

## ЗАДАНИЕ ДЛЯ ПРАКТИЧЕСКИХ РАБОТ В СУБД

1. Научится формировать модель БД с помощью инструментов СУБД (MySQL Workbench, dbForge Studio, PostgreSQL – по выбору студента) по своей теме.
2. Научится осуществлять перенос своей БД на другой сервер.
3. Изучить команды модификации данных (DML)
4. Осуществить выборку данных по своей теме с помощью различных операторов.
5. Изучить и применить к своей БД хранимые процедуры, функции и триггеры.

Для выполнения работ необходимо изучить методические материалы ниже. *Рассмотренный пример в методическом материале прикреплять в отчет не надо.* **Выполнить все пункты задания необходимо по своей выбранной теме** и результат работы зафиксировать в едином отчете по практическим работам. Загрузку отчета осуществлять по согласованию с преподавателем по практическим занятиям в соответствии с расписанием.

**Срок сдачи всех работ – 20 декабря 2021 года. Максимальное количество баллов за все практические работы – 40 баллов.**

## ВВЕДЕНИЕ

Базы данных являются одной из основных составляющих большинства современных приложений, особенно прикладного или аналитического характера. Любое предприятие имеет свою базу данных (а, возможно, и множество баз данных). Заходя в интернет, мы видим информацию из баз данных через сервисы социальных сетей, интернет-магазинов, электронных университетов и др. Немало математических задач связано с использованием баз данных. Примером тому являются задачи анализа данных или машинного обучения. Таким образом, знание и навыки работы с базами данных становятся неотъемлемой составляющей компетенции современного ИТ-специалиста. Данное учебно-методическое пособие призвано помочь студенту в практической форме приобрести необходимые навыки работы с базами данных и их использованием в различных приложениях.

Разработка приложения, использующего базу данных, включает в себя множество задач. Во-первых, требуется сформировать логическую модель базы данных и, как следствие, набор таблиц, которые будут хранить данные. Вторым моментом является выбор системы управления базами данных (СУБД), на котором будет храниться база. Именно СУБД отвечают за выполнение основных операций, выполняемых с базой данных. Во многом этот выбор зависит от масштабов создаваемого приложения. В дальнейшем следует определить серверную часть приложения, включающую определения целостности данных, серверные процедуры, позволяющие выполнять основные преобразования данных. Только после решения всех этих вопросов речь заходит о клиентской части приложения работы с базой данных. Некоторые СУБД имеют собственные средства создания клиентской части (например, MS FoxPro или более популярный MS Access), но в большинстве своем современные СУБД являются серверными, т.е. предоставляют средства доступа к данным из других приложений. Этот момент позволяет создавать гибкий пользовательский интерфейс на тех технологиях, которые являются более приемлемыми для пользователя. Отдель-

ным вопросом функционирования приложения базы данных являются вопросы экспорта и импорта данных из других источников информации и агрегация информации из различных источников для предоставления сводной и аналитической отчетности (концепция хранилищ данных).

За время практикума каждый студент должен разработать собственное приложение баз данных, которое обязательно должно включать следующие элементы:

1. Создание логической модели базы данных. Описание ER-модели, генерация на ее основе реляционной модели данных.
2. Реализация модели в СУБД. В качестве СУБД могут быть выбраны: MS SQL Server, MySQL или PostgreSQL или иное серверное СУБД.
3. Заполнение базы данных.
4. Создание различных запросов на получение данных (для формирования навыков работы с реализацией различных операций реляционной алгебры). Для каждой из операций (исключая деление) нужно показать минимум три запроса (хотя один и тот же запрос может демонстрировать выполнение нескольких операций).
5. Создание хранимых процедур и триггеров для обеспечения серверной части работы с данными.

Каждая из перечисленных задач рассматривается в учебно-методическом пособии на примере создания элементов приложения «Деканат», с помощью

которого предоставляются возможности отслеживать оценки, которые получают студенты во время сессии.

В качестве средств разработки (программного обеспечения) нужно выбрать сервер баз данных, т.е. СУБД, инструментальную оболочку для работы с выбранным сервером, технологию создания клиентского интерфейса.

В качестве сервера баз данных можно использовать:

- MS SQL Server – устанавливается вместе с MS Visual Studio, которая может использоваться как оболочка доступа к базам данных. При установке SQL Server'у присваивается определенное имя, по которому к нему можно будет обращаться (по умолчанию SQLEXPRESS). Для локальной работы с сервером можно использовать при подключении имя (local). Свободной оболочкой (для некоммерческого использования) для MS SQL Server является программный продукт dbForge Studio компании DEVART (<http://www.devart.com/ru/dbforge/sql/studio>):

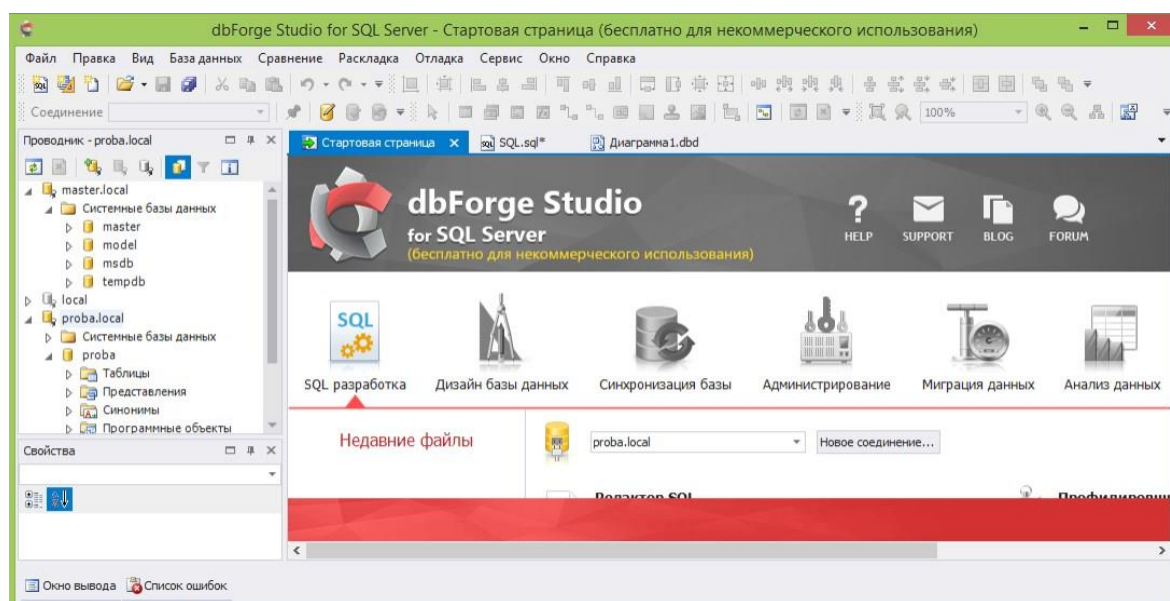


Рис. 1. Главное окно dbForge Studio для MS SQL Server.

Создание соединения оболочки с сервером производится с помощью меню «База данных» -> «Новое подключение...». Здесь вводятся параметры подключения и имя, по которому в дальнейшем к этому подключению можно будет обращаться:

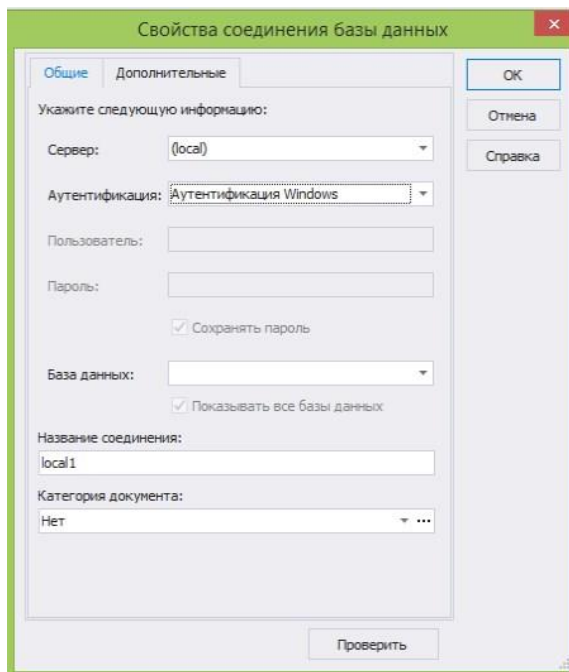


Рис. 2. Параметры соединения с базой данных MS SQL Server.

- MySQL (версии с 5.0). Этот бесплатный сервер баз данных устанавливается отдельно и конфигурируется с помощью специального wizard'a. Обратим внимание не то, что при конфигурировании экземпляра сервера требуется установить параметры учетной записи. По умолчанию, логин и пароль для сервера root. В качестве оболочки для работы с сервером MySQL можно использовать программный пакет MySQL Workbench – это свободное программное обеспечение, которое содержит средства моделирования, администрирования сервера и визуальной работы с базами данных, размещенными на нем.

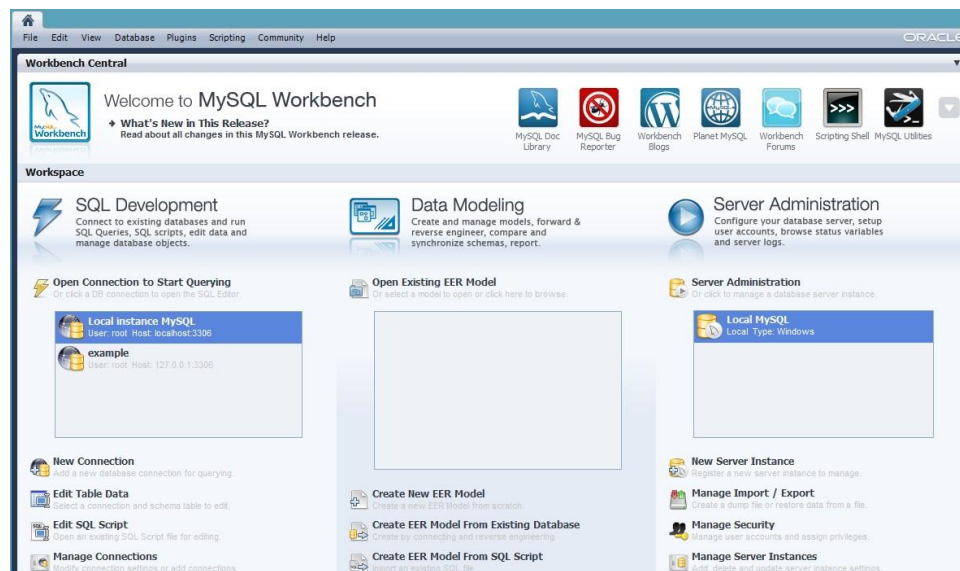


Рис. 3. Главное окно MySQL Workbench.

Для MySQL (аналогично MS SQL Server) компанией DEVART была разработана версия оболочки проектирования dbForge Studio. Она также является свободной для некоммерческого использования (<http://www.devart.com/ru/dbforge/mysql/studio>):

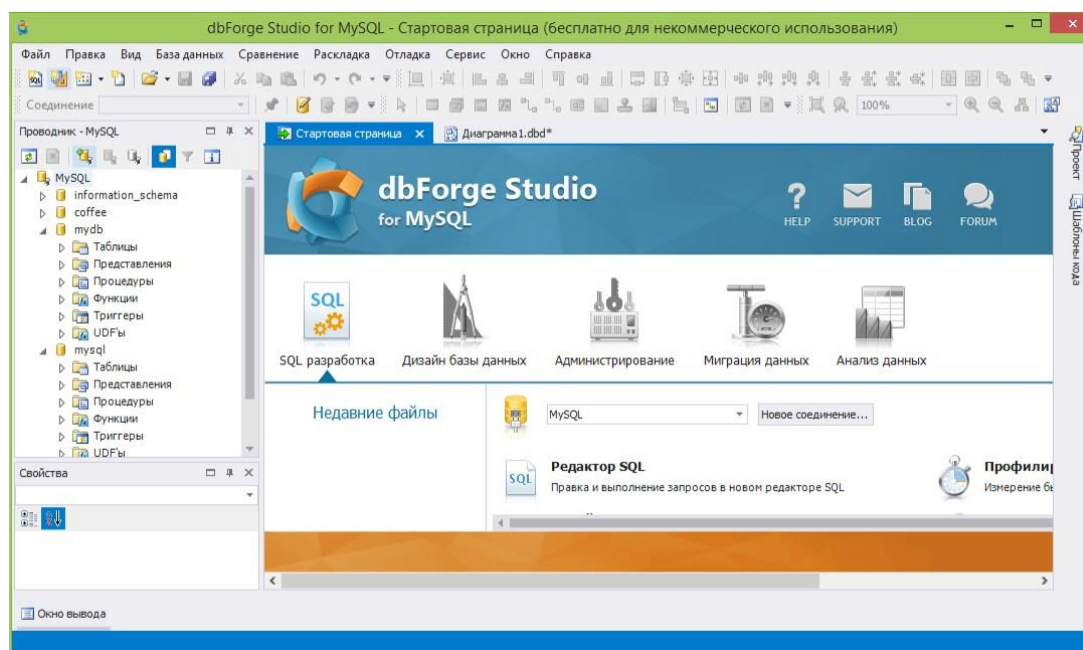


Рис. 4. Главное окно dbForge Studio для MySQL.

При создании подключения к MySQL серверу требуется указать другие параметры – это имя хоста, на котором установлен сервер баз данных (для локальных машин localhost), номер порта (по умолчанию MySQL ставится на порт 3306), логин и пароль учетной записи пользователя, а также

имя подключения. Еще не следует забывать на вкладке «Дополнительно» установить кодировку данных (сейчас настройки наиболее часто используют кодировку utf8) (MySQL очень чувствителен к кодировкам и отсутствие настройки кодировки может привести к проблемам с данными, написанными кириллицей):

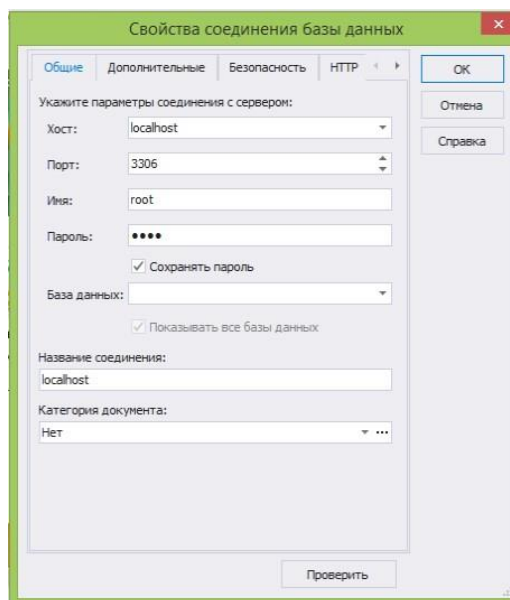


Рис. 5. Параметры соединения с базой данных MySQL.

- PostgreSQL также является свободным сервером баз данных. Также имеет оболочку проектирования pgAdmin. Существует уже оболочка dbForge Studio для PostgreSQL, однако на момент написания данного текста она была платным программным обеспечением.

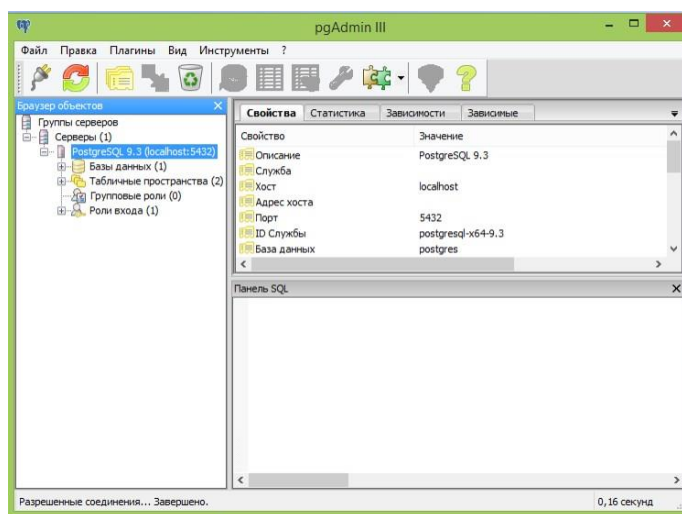


Рис. 6. Окно программы pgAdmin.

При установке сервера PostgreSQL и его дополнительного программного обеспечения будут запрошены параметры учетной записи пользователя. По умолчанию создается запись с логином postgres, пароль к которой устанавливает пользователь в момент установки. Аналогично MySQL, PostgreSQL идентифицируется хостом и номером порта (по умолчанию, 5432).



## ЧАСТЬ I. СЕРВЕРНЫЕ ТЕХНОЛОГИИ

### 1.1. МОДЕЛЬ ДАННЫХ

Разберем принципы формирования модели базы данных на примере приложения «Деканат». Модель будет создаваться с помощью инструментов моделирования данных в различных оболочках.

**Описание задачи.** Пусть требуется хранить и управлять информацией о результатах обучения студентов: об учебных группах; студентах, обучающихся в этих группах; дисциплинах, которые изучаются и сдаются в разные семестры; преподавателях, которые ведут эти дисциплины; оценках, которые были получены студентами при сдаче зачетов/экзаменов.

Существует несколько концепций моделей баз данных (иерархическая, сетевая, объектная, реляционная). Наиболее распространенной моделью является реляционная модель, которая очень тесно переплетается с принципами объектно-ориентированного анализа и еще одного популярного подхода в моделировании данных – ER-модели (модель «сущность-связь»).

ER-модель удобна для начального проектирования, поскольку она интуитивно понятна большинству пользователей. В ней выделяются понятия сущности (основные объекты базы), атрибуты (свойства сущности) и связи (взаимодействия между сущностями). В ряде оболочек именно в этих терминах и создан сервис создания модели данных.

Реляционная модель представляет всю базу данных как набор связанных таблиц. Большинство таблиц отвечает за хранение информации о сущностях (столбцы таблиц характеризуют их атрибуты). Среди атрибутов сущности выделяют ключевые атрибуты – атрибуты, которые являются идентифицирующими, точно определяющими запись, объект сущности. С помощью внедрения ключевых атрибутов одних сущностей (родительские таблицы) в качестве столбцов в другие таблицы (дочерние) реализуются различные связи между сущностями.

## Построение модели с помощью оболочки MySQL Workbench (версия 5.2.39 CE).

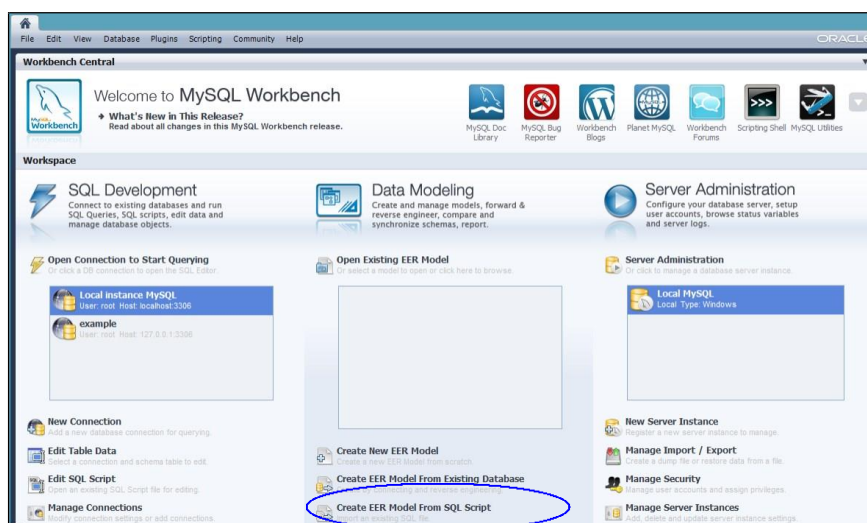


Рис. 7. Создание модели базы данных в MySQL Workbench.

Создаем новую ER-модель и диаграмму в модели. В полученном окне модели представлено полотно, на которое можно наносить новые таблицы, с помощью визуальных средств редактирования, создать столбцы (атрибуты) таблиц и с помощью панели инструментов создать связи между таблицами. При установке связи ключевые поля родительских сущностей добавляются в дочерние таблицы автоматически.

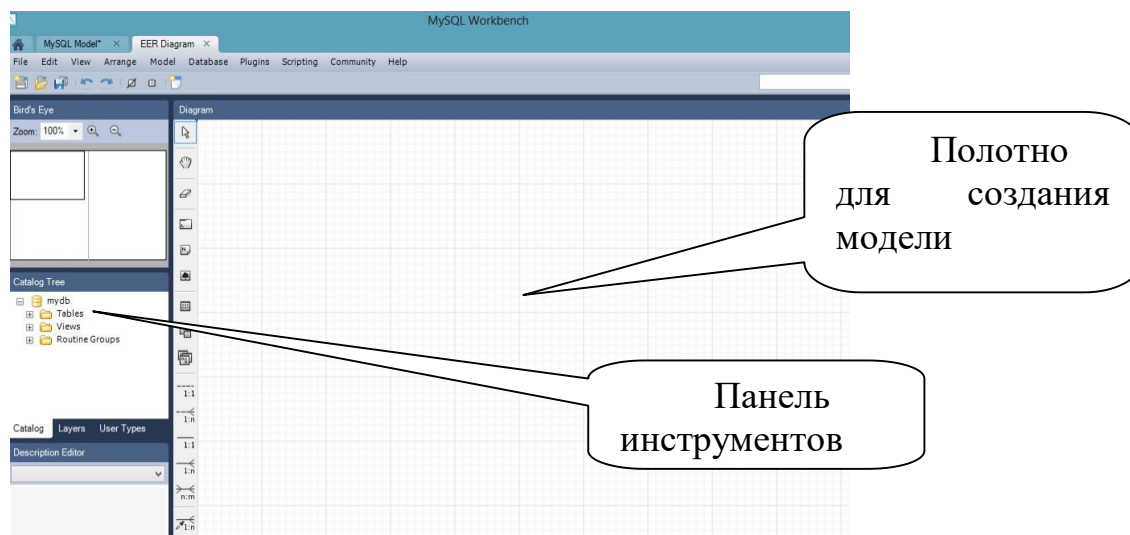


Рис. 8. Вид окна редактирования модели данных.

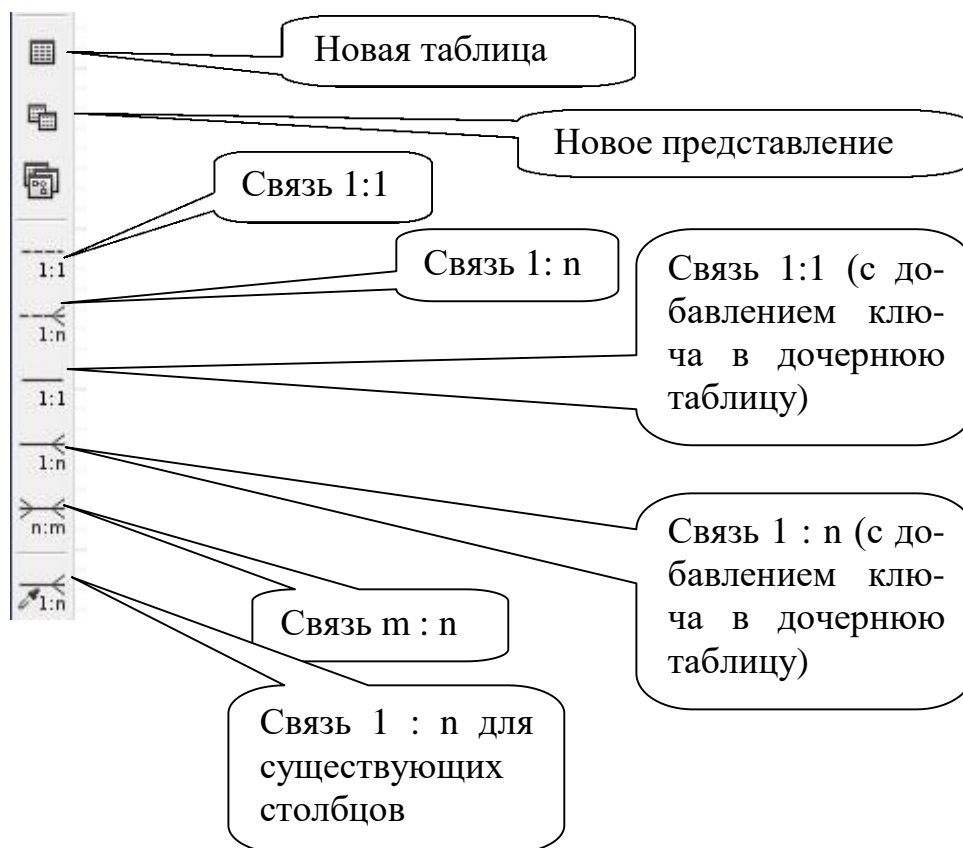


Рис. 9. Состав панели инструментов окна редактирования модели данных.

Проведем анализ состава таблиц для решаемой задачи. При описании столбцов таблицы поля, входящие в первичный ключ, будут подчеркнуты.

Имеется таблица **Студенты (Students):** (№Зач.книжки, ФИОСтудента, №Группы).

Для хранения групп не будем выделять отдельную таблицу.

Имеется таблица **Преподаватель (Teachers):** (№Преподавателя, ФИОПреподавателя, Должность, №Кафедры).

Чтобы избежать дублирования информации с названием кафедры введем справочную таблицу кафедр: таблица **Кафедра (Departments):** (№Кафедры, Название, Телефон).

Имеется таблица учебных дисциплин **Дисциплина (Subjects):** (№Дисциплины, Название).

Таблица **Сессия** содержит информацию о том, каков состав зачетов и экзаменов для каждой конкретной группы по семестрам, каким преподавате-

лям следует сдавать зачеты и экзамены: **Sessions** (№Группы, №Семестра, №Дисциплины, Отчетность, №Преподавателя). Заметим, что отчетность может определяться номером дисциплины и номером семестра, но в предположении наличии нескольких специальностей один и тот же предмет может сдаваться в разных семестрах разными группами. Поэтому отчетность и преподаватель зависят и от группы тоже.

Наконец, результаты сдачи сессии хранятся в таблице результатов **Results** (№Студента, №Группы, №Семестра, №Дисциплины, Баллы, ДатаСдачи, Оценка). Окончательную оценку хранить не требуется, так как она определяется количеством набранных баллов и таблицей оценок.

**Marks** (Оценка, НижняяГраница, ВерхняяГраница) – эта таблица является справочной и не связана с основными таблицами базы. Ее роль заключается в определении правильной оценки по набранным баллам.

В результате данного анализа задачи получится следующая модель:

- сначала формируется состав таблиц без связующих атрибутов:

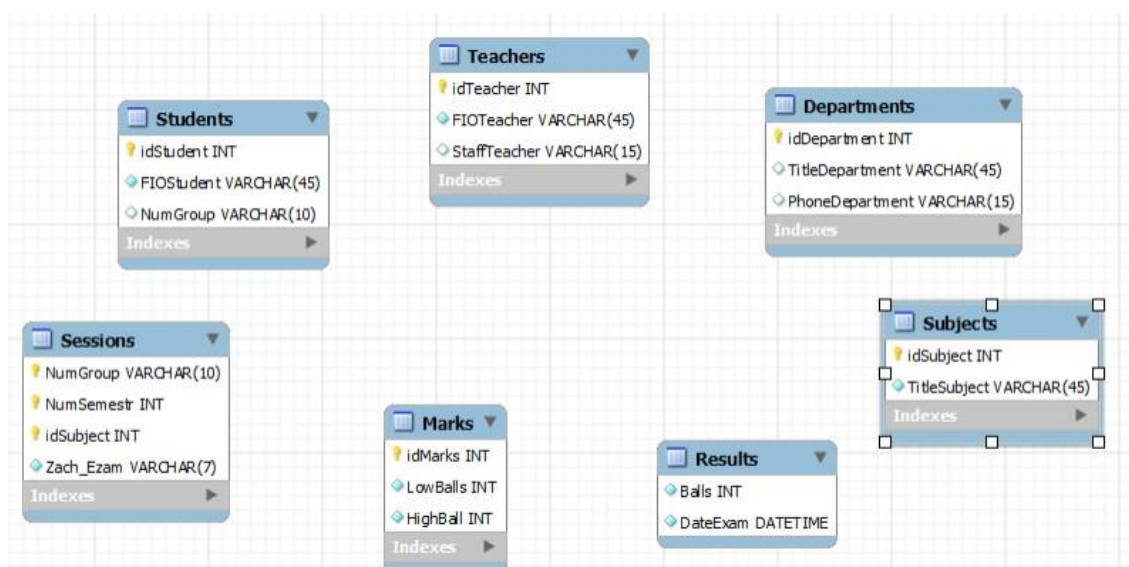


Рис. 10. Модель таблиц базы данных «Деканат» без указания связей.

- затем устанавливаем связи. Заметим, что можно было бы все связующие атрибуты сразу добавить в таблицы. Тогда все связи можно было бы добавить как связи «один-ко-многим» для существующих столбцов. Отметим также, что связь таблицы результатов и сессии не является очевидной, так как сессия зави-

сит от номера группы, а в таблице результатов указываются оценки конкретных студентов. Поэтому эту связь можно сделать идентифицирующей, а потом удалить из таблицы результатов атрибут номера группы. Другой вариант решения этой проблемы, добавить все поля в таблицу результатов и не устанавливать связь на уровне модели. Далее после создания таблиц в базе данных добавить ограничения внешних ключей для полей номера дисциплины и номера преподавателя.

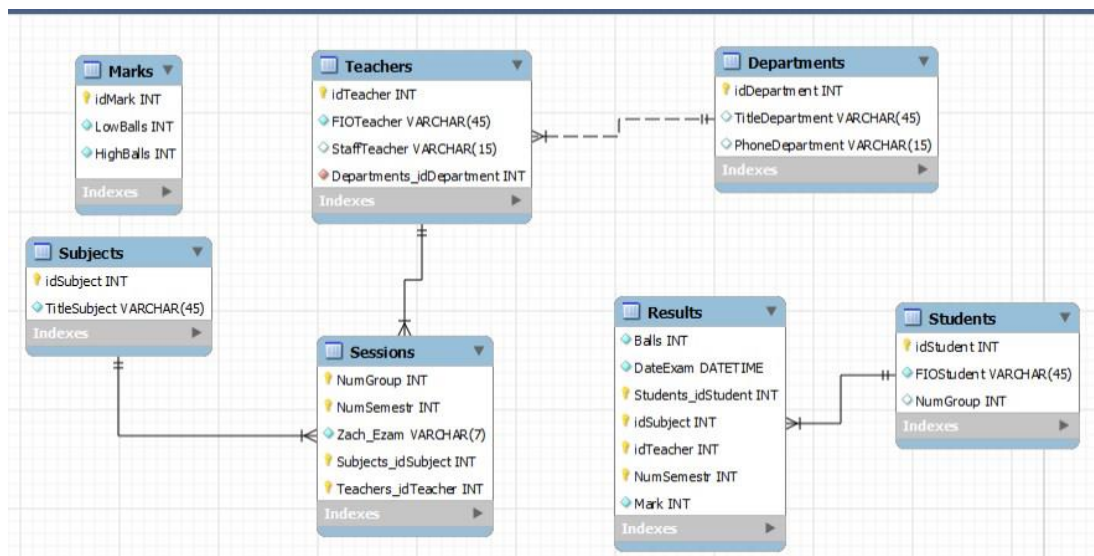


Рис. 11. Модель таблиц базы данных «Деканат» с указанием связей.

Отметим некоторую избыточность таблицы результатов относительно номера группы. Требуется обеспечить, чтобы номер группы и студенты были согласованы по таблицам студентов и результатов сессии.

### ***Построение модели в оболочке dbForge Studio для SQL Server***

Новую модель (диаграмму) базы данных можно создать с помощью меню «База данных»-> «Диаграмма БД».

Окно редактирования новой диаграммы состоит из полотна, на которое можно наносить новые таблицы с помощью визуальных средств редактирования, создать столбцы (атрибуты) таблиц и с помощью панели инструментов создать связи между таблицами.

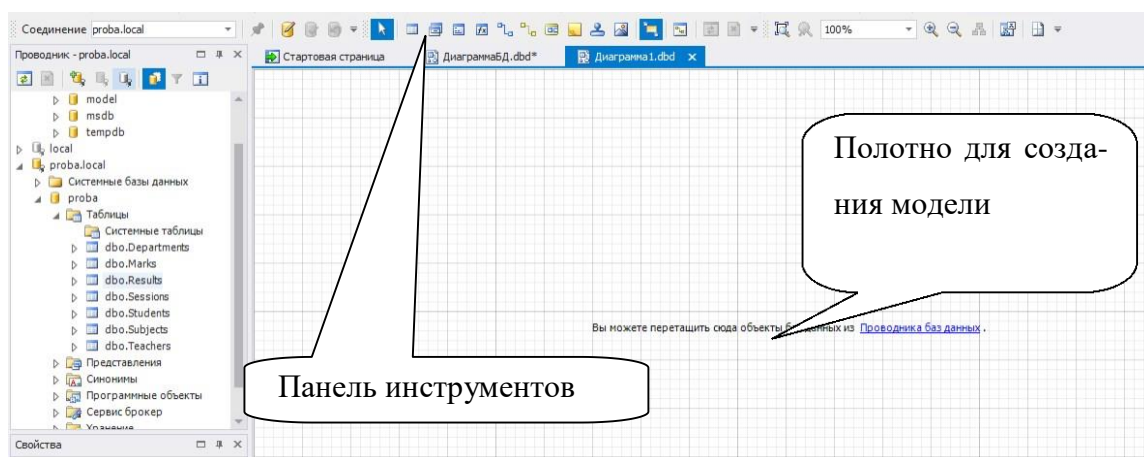


Рис.12. Окно редактирования диаграммы базы данных.

На панели инструментов следует отметить пока только две кнопки «Новая таблица», «Новая связь», которые позволяют создать новую таблицу, определив ее состав столбцов, первичные ключи и основные ограничения, и создать связи между таблицами, определив тем самым ограничения внешнего ключа.

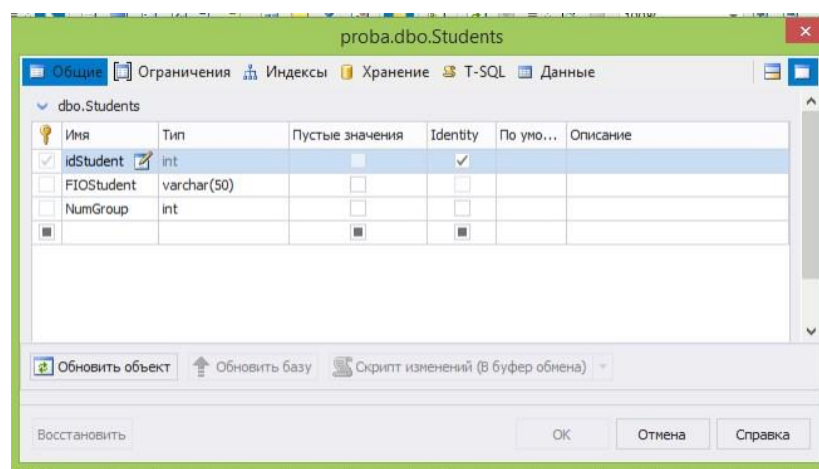


Рис. 13. Окно создания столбцов таблицы.

Для столбцов можно задать простые ограничения: допустимы ли пустые значения и определяет ли столбец поле-счетчик. Кроме того, можно выбрать столбцы, определяющие первичный ключ.

При создании связи требуется «нарисовать» мышью линию от дочерней таблицы к родительской. Для подтверждения параметров связи будет



показано окно, в котором нужно уточнить имена полей родительской и дочерней таблиц, которые будут связаны ограничением внешнего ключа:

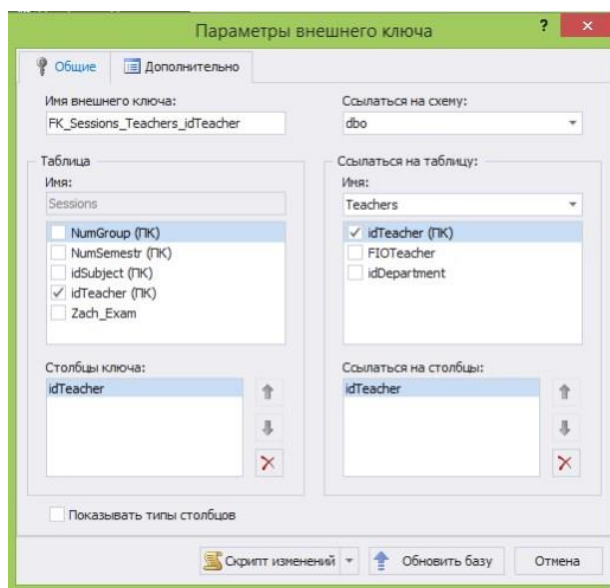


Рис. 14. Окно задания параметров внешнего ключа.

На вкладках «Ограничения» и «Индексы» можно увидеть все ограничения, которые сгенерируются в базе данных применительно к этой таблице. На вкладке T-SQL можно увидеть SQL-команду, выполнение которой эквивалентно выполнению всех сделанных настроек.

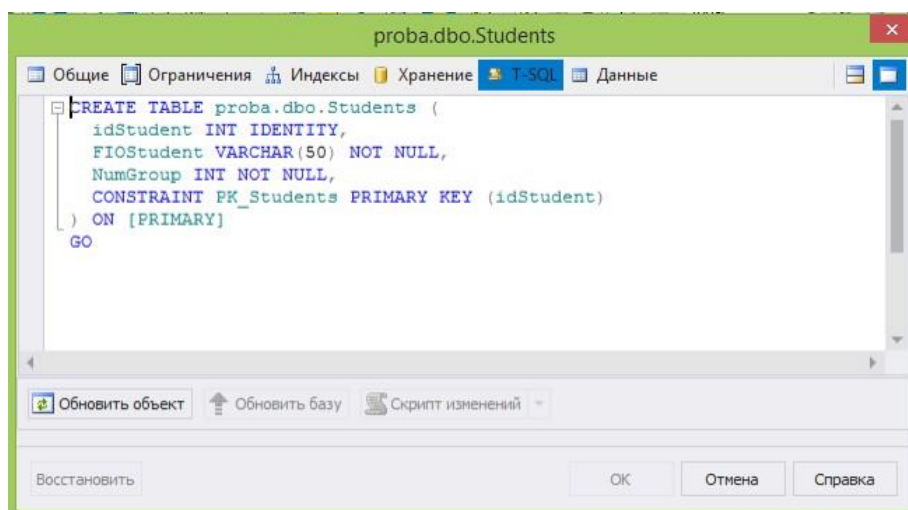


Рис. 15. Команда SQL создания таблицы «Студенты».

Отметим, что построитель модели синхронизирует все действия пользователя с базой данных, создавая указанные таблицы вместе со всеми ограничениями.

Таким образом, будет получена следующая модель:

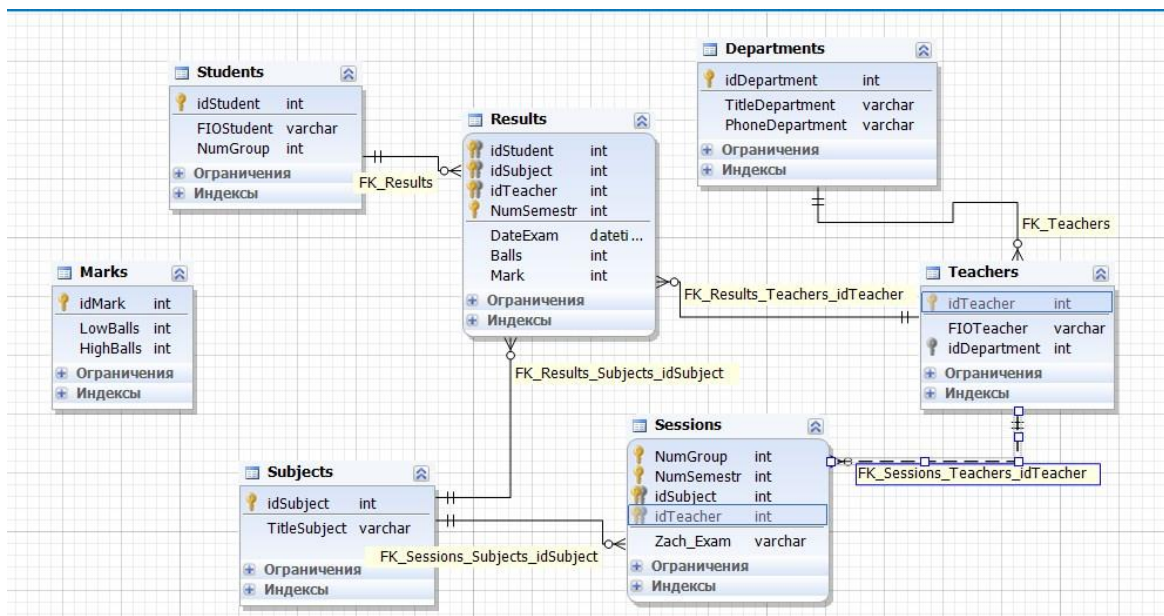


Рис. 16. Модель данных, построенная с помощью dbForge Studio.

Замечания относительно синхронизации номера группы в таблицах «Сессия» и «Студент» остаются на уровне модели нерешенным.

Аналогичным образом создается модель и, соответственно, база данных в среде dbForge Studio для MySQL.

Для PostgreSQL в стандартный набор инструмент формирования модели данных не входит. Поэтому состав таблиц нужно будет создать или с помощью специального SQL-оператора, или с помощью конструкторов таблиц:



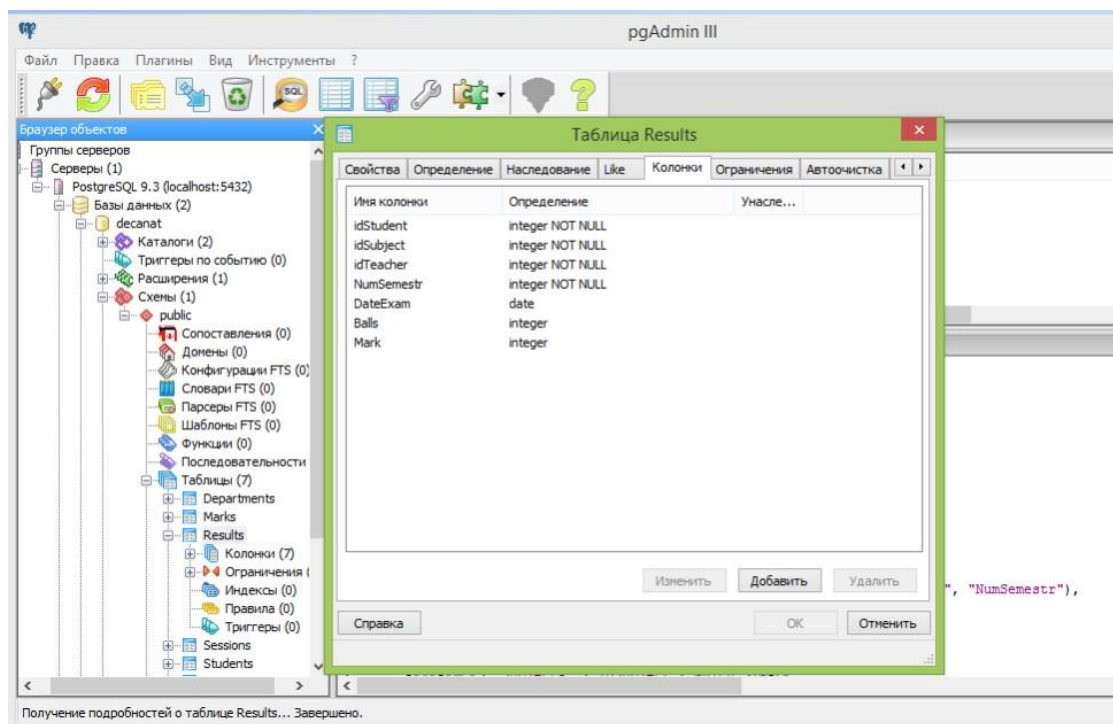


Рис. 17. Вид окна редактирования таблицы.

Действия с базой данных можно производить с помощью контекстного меню соответствующего элемента (таблицы, столбца, ограничения) в дереве объектов сервера. Настройки любого элемента производятся с помощью пункта контекстного меню «Свойства».

## 1.2. ПЕРЕНОС БАЗЫ ДАННЫХ НА ДРУГОЙ СЕРВЕР

Любое СУБД имеет средства резервного копирования базы данных. Такому копированию подвергаются как метаданные (структура данных базы), так и сами данные. Конечно, каждое СУБД имеет свои собственные форматы, но традиционным форматом является сохранение в виде последовательности SQL-команд (создания, вставки, изменения) (SQL-скрипт), выполнение которых приведет к текущему состоянию базы данных.

В оболочке dbForge Studio для SQL Server создание резервной копии (backup) можно осуществить двумя способами:

1. пункт меню «База данных» -> «Задачи» - > «Резервное копирование» (соответственно, для восстановления из резервной копии используется пункт меню «База данных» -> «Задачи» - > «Восстановление»). Этот способ связан с использованием специального формата MS SQL Server.

2. Генерация SQL-скрипта осуществляется с помощью пункта меню «База данных» -> «Задачи» - > «Сгенерировать скрипт...».

Кстати, многие важные опции, доступные через меню, доступны и на стартовой странице приложения, чтобы можно было получить к ним быстрый доступ:

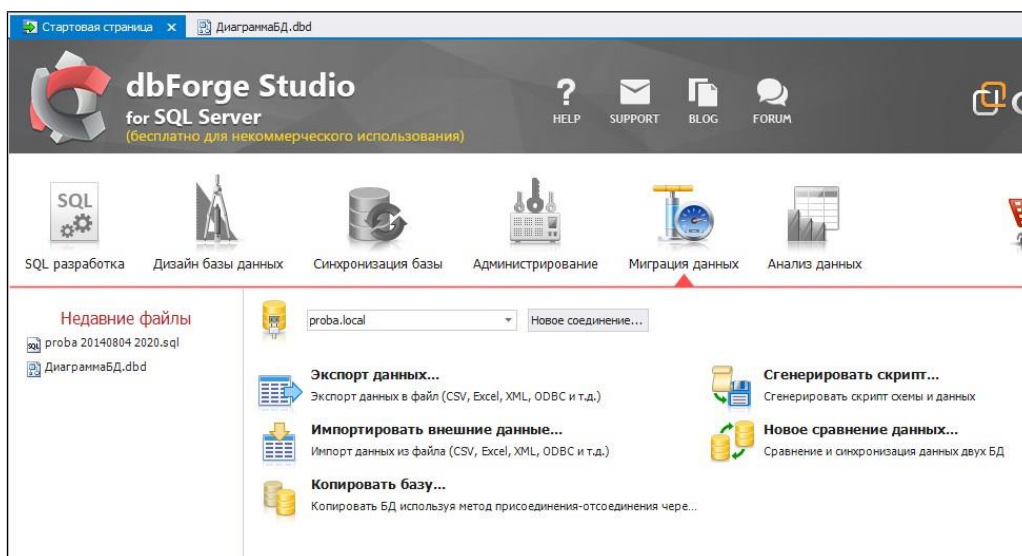


Рис. 18. Вид стартовой страницы для вкладки «Миграция данных».

В результате будет сгенерирован файл, содержащий следующие SQL-команды. Выделим полужирным шрифтом те команды, которые касаются создания базы данных и всех ее таблиц, а также определение ограничений:

```
--
-- Скрипт сгенерирован Devart dbForge Studio for SQL Server, Версия 3.8.180.1
-- Домашняя страница продукта: http://www.devart.com/ru/dbforge/sql/studio
-- Дата скрипта: 04.08.2014 23:36:06
-- Версия сервера: 11.00.2100
-- Версия клиента:
--
```

```
USE master
GO
```

```

IF DB_NAME() <> N'master' SET NOEXEC ON

--
-- Создать базу данных "proba"
--
PRINT (N'Создать базу данных "proba"')
GO

CREATE DATABASE proba
ON PRIMARY(
    NAME = N'proba',
    FILENAME = N'c:\Program Files\Microsoft SQL Serv-
er\MSSQL11.SQLEXPRESS\MSSQL\DATA\proba.mdf',
    SIZE = 4160KB,
    MAXSIZE = UNLIMITED,
    FILEGROWTH = 1024KB
)
LOG ON(
    NAME = N'proba_log',
    FILENAME = N'c:\Program Files\Microsoft SQL Serv-
er\MSSQL11.SQLEXPRESS\MSSQL\DATA\proba_log.ldf',
    SIZE = 1040KB,
    MAXSIZE = UNLIMITED,
    FILEGROWTH = 10%
)
GO

--
-- Изменить базу данных
--
PRINT (N'Изменить базу данных')
GO
ALTER DATABASE proba
SET
    ANSI_NULL_DEFAULT OFF,
    ANSI_NULLS OFF,
    ANSI_PADDING OFF,
    ANSI_WARNINGS OFF,
    ARITHABORT OFF,
    AUTO_CLOSE ON,
    AUTO_CREATE_STATISTICS ON,
    AUTO_SHRINK OFF,
    AUTO_UPDATE_STATISTICS ON,
    AUTO_UPDATE_STATISTICS_ASYNC OFF,
    COMPATIBILITY_LEVEL = 110,
    CONCAT_NULL_YIELDS_NULL OFF,
    CONTAINMENT = NONE,
    CURSOR_CLOSE_ON_COMMIT OFF,
    CURSOR_DEFAULT GLOBAL,
    DATE_CORRELATION_OPTIMIZATION OFF,
    DB_CHAINING OFF,
    HONOR_BROKER_PRIORITY OFF,
    MULTI_USER,
    NUMERIC_ROUNDABORT OFF,
    PAGE_VERIFY CHECKSUM,
    PARAMETERIZATION SIMPLE,

```

```

    QUOTED_IDENTIFIER OFF,
    READ_COMMITTED_SNAPSHOT OFF,
    RECOVERY SIMPLE,
    RECURSIVE_TRIGGERS OFF,
    TRUSTWORTHY OFF
    WITH ROLLBACK IMMEDIATE
GO

ALTER DATABASE proba
    SET ENABLE_BROKER
GO

ALTER DATABASE proba
    SET ALLOW_SNAPSHOT_ISOLATION OFF
GO

ALTER DATABASE proba
    SET FILESTREAM (NON_TRANSACTED_ACCESS = OFF)
GO
USE proba
GO
IF DB_NAME() <> N'proba' SET NOEXEC ON
GO

--
-- Создать таблицу "dbo.Teachers"
--
PRINT (N'Создать таблицу "dbo.Teachers"')
GO
CREATE TABLE dbo.Teachers (
    idTeacher int IDENTITY,
    FIOTeacher varchar(50) NOT NULL,
    idDepartment int NOT NULL,
    CONSTRAINT PK_Teachers PRIMARY KEY (idTeacher)
)
ON [PRIMARY]
GO

--
-- Создать таблицу "dbo.Subjects"
--
PRINT (N'Создать таблицу "dbo.Subjects"')
GO
CREATE TABLE dbo.Subjects (
    idSubject int IDENTITY,
    TitleSubject varchar(50) NOT NULL,
    CONSTRAINT PK_Subjects PRIMARY KEY (idSubject)
)
ON [PRIMARY]
GO

--
-- Создать таблицу "dbo.Students"
--
PRINT (N'Создать таблицу "dbo.Students"')
GO

```

```

CREATE TABLE dbo.Students (
    idStudent int IDENTITY,
    FIOStudent varchar(50) NOT NULL,
    NumGroup int NOT NULL,
    CONSTRAINT PK_Students PRIMARY KEY (idStudent)
)
ON [PRIMARY]
GO

--
-- Создать таблицу "dbo.Sessions"
--
PRINT (N'Создать таблицу "dbo.Sessions"')
GO
CREATE TABLE dbo.Sessions (
    NumGroup int NOT NULL,
    NumSemestr int NOT NULL,
    idSubject int NOT NULL,
    idTeacher int NOT NULL,
    Zach_Exam varchar(7) NOT NULL,
    CONSTRAINT PK_Sessions PRIMARY KEY (NumGroup, NumSemestr, idSubject, idTeacher)
)
ON [PRIMARY]
GO

--
-- Создать таблицу "dbo.Departments"
--
PRINT (N'Создать таблицу "dbo.Departments"')
GO
CREATE TABLE dbo.Departments (
    idDepartment int IDENTITY,
    TitleDepartment varchar(50) NOT NULL,
    PhoneDepartment varchar(50) NOT NULL,
    CONSTRAINT PK_Departments PRIMARY KEY (idDepartment)
)
ON [PRIMARY]
GO

--
-- Создать таблицу "dbo.Results"
--
PRINT (N'Создать таблицу "dbo.Results"')
GO
CREATE TABLE dbo.Results (
    idStudent int NOT NULL,
    idSubject int NOT NULL,
    idTeacher int NOT NULL,
    DateExam datetime NOT NULL,
    Balls int NOT NULL,
    Mark int NOT NULL,
    NumSemestr int NOT NULL,
    CONSTRAINT PK_Results PRIMARY KEY (idStudent, idSubject, idTeacher, NumSemestr)
)

```

```

ON [PRIMARY]
GO

--
-- Создать таблицу "dbo.Marks"
--
PRINT (N'Создать таблицу "dbo.Marks"')
GO
CREATE TABLE dbo.Marks (
    idMark int IDENTITY,
    LowBalls int NOT NULL,
    HighBalls int NOT NULL,
    CONSTRAINT PK_Marks PRIMARY KEY (idMark)
)
ON [PRIMARY]
GO
--
-- секция для команд вставки данных из всех таблиц - ее пропустим
--
-- Создать внешний ключ "FK_Teachers" для объекта типа таблица "dbo.Teachers"
--
PRINT (N'Создать внешний ключ "FK_Teachers" для объекта типа таблица
"dbo.Teachers"')
GO
ALTER TABLE dbo.Teachers
    ADD CONSTRAINT FK_Teachers FOREIGN KEY (idDepartment) REFERENCES
dbo.Departments (idDepartment)
GO
--
-- Создать внешний ключ "FK_Sessions_Subjects_idSubject" для объекта типа табли-
ца "dbo.Sessions"
--
PRINT (N'Создать внешний ключ "FK_Sessions_Subjects_idSubject" для объекта типа
таблица "dbo.Sessions"')
GO
ALTER TABLE dbo.Sessions
    ADD CONSTRAINT FK_Sessions_Subjects_idSubject FOREIGN KEY (idSubject) REFER-
ENCES dbo.Subjects (idSubject)
GO
--
-- Создать внешний ключ "FK_Sessions_Teachers_idTeacher" для объекта типа табли-
ца "dbo.Sessions"
--
PRINT (N'Создать внешний ключ "FK_Sessions_Teachers_idTeacher" для объекта типа
таблица "dbo.Sessions"')
GO
ALTER TABLE dbo.Sessions
    ADD CONSTRAINT FK_Sessions_Teachers_idTeacher FOREIGN KEY (idTeacher) REFER-
ENCES dbo.Teachers (idTeacher)
GO
--
-- Создать внешний ключ "FK_Results" для объекта типа таблица "dbo.Results"
--

```

```

PRINT (N'Создать внешний ключ "FK_Results" для объекта типа таблица
"dbo.Results"')
GO
ALTER TABLE dbo.Results
    ADD CONSTRAINT FK_Results FOREIGN KEY (idStudent) REFERENCES dbo.Students
(idStudent)
GO

--
-- Создать внешний ключ "FK_Results_Subjects_idSubject" для объекта типа таблица
"dbo.Results"
--
PRINT (N'Создать внешний ключ "FK_Results_Subjects_idSubject" для объекта типа
таблица "dbo.Results"')
GO
ALTER TABLE dbo.Results
    ADD CONSTRAINT FK_Results_Subjects_idSubject FOREIGN KEY (idSubject) REFER-
ENCES dbo.Subjects (idSubject)
GO

--
-- Создать внешний ключ "FK_Results_Teachers_idTeacher" для объекта типа таблица
"dbo.Results"
--
PRINT (N'Создать внешний ключ "FK_Results_Teachers_idTeacher" для объекта типа
таблица "dbo.Results"')
GO
ALTER TABLE dbo.Results
    ADD CONSTRAINT FK_Results_Teachers_idTeacher FOREIGN KEY (idTeacher) REFER-
ENCES dbo.Teachers (idTeacher)
GO
SET NOEXEC OFF
GO

```

При работе с оболочкой dbForge Studio для MySQL используется другой путь к пункту меню: «База данных» -> «Резервная копия» -> «Создать резервную копию БД» (аналогично можно использовать гиперссылку на стартовой странице в разделе «Миграция данных»). Приведем содержимое этого файла:

```

--
-- Скрипт сгенерирован Devart dbForge Studio for MySQL, Версия 6.2.233.0
-- Домашняя страница продукта: http://www.devart.com/ru/dbforge/mysql/studio
-- Дата скрипта: 04.08.2014 23:47:11
-- Версия сервера: 5.0.67-community-nt
-- Версия клиента: 4.1
---
-
-- Отключение внешних ключей
--
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0
*/;
--

```

```

-- Установить режим SQL (SQL mode)
--
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
--
-- Установка базы данных по умолчанию
--
USE decanat;
--
-- Описание для таблицы departments
--
DROP TABLE IF EXISTS departments;
CREATE TABLE departments (
  idDepartment INT(11) NOT NULL AUTO_INCREMENT,
  TitleDepartment VARCHAR(255) NOT NULL,
  PhoneDepartment VARCHAR(255) NOT NULL,
  PRIMARY KEY (idDepartment)
)
ENGINE = INNODB
AUTO_INCREMENT = 1
CHARACTER SET utf8
COLLATE utf8_general_ci;

--
-- Описание для таблицы marks
--
DROP TABLE IF EXISTS marks;
CREATE TABLE marks (
  idMark INT(11) NOT NULL,
  LowBalls INT(11) NOT NULL,
  HighBalls INT(11) NOT NULL,
  PRIMARY KEY (idMark)
)
ENGINE = INNODB
CHARACTER SET utf8
COLLATE utf8_general_ci;

--
-- Описание для таблицы students
--
DROP TABLE IF EXISTS students;
CREATE TABLE students (
  idStudent INT(11) NOT NULL AUTO_INCREMENT,
  FIOStudent VARCHAR(255) NOT NULL,
  NumGroup INT(11) NOT NULL,
  PRIMARY KEY (idStudent)
)
ENGINE = INNODB
AUTO_INCREMENT = 1
CHARACTER SET utf8
COLLATE utf8_general_ci;

--
-- Описание для таблицы subjects
--
DROP TABLE IF EXISTS subjects;
CREATE TABLE subjects (

```



```

    idSubject INT(11) NOT NULL AUTO_INCREMENT,
    TitleSubject VARCHAR(255) NOT NULL,
    PRIMARY KEY (idSubject)
)
ENGINE = INNODB
AUTO_INCREMENT = 1
CHARACTER SET utf8
COLLATE utf8_general_ci;

--
-- Описание для таблицы teachers
--
DROP TABLE IF EXISTS teachers;
CREATE TABLE teachers (
    idTeacher INT(11) NOT NULL AUTO_INCREMENT,
    FIOTeacher VARCHAR(255) NOT NULL,
    idDepartment INT(11) NOT NULL,
    PRIMARY KEY (idTeacher),
    CONSTRAINT FK_teachers_departments_idDepartment FOREIGN KEY (idDepartment)
        REFERENCES departments(idDepartment) ON DELETE NO ACTION ON UPDATE NO ACTION
)
ENGINE = INNODB
AUTO_INCREMENT = 1
CHARACTER SET utf8
COLLATE utf8_general_ci;

--
-- Описание для таблицы results
--
DROP TABLE IF EXISTS results;
CREATE TABLE results (
    idStudent INT(11) NOT NULL,
    idSubject INT(11) NOT NULL,
    idTeacher INT(11) NOT NULL,
    DateExam DATETIME NOT NULL,
    NumSemestr INT(11) NOT NULL,
    Balls INT(11) NOT NULL,
    Mark INT(11) NOT NULL,
    PRIMARY KEY (idStudent, idSubject, idTeacher, NumSemestr),
    CONSTRAINT FK_results_students_idStudent FOREIGN KEY (idStudent)
        REFERENCES students(idStudent) ON DELETE NO ACTION ON UPDATE NO ACTION,
    CONSTRAINT FK_results_subjects_idSubject FOREIGN KEY (idSubject)
        REFERENCES subjects(idSubject) ON DELETE NO ACTION ON UPDATE NO ACTION,
    CONSTRAINT FK_results_teachers_idTeacher FOREIGN KEY (idTeacher)
        REFERENCES teachers(idTeacher) ON DELETE NO ACTION ON UPDATE NO ACTION
)
ENGINE = INNODB
CHARACTER SET utf8
COLLATE utf8_general_ci;

--
-- Описание для таблицы sessions
--
DROP TABLE IF EXISTS sessions;
CREATE TABLE sessions (
    NumGroup INT(11) NOT NULL,

```

```

NumSemestr INT(11) NOT NULL,
idSubject INT(11) NOT NULL,
idTeacher INT(11) NOT NULL,
Zach_Exam VARCHAR(7) NOT NULL,
PRIMARY KEY (NumGroup, NumSemestr, idSubject, idTeacher),
CONSTRAINT FK_sessions_subjects_idSubject FOREIGN KEY (idSubject)
    REFERENCES subjects(idSubject) ON DELETE NO ACTION ON UPDATE NO ACTION,
CONSTRAINT FK_sessions_teachers_idTeacher FOREIGN KEY (idTeacher)
    REFERENCES teachers(idTeacher) ON DELETE NO ACTION ON UPDATE NO ACTION
)
ENGINE = INNODB
CHARACTER SET utf8
COLLATE utf8_general_ci;

--
-- секция для команд вставки данных из таблиц
--
--
-- Включение внешних ключей
--
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;

```

В PostgreSQL резервную копию можно создать с помощью пункта контекстного меню «Резервная копия» для элемента дерева, соответствующего копируемой базе данных. В результате будет сгенерирован следующий SQL-код:

```

--
-- PostgreSQL database dump
--
-- Dumped from database version 9.3.5
-- Dumped by pg_dump version 9.3.5
-- Started on 2014-08-04 23:54:32

SET statement_timeout = 0;
SET lock_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SET check_function_bodies = false;
SET client_min_messages = warning;
--
-- TOC entry 177 (class 3079 OID 11750)
-- Name: plpgsql; Type: EXTENSION; Schema: -; Owner:
--

CREATE EXTENSION IF NOT EXISTS plpgsql WITH SCHEMA pg_catalog;

--
-- TOC entry 1991 (class 0 OID 0)
-- Dependencies: 177
-- Name: EXTENSION plpgsql; Type: COMMENT; Schema: -; Owner:
--

```

```

COMMENT ON EXTENSION plpgsql IS 'PL/pgSQL procedural language';

SET search_path = public, pg_catalog;
SET default_tablespace = '';
SET default_with_oids = false;
--
-- TOC entry 171 (class 1259 OID 16397)
-- Name: Departments; Type: TABLE; Schema: public; Owner: postgres; Tablespace:
--

CREATE TABLE "Departments" (
    "idDepartment" oid NOT NULL,
    "TitleDepartment" text NOT NULL,
    "PhoneDepartment" text
);

ALTER TABLE public."Departments" OWNER TO postgres;

--
-- TOC entry 170 (class 1259 OID 16394)
-- Name: Marks; Type: TABLE; Schema: public; Owner: postgres; Tablespace:
--

CREATE TABLE "Marks" (
    "idMark" integer NOT NULL,
    "LowBalls" integer NOT NULL,
    "HighBalls" integer NOT NULL
);

ALTER TABLE public."Marks" OWNER TO postgres;

--
-- TOC entry 176 (class 1259 OID 16431)
-- Name: Results; Type: TABLE; Schema: public; Owner: postgres; Tablespace:
--

CREATE TABLE "Results" (
    "idStudent" integer NOT NULL,
    "idSubject" integer NOT NULL,
    "idTeacher" integer NOT NULL,
    "NumSemestr" integer NOT NULL,
    "DateExam" date,
    "Balls" integer,
    "Mark" integer
);

ALTER TABLE public."Results" OWNER TO postgres;

--
-- TOC entry 174 (class 1259 OID 16415)
-- Name: Sessions; Type: TABLE; Schema: public; Owner: postgres; Tablespace:
--

CREATE TABLE "Sessions" (
    "NumGroup" integer NOT NULL,
    "NumSemestr" integer NOT NULL,
    "idSubject" integer NOT NULL,
    "idTeacher" integer NOT NULL,
    "Zach_Exam" text

```

```

);

ALTER TABLE public."Sessions" OWNER TO postgres;
--
-- TOC entry 175 (class 1259 OID 16423)
-- Name: Students; Type: TABLE; Schema: public; Owner: postgres; Tablespace:
--

CREATE TABLE "Students" (
    "idStudent" oid NOT NULL,
    "FIOStudent" text,
    "NumGroup" integer
);

ALTER TABLE public."Students" OWNER TO postgres;
--
-- TOC entry 172 (class 1259 OID 16403)
-- Name: Subjects; Type: TABLE; Schema: public; Owner: postgres; Tablespace:
--

CREATE TABLE "Subjects" (
    "idSubject" oid NOT NULL,
    "TitleSubject" text NOT NULL
);

ALTER TABLE public."Subjects" OWNER TO postgres;
--
-- TOC entry 173 (class 1259 OID 16409)
-- Name: Teachers; Type: TABLE; Schema: public; Owner: postgres; Tablespace:
--

CREATE TABLE "Teachers" (
    "idTeacher" oid NOT NULL,
    "FIOTeacher" text NOT NULL,
    "idDepartment" integer NOT NULL
);

ALTER TABLE public."Teachers" OWNER TO postgres;
--
-- TOC entry 1978 (class 0 OID 16397)
-- Dependencies: 171
-- Data for Name: Departments; Type: TABLE DATA; Schema: public; Owner: postgres
--

COPY "Departments" ("idDepartment", "TitleDepartment", "PhoneDepartment") FROM
stdin;
\

--
-- TOC entry 1977 (class 0 OID 16394)
-- Dependencies: 170
-- Data for Name: Marks; Type: TABLE DATA; Schema: public; Owner: postgres
--

COPY "Marks" ("idMark", "LowBalls", "HighBalls") FROM stdin;

```

```

\..

--
-- TOC entry 1983 (class 0 OID 16431)
-- Dependencies: 176
-- Data for Name: Results; Type: TABLE DATA; Schema: public; Owner: postgres
--

COPY "Results" ("idStudent", "idSubject", "idTeacher", "NumSemestr", "DateExam",
"Balls", "Mark") FROM stdin;
\..

--
-- TOC entry 1981 (class 0 OID 16415)
-- Dependencies: 174
-- Data for Name: Sessions; Type: TABLE DATA; Schema: public; Owner: postgres
--

COPY    "Sessions"    ("NumGroup",    "NumSemestr",    "idSubject",    "idTeacher",
"Zach_Exam") FROM stdin;
\..

--
-- TOC entry 1982 (class 0 OID 16423)
-- Dependencies: 175
-- Data for Name: Students; Type: TABLE DATA; Schema: public; Owner: postgres
--

COPY "Students" ("idStudent", "FIOSStudent", "NumGroup") FROM stdin;
\..

--
-- TOC entry 1979 (class 0 OID 16403)
-- Dependencies: 172
-- Data for Name: Subjects; Type: TABLE DATA; Schema: public; Owner: postgres
--

COPY "Subjects" ("idSubject", "TitleSubject") FROM stdin;
\..

--
-- TOC entry 1980 (class 0 OID 16409)
-- Dependencies: 173
-- Data for Name: Teachers; Type: TABLE DATA; Schema: public; Owner: postgres
--

COPY "Teachers" ("idTeacher", "FIOTeacher", "idDepartment") FROM stdin;
\..

--
-- TOC entry 1853 (class 2606 OID 16439)
-- Name: pk_department; Type: CONSTRAINT; Schema: public; Owner: postgres; Ta-
blespace:
--

ALTER TABLE ONLY "Departments"

```

```

    ADD CONSTRAINT pk_department PRIMARY KEY ("idDepartment");

--
-- TOC entry 1851 (class 2606 OID 16441)
-- Name: pk_mark; Type: CONSTRAINT; Schema: public; Owner: postgres; Tablespace:
--

ALTER TABLE ONLY "Marks"
    ADD CONSTRAINT pk_mark PRIMARY KEY ("idMark");

--
-- TOC entry 1863 (class 2606 OID 16435)
-- Name: pk_results; Type: CONSTRAINT; Schema: public; Owner: postgres; Tablespace:
--

ALTER TABLE ONLY "Results"
    ADD CONSTRAINT pk_results PRIMARY KEY ("idStudent", "idTeacher", "idSubject", "NumSemestr");

--
-- TOC entry 1859 (class 2606 OID 16422)
-- Name: pk_sessions; Type: CONSTRAINT; Schema: public; Owner: postgres; Tablespace:
--

ALTER TABLE ONLY "Sessions"
    ADD CONSTRAINT pk_sessions PRIMARY KEY ("NumGroup", "NumSemestr", "idSubject", "idTeacher");

--
-- TOC entry 1861 (class 2606 OID 16430)
-- Name: pk_students; Type: CONSTRAINT; Schema: public; Owner: postgres; Tablespace:
--

ALTER TABLE ONLY "Students"
    ADD CONSTRAINT pk_students PRIMARY KEY ("idStudent");

--
-- TOC entry 1855 (class 2606 OID 16443)
-- Name: pk_subject; Type: CONSTRAINT; Schema: public; Owner: postgres; Tablespace:
--

ALTER TABLE ONLY "Subjects"
    ADD CONSTRAINT pk_subject PRIMARY KEY ("idSubject");

--
-- TOC entry 1857 (class 2606 OID 16445)
-- Name: pk_teacher; Type: CONSTRAINT; Schema: public; Owner: postgres; Tablespace:
--

ALTER TABLE ONLY "Teachers"
    ADD CONSTRAINT pk_teacher PRIMARY KEY ("idTeacher");

```

```

--
-- TOC entry 1864 (class 2606 OID 16446)
-- Name: fk_dep_tea; Type: FK CONSTRAINT; Schema: public; Owner: postgres
--

ALTER TABLE ONLY "Teachers"
    ADD CONSTRAINT fk_dep_tea FOREIGN KEY ("idDepartment") REFERENCES "Depart-
ments"("idDepartment");

--
-- TOC entry 1867 (class 2606 OID 16461)
-- Name: fk_res_stud; Type: FK CONSTRAINT; Schema: public; Owner: postgres
--

ALTER TABLE ONLY "Results"
    ADD CONSTRAINT fk_res_stud FOREIGN KEY ("idStudent") REFERENCES "Stu-
dents"("idStudent");

--
-- TOC entry 1868 (class 2606 OID 16466)
-- Name: fk_res_sub; Type: FK CONSTRAINT; Schema: public; Owner: postgres
--

ALTER TABLE ONLY "Results"
    ADD CONSTRAINT fk_res_sub FOREIGN KEY ("idSubject") REFERENCES "Sub-
jects"("idSubject");

--
-- TOC entry 1869 (class 2606 OID 16471)
-- Name: fk_res_tea; Type: FK CONSTRAINT; Schema: public; Owner: postgres
--

ALTER TABLE ONLY "Results"
    ADD CONSTRAINT fk_res_tea FOREIGN KEY ("idTeacher") REFERENCES "Teach-
ers"("idTeacher");

--
-- TOC entry 1865 (class 2606 OID 16451)
-- Name: fk_sess_subj; Type: FK CONSTRAINT; Schema: public; Owner: postgres
--

ALTER TABLE ONLY "Sessions"
    ADD CONSTRAINT fk_sess_subj FOREIGN KEY ("idSubject") REFERENCES "Sub-
jects"("idSubject");

--
-- TOC entry 1866 (class 2606 OID 16456)
-- Name: fk_sess_tea; Type: FK CONSTRAINT; Schema: public; Owner: postgres
--

ALTER TABLE ONLY "Sessions"
    ADD CONSTRAINT fk_sess_tea FOREIGN KEY ("idTeacher") REFERENCES "Teach-
ers"("idTeacher");

```

```
--
-- TOC entry 1990 (class 0 OID 0)
-- Dependencies: 5
-- Name: public; Type: ACL; Schema: -; Owner: postgres
--

REVOKE ALL ON SCHEMA public FROM PUBLIC;
REVOKE ALL ON SCHEMA public FROM postgres;
GRANT ALL ON SCHEMA public TO postgres;
GRANT ALL ON SCHEMA public TO PUBLIC;


-- Completed on 2014-08-04 23:54:32
--
-- PostgreSQL database dump complete
--
```

При внимательном рассмотрении трех сгенерированных скриптов видно, что основные команды по созданию таблиц и ограничений первичного и внешнего ключей почти не отличаются для этих трех СУБД. Дело в том, что язык SQL является стандартом для работы с базами данных и все современные реляционные базы данных стараются соблюдать этот стандарт. В основном, существенные различия заключаются только в используемых типах данных и небольших дополнениях, связанных с особенностями задания владельца таблицы (пользователя, который создал таблицу и, следовательно, имеет максимальные права для работы с ней), используемой кодировки и других настроек СУБД.

Нередко первоначальное проектирование выполняется с ошибками или недочетами (не все условия учтены, требуются новые столбцы или, напротив, какие-то столбцы являются лишними). Очевидно, что необходимы средства для обеспечения простого внесения изменений в таблицы. Этим средством является команда SQL **ALTER TABLE**, которую используют для корректировки списка столбцов таблицы и наложения разных ограничений как на отдельные столбцы, так и на таблицу в целом. Покажем на нескольких примерах, как можно использовать эту команду.

Выполнение SQL-команд осуществляется в оболочках с помощью специальных окон редактирования и выполнения SQL-скриптов. В dbForge Studio его можно создать с помощью меню «Новый»-> «SQL». В pgAdmin окно выполнения пользовательских запросов можно вызвать с помощью специальной кнопки



на панели инструментов . После создания ограничения можно увидеть как объекты соответствующих таблиц в дереве элементов базы данных (ограничения или индексы):

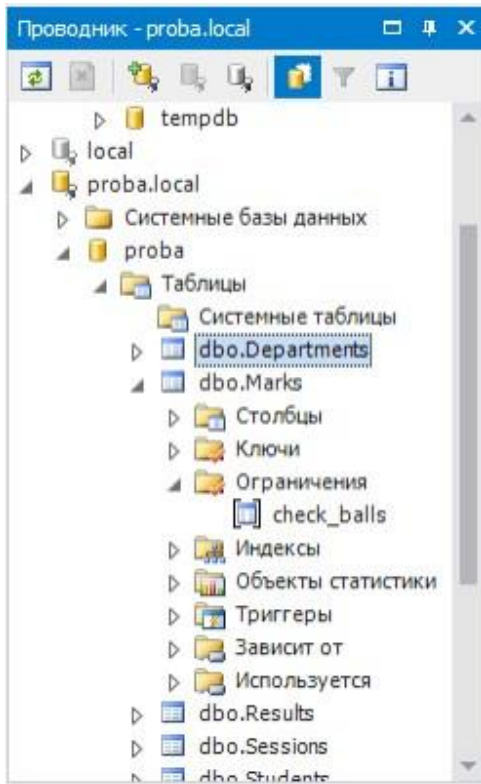


Рис. 19. Ограничение на проверку баллов в dbForge для SQL Server.

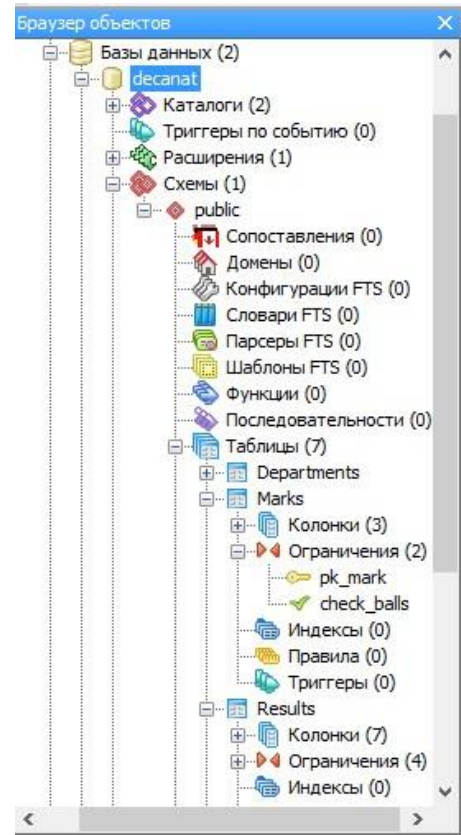


Рис.20. Ограничение на проверку баллов в pgAdmin.

Пример 1. Добавим ограничение уникальности на название кафедры в таблице Departments:

MS SQL Server, MySQL:

```
ALTER TABLE Departments ADD CONSTRAINT un_title
    UNIQUE (TitleDepartment);
```

PostgreSQL (отличием является заключением в кавычки имен таблиц, ограничения, столбцов):

```
ALTER TABLE "Departments" ADD CONSTRAINT "un_title"
    UNIQUE ("TitleDepartment");
```

Пример 2. Требуется добавить ограничение проверки условия для таблицы оценок, означающее, что нижняя граница баллов должна быть меньше верхней.

MS SQL Server:

```
ALTER TABLE Marks ADD CONSTRAINT check_balls  
CHECK (LowBalls<HighBalls);
```

Для MySQL эта команда выполняется, но как таковой объект базы данных не создается.

PostgreSQL:

```
ALTER TABLE "Marks" ADD CONSTRAINT "check_balls"  
CHECK ("LowBalls"<"HighBalls");
```

Пример 3. Требуется добавить ограничение проверки условия для поля телефона кафедры – телефон должен состоять из 7 цифр и иметь формат «xxx-xx-xx». Можно также, как и в предыдущем примере, добавить ограничение CHECK. Однако мы для демонстрации возможностей команды ALTER TABLE сначала удалим столбец телефона кафедры, а потом добавим новый столбец с учетом ограничения. В телефоне первая цифра 2 или 5, остальные цифры могут быть любыми. Для этого используется конструкция языка LIKE, задающая шаблон записи телефона (для PostgreSQL это конструкция SIMILAR TO):

MS SQL Server, MySQL:

```
ALTER TABLE Departments DROP COLUMN PhoneDepartment;  
ALTER TABLE Departments ADD PhoneDepartment VARCHAR(9) CHECK  
(PhoneDepartment LIKE '[2,5][0-9][0-9]-[0-9][0-9]-[0-9][0-9]');
```

PostgreSQL:

```
ALTER TABLE "Departments" DROP COLUMN "PhoneDepartment";  
ALTER TABLE "Departments" ADD "PhoneDepartment" text CHECK  
("PhoneDepartment" SIMILAR TO  
'(2|5)[0-9][0-9]-[0-9][0-9]-[0-9][0-9]');
```

Пример 4. Введем ограничение на согласованность баллов и оценки в таблице результатов сессии. Будем полагать для простоты, что в таблицу заносятся только положительные результаты сдачи зачетов и экзаменов. Таким об-

разом, оценка должна быть только 3, 4 или 5. При этом должна быть учтена согласованность баллов исходя из шкалы принятой балльно-рейтинговой системы:

#### MS SQL Server, MySQL:

```
ALTER TABLE Results ADD CONSTRAINT ch_res_marks CHECK (Mark IN (3,4,5) AND ((Mark=3 AND Balls BETWEEN 55 AND 70) OR (Mark=4 AND Balls BETWEEN 71 AND 85) OR (Mark=5 AND Balls BETWEEN 86 AND 100)));
```

#### PostgreSQL:

```
ALTER TABLE "Results" ADD CONSTRAINT "ch_res_marks" CHECK ("Mark" IN (3,4,5) AND (("Mark"=3 AND "Balls" BETWEEN 55 AND 70) OR ("Mark"=4 AND "Balls" BETWEEN 71 AND 85) OR ("Mark"=5 AND "Balls" BETWEEN 86 AND 100)));
```

Заметим, что для аналогичных целей была предназначена таблица Marks. В дальнейшем мы будем использовать ее.

Пример 5. В MySQL иногда требуется явно указать кодировку данных таблиц. Это также можно сделать с помощью команды ALTER TABLE:

```
ALTER TABLE `Departments` CONVERT TO CHARACTER SET utf8;
```

### 1.3. КОМАНДЫ МОДИФИКАЦИИ ДАННЫХ (DML)

Оболочки проектирования содержат средства визуального внесения данных в таблицы. Например, в dbForge Studio это делается с помощью контекстного меню таблицы, пункта «Редактировать таблицу», вкладка «Данные»:

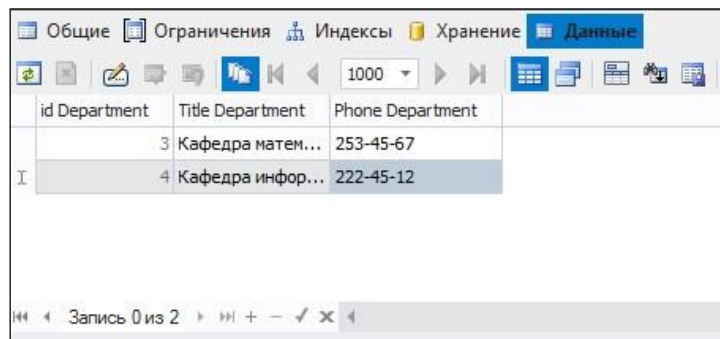


Рис. 21. Окно редактирования данных.

С помощью нижней панели можно добавлять, удалять записи и перемещаться по ним.

Сохранение данных происходит не сразу, а после закрытия окна. Но в процессе ввода данные проверяются на соответствие ограничениям. Например, внесем неправильный телефонный номер. В этом случае об ошибке будет сообщено следующим образом:

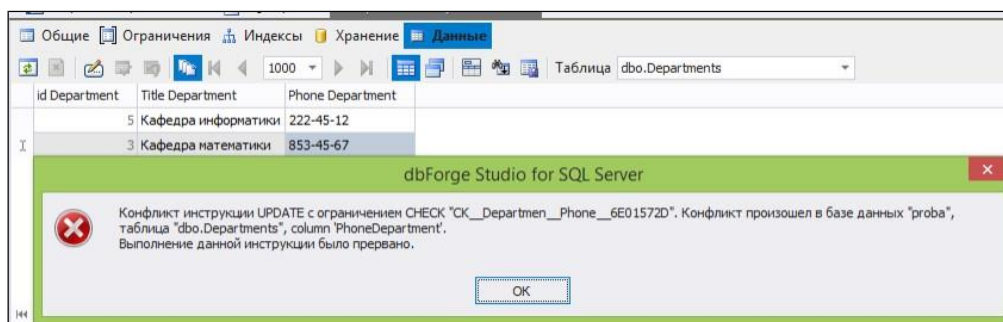


Рис. 22. Сообщение о нарушении ограничения.

Естественно, добавление записей (особенно массовое) может проводиться с помощью команд SQL, которые объединены в один сценарий. Итак, сценарий внесения данных в базу для MS SQL Server может быть таким:

```
USE proba
GO
```

```
-- вставка записей в таблицу Students
insert into Students (FIOSStudent,NumGroup)      VALUES      ('Иванов Иван
Иванович',901);
-- и другие записи
GO
```

```
-- вставка записей в таблицу Departments
insert into Departments (TitleDepartment,PhoneDepartment)  VALUES
('Кафедра математики','234-11-45');
-- и другие записи
GO
```

```
-- вставка записей в таблицу Teachers
insert into Teachers (FIOTeacher,idDepartment )  VALUES      ('Федосеев
Александр Иванович', 6);
-- и другие записи
GO
```

```
-- вставка записей в таблицу Subjects
insert into Subjects (TitleSubject) VALUES ('Математический анализ');
-- и другие записи
GO

-- вставка записей в таблицу Sessions
insert into Sessions (NumGroup, NumSemestr, Zach_Exam, idSubject, idTeacher)
VALUES (901, 1, 'зачет', 2, 18);
-- и другие записи
GO

-- вставка записей в таблицу Marks
insert into Marks (idMark, LowBalls, HighBalls) VALUES (5, 86, 100);
-- и другие записи
GO
```

Отметим, что свойство счетчика в MS SQL Server для поля фиксирует все операции с таблицей, поэтому, к примеру, несмотря на отсутствие записей в таблице, новый номер может быть отличен от 1. Оператор **GO**, который используется в скрипте, специфичен для MS SQL Server и разделяет скрипт на неделимые блоки, которые выполняются полностью или не выполняются вообще. Удобнее всего выполнять этот сценарий блоками, чтобы между ними можно было бы убедиться в корректности использования значений в полях внешнего ключа.

Команды вставки для MySQL не отличаются от приведенного выше кода (за исключением команды GO). Также внимательно следует относиться к ключевым полям при установке связи внешнего ключа.

Сценарий для PostgreSQL имеет следующий вид. Отсутствие полей-счетчиков требует указывать ключевые поля во всех записях.

```
-- сценарий вставки записей в таблицы базы данных
insert into "Students" ("idStudent", "FIOStudent", "NumGroup")
VALUES (1, 'Иванов Иван Иванович', 901);
-- и другие записи
```

При нарушениях тех или иных ограничений оператор вставки не срабатывает. Так, повторная вставка записи или вставка записи с уже существующим первичным ключом будет запрещена. Например, повторно осуществляем вставку записи:

```
insert into Marks (idMark, LowBalls, HighBalls) VALUES (5, 86, 100);
```

команда выполнена не будет. Мы увидим сообщение об ошибке:

#### MS SQL Server:

Ошибка: (53,1): Нарушено "PK\_Marks" ограничения PRIMARY KEY. Не удастся вставить повторяющийся ключ в объект "dbo.Marks". Повторяющееся значение ключа: (5).

#### PostgreSQL:

ОШИБКА: повторяющееся значение ключа нарушает ограничение уникальности "pk\_mark"

SQL-состояние: 23505

Подробности: Ключ "("idMark")=(5)" уже существует.

#### MySQL:

Duplicate entry '5' for key 1

При вставке записи с нарушением ограничения внешнего ключа также будет выведено сообщение об ошибке. Например, мы пытаемся вставить строку в таблицу сессии, в которой код преподавателя равен 100:

```
insert into Sessions (NumGroup, NumSemestr, Zach_Exam,
    idSubject,idTeacher) VALUES (905,1, 'экзамен', 3,100);
```

Сообщение об ошибке в этом случае будет таким:

#### MS SQL Server:

Ошибка: (53,63): Конфликт инструкции INSERT с ограничением FOREIGN KEY "FK\_Sessions\_Teachers\_idTeacher". Конфликт произошел в базе данных "proba", таблица "dbo.Teachers", column 'idTeacher'.

#### PostgreSQL:

ОШИБКА: INSERT или UPDATE в таблице "Sessions" нарушает ограничение внешнего ключа "fk\_sess\_tea"

SQL-состояние: 23503

Подробности: Ключ (idTeacher)=(100) отсутствует в таблице "Teachers".

#### MySQL:

Cannot add or update a child row: a foreign key constraint fails (`decanat/sessions`, CONSTRAINT `FK\_sessions\_teachers\_idTeacher` FOREIGN KEY (`idTeacher`) REFERENCES `teachers` (`idTeacher`) ON DELETE NO ACTION ON UPDATE NO ACTION)

## 1.4. ВЫБОРКА ДАННЫХ. ОПЕРАТОР SELECT (DQL)

Одной из основных задач, которую решают базы данных – это эффективный поиск необходимых данных. Универсальным подходом для решения этой задачи является применение специального оператора языка SQL - SELECT. Этот оператор достаточно сложный, имеет множество возможностей. Теоретической основой этого оператора является реляционная алгебра, которая доказывает возможность получения с помощью конечного набора операций любых возможных наборов данных.

Приведем несколько примеров запросов к таблицам базы данных, демонстрирующих различные возможности языка SQL и различные операции реляционной алгебры. Результаты будем показывать из различных вариантов реализации учебного проекта баз данных (при выполнении на различных СУБД).

Вспомним, что главным отличием синтаксиса команд SQL для PostgreSQL является заключение в кавычки имен таблиц, столбцов и пр. В следующих примерах будем приводить текст запроса в стиле MS SQL Server или MySQL. В случаях более серьезной разницы в записи запросов, будем приводить его текст для каждого СУБД в отдельности.

**Запрос 1. Операция проекции.** Осуществляется выбор только части полей таблицы, т.е. производится вертикальная выборка данных.

Распечатать ФИО всех студентов, зарегистрированных в базе данных:

```
SELECT FIOStudent FROM Students;
```

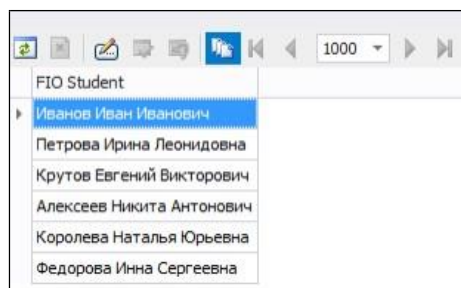
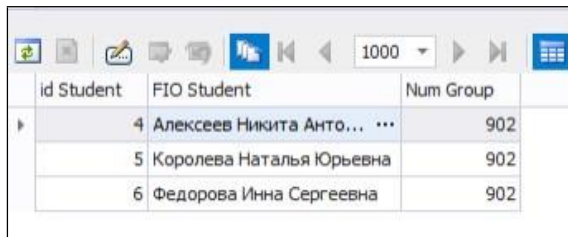


Рис. 23. Результат выполнения запроса для MS SQL Server.

**Запрос 2. Операция селекции.** Осуществляется горизонтальная выборка – в результат попадают только записи, удовлетворяющие условию.

Распечатать ФИО студентов группы 902.

```
SELECT * FROM Students WHERE NumGroup='902';
```



id Student	FIO Student	Num Group
4	Алексеев Никита Анто...	902
5	Королева Наталья Юрьевна	902
6	Федорова Инна Сергеевна	902

Рис. 24. Результат выполнения запроса для MySQL.

**Запрос 3. Операции соединения.** Здесь следует выделить декартово произведение и на его основе соединение по условию, а также естественное соединение (по одноименным полям или равенству полей с одинаковым смыслом).

Распечатать список зачетов и экзаменов, которые будут сдавать студенты группы 901 в первом семестре.

С помощью декартового произведения и соединения по условию данный запрос запишется так:

```
SELECT TitleSubject, Zach_Exam FROM Sessions, Subjects
WHERE Sessions.NumGroup='901' AND
Sessions.idSubject=Subjects.idSubject AND
Sessions.NumSemestr=1;
```

В этом запросе впервые мы выбираем информацию из нескольких таблиц. В случае использования одноименных полей следует дополнить их именами таблиц согласно схеме ИмяТаблицы.ИмяПоля. В случае же записи для СУБД PostgreSQL каждое из имен должно быть записано в кавычках:

```
SELECT "TitleSubject", "Zach_Exam" FROM "Sessions", "Subjects"
WHERE "Sessions"."NumGroup"='901' AND
"Sessions"."idSubject"="Subjects"."idSubject" AND
"Sessions"."NumSemestr"=1;
```



Панель вывода		
Вывод данных		Построить план выполнения
	TitleSubject text	Zach_Exam text
1	Математический анализ	экзамен
2	Алгебра и геометрия	зачет
3	Алгоритмизация	экзамен
4	Программирование	зачет

Рис. 25. Результат выполнения запроса для PostgreSQL.

Этот же запрос с помощью операции внутреннего соединения имеет следующий вид (соединение производится по равенству одноименных атрибутов idSubject таблиц Sessions и Subjects):

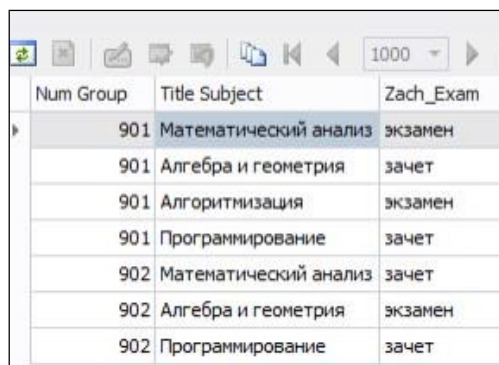
```
SELECT TitleSubject, Zach_Exam FROM Sessions INNER JOIN Subjects
ON Sessions.idSubject=Subjects.idSubject
WHERE Sessions.NumGroup='901' AND Sessions.NumSemestr=1;
```

**Запрос 4. Операция объединения.** Теоретико-множественные операции часто можно записать с помощью логических операций, примененных в конструкции WHERE запроса. Например, нужно получить список зачетов и экзаменов, которые сдают студенты 901 или 902 групп в 1 семестре. Таким образом, нужно объединить два множества, соответствующие двум разным группам. Объединение можно задать с помощью логического ИЛИ.

```
SELECT NumGroup, TitleSubject, Zach_Exam FROM
Sessions INNER JOIN Subjects
ON Sessions.idSubject=Subjects.idSubject
WHERE Sessions.NumSemestr=1 AND
(Sessions.NumGroup='901' OR Sessions.NumGroup='902');
```

Аналогичный результат будет получен с помощью объединения результатов двух запросов (подзапросов) с одинаковой структурой результата:

```
(SELECT NumGroup, TitleSubject, Zach_Exam FROM Sessions
INNER JOIN Subjects ON Sessions.idSubject=Subjects.idSubject
WHERE Sessions.NumSemestr=1 AND Sessions.NumGroup='901')
UNION
(SELECT NumGroup, TitleSubject, Zach_Exam FROM Sessions
INNER JOIN Subjects ON Sessions.idSubject=Subjects.idSubject
WHERE Sessions.NumSemestr=1 AND Sessions.NumGroup='902');
```



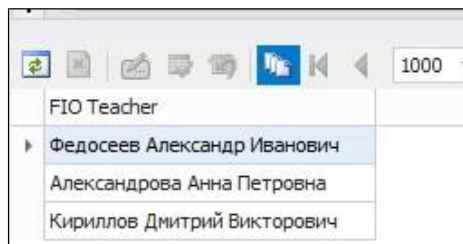
Num Group	Title Subject	Zach_Exam
901	Математический анализ	экзамен
901	Алгебра и геометрия	зачет
901	Алгоритмизация	экзамен
901	Программирование	зачет
902	Математический анализ	зачет
902	Алгебра и геометрия	экзамен
902	Программирование	зачет

Рис. 26. Результат выполнения запроса для MS SQL Server.

**Запрос 5. Операция пересечения.** В простых случаях эту операцию можно описать с помощью логической операции AND. В более сложных случаях эта операция определяется чаще всего с помощью подзапроса и ключевого слова EXISTS, которое показывает наличие похожего элемента во множестве, которое задается подзапросом.

Найти тех преподавателей, которым должны сдавать зачеты или экзамены в первом семестре студенты 901 и 902 групп. Отметим необходимость применения здесь операции переименования (AS) для того, чтобы различить два экземпляра таблицы Sessions (из основного запроса и подзапроса).

```
SELECT FIOTeacher FROM Teachers INNER JOIN Sessions
ON Teachers.idTeacher=Sessions.idTeacher
WHERE Sessions.NumSemestr=1 AND Sessions.NumGroup='901'
AND EXISTS (SELECT * FROM Sessions as s1
WHERE s1.idTeacher=Sessions.idTeacher AND s1.NumSemestr=1
AND s1.NumGroup='902');
```



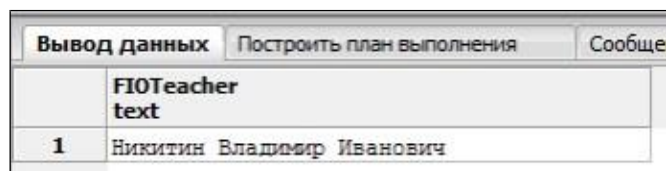
FIO Teacher
Федосеев Александр Иванович
Александрова Анна Петровна
Кириллов Дмитрий Викторович

Рис. 27. Результат выполнения запроса для MySQL.

**Запрос 6. Операция разности.** Эта операция также определяется часто с помощью подзапроса с ключевым словом NOT EXISTS, которое показывает отсутствие элемента во множестве, задаваемом подзапросом. Приведем аналогичный предыдущему пример.

Найти тех преподавателей, которым должны сдавать зачеты или экзамены в первом семестре студенты 901 группы, но не студенты из 902 группы.

```
SELECT FIOTeacher FROM Teachers INNER JOIN Sessions
ON Teachers.idTeacher=Sessions.idTeacher
WHERE Sessions.NumSemestr=1 AND Sessions.NumGroup='901'
AND NOT EXISTS (SELECT * FROM Sessions as s1
WHERE s1.idTeacher=Sessions.idTeacher AND s1.NumSemestr=1
AND s1.NumGroup='902');
```



	FIOTeacher text
1	Никитин Владимир Иванович

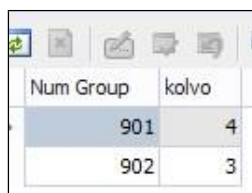
Рис. 28. Результат выполнения запроса для PostgreSQL.

Запрос 7. **Операция группировки.** Эта операция связана со своеобразной сверткой таблицы по полям группировки. Помимо полей группировки результат запроса может содержать итоговые агрегирующие функции по группам (COUNT, SUM, AVG, MAX, MIN).

Найти итоговое количество зачетов и экзаменов, которые должны сдавать студенты различных групп в 1 семестре.

Операция группировки здесь будет применяться к таблице **Sessions**. Поле группировки является номер группы. Агрегирующим полем является количество строк с заданной группой и номером семестра.

```
SELECT NumGroup, COUNT(*) AS kolvo FROM Sessions
WHERE NumSemestr=1 GROUP BY NumGroup;
```



Num Group	kolvo
901	4
902	3

Рис. 29. Результат выполнения запроса для MS SQL Server.

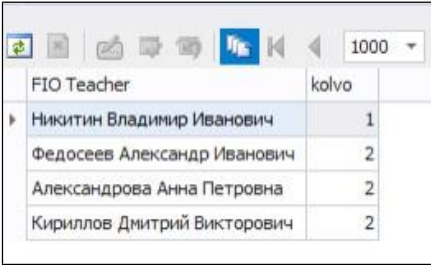
В случае, когда при группировке имеется условие отбора на группу, используется конструкция **HAVING**. Например, в предыдущем примере нам нужно вывести только те группы, в которых количество зачетов-экзаменов равно 3.

```
SELECT NumGroup, COUNT(*) AS kolvo FROM Sessions
WHERE NumSemestr=1 GROUP BY NumGroup HAVING COUNT(*)=3;
```

В СУБД MySQL возможно в условии в конструкции HAVING использовать псевдоним агрегирующего столбца kolvo.

**Запрос 8. Операция сортировки.** Вывести всех преподавателей, которым сдают студенты зачеты-экзамены в первом семестре, в порядке убывания количества зачетов-экзаменов. Для этого следует сначала выбрать нужные элементы таблицы Sessions, затем осуществить естественное соединение полученной таблицы с таблицей Teachers, после чего производится группировка записей в результате запроса и последующая сортировка.

```
SELECT FIOTeacher, COUNT(*) AS kolvo FROM sessions
INNER JOIN teachers
ON sessions.teachers_idTeacher=teachers.idTeacher
WHERE sessions.NumSemestr=1
GROUP BY FIOTeacher ORDER BY kolvo;
```



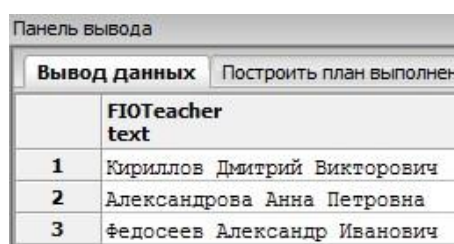
FIO Teacher	kolvo
Никитин Владимир Иванович	1
Федосеев Александр Иванович	2
Александрова Анна Петровна	2
Кириллов Дмитрий Викторович	2

Рис. 30. Результат выполнения запроса для MySQL.

**Запрос 9. Операция деления.** Это самая нетривиальная операция реляционной алгебры, которая обычно применяется тогда, когда требуется найти все записи первой таблицы, которые соединяются естественным образом со всеми записями второй таблицы. Например, нам требуется найти тех преподавателей, которым должны сдать в первом семестре зачеты-экзамены студенты всех групп факультета. Запрос получается достаточно сложный и он связан с выполнением двух операций разности (первая разность - из всевозможных комбинаций групп и преподавателей вычитаются реальные комбинации этих полей, т.е.

результатом становятся всевозможные нереальные пары, вторая разность – выбираются преподаватели, которые в нереальных парах не присутствуют).

```
SELECT FIOTeacher FROM Teachers WHERE idTeacher IN
  (SELECT DISTINCT s0.idTeacher FROM Sessions AS s0
   WHERE NumSemestr=1 AND
   NOT EXISTS (SELECT DISTINCT s1.idTeacher, s2.NumGroup
    FROM Sessions AS s1, Sessions AS s2
    WHERE s1.NumSemestr=1 AND s2.NumSemestr=1
    AND NOT EXISTS (SELECT * FROM Sessions
     AS s3 WHERE s3.idTeacher=s1.idTeacher AND
     s3.NumGroup=s2.NumGroup)
    AND s1.idTeacher=s0.idTeacher));
```



	FIOTeacher	text
1	Кириллов Дмитрий Викторович	
2	Александрова Анна Петровна	
3	Федосеев Александр Иванович	

Рис. 31. Результат выполнения запроса для PostgreSQL.

Разберем этот запрос по частям. Все возможные пары «преподаватель» - «группа» получаются с помощью подзапроса:

```
SELECT DISTINCT s1.idTeacher, s2.NumGroup
  FROM Sessions AS s1, Sessions AS s2
 WHERE s1.NumSemestr=1 AND s2.NumSemestr=1
```

добавлением к нему условия:

```
NOT EXISTS (SELECT * FROM Sessions AS s3
  WHERE s3.idTeacher=s1.idTeacher AND s3.NumGroup=s2.NumGroup)
```

из всевозможных пар вычитаются реальные пары, т.е. в результате получаем все возможные нереальные пары «преподаватель»-«группа». Результат этого подзапроса внедряется в другой подзапрос, получающий тех преподавателей, коды которых не присутствуют в этом списке. Далее подключением таблицы Teachers получаем их ФИО.

Наконец, рассмотрим возможность построения **представлений** – виртуальных таблиц, которые представимы как результат выполнения некоторого запроса. Представления в дальнейшем можно использовать в других запросах как таблицы, понимая при этом, что каждый раз при обращении к представлению производится выполнение запроса, по которому представление было создано.

Например, создадим представление, в котором находится информации о том, какие зачеты и экзамены должен сдать в первом семестре каждый студент.

```
CREATE VIEW Student_Session
AS
SELECT FIOStudent, TitleSubject FROM Students INNER JOIN Sessions
      ON Students.NumGroup=Sessions.NumGroup INNER JOIN Subjects
      ON Subjects.idSubject=Sessions.idSubject
WHERE NumSemestr=1;
```

Далее обратимся к этому представлению как к таблице. Например, найти те зачеты-экзамены, которые должен сдать студент Иванов:

```
SELECT TitleSubject FROM Student_Session
      WHERE FIOStudent LIKE 'Иванов%';
```

## 1.5. ХРАНИМЫЕ ПРОЦЕДУРЫ, ФУНКЦИИ И ТРИГГЕРЫ

Хранимые процедуры, функции и триггеры вводятся в базу данных для обеспечения бизнес-логики приложения на уровне серверной его компоненты. Обычно *хранимые процедуры и функции* представляют собой утилиты, которые определенным образом обрабатывают данные или реализуют достаточно сложный алгоритм вычисления некоторых показателей.

*Триггеры* – это частный случай хранимой процедуры, который выполняется автоматически при выполнении команд обновления данных (INSERT, DELETE, UPDATE). Триггеры привязываются к конкретным таблицам базы данных. Для каждой команды должны быть свои триггеры.

В дереве элементов базы данных в любом СУБД имеются группы для определения этих программных элементов:

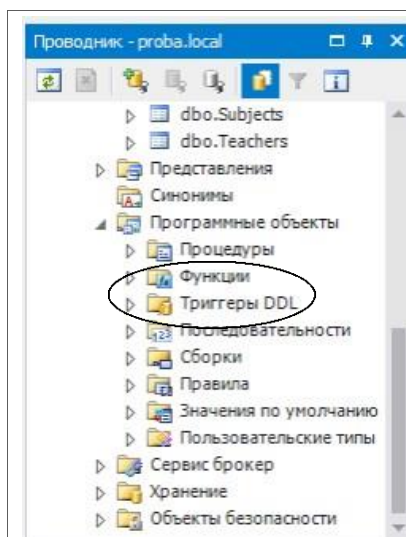


Рис. 32. Дерево элементов в MS SQL Server.

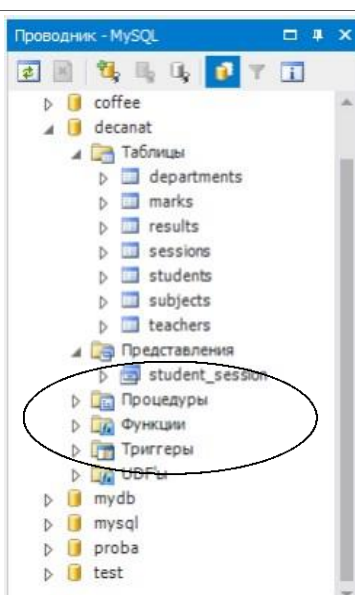


Рис. 33. Дерево элементов в MySQL.

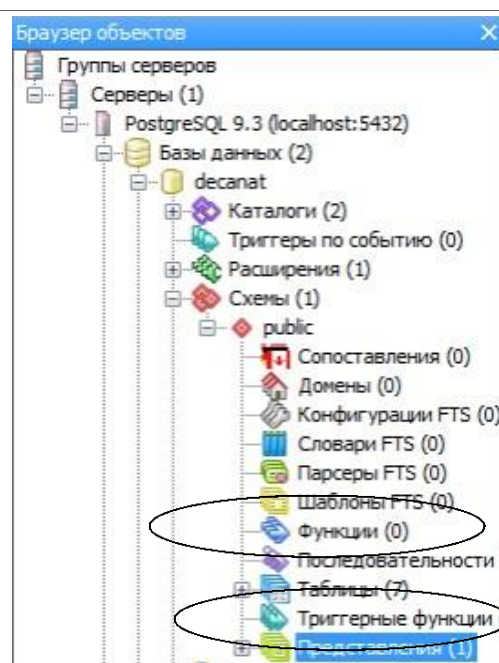


Рис. 34. Дерево элементов в PostgreSQL.

Для создания процедуры, функции или триггера требуется воспользоваться контекстным меню соответствующего элемента дерева. Для создания и редактирования существует специальное диалоговое окно, в котором, в частности, можно задать программный код процедуры, функции или триггера. Программный код формируется посредством перемешивания команд управления и SQL-команд.

Теперь приведем несколько примеров создания хранимых процедур и функций. Здесь мы уже заметим существенные отличия в синтаксисе используемых команд для различных СУБД. Поэтому для каждого из СУБД текст процедур, функций, триггеров и способы вызова укажем отдельно.

**Пример 1.** Напишем хранимую процедуру, которая получает в качестве входного параметра количество баллов и на основании шкалы оценок вычисляет полученную оценку. Результат возвращается через выходной параметр.

**MS SQL Server.** В SQL Server любая переменная именуется, начиная с символа '@'. Остальной код комментировать не требуется.

```
CREATE PROCEDURE dbo.GetMark1 (@ball int, @mark INT OUT)
AS
BEGIN
    IF @ball BETWEEN 55 AND 70
```



```

        SET @mark=3;
    ELSE IF @ball BETWEEN 71 AND 85
        SET @mark=4;
    ELSE IF @ball BETWEEN 86 AND 100
        SET @mark=5;
    ELSE SET @mark=2;
END
GO

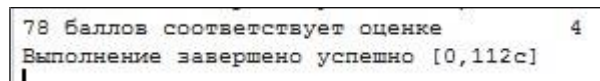
```

Для вызова процедуры требуется создать переменную для применения ее в качестве выходного параметра, после чего воспользоваться командой EXEC. Распечатать результат в выходном потоке можно с помощью оператора PRINT.

```

-- пример вызова процедуры GetMark1
DECLARE @mark INT;
EXEC GetMark1 78, @mark OUT;
PRINT '78 баллов соответствует оценке' + STR(@mark);

```



78 баллов соответствует оценке 4  
Выполнение завершено успешно [0,112с]

Рис. 35. Результат выполнения хранимой процедуры в MS SQL Server.

**MySQL.** Принципиальных отличий в программном коде нет. Стоит отметить, что символ '@' здесь используется только для глобальных переменных, поэтому имена параметров этот символ не имеют.

```

CREATE DEFINER = 'root'@'localhost'
PROCEDURE decanat.GetMark1(in ball INT, out mark INT)
BEGIN
    IF ball BETWEEN 55 AND 70 THEN
        SET mark=3;
    ELSEIF ball BETWEEN 71 AND 85 THEN
        SET mark=4;
    ELSEIF ball BETWEEN 86 AND 100 THEN
        SET mark=5;
    ELSE SET mark=2;
    END IF;
END

```

Вызов процедуры осуществляется следующим образом (достаточно отличным от MS SQL Server). В MySQL не предусмотрено окно сообщений, поэтому вывод осуществляется посредством выборки значения переменной:

```

call GetMark1(89,@m);
select ""+@m as "Оценка";

```



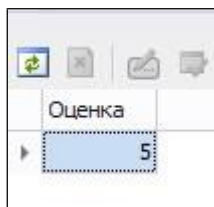


Рис. 36. Результат выполнения хранимой процедуры в MySQL.

**PostgreSQL.** Это СУБД не позволяет создавать процедуры. Здесь используются только функции. Еще одна особенность состоит в том, что функцию можно написать на разных языках. Наиболее распространены `sql` и `plpgsql`. Основное отличие языков состоит в том, что в `sql` доступны только операторы `sql`, а `plpgsql` имеет также операторы управления. Интересно, что именовать параметры вовсе не обязательно. К параметрам можно обращаться по номерам, предваренным символом “\$”. Итак, создадим скрипт, в котором запишем следующую функцию:

```
CREATE FUNCTION GetMark1 (integer) RETURNS integer AS $$
DECLARE res INTEGER;
BEGIN
    IF $1 BETWEEN 55 AND 70 THEN
        SELECT 3 INTO res;
    ELSE IF $1 BETWEEN 71 AND 85 THEN
        SELECT 4 INTO res;
    ELSE IF $1 BETWEEN 86 AND 100 THEN
        SELECT 5 INTO res;
    ELSE SELECT 2 INTO res;
    END IF; END IF; END IF;
    RETURN res;
END;
$$ LANGUAGE plpgsql;
```

Отметим применение символов “\$\$” в начале и конце функции. Они позволяют игнорировать символы-разделители внутри этих своеобразных скобок. Функция декларирует тип возвращаемого значения с помощью ключевого слова `RETURNS`. В теле функции создается переменная для хранения результата, которой присваивается значение в зависимости от ветки условных операторов, по которой пойдет управление. К параметру производится обращение посредством “\$1”. Отметим еще, что в конце следует указать используемый язык написания функции.

Вызов функции:

```
select GetMark1(68);
```

**Пример 2.** Чтобы при смене правил вычисления оценок не нужно было бы менять процедуру, мы создали справочную таблицу для хранения всех оценок и их диапазонов Marks. Пришло время ею воспользоваться. Второй вариант функции получения оценки по набранным баллам будет обращаться к этой таблице за информацией.

Оформим этот вариант в виде функции с одним параметром, хранящим набранные баллы, и возвращающую найденную оценку или 2 в случае, когда набранным баллам ничего в таблице не соответствует. В данной функции демонстрируется использование переменных, запросов и условного оператора. Приведем два варианта функции. С помощью запроса на количество таких записей алгоритм первого варианта предусматривает определение, есть ли в таблице соответствующая баллам оценка. Второй вариант пользуется специальной функцией EXISTS, которая, принимая в качестве аргумента запрос, возвращает логическое значение, определяющее, есть ли в результате запроса записи.

**MS SQL Server.** В предыдущем примере мы уже видели одно из отличий языка для MS SQL Server, связанное с присвоением значений переменным. В MS SQL Server для этого предназначен оператор SET. В других рассматриваемых нами СУБД в этой роли выступает оператор SELECT.

Итак, первый вариант функции. В нем определяются две переменные для хранения количества записей и оценки. Значением оценки по умолчанию является оценка 2. После определяется количество записей в таблице Marks, соответствующее набранным баллам и, если это количество больше 0, найденная оценка присваивается переменной @mark, которая в конце возвращается как результат функции. Отметим, что оператор RETURN должен быть последним в функции.

```
CREATE FUNCTION dbo.GetMark2(@ball int)
RETURNS INT
AS
BEGIN
    DECLARE @kolvo INT, @mark INT;
    SET @mark=2;
    SET @kolvo=(SELECT COUNT(*) FROM Marks WHERE
```

```

                                @ball between LowBalls and HighBalls);
IF @kolvo>0
    SET @mark=(SELECT idMark FROM Marks WHERE
                                @ball between LowBalls and HighBalls);
RETURN @mark;
END
GO

```

Во втором варианте функции в условном операторе вместо переменной @kolvo используется вызов функции EXISTS.

```

CREATE FUNCTION dbo.GetMark3(@ball int)
RETURNS INT
AS
BEGIN
    DECLARE @mark INT;
    SET @mark=2;
    IF EXISTS(SELECT * FROM Marks WHERE @ball
                                between LowBalls and HighBalls)
        SET @mark=(SELECT idMark FROM Marks WHERE
                                @ball between LowBalls and HighBalls);
    RETURN @mark;
END
GO

```

Вызов функции оформляется следующим образом:

```

DECLARE @mark INT;
SET @mark=dbo.GetMark3(93);
PRINT '93 балла соответствует оценке' + STR(@mark);

```

**MySQL.** Аналогичные функции для сервера MySQL определяются следующим образом:

```

CREATE FUNCTION decanat.GetMark2(ball int)
RETURNS int(11)
BEGIN
    DECLARE kolvo, mark INT;
    SELECT 2 INTO mark;
    SELECT COUNT(*) INTO kolvo FROM Marks WHERE
                                ball between LowBalls and HighBalls;
    IF kolvo>0 THEN
        SELECT idMark INTO mark FROM Marks WHERE
                                ball between LowBalls and HighBalls;
    END IF;
    RETURN mark;
END

```

```

CREATE FUNCTION decanat.GetMark3(ball int)
RETURNS int(11)
BEGIN
    DECLARE mark INT;
    SELECT 2 INTO mark;
    IF EXISTS (SELECT * FROM Marks WHERE
                                ball between LowBalls and HighBalls) THEN

```

```

        SELECT idMark INTO mark FROM Marks WHERE
                                ball between LowBalls and HighBalls;
    END IF;
    RETURN mark;
END

```

Вызов функции можно осуществлять непосредственно в выражении, например:

```
SELECT ""+GetMark2(89) as "Оценка";
```

**PostgreSQL.** Как уже было сказано, в PostgreSQL хранимых процедур нет, в этом СУБД используются только функции. Программный код практически не будет отличаться от кода для MySQL за исключением определения переменных за скобками функции, обращения к параметру и обращения к полям и таблице базы данных:

```

CREATE FUNCTION GetMark2 (integer) RETURNS integer AS $$
DECLARE kolvo INTEGER;
DECLARE mark INTEGER;
BEGIN
    SELECT 2 INTO mark;
    SELECT COUNT(*) INTO kolvo FROM "Marks"
                                WHERE $1 between "LowBalls" and "HighBalls";
    IF kolvo>0 THEN
        SELECT "idMark" INTO mark FROM "Marks"
                                WHERE $1 between "LowBalls" and "HighBalls";
    END IF;
    RETURN mark;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION GetMark3 (integer) RETURNS integer AS $$
DECLARE mark INTEGER;
BEGIN
    SELECT 2 INTO mark;
    IF EXISTS (SELECT * FROM "Marks"
                                WHERE $1 between "LowBalls" and "HighBalls") THEN
        SELECT "idMark" INTO mark FROM "Marks"
                                WHERE $1 between "LowBalls" and "High-
Balls";
    END IF;
    RETURN mark;
END;
$$ LANGUAGE plpgsql;

```

Теперь приведем примеры создания триггеров.

**Пример 1.** Создадим триггер для вставки в таблицу результатов сессии, в котором проверяются ограничения целостности (студент с заданным кодом существует, предмет с заданным кодом существует, дисциплину нужно сдавать

именно в этом семестре). Если произойдет нарушение этих ограничений, то требуется откатить транзакцию, т.е. не выполнять вставку записи. Если же все данные будут корректными, проведем заполнение значений полей даты сдачи зачета/экзамена как текущей и вычислим оценку по указанным баллам.

Для проверки корректности данных для вставки создадим вспомогательную хранимую функцию, чтобы код триггера был не очень сложным. (Для некоторых версий СУБД требуется, чтобы в триггере было упоминание только текущей записи, обращение к другим таблицам и записям недоступно).

### MS SQL Server:

```
CREATE FUNCTION dbo.IsCorrect (@idStud INT, @idSubj INT,
                               @Sem INT, @idTeach INT) RETURNS INT
AS
BEGIN
    IF EXISTS (SELECT * FROM Students INNER JOIN Sessions
               ON Students.NumGroup=Sessions.NumGroup
               INNER JOIN Subjects ON
               Sessions.idSubject=Subjects.idSubject
               INNER JOIN Teachers ON
               Sessions.idTeacher=Teachers.idTeacher
               WHERE Students.idStudent=@idStud AND
               Subjects.idSubject=@idSubj
               AND Teachers.idTeacher=@idTeach and NumSemestr=@Sem)
        RETURN 1;
    RETURN 0;
END
GO
```

### MySQL:

```
CREATE FUNCTION IsCorrect (idStud INT, idSubj INT, Sem INT, idTeach INT)
    RETURNS INT(11)
BEGIN
    RETURN EXISTS (SELECT * FROM Students INNER JOIN Sessions
                   ON Students.NumGroup=Sessions.NumGroup
                   INNER JOIN Subjects ON
                   Sessions.idSubject=Subjects.idSubject
                   INNER JOIN Teachers ON
                   Sessions.idTeacher=Teachers.idTeacher
                   WHERE Students.idStudent=idStud AND
                   Subjects.idSubject=idSubj
                   AND Teachers.idTeacher=idTeach and NumSemestr=Sem);
END
```

### PostgreSQL:

```
CREATE FUNCTION IsCorrect(integer, integer, integer, integer)
    RETURNS BOOLEAN AS $$
BEGIN
    RETURN EXISTS (SELECT * from "Students" INNER JOIN "Sessions"
                   ON "Students"."NumGroup"="Sessions"."NumGroup"
                   INNER JOIN "Subjects" ON
                   "Sessions"."idSubject"="Subjects"."idSubject"
```

```

INNER JOIN "Teachers" ON
"Sessions"."idTeacher"="Teachers"."idTeacher"
WHERE "Students"."idStudent"=$1 AND
"Subjects"."idSubject"=$2
AND "Teachers"."idTeacher"=$3 AND "NumSemestr"=$4);
END;
$$ LANGUAGE plpgsql;

```

Триггер на вставку записи в таблицу Results будет вызывать функцию проверки корректности, передавая в функцию поля из новой записи. Если запись будет корректной, будут скорректированы поля оценки и даты сдачи зачета/экзамена. В противном случае должен быть произведен откат транзакции.

### MS SQL Server:

При вставке записи сначала запись попадает в виртуальную таблицу inserted (при удалении будет использоваться таблица deleted, при изменении записи используются обе таблицы – в inserted хранятся новые значения записи, в deleted – прежние значения полей записи). Поэтому сначала получаем данные новой записи из таблицы inserted, после чего проверяем их на корректность. В случае корректных данных оставшиеся поля (дата и оценка) изменяются посредством команды UPDATE. Откат транзакции в случае некорректных данных производится с помощью команды ROLLBACK.

```

CREATE TRIGGER trigger1
ON dbo.Results
FOR INSERT
AS
BEGIN
-- объявление необходимых переменных для хранения данных новой записи
DECLARE @idStudent INT, @idSubject INT,
        @idTeacher INT, @NumSemestr INT, @Balls INT;

-- чтение данных новой записи
SET @idStudent =(SELECT idStudent FROM inserted);
SET @idSubject =(SELECT idSubject FROM inserted);
SET @idTeacher =(SELECT idTeacher FROM inserted);
SET @NumSemestr =(SELECT NumSemestr FROM inserted);
SET @Balls =(SELECT Balls FROM inserted);

-- проверка на корректность данных
IF dbo.IsCorrect(@idStudent, @idSubject, @NumSemestr, @idTeacher)=0
BEGIN
-- данные некорректны. Выводим сообщение об ошибке
-- и производим откат транзакции
PRINT 'Ошибка данных: данные некорректны';
ROLLBACK;
END
ELSE
-- изменение полей даты и вычисление оценки.

```

```

-- В условии указывается первичный ключ
UPDATE dbo.Results SET mark=dbo.GetMark3(@Balls), DateExam=GETDATE()
WHERE idStudent=@idStudent AND idSubject=@idSubject
AND idTeacher=@idTeacher AND NumSemestr=@NumSemestr;

END
GO

```

## MySQL:

Для MySQL данный триггер запишется проще, так как здесь проще получить данные новой записи. Новая запись хранится в виде объекта New (запись при удалении хранится в виде объекта Old). Однако имеется проблема, связанная с отсутствием команды отката триггера. В этом случае рекомендуется выполнить какую-нибудь ошибочную команду, например, вставить запись с уже существующим ключом. Ошибка в этой команде приведет к отмене действий всей транзакции (команды и триггера).

```

CREATE TRIGGER decanat.trigger1
BEFORE INSERT
ON decanat.results
FOR EACH ROW
BEGIN
IF IsCorrect(New.idStudent, New.idSubject, New.NumSemestr, New.idTeacher)
THEN
SET New.Mark=GetMark3(New.Balls);
SET New.DateExam=Now();
ELSE
insert into Departments values (1,"","");
END IF;
END

```

## PostgreSQL:

В PostgreSQL триггер как таковой связан со специальной триггерной функцией, в которой и осуществляется вся обработка данных. Триггерная функция возвращает объект-запись (NEW или OLD), с которой производится работа. При написании триггера мы указываем только для какой операции, для какой таблицы и каков тип триггера, после чего вызываем триггерную функцию. Откат производится генерацией исключительной ситуации с указанием сообщения об ошибке. В остальном код похож на тот, что писался для MySQL:

```

-- Создание триггерной функции на вставку результата сдачи экзамена
CREATE FUNCTION trigger_results_insert() RETURNS trigger AS $$ BEGIN
IF IsCorrect(NEW."idStudent", NEW."idSubject",
NEW."idTeacher", NEW."NumSemestr")
THEN
SELECT GetMark3(NEW."Balls") INTO NEW."Mark";

```

```

        SELECT Now() INTO New."DateExam";
    ELSE
        -- генерация исключительной ситуации
        RAISE EXCEPTION 'Ошибка корректности данных';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Создание триггера на вставку нового результата экзамена
CREATE TRIGGER tr_results_insert
BEFORE INSERT ON "Results" FOR EACH ROW
EXECUTE PROCEDURE trigger_results_insert();

```

Для проверки работы триггера (например, для MySQL) проведем следующие операции вставки:

```

INSERT INTO Results (idStudent, idSubject, idTeacher, NumSemestr, Balls)
VALUES (1,1,1,1,78);
INSERT INTO Results (idStudent, idSubject, idTeacher, NumSemestr, Balls)
VALUES (2,1,1,1,98);
INSERT INTO Results (idStudent, idSubject, idTeacher, NumSemestr, Balls)
VALUES (6,1,1,1,68);

```

Согласно данным, которые мы вносили в таблицу, последняя запись не должна быть добавлена.

**Пример 2.** Приведем еще один пример триггера на вставку новой записи в таблицу результатов. Этот триггер должен срабатывать после вставки и быть связан с подсчетом рейтинга студентов. Триггеры «после» часто используются для проведения специальной обработки данных на основании выполненной операции и могут быть связаны с другими таблицами.

Для этого введем в базу данных новую таблицу, например, с помощью следующей SQL-команды:

```

CREATE TABLE Reyting
( idStudent INT PRIMARY KEY,
  summ_balls INT,
  CONSTRAINT fk_reyting
    FOREIGN KEY (idStudent) REFERENCES Students (idStudent)
)

```

При вставке нового результата рейтинг студента должен меняться. Таким образом, нужно проанализировать, есть ли запись о студенте – в случае положительного ответа произвести суммирование баллов, иначе добавить новую запись в таблицу рейтинга.



## MS SQL Server:

```
CREATE TRIGGER trigger2
ON dbo.Results
AFTER INSERT
AS
BEGIN
    DECLARE @idStudent INT, @Balls INT;
    SET @idStudent = (SELECT idStudent FROM inserted);
    SET @Balls =(SELECT Balls FROM inserted);

    IF EXISTS(SELECT * FROM Reyting WHERE idStudent=@idStudent)
        UPDATE Reyting SET summ_balls=summ_balls+@Balls
                        WHERE idStudent=@idStudent;
    ELSE
        INSERT INTO Reyting (idStudent, summ_balls)
                        VALUES (@idStudent, @Balls);
END
GO
```

## MySQL:

```
CREATE TRIGGER decanat.trigger2
AFTER INSERT
ON decanat.results
FOR EACH ROW
BEGIN
    IF EXISTS(SELECT * FROM Reyting WHERE idStudent=new.idStudent) THEN
        UPDATE Reyting SET summ_balls=summ_balls+new.Balls
                        WHERE idStudent=new.idStudent;
    ELSE
        INSERT INTO Reyting (idStudent, summ_balls)
                        VALUES (New.idStudent, New.balls);
    END IF;
END
```

## PostgreSQL:

```
CREATE FUNCTION trigger_results_insert_after() RETURNS trigger AS $$
BEGIN
    IF EXISTS(SELECT * FROM "Reyting" WHERE "idStudent"=NEW."idStudent") THEN
        UPDATE "Reyting" SET "summ_balls"="summ_balls"+NEW."Balls"
                        WHERE "idStudent"=NEW."idStudent";
    ELSE
        INSERT INTO "Reyting" ("idStudent", "summ_balls")
                        VALUES (NEW."idStudent", NEW."Balls");
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER tr_results_insert_after
AFTER INSERT ON "Results" FOR EACH ROW
EXECUTE PROCEDURE trigger_results_insert_after();
```

Поэкспериментируйте сами со вставками записей, чтобы изменялся рейтинг студентов.

Разберем еще один пример хранимой функции для демонстрации использования **курсоров** – временных таблиц, представляющих собой результаты выполнения запроса и обрабатываемые построчно – от первой записи до последней. Для этого создадим еще одну версию функции перевода баллов в оценку – каждая строка таблицы Marks в ней будет обрабатываться построчно до получения строки с нужной оценкой или отсутствием соответствующей оценки.

### MS SQL Server:

```
CREATE FUNCTION dbo.GetMark4(@balls INT)
RETURNS INT
BEGIN
    DECLARE @res INT, @mark INT, @lowB INT, @highB INT;
    SET @res=2;

    -- декларация курсора, связанного с запросом
    DECLARE mark_cursor CURSOR FOR SELECT * FROM Marks;

    -- открытие курсора
    OPEN mark_cursor;
    -- считывание первой строки курсора в переменные @mark, @lowB, @highB
    FETCH NEXT FROM mark_cursor INTO @mark, @lowB, @highB;

    -- цикл продолжается, пока считывание возможно,
    -- на это укажет глобальная переменная
    WHILE @@FETCH_STATUS = 0
    BEGIN
        -- определяем, соответствуют ли баллы текущей оценке
        IF @balls BETWEEN @lowB AND @highB
        BEGIN
            SET @res=@mark;
            BREAK;
        END
        -- переход к следующей строке курсора
        FETCH NEXT FROM mark_cursor INTO @mark, @lowB, @highB;
    END

    -- закрытие курсора
    CLOSE mark_cursor;
    -- разрушение курсора
    DEALLOCATE mark_cursor;
    RETURN @res;
END
GO
```

**MySQL:** для обработки завершения записей курсора здесь требуется создать специальный обработчик CONTINUE HANDLER FOR NOT FOUND. В остальном работа с курсором аналогична.

```
CREATE DEFINER = 'root'@'localhost'
FUNCTION decanat.GetMark4(balls INT)
RETURNS int(11)
BEGIN
    -- переменные для хранения полей кортежа из таблицы Marks
    DECLARE mark, lowB, highB, res INT;
```

```

-- переменная для определения, завершен ли просмотр курсора
DECLARE is_end INT DEFAULT 0;
-- определение курсора для таблицы Marks
DECLARE mark_cursor CURSOR FOR SELECT * FROM Marks;

-- объявление обработчика ошибки завершения записей курсора
DECLARE CONTINUE HANDLER FOR NOT FOUND SET is_end=1;

SET res=2;
-- открытие курсора
OPEN mark_cursor;
-- считывание первой записи курсора
FETCH mark_cursor INTO mark, lowB, highB;

-- организация цикла просмотра строк из курсора
WHILE is_end=0 DO

    -- проверка диапазона баллов для текущей оценки
    IF balls BETWEEN lowB AND highB THEN
        SET res=mark;
        -- организация выхода из цикла
        SET is_end=1;
    END IF;
    -- считывание очередной записи курсора
    FETCH mark_cursor INTO mark, lowB, highB;
END WHILE;
CLOSE mark_cursor;
RETURN res;
END

```

**PostgreSQL:** как и в предыдущем случае отличия будут в организации цикла и выхода из него.

```

CREATE FUNCTION GetMark4 (integer) RETURNS integer AS $$
DECLARE res integer;
DECLARE mark integer;
DECLARE lowB integer;
DECLARE highB integer;
DECLARE mark_cursor CURSOR FOR SELECT * FROM "Marks";
BEGIN
    res:=2;
    OPEN mark_cursor;--открываем курсор
    LOOP --начинаем цикл по курсору
        --извлекаем данные из строки и записываем их в переменные
        FETCH mark_cursor INTO mark, lowB, highB;
        --если такого периода и не возникнет, то мы выходим
        IF NOT FOUND THEN EXIT;END IF;
        IF $1 BETWEEN lowB AND highB THEN
            res:=mark;
        END IF;
    END LOOP;--заканчиваем цикл по курсору
    CLOSE mark_cursor; --закрываем курсор
    return res;
END;
$$ LANGUAGE plpgsql;

```