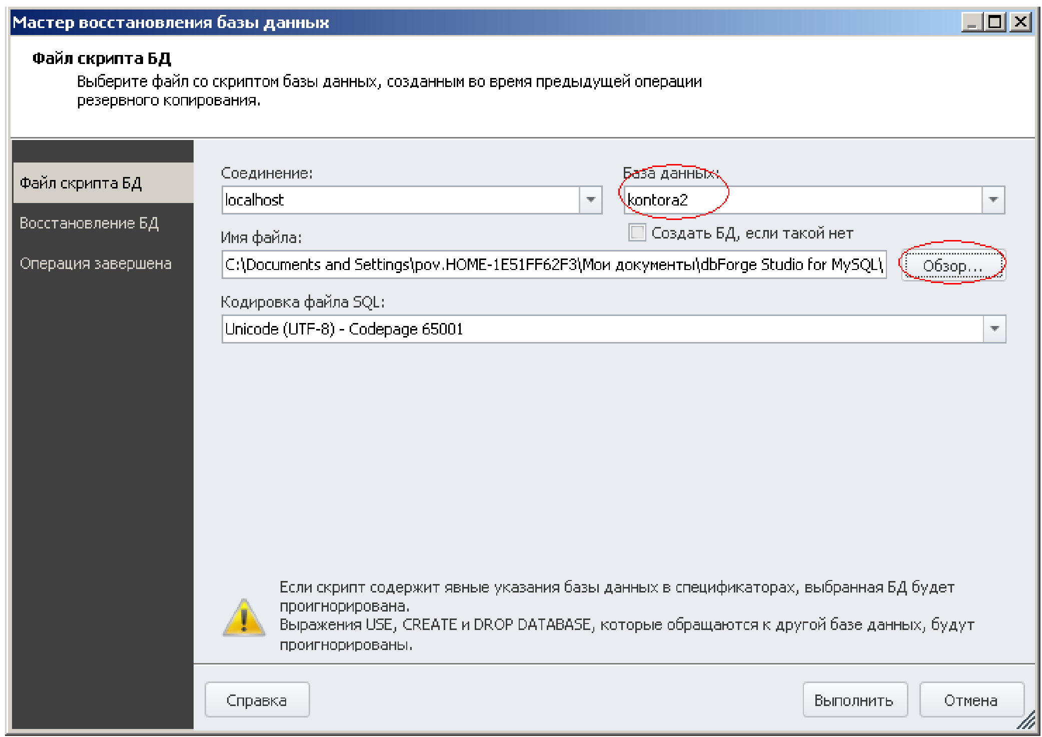


Восстановление базы данных в dbforge Studio for mySQL

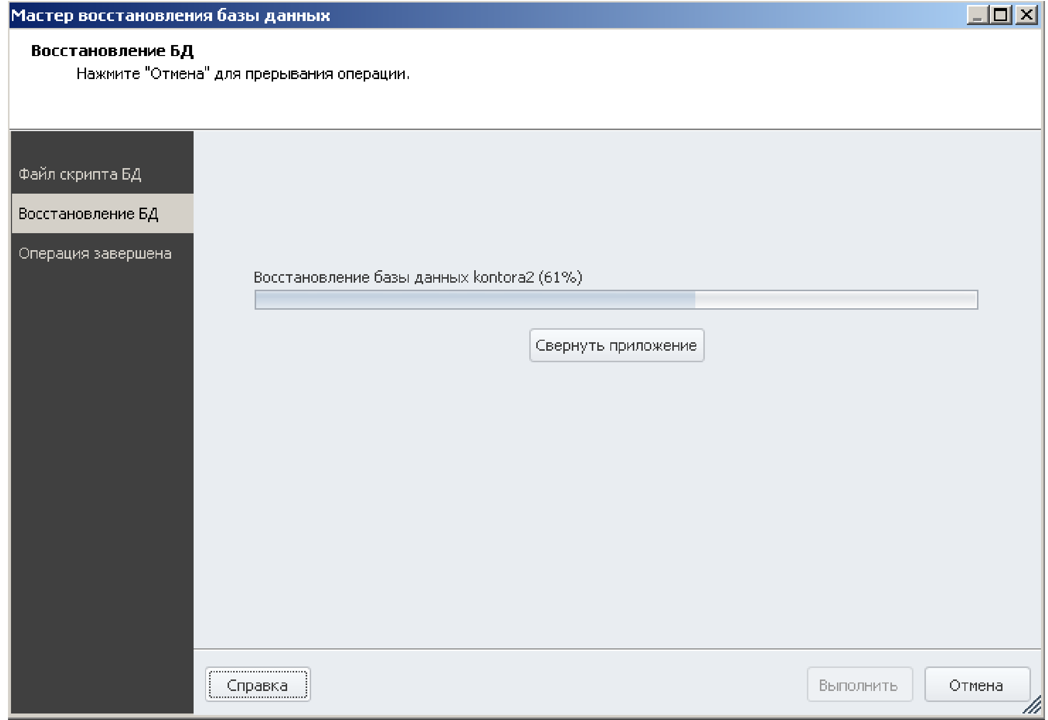
В секции **Администрирование** имеется пункт для восстановления базы данных из архивной копии:



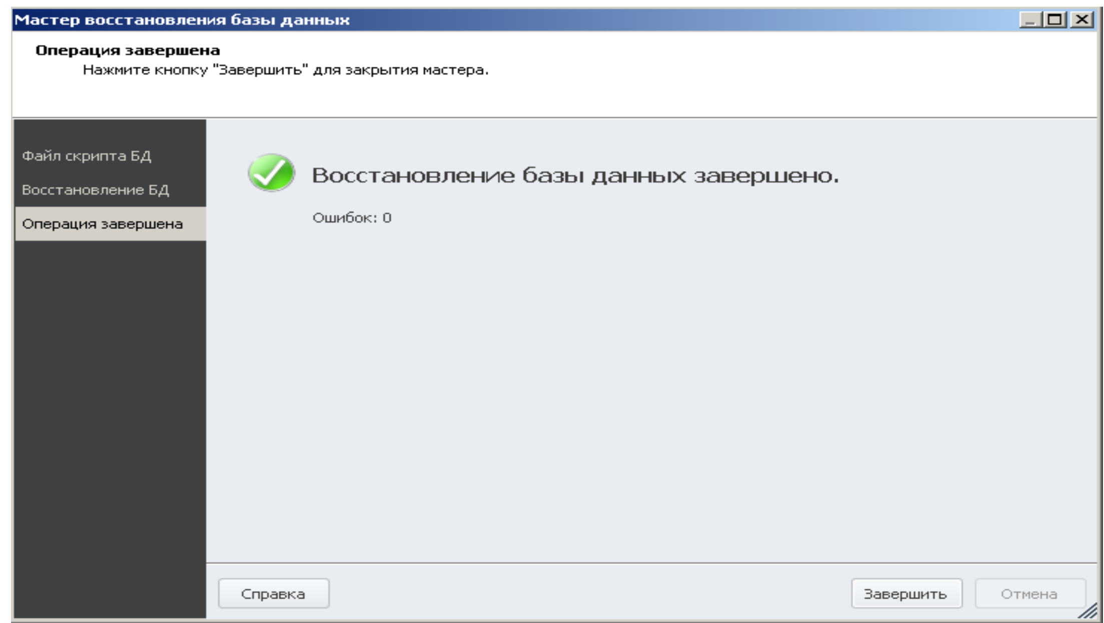
На первом шаге нужно задать имя для восстанавливаемой базы данных:



И нажать на кнопку «**Выполнить**»:



После выполнения операции восстановления будет выведено сообщение о ее результатах:



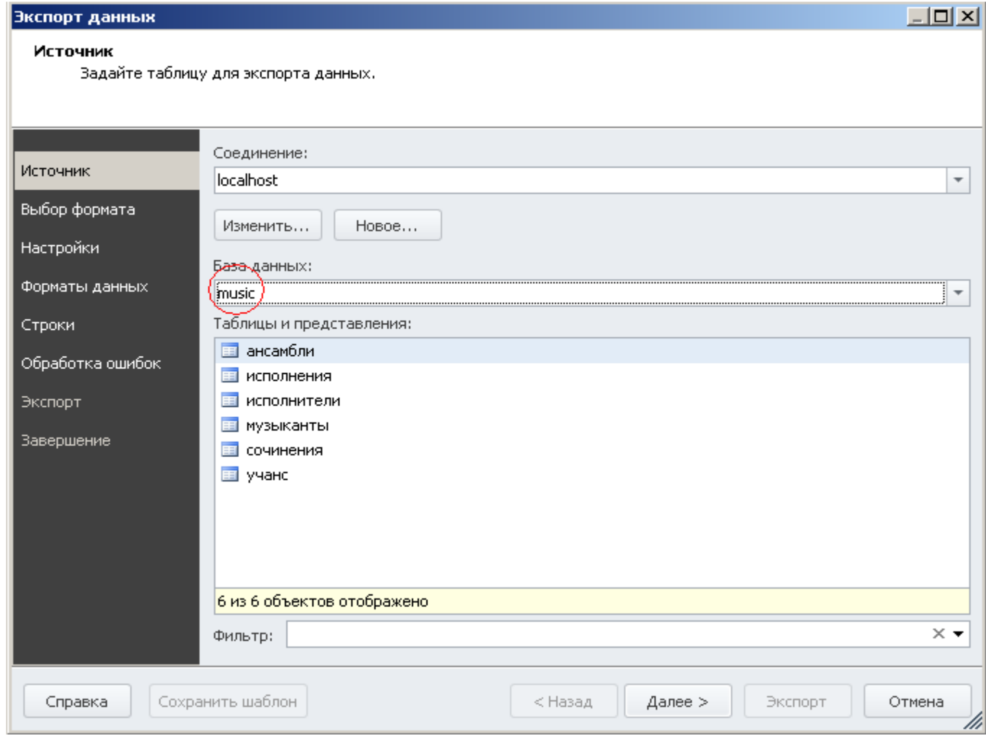
Миграция данных в dbforge Studio for mySQL

Под миграцией данных понимают операции импорта и экспорта – обмен данными с приложениями других форматов.

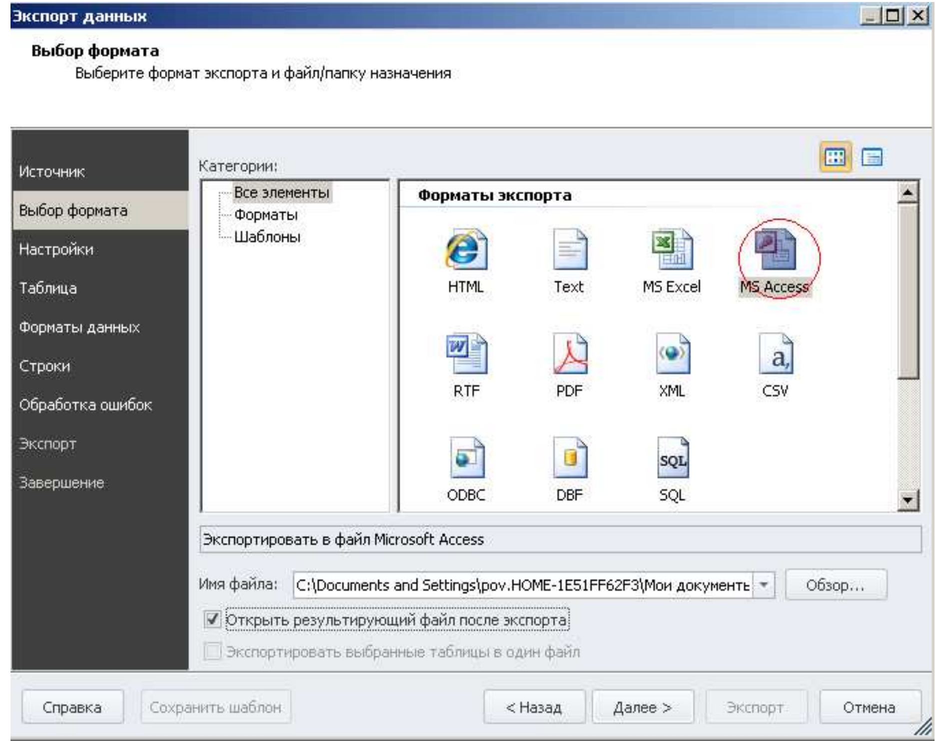
Рассмотрим операцию экспорта.

Экспорт данных в dbforge Studio for mySQL

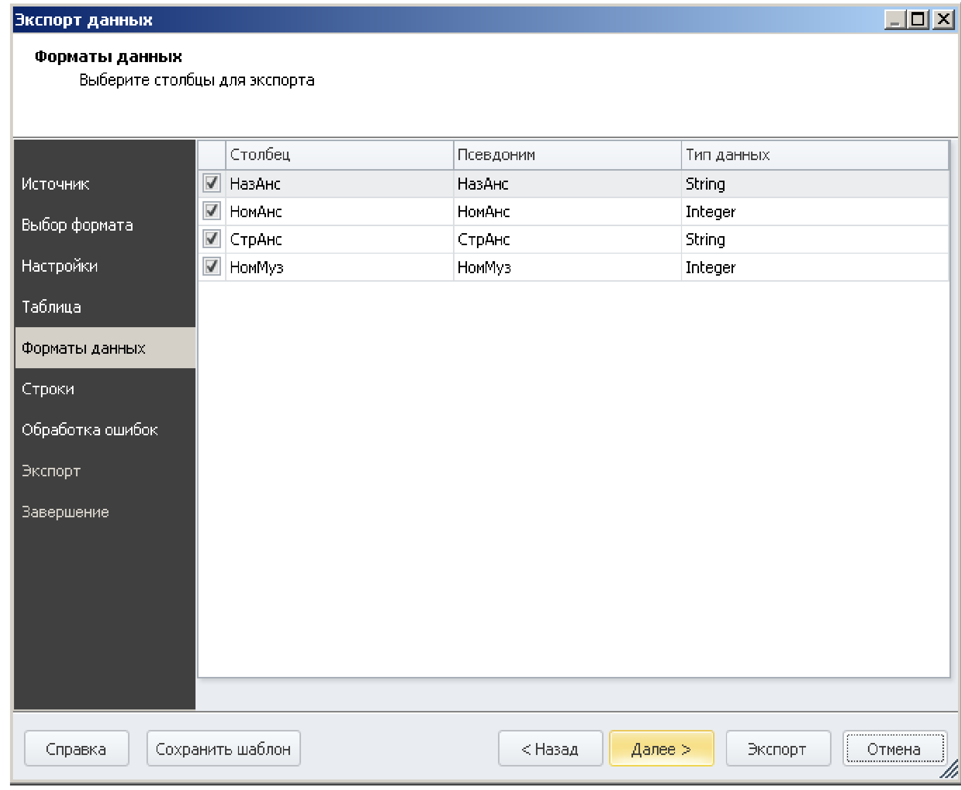
В первом окне следует выбрать базу данных и ее объекты для экспорта. Выберем базу данных **music** и таблицу **Ансамбли**:



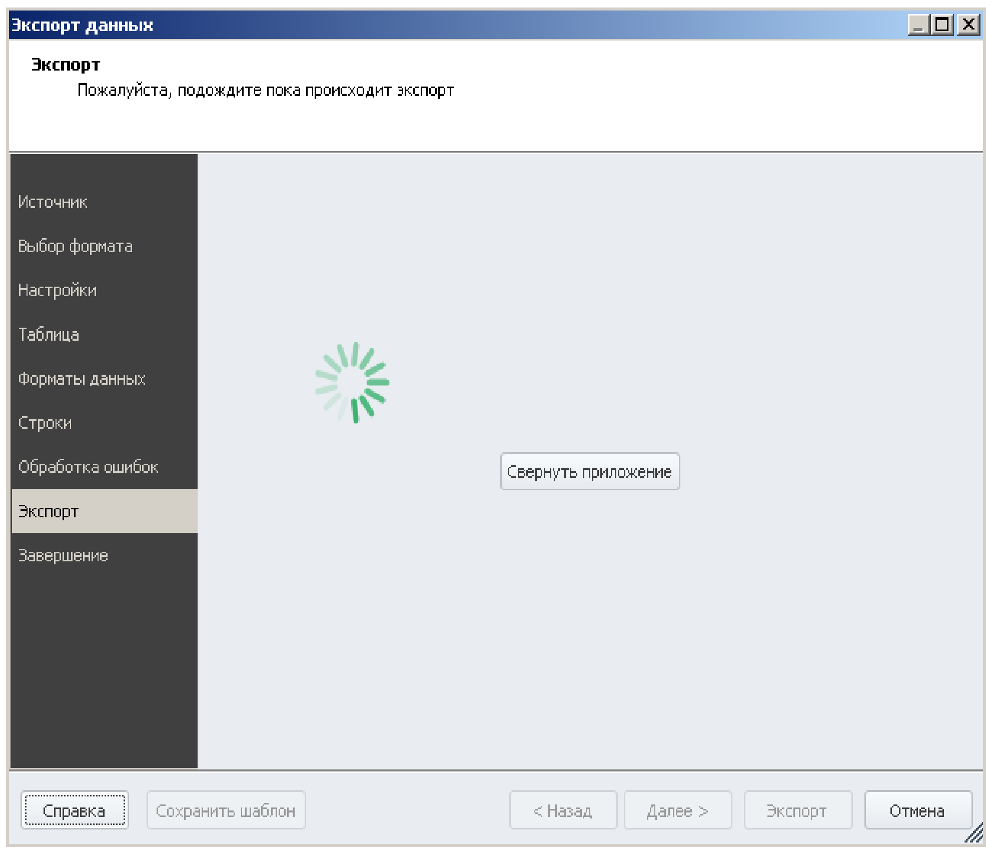
Далее выбираем формат, в который будут импортироваться данные. Для примера возьмем формат MS Access:

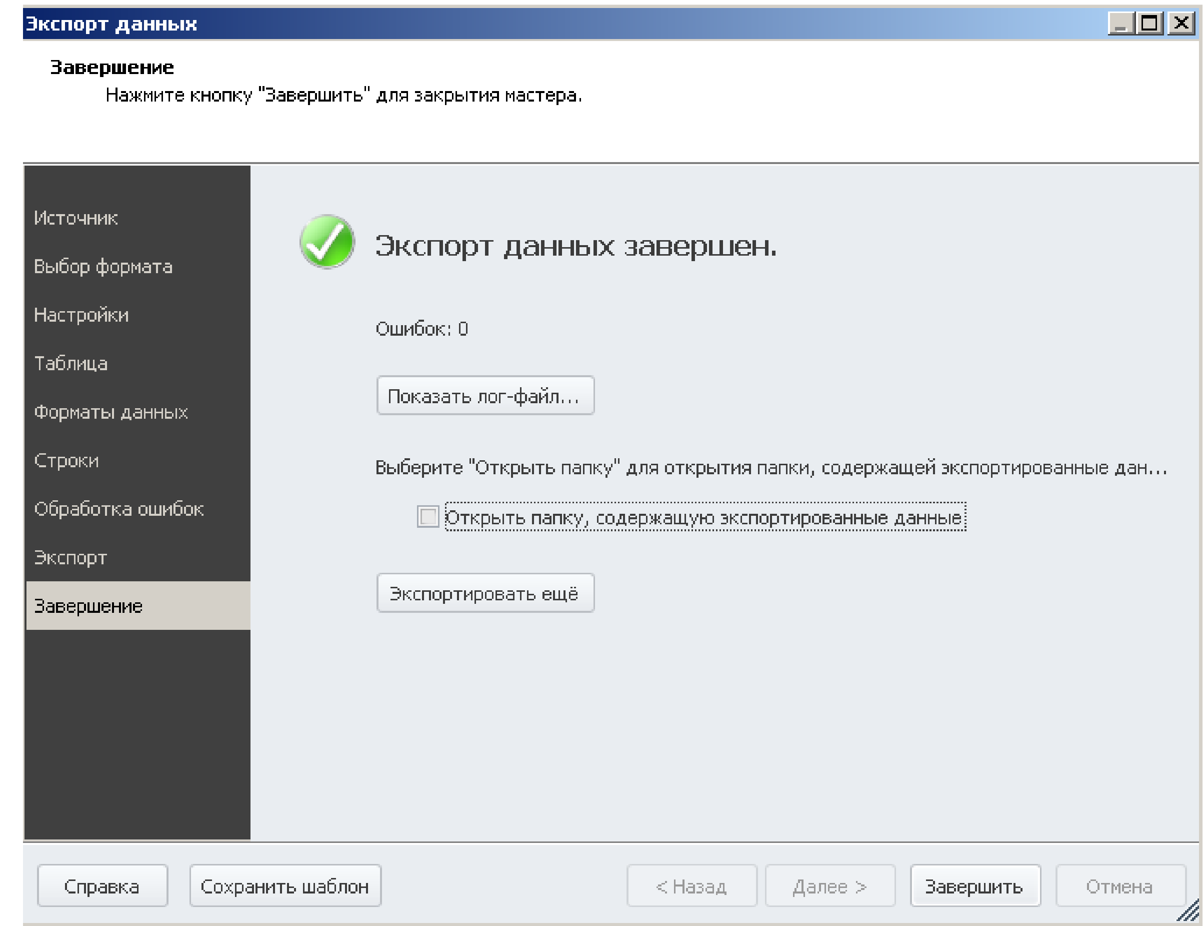


Далее можно уточнить структуру экспортируемых данных:



И можно запускать экспорт. Операция занимает некоторое время:





Импорт данных в dbforge Studio for mySQL

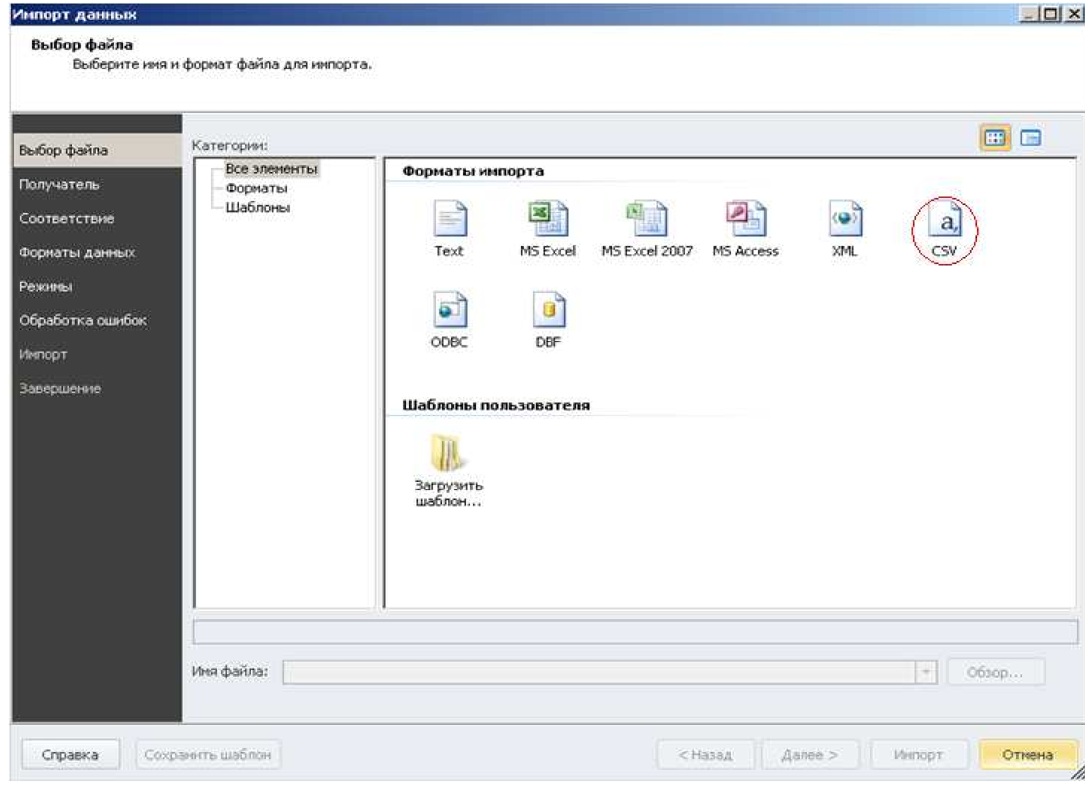
Подобным образом можно выполнять и обратный процесс – импорт данных из внешних источников. Пусть у нас есть структурированный текстовый файл с разделителями, содержащий информацию о товарах:

k_price.csv - Блокнот

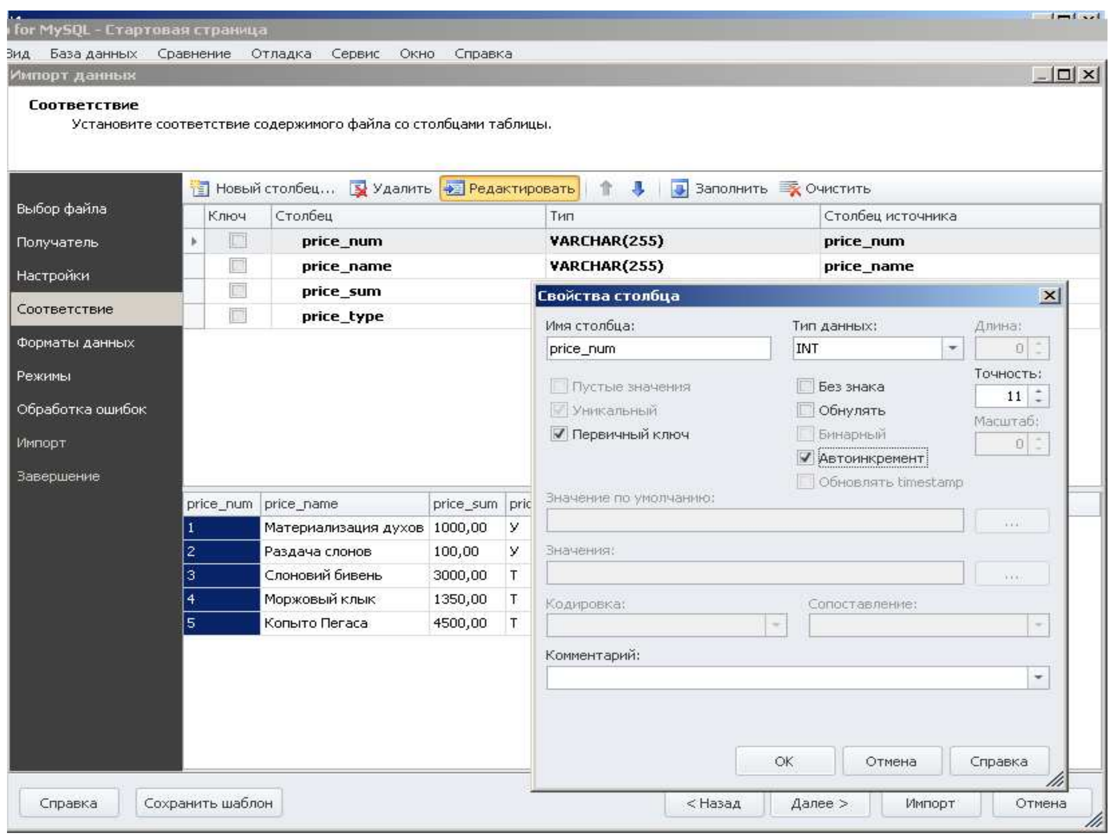
Файл Правка Формат Вид Справка

```
"price_num";"price_name";"price_sum";"price_type"
1;"Материализация духов";"1000,00";"У"
2;"Раздача слонов";"100,00";"У"
3;"Слоновий бивень";"3000,00";"Т"
4;"Моржовый клык";"1350,00";"Т"
5;"Копыто Пегаса";"4500,00";"Т"
```

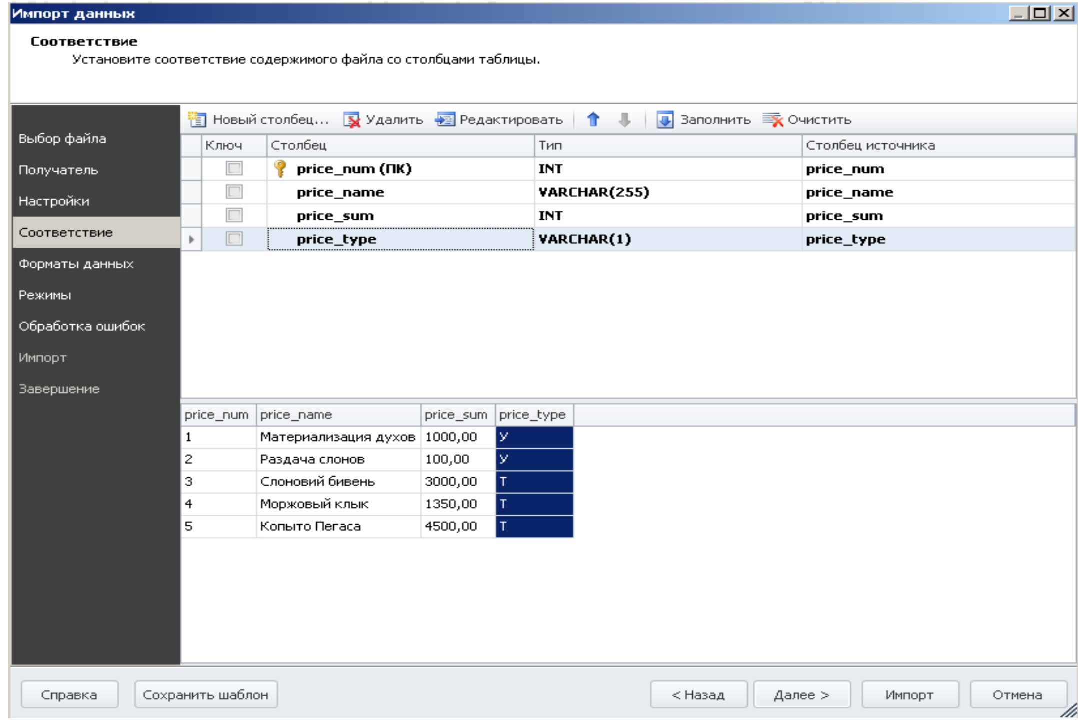
Такие файлы обычно имеют тип csv. Запускаем мастер для импорта, на первом экране выбираем тип файла:



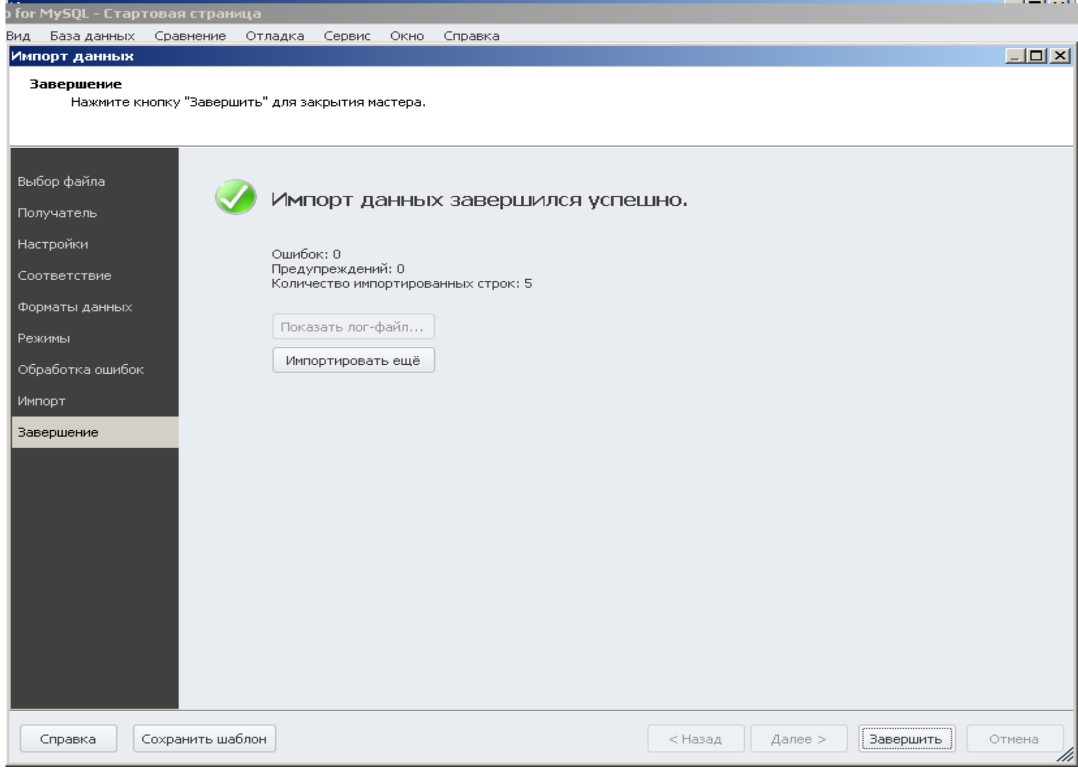
Далее мастер импорта анализирует данные и предлагает для них подходящие типы и размеры. Предложенные значения при необходимости можно поменять, а также назначить в таблице первичный ключ и другие ограничения.



Вот так выглядит структура таблицы после настройки:



После завершения операции выдается сообщение о ее результате:



Приложение 3. Реляционная алгебра и SQL

Рассмотрим, как связаны операции реляционной алгебры и язык SQL, т.е. приведем примеры запросов SQL, аналогичных операциям реляционной алгебры. В качестве примера базы данных будем использовать «Музыкантов».

Операция проекции **proj** выражается через `SELECT` с ключевым словом `DISTINCT`.

Получить все названия ансамблей:

proj НазАнс (Ансамбли)

```
SELECT DISTINCT НазАнс FROM Ансамбли
```

Операция выбора **sel** выражается через `SELECT` с ключевым словом `WHERE`.

Получить данные об ансамблях из России:

sel СтрАнс='Россия' (Ансамбли)

```
SELECT * FROM Ансамбли WHERE СтрАнс='Россия'
```

Условия также могут быть и сложными.

Получить имена музыкантов, родившихся в 20-м веке

```
SELECT ИмяМуз FROM Музыканты
WHERE ДатаРожд>'1900-12-31' AND ДатаРожд<'2001-01-01'
```

Операция соединения таблиц **join** может быть выражена несколькими способами.

Получить имена композиторов:

```
proj ИмяМуз (Музыканты join Сочинения)
```

Можно использовать связь таблиц через условие `WHERE`:

```
SELECT DISTINCT ИмяМуз FROM Музыканты M, Сочинения C WHERE
C.НомМуз=M.НомМуз
```

Можно использовать более современный синтаксис `JOIN ... ON`

```
SELECT DISTINCT ИмяМуз FROM Музыканты M JOIN Сочинения C ON
C.НомМуз=M.НомМуз
```

Если требуется вывести данные из одной таблицы, а условие накладывать на другую таблицу, то удобно использовать подзапросы, связанные и несвязанные.

```
SELECT DISTINCT ИмяМуз FROM Музыканты WHERE НомМуз IN
       (SELECT НомМуз FROM Сочинения)
```

или

```
SELECT DISTINCT ИмяМуз FROM Музыканты WHERE НомМуз = Any
       (SELECT НомМуз FROM Сочинения)
```

или

```
SELECT DISTINCT ИмяМуз FROM Музыканты M WHERE EXISTS
       (SELECT * FROM Сочинения C WHERE
             C.НомМуз=M.НомМуз)
```

Приведем пример сложного запроса, использующего данные из всех 6 таблиц базы данных.

Получить названия ансамблей, которые играли Моцарта на саксофоне:

```
proj  НазАнс
     (proj НомСоч (sel ИмяМуз='Моцарт' (Музыканты) join
              Сочинения)
       join
       proj НомСоч, НомАнс
          (proj НомИсп
          (sel Инструмент ='Саксофон' (Исполнители))
               join УчАнс join Исполнения)
       join Ансамбли )
```

```
SELECT НазАнс FROM Ансамбли WHERE НомАнс IN
(
	SELECT И1.НомАнс
	FROM Исполнения И1, Исполнители И2, Музыканты М,
		Сочинения С, УчАнс У
	WHERE И1.НомСоч=С.НомСоч AND С.НомМуз=М.НомМуз AND
		И1.НомАнс=У.НомАнс AND И2.НомИсп=У.НомИсп AND
		М.ИмяМуз='Моцарт' AND
		И2.Инструмент='Саксофон'
)
```

Операция объединения **union** соответствует нескольким командам SELECT, связанным ключевым словом UNION.

Получить общий список фамилий композиторов и дирижеров:

```
proj ИмяМуз (Музыканты join Сочинения)
     union
proj ИмяМуз (Музыканты join Исполнения)
```

```
SELECT DISTINCT ИмяМуз FROM Музыканты М, Сочинения С WHERE
С.НомМуз=М.НомМуз
UNION
SELECT DISTINCT ИмяМуз FROM Музыканты М, Исполнения И WHERE
И.НомМуз=М.НомМуз
```

Операция пересечения **intersection** может быть выражена несколькими способами.

Получить имена музыкантов, которые играют и на саксофоне, и на кларнете:

```
proj ИмяМуз (Музыканты join sel
     Инструмент='Саксофон'(Исполнители))
intersection
proj ИмяМуз (Музыканты join sel
     Инструмент='Кларнет'(Исполнители))
```

```
SELECT DISTINCT ИмяМуз FROM Музыканты М1,
	Исполнители И1, Исполнители И2
WHERE М1.НомМуз=И1.НомМуз AND
```

```
      И1.Инструмент='Саксофон' AND
      И2.Инструмент='Кларнет' AND
      И2.НомМуз=И1.НомМуз
```

или

```
SELECT DISTINCT ИмяМуз
      FROM Музыканты М1, Исполнители И1
WHERE М1.НомМуз=И1.НомМуз AND
      И1.Инструмент='Саксофон' AND
      М1.НомМуз IN
            (SELECT НомМуз FROM Исполнители И2
                  WHERE И2.Инструмент='Кларнет')
```

или

```
SELECT DISTINCT ИмяМуз
      FROM Музыканты М1, Исполнители И1
      WHERE М1.НомМуз=И1.НомМуз AND
            И1.Инструмент='Саксофон' AND
            М1.НомМуз =ANY
            (SELECT НомМуз FROM Исполнители И2
                  WHERE И2.Инструмент='Кларнет')
```

или

```
SELECT DISTINCT ИмяМуз
      FROM Музыканты М1, Исполнители И1
WHERE М1.НомМуз=И1.НомМуз AND
      И1.Инструмент='Саксофон' AND
      EXISTS
            (SELECT * FROM Исполнители И2
            WHERE И2.Инструмент='Кларнет'
            AND И2.НомМуз=И1.НомМуз)
```

Операция вычитания **`difference`** также может быть выражена несколькими способами.

Получить имена музыкантов, которые играют на саксофоне, но не играют на кларнете:

```
proj ИмяМуз (Музыканты join sel
         Инструмент='Саксофон'(Исполнители))
     diffrence
proj ИмяМуз (Музыканты join sel
         Инструмент='Кларнет'(Исполнители))
```

```
SELECT DISTINCT ИмяМуз
```

```
      FROM Музыканты М1, Исполнители И1
WHERE М1.НомМуз=И1.НомМуз AND
      И1.Инструмент='Саксофон' AND
      М1.НомМуз NOT IN
             (SELECT НомМуз FROM Исполнители И2
              WHERE И2.Инструмент='Кларнет')
```

или

```
SELECT DISTINCT ИмяМуз
      FROM Музыканты М1, Исполнители И1
      WHERE М1.НомМуз=И1.НомМуз AND
            И1.Инструмент='Саксофон' AND
            М1.НомМуз !=ALL
             (SELECT НомМуз FROM Исполнители И2
            WHERE И2.Инструмент='Кларнет')
```

или

```
SELECT DISTINCT ИмяМуз
      FROM  Музыканты М1, Исполнители И1
      WHERE М1.НомМуз=И1.НомМуз AND
            И1.Инструмент='Саксофон' AND
 NOT EXISTS
             (SELECT * FROM Исполнители И2
            WHERE И2.Инструмент='Кларнет'
            AND И2.НомМуз=И1.НомМуз)
```

Операция умножения **product** получается, если мы выполняем выборку из 2 таблиц, но не указываем условия связи.

Получить всевозможные пары имен музыкантов:

```
Музыканты2 aliases Музыканты

proj Музыканты.ИмяМуз, Музыканты2.ИмяМуз
     (Музыканты product Музыканты2)
```

```
SELECT М1.ИмяМуз, М2.ИмяМуз
FROM Музыканты М1, Музыканты М2
```

Очень интересно выглядит операция деления **division**. Она представляет собой двойное отрицание существования.

*Получить названия ансамблей, которые играли **все** произведения Моцарта (т.е., нет **ни одного** произведения Моцарта, которого они бы **не** играли):*

```
proj НазАнс
     (proj НомАнс, НомСоч (Исполнения)
              division
     proj НомСоч (sel ИмяМуз='Моцарт' (Музыканты)
              join Сочинения)
          join Ансамбли)
```

```
SELECT НазАнс FROM Ансамбли A WHERE NOT EXISTS
(
      SELECT * FROM Сочинения C, Музыканты M
      WHERE C.НомМуз=M.НомМуз AND ИмяМуз='Моцарт'
      AND NOT EXISTS
      (
            SELECT * FROM Исполнения И
            WHERE И.НомСоч=C.НомСоч AND
            И.НомАнс=A.НомАнс
      )
)
```

Литература

1. *Гольцман В.* MySQL 5.0 / В. Гольцман. – СПб.: Питер, 2009. – 256 с.

2. *Кузнецов М.* MySQL 5 / М. Кузнецов, И. Симдянов. – СПб: БХВ-Петербург, 2010. – 1024 с.

3. *Веллинг Л.* MySQL: учебное пособие / Л. Веллинг, Л. Томсон. – М.:Вильямс, 2005. – 304 с.

4. *Крёнке Д.* Теория и практика построения баз данных. 8-е изд. / Д. Крёнке. – СПб.: Питер, 2003. – 800 с.

Дистрибутивы

http://www.mysql.com/downloads/installer/
Пакет программ для Windows, содержащий **MySQL server**, среду для разработки и администрирования **MySQL Workbench** *и много других полезных компонентов*.

http://www.devart.com/ru/dbforge/mysql/studio/download.html
dbForge Studio for MySQL 6.0. Содержит набор инструментов для профессиональной разработки и управления MySQL-базы данных.