МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ КЫРГЫЗСКОЙ РЕСПУБЛИКИ

КЫРГЫЗСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. И.Раззакова

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ КОМПЬЮТЕРНЫХ СИСТЕМ

РАЗРАБОТКА КЛИЕНТ-СЕРВЕРНЫХ ПРИЛОЖЕНИЙ MS SQL SERVER 2014 + MS ACCESS 2013

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ ПО ДИСЦИПЛИНЕ «БАЗЫ ДАННЫХ – 1»

Для студентов направления 710400 «Программная инженерия»

Бишкек 2014

«Рассмотрено»
На заседании кафедры
«Программное обеспечение
компьютерных систем»
Прот. №1 от 26.08.2014г.

«Одобрено»
Методическим советом
ФИТ
Прот. №2 от 29.09.2014г.

УДК 681.3.01

Составитель: доц. Раматов К.С.

Разработка клиент-серверных приложений SQL SERVER 2014+Access 2013: Методические указания к выполнению лабораторных работ по базам данных //КГТУ им. И. Раззакова; Сост.: / - Б.: ИЦ «Текник», 2015. - 48 с.

Представлены краткие теоретические сведения, примеры лабораторных заданий, методика выполнения и задания для лабораторных работ.

Предназначено для студентов направления «Программная инженерия» всех форм обучения.

Табл.4. Рис.3. Библиогр. 5 назван.

Рецензент:

Корректор ***Эркинбек к. Ж.***
Редактор ***Турдукулова А.К.***
Тех.редактор ***Кочоров А.Д.***

Подписано к печати 25.09.2015 г. Формат бумаги 60х84$^{1}/_{16}$.
Бумага офс. Печать офс. Объем 3 п.л. Тираж 50 экз. Заказ 200. Цена 51,3с.

Бишкек, ул. Сухомлинова, 20. ИЦ “Текник” КГТУ им. И.Раззакова, т.: 54-29-43
e-mail: beknur@mail.ru

ВВЕДЕНИЕ

Методические указания по выполнению лабораторных работ по дисциплине «Базы данных» ориентированы на студентов III курса. Они охватывают ряд вопросов по программе данной дисциплины.

Указания содержат:

- основные теоретические сведения;
- перечень обязательных заданий для выполнения лабораторных работ;
- указания к выполнению лабораторных заданий;

В методических указаниях приводятся наиболее полное представление основных понятий таких разделов курса «Базы данных» как “Таблицы”, “Представления”, “Ограничения целостности”, “Хранимые процедуры”, “Триггеры”, а также примеры написания кодов и решения задач с использованием языка Transact-SQL. Это позволит студентам как самостоятельно изучить теоретический материал по данным разделам, так и получить задание и решить задачи по лабораторным занятиям.

Методические указания могут быть полезны также студентам заочной и дистантной форм обучения.

Лабораторная работа №1
Создание базы данных, таблиц и диаграмм (4 часа)

Цель работы

Освоение навыков построения архитектуры и обеспечения целостности баз данных (БД) с помощью СУБД SQL SERVER 2014.

1. Теоретические сведения

В реляционной модели достигается гораздо более высокий уровень абстракции данных, чем в иерархической или сетевой. В статье Е.Ф.Кодда утверждается, что "*реляционная модель предоставляет средства описания данных на основе только их естественной структуры, т.е. без потребности введения какой-либо дополнительной структуры для целей машинного представления*". Другими словами, представление данных не зависит от способа их физической организации. Это обеспечивается за счет использования математической теории отношений (само название "*реляционная*" происходит от английского *relation* - "отношение").

Перейдем к рассмотрению структурной части реляционной модели данных. Прежде всего необходимо дать несколько определений.

Определения:

- Декартово произведение: Для заданных конечных множеств $D_1, D_2, ..., D_n$ (не обязательно различных) декартовым произведением $D_1 * D_2 * ... * D_n$ называется множество произведений вида $d_1 * d_2 * ... * d_n$, где $d_1 \in D_1, d_2 \in D_2, ..., d_n \in D_n$

Пример: *если даны два множества A (a_1,a_2,a_3) и B (b_1,b_2), их декартово произведение будет иметь вид C=A*B (a_1*b_1, a_2*b_1, a_3*b_1, a_1*b_2, a_2*b_2, a_3*b_2)*

- Отношение: Отношением *R*, определенным на множествах $D_1, D_2, ..., D_n$, называется подмножество декартова произведения $D_1 * D_2 * ... * D_n$. При этом:

  - множества $D_1, D_2, ..., D_n$ называются **доменами** отношения;
  - элементы декартова произведения $d_1 * d_2 * ... * d_n$ называются **кортежами**;
  - число *n* определяет **степень отношения** (*n=1* - унарное, *n=2* - бинарное, ..., *n*-арное);
  - количество кортежей называется **мощностью отношения**;

Пример: *на множестве C из предыдущего примера могут быть определены отношения R_1 (a_1*b_1, a_3*b_2) или R_2 (a_1*b_1, a_2*b_1, a_1*b_2)*

Отношения удобно представлять в виде таблиц. На рис. 1 представлена таблица (отношение степени 5), содержащая некоторые сведения о работниках гипотетического предприятия. Строки таблицы соответствуют кортежам. Каждая строка фактически представляет собой описание одного объекта предметной области (в данном случае работника), характеристики которого содержатся в столбцах. Можно провести аналогию между элементами реляционной модели данных и элементами модели "сущность-связь". Реляционные отношения соответствуют наборам сущностей, а кортежи - сущностям. Поэтому, также как и в модели "сущность-связь" столбцы в таблице, представляющей реляционное отношение, называют **атрибутами**.

Рис. 1 Основные компоненты реляционного отношения.

Каждый атрибут определен на домене, поэтому домен можно рассматривать как множество допустимых значений данного атрибута.

Несколько атрибутов одного отношения и даже атрибуты разных отношений могут быть определены на одном и том же домене. В примере, показанном на рис.1 атрибуты "Оклад" и "Премия" определены на домене "Деньги". Поэтому, понятие домена имеет семантическую нагрузку: данные можно считать сравнимыми только тогда, когда они относятся к одному домену. Таким образом, в рассматриваемом нами примере сравнение атрибутов "Табельный номер" и "Оклад" является семантически некорректным, хотя они и содержат данные одного типа.

Именованное множество пар "имя атрибута - имя домена" называется схемой **отношения**. Мощность этого множества - называют **степенью** или *"арностью"* отношения. Набор именованных схем отношений представляет из себя **схему базы данных.**

Атрибут, значение которого однозначно идентифицирует кортежи, называется **ключевым** (или просто **ключом**). В нашем случае ключом является атрибут "Табельный номер", поскольку его значение уникально для каждого ра-

ботника предприятия. Если кортежи идентифицируются только сцеплением значений нескольких атрибутов, то говорят, что отношение имеет составной ключ.

Отношение может содержать несколько ключей. Всегда один из ключей объявляется **первичным** (*primary key*, в дальнейшем будем обозначать ***PK***), его значения не могут обновляться. Все остальные ключи отношения называются возможными ключами.

Комплекс программных средств, осуществляющих управление БД, основанных на реляционной модели, называют системой управления реляционными БД (СУРБД),

Пример базы данных, содержащей сведения о подразделениях предприятия и работающих в них сотрудниках, применительно к реляционной модели будет иметь вид:

Рис. 2. База данных о подразделениях и сотрудниках предприятия.

Например, связь между отношениями ОТДЕЛ и СОТРУДНИК создается путем копирования первичного ключа *"Номер_отдела"* из первого отношения во второе. Таким образом:

- для того, чтобы получить список работников данного подразделения, необходимо
  1. из таблицы ОТДЕЛ установить значение атрибута *"Номер_отдела"*, соответствующее данному "Наименованию_отдела"
  2. выбрать из таблицы СОТРУДНИК все записи, значение атрибута *"Номер_отдела"* которых равно полученному на предыдущем шаге.
- для того, чтобы узнать в каком отделе работает сотрудник, нужно выполнить обратную операцию:
  1. определяем *"Номер_отдела"* из таблицы СОТРУДНИК
  2. по полученному значению находим запись в таблице ОТДЕЛ.

Атрибуты, представляющие собой копии ключей других отношений, называются **внешними ключами** (*foreign key*, в дальнейшем будем обозначать ***FK***).

Связи или отношения между таблицами (*relationships*) позволяют обеспечить целостность данных. Например, связь между таблицами ОТДЕЛ и СОТРУДНИК обеспечивает целостность поля *"Номер_отдела"* в СОТРУДНИКе (а именно, в таблицу СОТРУДНИК нельзя будет ввести номер отдела, которого нет в ОТДЕЛе). Кроме того, в СУРБД предусмотрены различные настройки каждой связи:

1) *Каскадное обновление связанных полей*. Эта настройка позволяет автоматически обновлять связанное поле в FK при изменении значения PK. Если данную настройку не применять, то система не позволит обновлять значения PK, пока не произвести соответствующих изменений в FK.

2) *Каскадное удаление связанных записей*. При установлении данного свойства связи удаление записей из таблицы с PK производится удаление соответствующих записей из таблицы FK. Аналогично, если эта настройка не установлена, то невозможно удаление записей с PK, пока существуют записи с FK.

Связи в СУРБД, как правило, имеют тип «один-ко-многим», т.е. для одной записи в таблице с PK соответствует несколько записей в таблице с FK.

Свойства отношений.

1. Отсутствие кортежей-дубликатов. *Из этого свойства вытекает наличие у каждого кортежа первичного ключа. Для каждого отношения, по крайней мере, полный набор его атрибутов является первичным ключом. Однако, при определении первичного ключа должно соблюдаться требование "минимальности", т.е. в него не должны входить те атрибуты, которые можно отбросить без ущерба для основного свойства первичного ключа - однозначно определять кортеж.*
2. Отсутствие упорядоченности кортежей.
3. Отсутствие упорядоченности атрибутов. Для ссылки на значение атрибута всегда используется имя атрибута.
4. Атомарность значений атрибутов, т.е. среди значений атрибута не могут содержаться множества значений (отношения)

В СУБД SQL Server, как и во многих других системах управления базами данных, отношение называется таблицей, кортеж записью и атрибут – полем.

В реляционной базе данных данные хранятся в базовых таблицах. В одной базе данных SQL Server их может быть до двух миллиардов. Для создания новой таблицы используется инструкция `CREATE TABLE` со следующими опциями:

- имя базы данных, которая будет содержать создаваемую таблицу;
- владелец таблицы;
- имя таблицы, которое должно быть уникальным среди имен базовых таблиц и представлений в этой базе данных, принадлежащих одному владельцу;
- спецификации от 1 до 1024 столбцов;
- ограничение первичного ключа (не обязательно);

- от 1 до 250 ограничений уникальности (не обязательно);
- от 1 до 253 ограничений внешнего ключа (не обязательно);
- одно (или более) ограничение на значения (CHECK), определяющее, какие данные могут быть добавлены в таблицу (не обязательно);
- группа файлов, в которой будет храниться таблица (не обязательно).

В инструкции *CREATE TABLE* задается имя создаваемой таблицы, а за ним в скобках следуют определения столбцов и ограничений, разделенных запятыми. У языка SQL довольно свободный формат, для удобства чтения можно разбивать инструкции на любое количество строк и вставлять пробелы между словами и символами.

Следующая инструкция создает базовую таблицу для хранения сведений о клиентах:

```
CREATE TABLE Customer
   (CustID       int           NOT NULL,
    Name         char    ( 30) NOT MULL,
    ShipLine     varchar(100),
    ShipCity     char    ( 30),
    ShipState    char    (  2),
    Status       char    (  1),
CONSTRAINT CustomerPk PRIMARY KEY (CustID))
```

2. Задание к работе

Создать базу данных **Students**, таблицы, а также диаграмму для обеспечения связей между созданными таблицами.

3. Порядок выполнения лабораторной работы

- ✓ Откройте программу Microsoft SQL Server 2014, утилиту SQL Server Management Studio (SSMS): ***Пуск > Программы > MS SQL Server > Среда SQL Server Management Studio.***
- ✓ Войдите в объект ***Базы данных***, правой кнопкой мыши выберите команду ***Создать базу данных***.
- ✓ В диалоговом окне задайте имя создаваемой базы данных (В столбце ***Путь*** можно указать локализацию базы данных, если хотите, чтобы она отличалась от принятой по умолчанию), после чего нажмите кнопку ***ОК***. В результате в структуре ***Базы данных*** появится новая база данных с указанным вами именем.
- ✓ Откройте созданную БД. В объекте ***Таблицы*** правой кнопкой мыши выберите команду ***Создать таблицу***. В появившемся окне конструктора таблицы задайте имена полей, типы данных и ограничения для таблицы **Regions** (ID (первичный ключ), Region). Отметим, что тип для первичного ключа желательно задавать целочисленный, причем использовать различные варианты, предоставляемые SQL Server (*tinyint* (1 байт), *smallint* (2 байта), *int* (4 байта), *bigint* (8 байтов). Для каждой таблицы выбирайте оптимальный

тип данных.

- ✓ Аналогично создайте таблицы **Nationalities**, **Faculties**, **Specialties**, **Groups**, **Subjects**, определив структуру таблиц в виде (связываемые поля должны иметь одинаковый тип):

№	Наименование таблицы	Поля
1	Nationalities	ID (первичный ключ), Nationality
2	Faculties	ID (первичный ключ), Faculty
3	Specialties	ID (первичный ключ), Specialty, Faculty (внешний ключ)
4	Groups	ID (первичный ключ), Group, Specialty (внешний ключ)
5	Subjects	ID (первичный ключ), Subject

- ✓ Используя инструкцию ***CREATE TABLE***, пример которой показан выше, создайте таблицы **Students**, **Teachers**, **ProgressInStudy** (успеваемость студентов), **Lectures** (проведение занятий):

№	Наименование таблицы	Поля
6	Students	ID (первичный ключ), Name, DateBorn, Region (внешний ключ), Nationality (внешний ключ), Group (внешний ключ)
7	Teachers	ID (первичный ключ), Name, Address
8	ProgressInStudy	ID (первичный ключ), Student (внешний ключ), Subject (внешний ключ), Term (семестр, условие на значение: между 1 и 10), Prize (балл, условие на значение: между 0 и 100)
9	Lectures	Teacher (внешний ключ), Subject (внешний ключ), Group (внешний ключ)

- ✓ В объекте ***Диаграммы базы данных*** с помощью правой кнопки мыши выберите команду *Создать диаграмму базы данных*.

- Следуя указаниям мастера создания диаграмм, вынесите на рабочее поле все созданные таблицы.
- Создайте связи между таблицами таким образом, чтобы схема базы данных выглядела как показано на рис. 3.

4. Контрольные вопросы:

4.1. Какими способами можно создать таблицу?
4.2. Какие бывают типы данных в SQL Server?
4.3. Как задать ограничения данных в таблицах?
4.4. Как задаются ключи таблицы?
4.5. Назовите типы связей таблицы.
4.6. Каким образом осуществляется целостность базы данных?

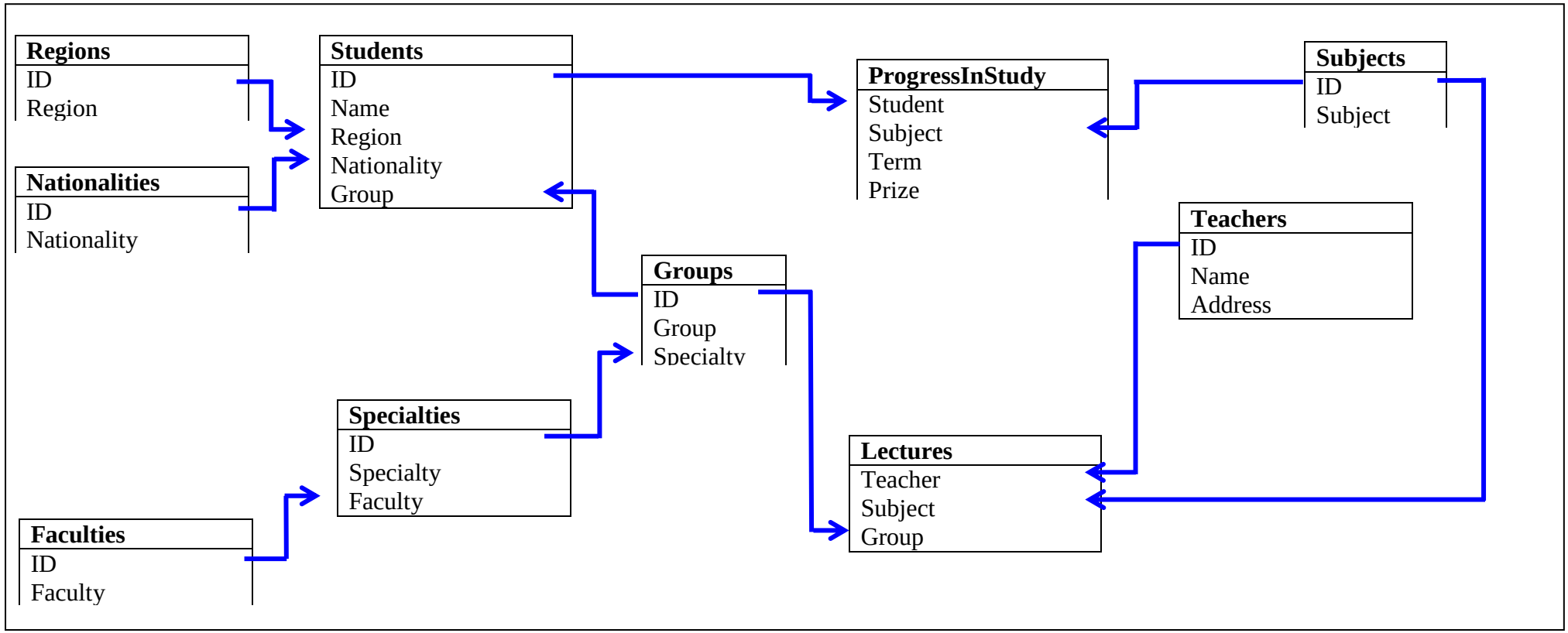

Рис. 3. Структура базы данных Students

Лабораторная работа №2
Создание представлений (Views) (4 часа)

Цель работы

Освоение навыков создания представлений в СУБД SQL SERVER 2014 для выборки данных.

1. Теоретические сведения

Согласно терминологии SQL *представление (view)* – это объект, который для пользователей и приложений, выполняющих запросы, почти ничем не отличается от таблицы. Представления можно использовать для просмотра и обновления данных, но на самом деле никаких данных они не содержат. Представление лишь предоставляет доступ к.данным одной или нескольких таблиц, на которых оно основано. Вот что можно делать с помощью представлений:

- отбирать из базовой таблицы подмножества строк;
- отбирать из базовой таблицы подмножества столбцов;
- формировать новые вычисляемые столбцы на основе одного или нескольких столбцов базовой таблицы;
- объединять связанные строки нескольких базовых таблиц в одну строку представления;
- объединять наборы строк нескольких базовых таблиц с помощью операции `UNION`.

Представление может с одной стороны упростить, а с другой ограничить доступ к данным. Вот как, например, создается представление, позволяющее просматривать информацию только о тех клиентах, чей лимит кредита не меньше $5000:

```
CREATE VIEW CuatHighCredit AS
    SELECT * FROM Customer WHERE CreditLimit >= 5000
```

Получив такую инструкцию, SQL Server создаст в каталоге базы данных определение представления `CustHighCredit`. После этого представление `CustHighCredit` можно будет использовать в SQL-инструкциях как самую обыкновенную таблицу. Например, можно выполнить вот такое обновление его строк:

```
UPDATE CustHighCredit
    SET Status = 'B' WHERE ShipCity = 'Seattle' AND Sta-
tus = 'X'
```

Однако эта инструкция не так проста, как кажется на первый взгляд. Во-первых, важно понимать, что она обновляет таблицу `Customer`, лежащую в

основе представления CustHighCredit. А во-вторых, она обновляет только строки, соответствующие трем условиям: CreditLimit >= 5000, ShipCity = 'Seattle' и Status = 'X'. Выполняя эту инструкцию, SQL Server обрабатывает только строки, удовлетворяющие критерию отбора, заданному в представлении CustHighCredit. Эти строки он проверяет на соответствие условиям, заданным в инструкции UPDATE, и выясняет, какие из них нужно обновить, а какие нет. Как видите, о представлении можно думать как о таблице, содержащей только те строки, которые соответствуют указанному в нем критерию.

Инструкция CREATE VIEW – это одна из самых сложных инструкций языка определения данных в SQL, и поэтому мы будем рассматривать ее по частям. После ключевых слов CREATE VIEW задается имя создаваемого представления, которое при желании можно уточнить именем его владельца. Имя базы данных в инструкции CREATE VIEW не задается, поскольку представления можно создавать только в текущей базе данных. Имена таблиц и представлений, на основе которых создается новое представление, можно уточнять и именами владельцев, и именами баз данных, эти объекты могут находиться и в других базах данных. Имя представления не может совпадать с именем другой .таблицы или представления, принадлежащих тому же владельцу и хранящихся в той же базе данных.

Примечание. Если не указать имя владельца создаваемого представления, то им станет текущий пользователь.

Следующей частью инструкции ***CREATE VIEW*** является необязательный список столбцов представления (их может быть до 1024). Если не задать список столбцов, как в предыдущем примере, в представление войдут столбцы, указанные в предложении SELECT. Предложение SELECT, по своей структуре подобное инструкции SELECT задается после ключевого слова AS. Оно определяет результирующую таблицу представления, т.е. столбцы и строки исходных таблиц и представлений, которые войдут в новое представление. Результирующая таблица представления — это абстрактная таблица, которой вовсе не обязательно должна соответствовать реальная таблица, хранящаяся на диске или в памяти. В следующем примере предложение SELECT определяет результирующую таблицу, содержащую все столбцы таблицы Customer и только те ее строки, в которых лимит кредита больше или равен $5000:

```
SELECT * FROM Customer WHERE CreditLimit >= 5000
```

Символ звездочки (*), следующий за ключевым словом SELECT, означает "все столбцы". SQL Server генерирует список столбцов представления при его создании, и если в дальнейшем в исходную таблицу будут добавлены новые столбцы, они не войдут в представление.

Выборка строк с помощью инструкции ***SELECT***

Инструкция SQL `SELECT` извлекает строки из одной или нескольких таблиц или представлений. Если ввести инструкцию `SELECT`, результаты ее выполнения будут выведены на панели `Results`. Эти результаты можно редактировать и печатать, а также сохранять в текстовом файле.

В инструкции `SELECT` задается список столбцов, содержащих нужные вам данные, имена таблиц и представлений, в которых они находятся, и критерий отбора строк. Еще в этой инструкции можно указать, как вы хотите сгруппировать и отсортировать данные. Дополнив вашу информацию сведениями из каталога, SQL получит всю необходимую информацию о том, какие данные вам требуются и как их получить. Вот как выглядит базовая структура инструкции `SELECT`:

```
SELECT        список-столбцов
  FROM        список-таблиц
  WHERE       условие-отбора
  GROUP BY    столбцы-для-группировки
  HAVING      условие-отбора
  ORDER BY    столбцы-для-сортировки
```

Инструкция `SELECT` — это один из самых мощных и гибких инструментов языка SQL, и овладение им является ключом к освоению всего SQL.

Арифметические операции с датами и временем

Выражения и скалярные функции часто используются для работы с датами и временем. Для пользователя SQL представляет даты в виде символьных строк, содержащих числовые или текстовые обозначения года, месяца и дня (иногда с разделителями). При вводе SQL-инструкции литеральное значение даты задается в виде строки. Ее формат зависит от языка (например, `us_english`) и формата даты, установленных с помощью инструкции `SET DATEFORMAT`. Например, следующая строка представляет 1 мая 2000 года в формате `ymd`:

```
'2000-05-01'
```

Та же дата в формате `mdy` вводится так:

```
'05-01-2000'
```

Вместо дефиса (-) в качестве разделителя может использоваться косая черта (/) или точка (.). Как правило, лучше пользоваться четырехзначным обозначением года. Если же вы пользуетесь двузначным обозначением, учтите, что

все значения, меньшие, чем установлено параметром `two digit year cut-off` относятся к тому же столетию, что и граничный год (обычно 21-му), а значения, равные или большие граничного года, относятся к предыдущему столетию (обычно 20-му). Граничным годом (`cutoff year`) SQL Server по умолчанию считает 2049 год.

SQL поддерживает сложение и вычитание дат с помощью операторов + и -, а также функций `DATEADD` и `DATEDIFF`. В следующем примере е помощью функции `DATEDIFF` вычисляется количество дней между `ShipDate` и `SaleDate` для определения задержки между оформлением заказа и отправкой товара:

```
SELECT CustID, OrderlD, SaleDate, ShipDate,
       DATEDIFF(Day, SaleDate, ShipDate) AS DaysToShip
  FROM Sale
 WHERE ShipDate IS NOT NULL
```

Вот как еще можно было бы составить выражение, дающее тот же результат:

```
SELECT CustID, OrderlD, SaleDate, ShipDate,
       CONVERT(Int, ShipDate - SaleDate) AS DaysToShip
  FROM Sale
 WHERE ShipDate IS NOT NULL
```

В этой инструкции для вычисления разности дат используется оператор минус (-), результатом которого является значение типа `DateTime`. Функция `CONVERT` преобразует это значение в количество дней. Без нее результатом выражения `ShipDate - SaleDate` для первой строки таблицы `Sale` (т.е. `15-05-2000 - 01-05-2000`) было бы: `15-01-1900  00:00:00.000`

Началу 1 января 1900 года SQL Server ставит в соответствие значение 0 и для каждого полного дня добавляет по 1.

Для добавления к дате определенного периода времени (или для его вычитания) используется функция `DATEADD`. У нее три аргумента: тип периода, вычитаемое или прибавляемое количество единиц и дата. В следующем примере к дате добавляется 30 дней:

```
ADDDATE (Day, 30, SaleDate)
```

Для вычитания периода из даты нужно просто задать во втором аргументе отрицательное число.

В приведенном выше примере инструкции `SELECT` показано, как с помощью ключевого слова `AS` задать имя столбца результирующей таблицы, который будет содержать результаты вычисления выражения:

```
DATEDIFF(Day, SaleDate, ShipDate) AS DaysToShip
```

С помощью того же ключевого слова AS можно задать другое имя (т.е. псевдоним) и для существующего столбца.

Примечание. Во избежание путаницы не стоит злоупотреблять возможностью задавать псевдонимы для существующих столбцов. Гораздо лучше, если во всех запросах и представлениях каждый столбец будет фигурировать под одним и тем же именем. Кроме того, учтите, что имя, заданное для столбца или выражения в списке SELECT, нельзя использовать в предложениях WHERE, GROUP BY и HAVING, поскольку в них могут фигурировать только столбцы таблиц, заданных в предложении FROM.

Если вы хотите, чтобы имя столбца содержало пробелы, заключите его в двойные кавычки:

```
DATEDIFF(Day, SaleDate, ShipDate) AS "Days To Ship"
```

Обратите внимание, что заданное в нашем примере условие отбора исключает из результирующей таблицы строки с пустым столбцом ShipDate. Мы должны учитывать, что в таблице Sale для этого столбца допускаются значения NULL, поскольку для арифметических выражений это очень важно (если хотя бы один из операндов арифметического выражения имеет значение NULL, все выражение также получает значение NULL). И это совершенно оправдано. Возьмем, например, наше выражение, вычисляющее разность двух дат: очевидно, что если одна из дат неизвестна, то неизвестна и их разность, т.е. она равна NULL. Условие отбора IS NOT NULL позволяет пропустить все строки, в которых дата отгрузки (ShipDate) неизвестна, поскольку товар еще не отгружен.

Примечание. Описанное правило сравнения столбца ShipDate со значением NULL основано на стандарте ANSI, которому соответствует уже упоминавшаяся установка ANSI_NULLS. Никогда не используйте операторы сравнения с ключевым словом NULL (например, COlA = NULL или COIA <> NULL), хотя Transact-SQL это и допускает. Результат сравнения любого значения с NULL согласно стандарту ANSI всегда неизвестен.

Литеральные значения времени и арифметические операции со значениями времени в Transact-SQL такие же, как и для дат. Например, символьная строка ' 13:30:10' в тексте SQL-инструкции означает 13 часов 30 минут 10 секунд.

Агрегатные функции

Скалярные функции обрабатывают значения из одной строки таблицы или объединения таблиц. В SQL определен и другой тип функций, предназначенных для вычислений на основе значений столбца в целом наборе строк. Такие функции называются *агрегатными (статистическими) функциями* (*aggre-*

gate function) или *функциями столбца (column function)*. Список всех агрегатных функций SQL приведен в таблице (Рис.4). С функциями AVG, CHECK-SUM_AGG, COUNT, MAX, MIN и SUM может использоваться ключевое слово DISTINCT, означающее, что перед вычислением значения функции из набора обрабатываемых ею данных должны быть исключены все повторяющиеся значения.

Таблица 4. Агрегатные функции SQL Server

Функция	Описание
AVG (выражение)	Среднее значение набора непустых значений выражения
CHECKSUM_AGG (выражение)	Контрольная сумма непустых значений выражения
COUNT(*) COUNT(выражение) COUNT(DISTINCT выражение)	COUNT(*) возвращает количество строк в заданном наборе строк. В этой форме инструкции COUNT ключевое слово DISTINCT использоваться не может. COUNT (выражение) возвращает количество непустых значений выражения. Если задано ключевое слово DISTINCT, функция COUNT возвращает количество уникальных непустых значений выражения. Все формы функции COUNT возвращают значение типа int
COUNT_BIG(*) COUNT_BIG(выражение) COUNT_BIG(DISTINCT выражение)	Функция COUNT_BIG подобна функции COUNT с той разницей, что COUNT_BIG возвращает значение типа bigint
GROUPING (столбец)	Возвращает 1, если агрегируемая строка добавлена оператором CUBE или ROLLUP, в противном случае возвращает 0
MAX (выражение)	Максимальное значение из всех непустых значений выражения
MIN(выражение)	Минимальное значение из всех непустых значения выражения
SUM(выражение)	Сумма всех непустых значений выражения
STDEV (выражение)	Среднеквадратичное отклонение непустых значений выражения
STDEVP(выражение)	Смещенное среднеквадратичное отклонение непустых значений выражения
VAR {выражение)	Дисперсия непустых значений выражения
VARP (выражение)	Смещенная дисперсия непустых значений выражения

Следующая инструкция SELECT возвращает одну строку с общим количеством клиентов и средней скидкой для них:

```
SELECT 'Average discount for', COUNT(*), ' customers is',
AVG(Discount) FROM Customer
```

Агрегатная функция COUNT(*) возвращает количество строк в исходной таблице, а агрегатная функция AVG(Discount) возвращает среднее значение набора непустых значений заданного столбца.

Со столбцами, допускающими значение NULL, агрегатные функции должны использоваться очень осторожно. Предположим, например, что столбец Discount допускает значения NULL и такое значение имеется, по меньшей мере, в одной строке таблицы Customer. В этом случае следующая инструкция вернет неожиданный результат:

```
SELECT COUNT(*), AVG(Discount), SUM(Discount)
  FROM Customer
```

Сумма окажется не равной средней скидке, помноженной на количество строк. Дело в том, что функция COUNT(*) сосчитает все строки исходной таблицы, а функции AVG и SUM проигнорируют те строки, в которых столбец Discount имеет значение NULL.

Чтобы избежать такого расхождения данных, можно воспользоваться альтернативной формой функции COUNT, которая тоже пропустит строки с пустым столбцом Discount:

```
SELECT COUNT(Discount), AVG(Discount), SUM(Discount )
  FROM Customer
```

Возможно также более универсальное и наглядное решение – вовсе исключить из обработки все строки с пустым столбцом Discount при помощи предложения WHERE:

```
SELECT COUNT(*), AVG(Discount), SUM( Discount )
  FROM Customer
 WHERE Discount IS NOT NULL
```

Когда в списке столбцов инструкции SELECT задается агрегатная функция, и при этом в инструкции отсутствует предложение GROUP BY (о котором рассказывается в следующем разделе), агрегатная функция обрабатывает весь набор строк, определяемый предложением WHERE, и результирующая таблица запроса всегда содержит только одну строку. Если же задать предложение GROUP BY, результирующая таблица будет содержать по одной строке на каждую группу исходных строк, или же, если задано еще и предложение HAVING, - по одной строке на каждую группу, удовлетворяющую условию HAVING.

Если агрегатная функция применяется к пустому набору строк (т.е. не содержащему ни одной строки), ее результатом будет NULL. Исключение составляет только функция COUNT(*), которая для пустого набора строк возвращает ноль.

Для удаления повторяющихся значений из набора значений, обрабатываемых агрегатной функцией, нужно сразу после ее открывающейся скобки поместить ключевое слово DISTINCT. Чаше всего этот параметр используется с функцией COUNT для подсчета различных значений заданного столбца:

```
SELECT COUNT(DISTINCT ShipCity) FROM Customer
```

Этот пример возвращает одну строку с единственным столбцом, содержащим количество городов, в которых у компании имеется хоть один клиент.

Предложение GROUP BY

Предложение GROUP BY используется для применения агрегатных функций к подгруппам строк, отобранных инструкцией SELECT. Например, следующая инструкция

```
SELECT ShipCity, COUNT(*) AS "Customer Count",
       AVG(Discount)    AS "Average Discount"
  FROM Customer
  GROUP BY ShipCity
```

возвращает по одной строке на каждую группу клиентов, проживающих в одном городе.

В этом примере ShipCity играет роль столбца группировки, по значениям которого строки таблицы Customer разделяются на группы — отдельная группа для каждого значения столбца ShipCity. Агрегатные функции COUNT и AVG по очереди применяются к каждой группе, так что в результирующей таблице оказывается по одной строке на каждую группу. Если столбец группировки допускает пустые значения, для значения NULL тоже создается отдельная группа.

Примечание. В данном примере намеренно не исключены строки с пустыми значениями столбца Discount, чтобы и результирующей таблице было указано полное количество клиентов, проживающих в каждом городе. При этом средняя скидка вычисляется только на основе строк с непустыми значениями столбца Discount.

В предложении GROUP BY указывается список столбцов и скалярных выражений. Например, в следующей инструкции SELECT создается отдельная группа для каждой комбинации вычисляемого столбца DaysToShip и столбца SaleDate:

```
SELECT SaleDate,
      DATEDIFF(Day, SaleDate, ShipDate) AS DaysToShip,
       AVG(TotalAmt) AS DaysToShip
  FROM Sale
  WHERE ShipDate IS NOT NULL
  GROUP BY  SaleDate, DATEDIFF(Day, SaleDate, ShipDate)
```

Обычно столбцы и выражения группировки включаются и в список SELECT, чтобы каждая строка результирующей таблицы содержала значения, идентифицирующие группу. Остальные столбцы, указанные в списке SELECT, должны быть аргументами агрегатных функций.

Если для отбора строк используется предложение WHERE, оно может исключить строки с некоторым значением столбца (или столбцов) группировки. Если в группу попала ни одна строка, SQL Server не генерирует для нее строку результирующей таблицы. Таким образом, следующая инструкция

```
SELECT ShipCity, COUNT(*) AS "Customer Count",
  FROM Customer
  WHERE Discount > .01
  GROUP BY ShipCity
```

сгенерирует таблицу, не имеющую строк с данными для Albany и Seattle

Для того чтобы получить строку для каждой группы, даже если она окажется пустой, поместите за ключевыми словами GROUP BY ключевое слово ALL:

```
SELECT ShipCity, COUNT(*) AS "Customer Count",
  FROM Customer
  WHERE Discount > .01
  GROUP BY ALL ShipCity
```

Предложение HAVING

Предложение HAVING используется для отбора строк результирующей таблицы уже после применения к сгруппированным строкам агрегатных функций. Оно похоже на предложение WHERE, служащее для отбора строк перед группировкой. Например, следующая инструкция SELECT возвращает данные о городах, в которых средняя скидка клиентам превысила один процент:

```
SELECT    ShipCity, COUNT(*)   AS  "Customer Count",
         AVG(Discount)         AS  "Average Discount"
  FROM    Customer
  WHERE   Discount IS NOT NULL
  GROUP BY ShipCity
  HAVING   AVG(Discount) > .01
```

Условие отбора, заданное в предложении HAVING, может содержать столбцы группировки, как например ShipCity, или агрегатные функции, как например AVG(Discount)..

Примечание. Хотя предложение HAVING может быть задано и без предложения GROUP BY, так поступают редко.

Предложение ORDER BY

Предложение ORDER BY служит для сортировки результирующей таблицы инструкции SELECT перед ее возвращением приложению. В этом предложении задается список столбцов и порядок сортировки значений каждого из этих столбцов: по возрастанию (задается ключевым словом ASC и подразумевается по умолчанию) или по убыванию (задается ключевым словом DESC).

Примечание. Предложение ORDER BY не является частью предложения SELECT, включаемого в другие инструкции, оно используется только в инструкции SELECT. Его можно включать в инструкции SELECT, INSERT, но не в определения представлений.

Следующая инструкция возвращает информацию о заказах на уже отгруженные товары, отсортированную по кодам клиентов, а для каждого клиента еще и по количеству дней, прошедшему от момента оформления заказа до времени отгрузки:

```
SELECT   CustId, OrderId, SaleDate, ShipDate,
         DATEDIFF(Day, SaleDate, ShipDate)
FROM     Sale
WHERE    ShipDate IS NOT NULL
ORDER BY Custld, 5 DESC
```

В этом примере безымянный пятый столбец, получающийся путем вычисления выражения DATEDIFF(Day, SaleDate, ShipDate), используется для сортировки по убыванию строк с одинаковыми значениями столбца Custld.

Чтобы сделать эту инструкцию более понятной, можно определить для вычисляемого столбца псевдоним-и использовать его в предложении ORDER BY:

```
SELECT   CustId, OrderId, SaleDate, ShipDate,
         DATEDIFF(Day, SaleDate, ShipDate) AS DaysToShip
FROM     Sale
WHERE    ShipDate IS NOT NULL
ORDER BY Custld,
         DaysToShip DESC
```

2. Задание к работе

Создать представления с помощью мастера SQL Server и языка Transact-SQL для выборки данных.

3. Порядок выполнения лабораторной работы

- В ***обозревателе объектов*** разверните базу данных, в которой необходимо создать новое представление.
- Щелкните правой кнопкой папку ***Представления*** и выберите ***Создать представление...***
- В диалоговом окне ***Добавить таблицу*** выберите один или несколько элементов, которые необходимо включить в новое представление, на вкладку «***Таблицы***».
- Щелкните ***Добавить***, из предложенного списка выберите таблицы **Students, Regions, Nationalities, Faculties, Specialties, Groups,** а затем выберите ***Закрыть***.
- На ***Панели диаграмм*** выберите столбцы таблиц, которые будут участвовать в представлении, а также задайте сортировку выборки в следующем порядке:

№	Table	Column	Sort Type	Sort Order
1	Faculties	Faculty	ascending	1
2	Specialties	Specialty	ascending	2
3	Groups	Group	ascending	3
4	Students	Name	ascending	4
5	Nationalities	Nationality		
6	Regions	Region		

- Добавьте еще одно поле **Age**, которое будет вычислять возраст студента с использованием функций обработки дат, показанных выше.
- Назовите данное представление **ListOfStudents**.
- Следующее представление создайте с помощью T-SQL.
- Используя инструкцию *Select*, наберите код, который создаст представление **CountOfStudents**, выбирающее количество студентов для каждой группы каждой специальности по всем факультетам.
- Третье представление **CountByStudentsAges** можно создать с использованием мастера представления на основе уже созданного представления **ListOfStudents**. Для этого в мастере представления выберите это представление, отметьте столбцы ***Faculties***, ***Specialties***, ***Groups***, ***Age***, задайте группировку по выбранным полям и вычисляемое значение количества студентов данного возраста по каждой группе. Внимательно ознакомьтесь с кодом на Transact-SQL, который выполняет каждое из созданных представлений.

4. Контрольные вопросы:

4.1. Как создается представление с помощью мастера представлений?
4.2. Как создается представление с помощью языка ***T-SQL***?
4.3. Какую структуру имеет инструкция ***Select***?
4.4. Что значит инструкция ***Group by*** и как она выполняется?
4.5. Как применяются агрегатные функции?

Лабораторная работа №3
Создание клиентского приложения к базе данных (4 часа)

Цель работы

Освоение навыков использования инструментов Microsoft Access 2013 для создания интерфейса пользователя к построенной базе данных в СУБД SQL SERVER 2014.

1. Теоретические сведения

Форма — это объект БД, предназначенный для ввода и отображения информации. Формы позволяют выполнить проверку корректности данных при вводе, проводить вычисления, обеспечивают доступ к данным в связанных таблицах с помощью подчиненных форм.

Работа с формами может происходить в трех режимах: в режиме *Формы*, в режиме *Таблицы*, в режиме *Конструктора*. Выбрать режим работы можно при помощи кнопки *Вид* панели инструментов, *Конструктор форм* либо с помощью команды меню Вид.

В режимах *Формы* и *Таблицы* можно осуществлять добавление, удаление и редактирование записей в таблице или запросе, являющемся источником данных для форм.

В режиме *Конструктор* можно производить изменение внешнего вида формы, добавление и удаление элементов управления, разработку.

Виды форм. В Access можно создать формы следующих видов:

- форма в столбец или полноэкранная форма;
- ленточная форма;
- табличная форма;
- форма главная/подчиненная;
- сводная таблица;
- форма-диаграмма.

Форма в столбец представляет собой совокупность определенным образом расположенных полей ввода с соответствующими им метками и элементами управления. Чаще всего эта форма используется для ввода и редактирования данных.

Ленточная форма служит для отображения полей группы записей. Поля не обязательно располагаются в виде таблицы, однако для одного поля отводится столбец, а метки поля располагаются как заголовки столбцов.

Табличная форма отображает данные в режиме таблицы.

Форма главная/подчиненная представляет собой совокупность формы в столбец и табличной. Ее имеет смысл создавать при работе со связанными *таблицами, в которых установлена связь типа один-ко-многим.*

Форма *Сводная таблица* выполняется мастером создания сводных таблиц Excel на основе таблиц и запросов Access (мастер сводных таблиц является объектом, внедренным в Access, чтобы использовать его в Access, необходимо установить Excel). *Сводная таблица* представляет собой перекрестную таблицу данных, в которой итоговые данные располагаются на пересечении строк и столбцов с текущими значениями параметров.

Форма с диаграммой. В Access в форму можно вставить диаграмму, созданную Microsoft Graph. Graph является внедряемым OLE-приложением и может быть запущен из Access. С внедренной диаграммой можно работать так же, как и с любым объектом OLE.

Структуры формы. Любая форма может включать следующие разделы:

заголовок формы — определяет верхнюю часть формы и может содержать текст, графику и другие элементы управления;

верхний колонтитул — раздел отображается только в режиме предварительного просмотра и обычно содержит заголовки столбцов;

область данных — определяет основную часть формы, содержащую поля, полученные из источника данных;

нижний колонтитул — раздел отображается только в режиме предварительного просмотра в нижней части экранной страницы и обычно содержит номер страницы, дату и т. д.;

примечание формы — отображается внизу последней экранной страницы формы.

Форма может содержать все разделы или некоторые из них.

Как и любой объект базы данных, форма имеет свойства. Значения этих свойств для всей формы, ее разделов или элементов управления задаются в окнах свойств соответствующих объектов. Для отображения на экране окна свойств нужно нажать кнопку *Свойства* на панели инструментов *Конструктор форм*.

Окно свойств выделенного объекта содержит следующие вкладки:

Макет — с помощью этих свойств задается макет формы;

Данные — с помощью этих свойств задается источник данных;

События — содержит перечень свойств, связанных с объектом;

Другие — перечень остальных свойств;

Все — перечень всех свойств.

Основные свойства формы:

- ✓ *подпись* — позволяет задать название формы, которое будет выводиться в области заголовка;
- ✓ *режим по умолчанию* — определяет режим открытия формы (простая, ленточная, табличная формы);
- ✓ *допустимые режимы* — свойство, которое определяет, можно ли с помощью команд меню ***Вид*** переходить из режима формы в режим конструктора;
- ✓ свойства: *полосы прокрутки, область выделения, кнопки перехода, разделительные линии, кнопка оконного меню, размеров окна, кнопка закрытия, кнопка контекстной справки, тип границы* — определяют, будут ли выводиться эти элементы в окно формы;

- ✓ *свойства разрешить добавления, удаления, изменения* — определяют, можно ли пользователю редактировать данные через форму. Эти свойства могут принимать значения Да/Нет;
- ✓ *ввод данных* — определяет режим открытия формы и принимает значения Да/Нет. Режим Да — открытие формы только для добавления новых записей. Режим Нет — просмотр существующих записей и добавление новых;

блокировка записей — определяет способы блокировки записей в режиме многопользовательской работы с базой данных.

Для создания форм в Access используются следующие виды.

Автоформа — автоматизированное средство для создания форм трех стандартных типов: в столбец, ленточная, табличная. При этом в форму вставляются все поля источника данных.

Мастер форм — программное средство, которое позволяет создавать структуру одного из трех стандартных типов формы в режиме диалога с разработчиком формы. При этом в форму вставляются выбранные пользователем поля из источника данных.

Конструктор форм — позволяет конструировать форму пользователем в окне конструктора форм.

Самым удобным способом создания новой формы является следующая технология: форма создается с использованием *автоформы* или *мастером форм,* а затем дорабатывается в режиме конструктора.

Источником данных формы являются одна или несколько связанных таблиц и/или запросов.

Элементом управления называют любой объект формы или отчета, который служит для вывода данных на экран, оформления или выполнения макрокоманд. Элементы управления могут быть связанными, вычисляемыми или свободными.

Связанный (присоединенный) элемент управления присоединен к полю базовой таблицы или запроса. При вводе значения в связанный элемент управления поле таблицы текущей записи автоматически обновляется. Поле таблицы является источником данных связанного элемента управления.

Вычисляемый элемент управления создается на основе выражений. В выражениях могут использоваться данные полей таблицы или запроса, данные другого элемента управления формы или отчета и функции.

Свободные элементы управления предназначены для вывода на экран данных, линий, прямоугольников и рисунков. Свободные элементы управления называют также переменными или переменными памяти.

Все элементы управления могут быть добавлены в форму или отчет с помощью панели инструментов элементов управления, которая появляется при работе с формой или отчетом.

2. Задание к работе

Создать базу данных на Microsoft Access 2013 и установить связь к объектам на SQL SERVER базе данных, создать в ней формы и отчеты для обновления каждой из таблиц, отображения данных на основе представлений.

3. Порядок выполнения лабораторной работы

- Откройте программу ***Microsoft Access 2013***.
- Щелкните на кнопке «*Создать*», в открывшейся форме выберите «*Новая база данных*», задайте путь и наименование БД в форме «*Имя файла*» и нажмите на кнопку «*Создать*».
- В открывшейся основной форме Access войдите во вкладку «*Внешние данные*», щелкните на «*Базы данных ODBC*».
- В открывшейся форме «*Внешние данные – Базы данных ODBC*» поставьте галочку на варианте «*Создать связанную таблицу для связи с источником данных*» и нажмите на «*Ок*».
- В форме «*Выбор источника данных*» выберите вкладку «*Источник данных компьютера*» и щелкните на кнопке «*Создать*».
- В появившемся окне «*Создание нового источника данных*» выберите вариант «*пользовательский*» и нажмите на кнопке «*Далее*».
- Из предложенного списка драйверов выберите «*SQL Server*» и щелкните на кнопке «*Далее*».
- В следующей форме нажмите на кнопке «*Готово*».
- В окне о сведениях создаваемого источника введите имя создаваемого источника, из списка серверов выберите имя сервера, на котором создана БД SQL Server и нажмите «*Готово*».
- В появившейся информационной форме о созданном источнике щелкните на «*Ok*».
- В результате увидите, что созданный источник данных внесен в список всех источников. Убедитесь, что созданный источник выделен и нажмите «*Ok*».
- В следующей форме отметьте все таблицы и представления, которые созданы в БД SQL Server и нажмите «*Ok*».
- Отвечая положительно на запросы относительно каждого объекта, убедитесь, что выбранные объекты БД появились в таблицах Access как связанные таблицы.
- Откройте страницу «***Формы***». Используя мастер или конструктор, создайте формы для каждой таблицы и каждого представления.
- Для форм, источником которых являются таблицы с внешними ключами, необходимо создать поля со списками для тех значений, которые ссылаются на другие таблицы. К примеру, если источником формы «**Специальности**» является таблица **Specialties**, то поле **Faculty** должно принимать значения из таблицы **Faculties.** Поэтому откройте форму «**Специальности**».
- На элементе управления **Faculty** щелкните правой кнопкой мыши и выберите *Преобразовать элемент в / Поле со списком*.
- Настройте полученное поле со списком. Нажмите правую кнопку мыши на нем и выберите *Свойства*.
- В наборе вкладок свойств элемента войдите во вкладку «***Данные***». В графе *Источник строк* задайте таблицу **Faculties**.
- Откройте вкладку «***Макет***». В графе *Количество столбцов* введите «2» (по-

скольку в таблице **Faculties** используется два поля). В графе *Ширина столбцов* введите, к примеру «0;3», т.е. первое поле **ID** будет невидимым, для поля **Faculty** отводится 3 см.

- Откройте вкладку «***Данные***». В графе *Присоединенный столбец* введите «1». Это говорит о том, что из двух полей таблицы **Faculties** будет выбираться значение первого поля **ID** для поля **Faculty** таблицы **Specialties**. Откройте форму «**Специальности**» в режиме формы и ознакомьтесь с работой созданного поля со списком.
- Аналогично преобразуйте все другие формы, имеющие источником таблицы с внешними ключами.
- Создайте отчеты для отображения данных, получаемых при запуске представлений. Эти отчеты должны иметь вид ленточных форм, группироваться по факультетам, специальностям и группам и открываться нажатием соответствующих кнопок в главной кнопочной форме.
- Создайте главную кнопочную форму, в которой поместите кнопки для открытия каждой из форм и отчетов.
- В меню MS Access войдите в ***Файл*** / ***Параметры***. Выберите вариант «*Текущая база данных»,* задайте заголовок приложения, которое будет отражаться при открытии программы, а также выберите главную кнопочную форму в окне «*Форма просмотра*». После этого при запуске проекта будет автоматически открываться главная кнопочная форма.
- Откройте созданный интерфейс пользователя и заполните данными все созданные формы.

4. Контрольные вопросы:

4.1. Как создаются связи с объектами SQL Server из клиентского приложения в MS Access?
4.2. Какими способами можно создать форму?
4.3. Что такое поле со списком и как он создается?
4.4. Чем отличается работа с объектами MS Access от работы с объектами SQL Server?

Лабораторная работа №4
Создание хранимых процедур (4 часа)

Цель работы

Освоение навыков создания хранимых процедур в СУБД SQL SERVER 2014 для выборки данных и демонстрация результатов их работы в клиентском приложении.

1. Теоретические сведения

Хранимая процедура (`stored procedure`) — это несколько последовательных инструкций Transact-SQL, которые во время ее создания компилируются в специальный формат (план выполнения). Хранимые процедуры — это мощный и гибкий инструмент, использующийся для реализации различных функций администрирования базой данных, управления и обработки данных, например, для создания таблиц, предоставления или изменения базы данных. Хотя сам язык SQL является непроцедурным языком, его диалект Transact-SQL для SQL Server содержит несколько дополнительных возможностей, включающих использование ключевых слов управления потоком данных. Это позволяет применять в хранимых процедурах SQL Server сложную логику обработки данных и решать с их помощью, самые разнообразные задачи.

После компиляции хранимой процедуры SQL Server оптимизирует план выполнения так, чтобы хранимая процедура выполнялась наилучшим образом. Такая оптимизация обеспечивает высокую эффективность выполнения хранимых процедур. Хранимые процедуры могут возвращать значения параметров, наборы данных, коды или создавать курсоры, Одна хранимая процедура может быть доступна многим пользователей. Хранимые процедуры могут принимать до 1024 параметров и выполняться как на локальных, так и на удаленных системах SQL Server.

Для создания хранимой процедуры используется инструкция `CREATE PROCEDURE`, а для выполнения хранимой процедуры — инструкция `EXECUTE` или соответствующая функция используемого приложением программного интерфейса для доступа к SQL Server (такого, как OLE DB). В хранимых процедурах могут использоваться любые SQL-инструкции, за исключением `CREATE DEFAULT`, `CREATE RULE`, `CREATE TRIGGER` и `CREATE VIEW`.

Хранимые процедуры очень похожи на обыкновенные процедуры языков высокого уровня, у них могут быть входные и выходные параметры и локальные переменные, в них могут производиться числовые вычисления и операции над символьными данными, результаты которых могут присваиваться переменным и параметрам. Кроме того, в хранимых процедурах могут выполняться стандартные операции с базами данных (как DDL, так и DML). И особенно важно то, что в хранимых процедурах возможны циклы и ветвления, т.е. в них могут использоваться инструкции управления потоком.

Хранимая процедура всегда создается в текущей базе данных. Вот простейший пример инструкции, создающей хранимую процедуру:

```
CREATE PROCEDURE ListCustWithDiscount @MinDiscount
   DEC(5,3) AS  SELECT   *
                FROM     Customer
                WHERE    Discount >= @MinDiscount
```

А вот как эта процедура выполняется:

```
EXECUTE ListCustWithDiscount   .1
```

Примечание. В случае если хранимая процедура выполняется с помощью командной строки и ее вызов является единственной инструкцией пакета, ключевое слово `EXECUTE` можно опустить**.**

Максимальная глубина вложенности хранимых процедур (т.е. длина цепочки их вызовов) составляет 32. Текущий уровень вложенности можно узнать с помощью встроенной функции SQL Server `@@NESTLEVEL`.

В определении хранимой процедуры за ключевыми словами `CREATE PROCEDURE` следует имя процедуры и определения ее параметров. Ключевое слово `AS` указывает начало тела процедуры, состоящего из одной или более SQL-инструкций. Подобно функциям в языках высокого уровня, процедура может (но не обязательно должна) возвращать целочисленное значение. Кроме того, она может возвращать данные вызывающей процедуре или приложению через определенные вами выходные параметры.

▼ Примечание. Transact-SQL позволяет идентифицировать процедуру по имени и номеру (например, `ListCust; 1`). Так можно определить несколько процедур с одинаковыми именами и разными номерами. Если в определении процедуры задан ее номер, то этот номер нужно указывать в ее вызове. С помощью одной инструкции `DROP PROCEDURE` можно удалить все процедуры с одинаковыми именами и разными номерами. Однако мы не рекомендуем пользоваться возможностью нумерации процедур и советуем назначать им уникальные и информативные имена без номеров.

Изменение и удаление хранимых процедур

Инструкция `ALTER PROCEDURE` позволяет изменить код хранимой процедуры, сохранив все назначенные ей разрешения. Синтаксис этой инструкции подобен синтаксису инструкции `CREATE PROCEDURE`, они различаются лишь первым ключевым словом. Например, следующая инструкция переопределяет процедуру

```
ALTER PROCEDURE ListCustWithDiscount @MinDiscount DEC(5,3)
        AS   SELECT   *
                  FROM     Customer
                  WHERE    Status = 'Active'
                           AND Discount >= @MinDiscount
```

Для переименования хранимой процедуры используется системная хранимая процедура sp_rename, принимающая три аргумента: старое имя, новое имя и тип объекта (для хранимой процедуры типом объекта является 'object', его можно не указывать). Например, следующая инструкция переименовывает процедуру ListCustWithDiscount:

```
sp_rename 'ListCustWithDiscount',
'ListCustonerWithDiscount', 'object'
```

При первом запуске процедуры SQL Server ее компилирует и оптимизирует. Кроме того, он автоматически компилирует и оптимизирует процедуру, когда изменяются таблицы, которые она использует. Однако это не касается добавления в таблицу нового индекса, и если вы хотите, чтобы процедура использовала этот индекс, откомпилируйте ее сами с помощью системной хранимой процедуры sp_recompile:

```
sp_recompile 'ListCustWithDiscount'
```

Эта инструкция не вызывает немедленную компиляцию процедуры ListCustWithDiscount, а только помечает ее как требующую компиляции, чтобы SQL Server сделал это при следующем вызове процедуры.

Для удаления хранимой процедуры используется инструкция DROP PROCEDURE:

```
DROP PROCEDURE ListCustWithDiscount
```

Параметры хранимой процедуры

У хранимой процедуры может быть до 1024 параметров. Определение каждого параметра имеет такую форму:

```
@имя-параметра тип-данных
```

Имя параметра должно начинаться с символа @, за которым могут следовать любые буквы Unicode, цифры и символы @, $, # и _. Никогда не начитайте имена параметров с символов @@, поскольку этот префикс SQL Server использует для некоторых своих встроенных функций. Для параметров допускаются те же типы данных, что и для столбцов таблиц, а также особый тип данных cursor, о котором мы поговорим позднее. Если у процедуры несколько параметров, разделите их определения запятыми.

Для параметра можно задать значение по умолчанию, которое будет присваиваться ему в том случае, если значение этого параметра не задано в вызове процедуры. Вот как можно модифицировать предыдущий пример, чтобы по умолчанию процедура ListCustWithDiscount возвращала все строки со скидкой, большей 0,001:

```
CREATE PROCEDURE ListCustWithDiscount
    @MinDiscount DEC(5,3) = 0.001
AS  SELECT   *
    FROM     Customer
    WHERE    Discount >= @MinDiscount
```

Эту процедуру можно вызвать вовсе без аргументов, что будет равнозначно вызову ее с аргументом 0,001:

```
    EXECUTE ListCustWithDiscount
```

По умолчанию все параметры хранимой процедуры являются входными. Для того чтобы определить выходной параметр, нужно добавить после его имени, типа и значения по умолчанию ключевое слово OUTPUT, как в следующем примере:

```
CREATE PROCEDURE GetCustDiscount @CustId INT,
    @Discount DEC(5,3) OUTPUT
AS  SET @Discount =
(SELECT Discount FROM Customer WHERE Custld = @CustId)
```

Обратите внимание на то, как в этом примере используется скалярный запрос: возвращаемое им значение присваивается выходному параметру процедуры @Discount. Когда процедура завершает свою работу, вызывающей процедуре или приложению возвращаются последние значения, присвоенные ее выходным параметрам. Для параметров хранимых процедур допускаются значения NULL. Это значит, что и в процедуру, и из процедуры можно передавать значения NULL.

В вызове процедуры, имеющей выходной параметр, должна быть задана переменная, которой будет присвоено его значение. Эта переменная задается на месте выходного параметра и сопровождается ключевым словом OUTPUT:

```
EXECUTE GetCustDiscount 123789, @CustDiscount OUTPUT
```

Вместо аргумента, для которого в процедуре определено значение по умолчанию, в инструкции EXECUTE можно использовать ключевое слово DEFAULT. В конце списка аргументов такие аргументы можно просто опустить.

Примечание. Поскольку параметры, для которых определены значения по

умолчанию, являются необязательными, лучше всего поместить их в конец списка параметров. Так удобнее будет опускать соответствующие аргументы в вызовах процедуры.

Существует альтернативный и часто более удобный способ задания аргументов вызываемой хранимой процедуры. Его иллюстрирует следующий пример:

```
EXECUTE GetCustDiscount @CustId = 123789,
    @Discount = @CustDiscount OUTPUT
```

Используя имена параметров, можно задавать аргументы в любом порядке, не привязываясь к их порядку в определении процедуры. Например, следующий вызов будет абсолютно правильным:

```
EXECUTE GetCustDiscount
    @Discount = @CustDiscount OUTPUT, @CustId = 123789
```

Transact-SQL допускает даже смешанный способ вызова параметров: первые несколько аргументов можно задать позиционно, а остальные в любом порядке по именам. Но учтите, что вслед за параметром, заданным по имени, позиционно задать остальные параметры невозможно, их тоже нужно будет задать по именам.

Результирующие наборы строк

Хранимая процедура возвращает отдельный набор строк для каждой входящей в нее инструкции SELECT, которая не является скалярным вложенным запросом (т.е. не используется вместо одного значения) и значения столбцов которой не присваиваются переменным или параметрам. Например, следующая хранимая процедура возвращает два набора строк:

```
CREATE PROCEDURE ListLowHighDiscCust AS
    SELECT *    FROM Customer    WHERE Discount < .01
    SELECT *    FROM Customer    WHERE Discount > .1
```

Возвращаемый код состояния

В хранимой процедуре может использоваться инструкция RETURN, прекращающая выполнение процедуры и возвращающая заданное в этой инструкции значение, которое обычно интерпретируется как код состояния:

```
CREATE PROCEDURE ListCustWithDiscount
    @MinDiscount DEC(5,3) = 0.001
AS  IF (@MinDiscount > 1.0) RETURN (1)
    SELECT *
```

```
  FROM Customer
 WHERE Discount >= @MinDiscount
RETURN (0)
```

Возвращаемое значение не является обязательным. Им может быть любое целочисленное выражение. Для его получения в вызывающей процедуре используется оператор присваивания:

```
EXECUTE @Status = ListCustWithDiscount 0.1
```

А можно так:

```
EXECUTE @Status = ListCustWithDiscount @MinDiscount=0.1
```

Примечание. Для своих системных процедур SQL Server интерпретирует значение `0` как успешный вызов, а значения от `-1` до `-99` как системные коды ошибок (например, ошибка преобразования данных). В пользовательских процедурах можно тоже использовать значение `0` как индикатор успешного вызова, но не стоит возвращать значения от `-1` до `-99`.

2. Задание к работе

Создать хранимые процедуры и показать в клиентском приложении результаты их выполнения.

3. Порядок выполнения лабораторной работы

- Создайте хранимую процедуру **SP_Specialty** для выборки специальностей выбранного факультета с использованием среды SQL Server Management Studio. Для этого:
- Последовательно разверните узел ***Базы данных***, свою базу данных и узел *Программирование*.
- Щелкните правой кнопкой мыши элемент *Хранимые процедуры* и выберите пункт *Создать хранимую процедуру*.
- Удалите весь предлагаемый код и в меню *Запрос* выберите пункт *Создать запрос в редакторе*.
- Аналогично действиям при создании представлений с помощью построителя выберите поля таблицы **Specialties**, для поля **Faculty** задайте фильтр @fac и нажмите *Ok*.
- В появившемся окне кода создаваемой хранимой процедуры добавьте описание входного параметра ***@fac***, задав ему целочисленный тип.
- Для проверки синтаксиса выберите пункт *Начать отладку* в меню *Отладка*. Если возвращается сообщение об ошибке, сравните инструкции с приведенными выше и при необходимости внесите исправления.
- Чтобы увидеть процедуру в обозревателе объектов, щелкните правой кнопкой мыши элемент *Хранимые процедуры* и выберите пункт *Обновить*.

- Чтобы выполнить процедуру, в обозревателе объектов щелкните правой кнопкой мыши имя хранимой процедуры **SP_Spec** и выберите пункт *Выполнение хранимой процедуры*.
- В окне *Выполнение процедуры* введите любое целое число, соответствующее номеру факультета в качестве значения для параметра ***@fac***. В результате увидите специальности выбранного факультета.
- Аналогичным образом создайте хранимые процедуры **SP_Groups**, осуществляющую выборку групп заданной специальности, **SP_Students**, выбирающую студентов выбранной группы.
- На проекте MS Access создайте форму для просмотра студентов заданной группы. В свойствах формы источником формы выберите таблицу **Students**. В области данных формы поместите поле, показывающее значение поля **Name** таблицы **Students**. В заголовке формы создайте три поля со списком, источниками которых выберите таблицы **Faculties**, **Specialties**, **Groups**, задав такими же и имена полей со списком и настроив их так, чтобы их значения выбирались из **ID**, а видимыми были соответственно поля **Faculty**, **Specialty** и **Group**, используя приемы, описанные в Лабораторной работе №2.
- Войдите в свойства поля со списком **Faculties**. Войдите во вкладку «События». В позиции «После обновления» задайте значение «Процедура обработки событий» (для этого выбирается значение с помощью кнопочки, находящейся справа от данной строки или двойным щелчком на строке).
- В появившемся рабочем окне наберите код на VBA, который источником поля со списком **Specialties** назначит хранимую процедуру **SP_Specialty**:

```
Dim cn As New ADODB.Connection
Dim rs As New ADODB.Recordset
Set cn = New ADODB.Connection
Set rs= New ADODB.Recordset
cn.Provider = "Microsoft.Access.OLEDB.10.0"
cn.Properties("Data Provider").Value = "SQLOLEDB"
cn.Properties("Data Source").Value = "имя сервера"
cn.Properties("User ID").Value = "имя пользователя"
cn.Properties("Password").Value = "пароль"
cn.Properties("Initial Catalog").Value = "имя БД"
cn.Open
Set rs.ActiveConnection = cn
rs.Source = "Exec SP_Specialty " & Me.Faculty
rs.CursorLocation = adUseClient
rs.Open
Set Me![Specialties].Recordset = rs
Set rs = Nothing
Set cn = Nothing
```

- После этого войдите в свойства поля со списком **Specialties** и уберите источник строк, поскольку источник будет определяться теперь после обновления поля **Faculties**.
- Аналогично преобразуйте событие поля со списком **Specialties**, который задаст источником для поля **Groups** хранимую процедуру **SP_Groups**.

- Войдите в свойства поля со списком **Groups**. В событиях данного поля в графе «После обновления» задайте аналогичный код с единственной разницей – после формирования `recordset rs` на основе хранимой процедуры **SP_Students** `перед открытием rs` уберите строку

  ```
  rs.CursorLocation = adUseClient
  ```

  и объявите источник записей для формы:

  ```
  Me.Recordset = rs
  ```

- Войдите в свойства формы и уберите источник строк, поскольку источник будет определяться теперь после обновления поля **Groups**.
- Перейдя в режим формы, проверьте работу созданных полей со списком, а именно: при выборе факультета поле специальностей должно показывать специальности выбранного факультета; поле групп – показывать список групп выбранной специальности; после выбора группы форма должна показывать студентов данной группы.
- Добавьте кнопки в главную кнопочную форму для открытия этой формы.

4. Контрольные вопросы:

4.1. Что такое хранимая процедура?
4.2. Как создается хранимая процедура с помощью мастеров?
4.3. Как создается хранимая процедура с помощью T-SQL?
4.4. Какую структуру имеет инструкция хранимой процедуры?
4.5. Как с помощью VBA вызвать хранимую процедуру?
4.6. Как назначить хранимую процедуру источником формы, поля со списком?

Лабораторная работа №5
Создание триггеров (4 часа)

Цель работы

Освоение навыков создания триггеров в СУБД SQL SERVER 2014 для проведения автоматических действий при изменении данных в таблицах.

1. Теоретические сведения

Триггер (trigger) — это особый вид хранимой процедуры, выполняемый только тогда, когда одна или несколько строк таблицы, с которой связана эта процедура, изменяются с помощью инструкции `INSERT, UPDATE` или `DELETE`. При этом в определении триггера указывается, с какой именно из этих операций он будет связан. Поскольку SQL Server вызывает триггер *каждый раз,* когда выполняется соответствующая операция, в нем можно запрограммировать дополнительные действия по обеспечению реляционной целостности данных, не поддерживаемые стандартными средствами SQL Server. Кроме того, с помощью триггеров можно определять более сложные операции с данными, охватывающие несколько таблиц и имеющие логику, требующую программирования. Как правило, триггеры связывают с базовыми таблицами. Однако существует особый вид триггеров, называемый *замещающими триггерами (instead of triggers),* который можно связывать и с представлениями. Сначала мы рассмотрим триггеры, связываемые с базовыми таблицами. Это обычные триггеры (after-триггеры), используемые в предыдущих версиях SQL Server.

Примечание**.** В отличие от инструкции `DELETE`, инструкция `TRUNCATE TABLE` не вызывает выполнение триггера. Не выполняется триггер и в ответ на инструкцию `WRITETEXT`. Для пакетного добавления строк в таблицу с помощью программы `BCP` (Bulk Copy Program) триггер выполняется только в том случае, если установлена опция протоколирования пакетных инструкций.

SQL Server всегда сам вызывает триггеры, и выполнение их с помощью инструкции `execute` невозможно. У триггера не может быть ни параметров, ни возвращаемого значения. Для создания триггера используется инструкция `CREATE TRIGGER:`

```
CREATE TRIGGER TrackCustomerUpdates
    ON AppDta.dbo.Customer
    FOR UPDATE
  AS     INSERT INTO AppDta.dbo.CustUpdLog
              (Custld, Action, UpdUser, UpdDateTime)
    SELECT    Custld,'Update', CURRENT_USER, CUR-
RENT_TIMESTAMP FROM INSERTED
```

Этот пример демонстрирует важнейшие элементы определения триггера. Имя триггера следует за ключевыми словами `CREATE TRIGGER`. Далее идет

предложение ON, где указывается одна базовая таблица, с которой связывается создаваемый триггер. (Как уже говорилось, замещающие триггеры можно связывать с представлениями.) Предложение FOR определяет, для какого действия предназначен данный триггер. В нем задаются ключевые слова UPDATE, INSERT и DELETE в любом наборе и в любой последовательности. В качестве синонима ключевого слова FOR можно использовать ключевое слово AFTER. Далее следует ключевое слово AS, а за ним тело процедуры. В приведенном примере тело триггера состоит из единственной инструкции INSERT, которая добавляет строки в таблицу, предназначенную для контролирования обновлений в таблице Customer.

После создания этого триггера SQL Server будет автоматически выполнять его после каждой инструкции UPDATE, обновляющей строки таблицы Customer. В теле триггера можно использовать ключевое слово INSERTED для ссылки на временную, расположенную в оперативной памяти таблицу, содержащую все новые (обновленные) строки таблицы, с которой связан данный триггер.

Примечание. Таблицу INSERTED, которая используется в триггерах, не нужно создавать вручную. SQL Server сам создает и удаляет временные таблицы, располагающиеся в оперативной памяти. Ключевое слово INSERTED можно вводить в любом регистре, даже если для базы данных задано сопоставление с опцией учета регистра.

В нашем примере новые значения столбца Custld из обновленных строк таблицы Customer включаются в состав строк, добавляемых в пользовательскую таблицу CustUpdLog. В остальные три столбца этой таблицы записывается тип действия (т.е. 'Update'), имя пользователя, выполнившего это действие, и штамп времени. Для того чтобы контролировать операции добавления и удаления строк таблицы Customer, можно создать аналогичные триггеры, например:

```
CREATE TRIGGER TrackCustomerlnserts
    ON AppDta.dbo.Customer    FOR INSERT
  AS     INSERT INTO AppDta.dbo.CustUpdLog
         (Custld, Action,   UpdUser,      UpdDateTime)
  SELECT Custld, 'Insert', CURRENT_USER, CUR-
RENT_TIMESTAMP  FROM INSERTED

CREATE TRIGGER TrackCustomerDeletes
    ON AppDta.dbo.Customer
    FOR DELETE
  AS     INSERT INTO AppDta.dbo.CustUpdLog
         (Custld, Action,   UpdUser,      UpdDateTime)
SELECT    Custld, 'Delete', CURRENT_USER, CUR-
RENT_TIMESTAMP   FROM DELETED
```

Во втором из этих триггеров используется ключевое слово DELETED, предназначенное для ссылки на временную расположенную в оперативной памяти таблицу, содержащую старые (удаленные) строки таблицы, с которой связан данный триггер.

Примечание. Временные таблицы, на которые указывают идентификаторы INSERTED и DELETED, имеют ту же структуру, что и базовая таблица, с которой связан выполняемый триггер.

Вместо того чтобы создавать отдельный триггер для каждой операции, можно создать для них один общий триггер, в котором программным путем определять для какой операции он вызван, и выполнять соответствующие действия. В следующем примере в предложении FOR перечисляются все три операции INSERT, UPDATE и DELETE, а для добавления соответствующих данных в таблицу CustUpdLog используется инструкция IF:

```
CREATE TRIGGER TrackCustomerUpdates
    ON AppDta.DBO.CUSTOMER FOR INSERT, UPDATE, DELETE
AS  DECLARE @InsertedCount INT
    DECLARE @DeletedCount  INT
    SET @InsertedCount = (SELECT COUNT (*)
    FROM INSERTED)
    SET @DeletedCount = (SELECT COUNT (*) FROM DELETED)
    IF (@InsertedCount > 0) BEGIN
      INSERT INTO AppDta.dbo.CustUpdLog
              (Custld, Action, UpdUser, UpdDateTime )
        SELECT Custld,
          CASE WHEN (@DeIetedCount > 0) THEN 'Update'
               ELSE 'Insert' END,
                CURRENT_USER, CURRENT_TIMESTAMP
         FROM INSERTED
    END
    ELSE IF (@DeletedCount > 0) BEGIN
      INSERT INTO AppDta.dbo.CustUpdLog
       (Custld, Action,   UpdUser,      UpdDateTime )
SELECT Custld, 'Delete', CURRENT_USER, CURRENT_TIMESTAMP
    FROM DELETED
END
```

Как видно из этого примера, временная таблица INSERTED содержит строки в том случае, когда инструкция INSERT или UPDATE обработала хотя бы одну строку. А временная таблица DELETED содержит строки в том случае, когда инструкция DELETE или UPDATE обработала хотя бы одну строку. Для инструкции UPDATE таблица DELETED содержит обновленные строки со *старыми* значениями, а таблица INSERTED содержит те же строки с *новыми* значениями. В приведенном примере учтена еще одна особенность триггеров:

триггеры операций DELETE и UPDATE вызываются даже в том случае, если операция не удалила или не обновила ни одной строки (потому что не нашлось строк, соответствующих условию WHERE). Не забывайте об этом, когда будете создавать собственные триггеры.

Проверка факта изменения конкретных столбцов

В теле триггера INSERT или UPDATE с помощью особой проверки IF UPDATE (имя-столбца) можно выяснить, обновлен ли конкретный столбец таблицы. Следующий пример показывает, как это делается:

```
CREATE TRIGGER TrackDiscountUpdates
    ON AppDta.dbo.Customer FOR UPDATE
  AS     IF UPDATE(Discount)
INSERT INTO AppDta.dbo.CustUpdLog (Custld, Action,   Dis-
count, UpdUser,  UpdDateTime)
SELECT Custld, 'Update', Discount, CURRENT_USER, CUR-
RENT_TIMESTAMP FROM INSERTED
    ENDIF
```

Так же как в обычной структуре IF. . .ELSE. . .ENDIF, в инструкции IF UPDATE можно использовать логические операторы AND и OR, объединяющие несколько условий. Например:

```
CREATE TRIGGER TrackCustomerUpdates
    ON AppDta.dbo.Customer FOR UPDATE
AS       IF UPDATE(Name) AND UPDATE(Discount) ...

CREATE TRIGGER TrackCustomerUpdates
    ON AppDta.dbo.Customer FOR UPDATE
AS   IF UPDATE( Name )  OR UPDATE( Discount )    ...
```

Примечание. SQL Server позволяет помещать проверку условия IF UPDATE (имя-столбца) прямо за ключевым словом AS и без ключевого слова ENDIF. В этом случае все тело триггера выполняется при условии, что заданный столбец обновлен. Однако мы рекомендуем использовать стандартный синтаксис IF UPDATE (имя-столбца) ...ENDIF и рассматривать UPDATE (имя-столбца) как обычную логическую функцию.

Для инструкции UPDATE проверка UPDATE (имя-столбца) возвращает True, если столбец указан в предложении SET. SQL Server *не* проверяет, действительно ли его новое значение отличается от старого. Для инструкции INSERT проверка UPDATE (имя-столбца) возвращает True, если столбец явно указан в списке столбцов этой инструкции или если список столбцов в ней вообще отсутствует (что неявно означает "все столбцы"). И не имеет значе-

ния, каким образом столбцу присвоено новое значение: задано ли оно как литерал, как выражение или вместо значения задано ключевое слово NULL или DEFAULT.

Дополнительные опции триггеров

В инструкции CREATE TRIGGER могут использоваться две необязательные опции. Первая из них, WITH ENCRYPTION, шифрует сохраняемый в системном каталоге код триггера:

```
CREATE TRIGGER TrackCustomerUpdates
    ON AppDta.dbo.Customer WITH ENCRYPTION FOR UPDATE AS
...
```

Для того чтобы при модификации данных процессом репликации триггер не вызывался, нужно включить в его определение опцию NOT FOR REPLICATION:

```
CREATE TRIGGER TrackCustomerUpdates
    ON AppDta.dbo.Customer FOR UPDATE
       NOT FOR REPLICATION AS ...
```

Изменение и удаление триггеров

Для изменения определения триггера используется инструкция ALTER TRIGGER. Можно просто удалить триггер и создать его заново. Удаление триггера выполняется так:

```
DROP TRIGGER TrackCustomerUpdates
```

Инструкция ALTER TRIGGER имеет точно такой же синтаксис, что и инструкция CREATE TRIGGER, за исключением первого ключевого слова.

Включение и отключение триггера

Инструкция ALTER TABLE позволяет включать и отключать отдельные триггеры таблицы. Отключение триггера выполняется так:

```
ALTER TABLE Customer
DISABLE TRIGGER TrackCustomerUpdates
```

Для того чтобы снова включить триггер, повторите эту инструкцию, заменив ключевое слово DISABLE ключевым словом ENABLE. Можно включать и отключать все триггеры сразу, заменив имя триггера ключевым словом ALL.

Работа с триггерами

Для одной таблицы можно создавать несколько триггеров, более того, их может быть несколько даже для одной и той же инструкции (например UPDATE). Каждая новая инструкция CREATE TRIGGER связывает с таблицей еще один триггер в дополнение к уже существующим. Когда вызывается инструкция, с которой связано несколько триггеров, для членов которой не определен конкретный порядок, все они выполняются по очереди. Единственное, что можно сделать в отношении последовательности их выполнения, это указать, какой из триггеров должен запускаться первым, а какой — последним. Для этого используется системная хранимая процедура sp_settriggerorder:

```
sp_settriggerorder 'TrackCustomerUpdates', 'First', 'Update'
```

У процедуры sp_settriggerorder три аргумента: имя триггера, порядок его выполнения (First, Last или None – первый, последний или порядок не определен) и тип инструкции (INSERT, UPDATE или DELETE). Если триггер связан с двумя или тремя инструкциями, нужно выполнить процедуру sp_settriggerorder для каждой инструкции, для которой вы хотите задать порядок его выполнения. После окончания инструкции ALTER TRIGGER порядок выполнения измененного триггера устанавливается в None, т.е. снова становится неопределенным.

Таким образом, если более двух триггеров таблицы связано с одной и той же операцией, порядок выполнения всех этих триггеров задать нельзя. А потому при написании этих триггеров не рассчитывайте на определенную последовательность их выполнения, т.е. не основывайте действия одного триггера на результатах выполнения другого. Эта особенность выполнения триггеров не так уж важна, поскольку в случае, когда необходимо разбить триггер на несколько отдельных модулей, всегда можно написать несколько обыкновенных хранимых процедур и по очереди вызывать их из триггера. В этом случае в порядке их выполнения можно быть уверенным.

Триггер всегда создается в той же базе данных, что и таблица, с которой он связан. В пределах базы данных имена триггеров должны быть уникальны. При желании можно уточнить имя триггера именем владельца (например, dbo.TrackCustomerUpdates), но владельцем триггера может быть только владелец таблицы.

Примечание. Хотя триггер всегда связан с базовой таблицей из той же базы данных, в которой хранится он сам, в нем могут быть ссылки на объекты из других баз данных. Это позволяет создавать триггеры, обеспечивающие целостность связей между двумя таблицами из разных баз данных. Внешние ключи, определяемые в инструкции CREATE TABLE, этого не позволяют.

Несмотря на то, что триггер является разновидностью хранимой процеду-

ры, его нельзя вызывать с помощью инструкции EXECUTE. Если какой-то программный код должен совместно использовать триггеры и обычные хранимые процедуры или просто несколько триггеров, поместите его в хранимую процедуру и вызывайте из этих триггеров.

Вложенные и рекурсивные вызовы триггеров

Если триггер обновляет таблицу, это обновление может вызвать выполнение другого триггера или даже того же самого. По умолчанию SQL Server допускает до 32 вложенных не рекурсивных вызовов триггеров, а рекурсивные вызовы запрещает. Мы же рекомендуем разрешить рекурсивное выполнение триггеров и пользоваться им по мере необходимости. Опция, разрешающая рекурсивное выполнение триггеров, устанавливается с помощью инструкции ALTER DATABASE или системной хранимой процедуры sp_dboption:

```
ALTER DATABASE AppDta
SET RECURSIVE_TRIGGERS ON
```

или

```
sp_dboption AppDta 'recursive triggers', 'true'
```

Хотя мы не рекомендуем так поступать, но учтите, что с помощью следующей инструкции можно запретить все вложенные вызовы триггеров во всех базах данных:

```
sp_configure 'nested triggers', 0
```

Вывод информации о триггерах

Информацию о триггерах возвращают четыре системные хранимые процедуры:

- ✓ sphelp имя-триггера — возвращает имя владельца триггера, а также дату и время его создания;
- ✓ sphelptext имя-триггера — возвращает исходный код триггера (если он не зашифрован);
- ✓ sp_depends имя-триггера — возвращает список объектов, на которые в триггере имеются ссылки;
- ✓ sphelptrigger имя-таблицы — возвращает список триггеров, определенных для заданной таблицы.

Процедура sphelptext выручит вас, если вы потеряли исходный код одного из триггеров.

Программирование триггеров

В основном программирование триггеров ничем не отличается от программирования хранимых процедур, однако некоторые ограничения все же имеются. Прежде всего, в триггерах нельзя использовать следующие инструкции:

```
ALTER DATABASE   CREATE DATABASE      DISK INIT
DISK RESIZE      DROP DATABASE        LOAD DATABASE
LOAD LOG         RECONFIGURE          RESTORE DATABASE
RESTORE LOG
```

Кроме того, триггер не может возвращать вызывающей программе результирующие наборы строк, т.е. в нем не должны выполняться самостоятельные инструкции `SELECT`.

Скалярные вложенные запросы, значения которых присваиваются переменным триггера, можно выполнять совершенно свободно, поскольку они не создают результирующих наборов строк. В начале триггера желательно выполнять инструкцию `SET NOCOUNT ON`, которая отключает отправку сообщений, указывающих, сколько строк обработано каждой SQL-инструкцией.

Примечание. Если в триггере (или в обычной хранимой процедуре) используется инструкция `SET`, после завершения работы триггера устанавливаемые ею опции возвращаются в исходное состояние.

Одно из назначений триггеров состоит в том, чтобы дополнить стандартные средства обеспечения реляционной целостности (т.е. первичные ключи, внешние ключи, ограничения уникальности и ограничения на значения) пользовательскими средствами, учитывающими специфику конкретной базы данных. После проверки определенных условий в триггере можно отменить всю транзакцию, выполнив инструкцию `ROLLBACK`. Эту возможность демонстрирует следующий триггер, отменяющий назначение скидки клиенту, когда эту операцию попытался выполнить пользователь, не имеющий на это права:

```
CREATE TRIGGER CheckDiscountUpdates
    ON AppDta.dbo.Customer FOR INSERT, UPDATE
  AS
-- Проверяем,   задан ли в инструкции столбец Discount
    IF UPDATE(Discount) BEGIN
      DECLARE @User VARCHAR(256) SET NOCOUNT ON
-- Проверяем,  действительно ли задано значение столбца
-- Discount
      IF (EXISTS (SELECT * FROM INSRETED WHERE Discount >
0))
       BEGIN
-- Убеждаемся,  что пользователь имеет право назначать
-- скидки
```

```
    IF (NOT EXISTS (SELECT * FROM Employee
    WHERE UserName = CURRENT_USER
    AND DiscountAuthority = 'Y'))
           BEGIN
-- Отправляем сообщение и отменяем транзакцию
    SET @User = CURRENT_USER
    RAISERROR('User %s not approved to assign dis-
                        counts.', 16, 1, @User )
              ROLLBACK
           END
       END
    END
```

Триггер вызывается один раз после выполнения инструкции, с которой он связан, и только в том случае, если эта инструкция не нарушает ограничений первичного и внешнего ключей, уникальности и целостности. К моменту вызова триггера результаты выполнения инструкции уже отражены в таблице, с которой он связан, как и во временных таблицах `INSRETED` и `DELETED`, но соответствующая транзакция еще не сохранена, и все эти изменения можно отменить.

Поскольку инструкция должна удовлетворять всем ограничениям таблицы *еще до* вызова триггера, триггеры нельзя использовать для обеспечения выполнения этих ограничений. Предположим, например, что для таблицы `Sale` определено ограничение внешнего ключа без каскадного удаления. Это ограничение препятствует удалению строки таблицы `Customer`, если с ней связана хоть одна строка таблицы `Sale`. Возможно, вы захотите создать `DELETE`-триггер для таблицы `Customer`, удаляющий из таблицы `Sale` все заказы удаляемого клиента, сделанные в прошлом году, и тем самым разрешающий удаление строки таблицы `Customer`, если больше заказов не осталось. Но это не так просто. Поскольку, как уже говорилось, ограничение внешнего ключа проверяется еще до вызова триггера, SQL Server 2000 просто отменит операцию удаления клиента, имеющего хоть один заказ. В итоге ваш триггер даже не будет вызван. Впрочем, у этой проблемы все же имеется решение. Таким решение является использование замещающих триггеров, о которых рассказывается в следующем разделе.

Триггер может выполнять различные операции с таблицей, с которой он связан, в частности, выполнять дополнительную обработку добавленных или обновленных строк. Но если вы решите воспользоваться этой возможностью, подумайте, что делать с рекурсивными вызовами триггера.

Замещающие триггеры

До сих пор мы говорили об обычных триггерах (after-триггерах), т.е. триггерах, выполняемых после операции, с которой они связаны, и после проверки всех относящихся к этой операции ограничений. Триггеры другого типа, называемые *замещающими триггерами (instead of triggers)*, выполняются *вме-*

сто операции, с которой они связаны. Замещающие триггеры могут выполнять множество действий, причем не обязательно связанных с исходной операцией (например, добавлением строк в таблицу). И если обычные триггеры определяются исключительно для базовых таблиц (поскольку именно там модифицированы данные, с которыми они работают), то замещающие триггеры можно определять и для представлений, причем даже для тех представлений, которые без триггеров были бы доступны только для чтения.

Для создания замещающего триггера ключевое слово `FOR` (или `AFTER`) в инструкции `CREATE TRIGGER` нужно заменить ключевым словом `INSTEAD OF`. Ниже приведен пример замещающего триггера, который решает описанную в предыдущем разделе проблему: автоматически удаляет зависимые строки таблицы `Sale` при удалении родительской строки таблицы `Customer`:

```
CREATE TRIGGER CustomerDelete ON   Customer INSTEAD OF
DELETE
  AS SET NOCOUNT ON
  -- Проверяем,  имеются ли в таблице Sale прошлогодние
заказы
  -- удаляемых клиентов
    IF (EXISTS (SELECT * FROM DELETED JOIN Sale
                    ON DELETED.Custld = Sale.Custld
                WHERE DATEDIFF (DAY, SaleDate,
                  CURRENT__TIMESTAMP) < 365)) BEGIN
      RAISERROR('One or more customers have recent
sales', 16, 1)
      RETURN
    END
    -- Удаляем соответствующие строки таблицы Sale
    -- (все они устарели)
    DELETE FROM Sale WHERE Custld IN (SELECT Custld FROM
DELETED)
    IF (@@ERROR > 0) BEGIN
      ROLLBACK
      RAISERROR ('Could not delete all old sales', 16, 2)
RETURN
    END
  -- Удаляем клиентов
  DELETE FROM Customer WHERE Custld IN
    (SELECT Custld FROM DELETED )
  -- В случае неудачи отменяем все изменения
  IF (@@ERROR > 0) BEGIN
    ROLLBACK
 RAISERROR('Could not delete all old customers', 16, 3)
    RETURN
  END
RETURN
```

Обратите внимание, что в этом примере используется ключевое слово `DELETED`. Это уже знакомая вам ссылка на временную таблицу, создаваемую SQL Server специально для выполняемого триггера. Только в отличие от обычного триггера эта таблица содержит не удаленные строки, а строки, которые *были бы* удалены, если бы SQL Server выполнил исходную инструкцию.

На самом же деле к моменту вызова замещающего триггера в связанную с ним таблицу не внесено *никаких* изменений. Как показывает приведенный пример, для выборки тех строк, которые были бы обработаны исходной инструкцией, можно использовать первичные ключи строк таблицы `DELETED` (или `INSERTED`).

В отличие от обычных триггеров с каждым типом инструкций, которые могут быть выполнены для таблицы, можно связать только один замещающий триггер. Однако наряду с замещающим триггером для этой инструкции может быть определено и несколько обычных триггеров. Если замещающий триггер выполняет ту же SQL-инструкцию, для которой он был вызван, рекурсивного вызова этого триггера не происходит. Вместо этого просто выполняется соответствующая операция, и после нее вызываются обычные триггеры (если таковые имеются).

2. Задание к работе

Создать триггеры для осуществления контроля над изменением данных в таблицах.

3. Порядок выполнения лабораторной работы

- Создайте таблицы **TableForTriggers**, задав структуру, подобную таблице **Students**, а также добавьте поле **Activity (char(10))**. Данная таблица будет содержать записи, обновляемые в таблице **Students**, а поле **Activity** будет иметь значения *Update, Insert* или *Delete*.
- Последовательно разверните узел ***Базы данных***, свою базу данных и узел *Программирование*.
- Щелкните правой кнопкой мыши элемент *Триггеры базы данных* и выберите пункт *Создать триггер базы данных*.
- В рабочем поле, используя синтаксис триггера, введите код для создания триггера, который будет вставлять обновленную в таблице **Students** запись в таблицу **TableForTriggers**, причем значение поля **Activity** присвоить *Update*.
- Аналогично создайте триггер, который будет вставлять добавленную в таблицу **Students** запись в таблицу **TableForTriggers**, значение поля **Activity** присвоить *Insert*, а также триггер, вставляющий удаленную из таблицы **Students** запись в таблицу **TableForTriggers**, значение поля **Activity** будет *Delete*.
- Для проверки созданных триггеров попробуйте изменить, удалить и вставить несколько записей в таблицу **Students**, а затем откройте таблицу **TableForTriggers** и убедитесь, что результаты обновлений отражены верно.

4. Контрольные вопросы:

4.7. Что такое триггер?
4.8. Какие существуют типы триггеров?
4.9. Как создается триггер?
4.10. В чем отличие триггера от хранимой процедуры?
4.11. Какую структуру имеет инструкция триггера?

ЛИТЕРАТУРА

1. А. Бондарь. Microsoft SQL Server 2012, Санкт-Петербург: БХВ-Петербург, 2013
2. Д. Петкович. Microsoft SQL Server 2012. Руководство для начинающих, Санкт-Петербург : БХВ-Петербург , 2013
3. М. Оутей, П. Конте. SQL Server 2000. Эффективная работа. Санкт-Петербург, 2002.
4. Г. Гурвиц. Microsoft Access 2010. Разработка приложений на реальном примере. БХВ-Петербург, 2013
5. А. Сенов. Access 2010 – учебный курс. Изд. Питер, 2010.