МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Белгородский государственный технологический университет
им. В.Г. Шухова
Кафедра программного обеспечения вычислительной техники
и автоматизированных систем

Утверждено
научно-методическим советом
университета

БАЗЫ ДАННЫХ

Методические указания к выполнению лабораторных работ
для студентов направлений бакалавриата
09.03.01 — Информатика и вычислительная техника и
09.03.04 — Программная инженерия

Белгород
2018

УДК 004(075.8)
ББК 32.973.26я73
Б17

Составители: ст. преп. А. И. Гарибов
ст. преп. Т. В. Бондаренко

Рецензент: канд. техн. наук, доцент Белгородского государственного технологического университета им. В. Г. Шухова Е. Н. Коробкова

Б17 **Базы** данных: методические указания к выполнению лабораторных работ для студентов направлений бакалавриата 09.03.01 — Информатика и вычислительная техника и 09.03.04 — Программная инженерия / сост.: А. И. Гарибов, Т. В. Бондаренко. — Белгород: Изд-во БГТУ, 2018. — 52 с.

В методических указаниях содержатся задания для лабораторных работ, посвящённых практическому использованию СУБД Microsoft SQL Server и созданию приложений для баз данных на основе технологии Microsoft .NET Framework. Методические указания предназначены для студентов направлений бакалавриата 09.03.01 — Информатика и вычислительная техника и 09.03.04 — Программная инженерия

Данное издание публикуется в авторской редакции.

УДК 004(075.8)
ББК 32.973.26я73

Содержание

Лабораторная работа № 1
Разработка структуры базы данных

Цель работы: научиться создавать инфологическую модель данных и структуру базы данных по заданной предметной области.

Основные теоретические сведения

Первым этапом разработки структуры базы данных является анализ предметной области. В ходе анализа выявляются основные сущности и их атрибуты, а также взаимосвязи между ними.

Сущность — это любой конкретный или абстрактный объект в рассматриваемой предметной области, информацию о котором необходимо хранить в базе данных.

Атрибут — это именованная характеристика сущности. Наименование атрибута должно быть уникальным для конкретного типа сущности, но может быть одинаковым для различного типа сущностей. Атрибуты используются для определения того, какая информация должна быть собрана о сущности.

Связь — это ассоциирование двух или более сущностей, которое указывает, каким образом связаны сущности. Эта информация необходима для поддержания целостности данных.

Результат анализа представляется в виде текста, в котором отражена вся информация, которую необходимо хранить в базе данных. Характеристики сущностей, которые хранить в базе данных не требуется, в результаты анализа не вносятся.

На основе проведённого анализа составляется ***инфологическая модель*** данных. Одной из форм инфологической модели является диаграмма «сущность — связь», или ER-диаграмма.

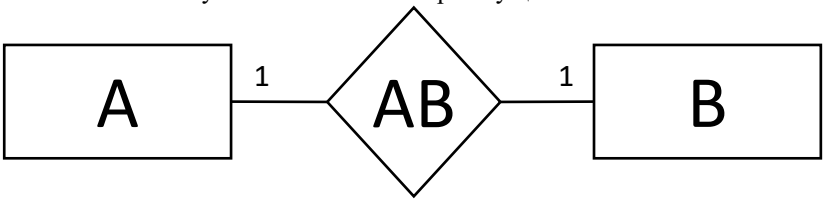
Разработка диаграммы «сущность — связь»

Основными элементами диаграммы «сущность — связь» являются:

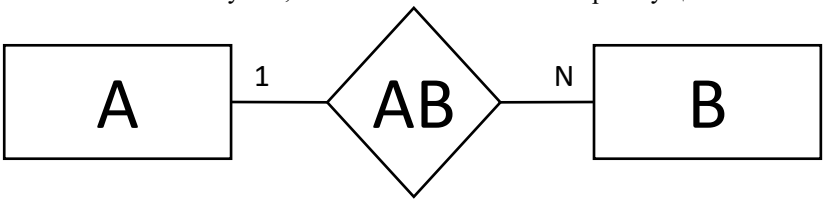
- сущности — представляют собой сущности предметной области. Изображаются прямоугольниками, внутри которых указано наименование сущности;
- ассоциации — представляют собой связи между сущностями. Изображаются ромбами, внутри которых может быть указано наименования связи;
- соединительные линии — соединяют связанные сущности, проходят через вершины ромба, представляющего ассоциацию. Над линиями проставляется степень связи (1 или буква, заменяющая слово «много»).

Между двумя сущностями возможны следующие виды связей:

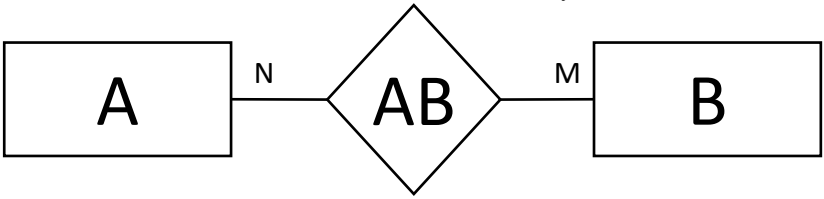
– ***связь «один-к-одному»*** (1:1) — одному экземпляру сущности A соответствует 1 или 0 экземпляров сущности B:



– ***связь «один-ко-многим»*** (1:N) — одному экземпляру сущности A соответствует 0, 1 или несколько экземпляров сущности B:



– ***связь «многие-к-одному»*** (N:1) — аналогична связи «один-ко-многим», если сущности поменять местами, поэтому её обычно не рассматривают как отдельный вид связи;
– ***связь «многие-ко-многим»*** (N:M) — является комбинацией связей «один-ко-многим» и «многие-к-одному»:



Существуют также более сложные связи, которые встречаются редко и зачастую сводятся к комбинации основных видов связей.

Атрибуты на диаграмму «сущность — связь» допускается не выносить, поскольку изображение большого количества атрибутов существенно увеличивает размер диаграммы, что приводит к уменьшению её наглядности и удобочитаемости. Атрибуты каждой сущности можно указывать в виде отдельного описания, которое можно выполнить в виде приложения к диаграмме.

По полученной диаграмме «сущность — связь» можно создать схему структуры базы данных, которая удовлетворяет требованиям *третьей нормальной формы*, не содержит аномалий модификации и позволяет поддерживать ссылочную целостность данных.

Проектирование структуры базы данных

Структуру базы данных, не зависящей от конкретной СУБД, удобно представить в виде табличной структуры.

Для каждой сущности диаграммы «сущность — связь» создаётся *таблица* базы данных (в некоторых случаях несколько таблиц). Для каждого атрибута сущности создаётся *столбец* в соответствующей таблице, который также называют *полем*. Экземпляры сущности хранятся в *строках* таблицы, которые также называют *записями*.

Каждая таблица должна иметь имя, уникальное для всех объектов базы данных. Имя каждого столбца должно быть уникальным для данной таблицы. Рекомендуется, чтобы имена объектов базы данных состояли только из латинских букв, цифр и знака подчёркивания и не начинались с цифры.

В каждой таблице требуется определить ***первичный ключ*** — минимальный набор столбцов таблицы, значения которых однозначно идентифицируют каждую строку этой таблицы.

Рекомендуется выбрать в качестве первичного ключа один столбец, соответствующий атрибуту сущности, значения которого являются уникальными для каждой строки таблицы, и требует небольшой объём памяти для их хранения в базе данных (обычно не более 8–10 байт).

Предпочтительными типами данных столбцов поля первичного ключа являются целочисленный и строковый длиной до 10 символов. Если в таблице нет подходящих столбцов, добавляется ***суррогатный ключ*** — столбец, не являющийся атрибутом сущности, предназначенный для обеспечения уникальности каждой строки таблицы базы данных. В качестве типа данных обычно выбирается целочисленный тип данных. Значение суррогатного ключа для каждой строки таблицы генерируется автоматически средствами СУБД и называется ***идентификатором записи***. Чтобы отличить столбец идентификатора записи от столбцов, соответствующих атрибутам сущности, ему часто присваивают имя **ID**.

Для реализации связи между сущностями используется ***внешний ключ*** — набор из одного или нескольких столбцов одной таблицы (*подчинённой*), которая имеет связь с другой таблицей (*главной*). При этом, количество и типы столбцов внешнего ключа подчинённой таблицы, должны совпадать с количеством и типами столбцов первичного ключа главной таблицы. Обычно внешний ключ, как и первичный ключ главной таблицы, состоит из одного столбца, имя которого имеет вид **ИмяГлавнойТаблицыID**.

Если между таблицами установлена связь, то для каждой записи подчинённой таблицы значения полей, соответствующих внешнему

ключу, должны совпадать со значениями полей первичного ключа связанной записи главной таблицы. Однако, для любой записи таблицы некоторые её поля могут не содержать значений, если это разрешено в соответствующих столбцах таблицы. В этом случае говорят, что поле содержит пустое значение. Если все поля внешнего ключа одной из записей подчинённой таблицы содержат пустое значение, то такая запись не связана ни с одной из записей главной таблицы.

Для связи «один-ко-многим» к таблице, соответствующей сущности B, добавляется столбец внешнего ключа, значения которого ссылаются на значения первичного ключа таблицы, соответствующей сущности A. Связь «один-к-одному» является частным случаем связи «один-ко-многим» и реализуется аналогичным образом.

Для связи «многие-ко-многим» в базе данных создаётся дополнительная таблица и две связи «многие-к-одному» к таблицам, соответствующим сущностям A и B. Эта таблица состоит из столбцов внешнего ключа для обеих связей, и для всех этих столбцов создаётся первичный ключ.

Таким образом, все типы связи между сущностями сводятся к связи «один-ко-многим».

В соответствии с требованиями первой нормальной формы каждое поле любой записи должно содержать только одно значение определённого типа. Все современные СУБД поддерживают следующие основные типы данных:

- целочисленный, при необходимости с указанием диапазона значений или необходимого количества байт;
- вещественный тип, при необходимости с указанием количества цифр после запятой;
- строковый с указанием максимального количества символов;
- дата;
- дата и время;
- логический;
- массив байтовых значений.

Столбцы таблицы должны иметь один из указанных типов данных. Также необходимо указать, допускает ли столбец пустые значения (NULL).

Задание к работе

По выбранной (по согласованию с преподавателем) предметной области:

1. Выполнить анализ предметной области, выделить основные сущности, атрибуты и связи.

2. Создать диаграмму «сущность — связь».
3. Разработать структуру базы данных.

Содержание отчёта

1. Результаты анализа предметной области в виде текста с указанием сущностей, их атрибутов и связей между сущностями, которые должны храниться в базе данных.
2. Диаграмма «сущность — связь» с перечислением атрибутов каждой сущности.
3. Схема структуры базы данных.
4. Описание столбцов таблиц базы данных, включающее: имя столбца, назначение (какие данные хранятся), тип данных, допускает ли столбец пустые значения.

Лабораторная работа № 2
Создание объектов базы данных в СУБД Microsoft SQL Server

Цель работы: изучить основные возможности языка SQL для создания структуры базы данных, научиться создавать базы данных, таблицы, связи, ограничения, а также создавать, изменять и удалять данные.

Основные теоретические сведения

Управление созданием и использованием баз данных является основной задачей *системы управления базами данных (СУБД)*. Базы данных, которыми управляет СУБД, физически располагаются на *сервере баз данных*. Для получения доступа к серверу баз данных необходимо знать имя или IP-адрес сервера, имя пользователя и пароль.

Взаимодействие с сервером баз данных для создания и модификации структуры базы данных выполняется с помощью специальных команд *языка описания данных (DDL — Data Definition Language)*. В СУБД Microsoft SQL Server это SQL-запросы, предназначенные для создания, изменения и удаления объектов баз данных, а также самих баз данных.

Создание, изменение и удаление данных в таблицах базы данных выполняется командами *языка манипулирования данными (DML — Data Manipulation Language)*, которые также являются частью языка SQL.

Для создания и запуска SQL-запросов необходима специальная программа, предназначенная для взаимодействия с сервером баз данных под управлением конкретной СУБД. В состав СУБД Microsoft SQL Server входит программа *SQL Server Management Studio (SSMS)*,

которая содержит текстовый редактор для создания и выполнения SQL-запросов, а также графические средства для управления структурой базы данных. Также существуют аналогичные программы сторонних производителей, большинство из которых являются платными и содержат дополнительные возможности.

Создание и модификация базы данных

Для создания базы данных используется следующий запрос:

```
CREATE DATABASE database_name
```

где *database_name* — имя базы данных.

Удаление созданной ранее базы данных выполняется с помощью запроса:

```
DROP DATABASE database_name
```

Чтобы изменить имя существующей базы данных, необходимо выполнить следующий запрос:

```
ALTER DATABASE database_name
  MODIFY NAME = new_database_name
```

где *new_database_name* — новое имя базы данных.

На одном сервере баз данных может быть создано несколько баз данных. При выполнении запросов к объектам базы данных требуется указать, к какой базе данных происходит обращение. Чтобы избежать этого, можно задать базу данных по умолчанию:

```
USE database_name
```

При выполнении следующих запросов если при обращении к объекту данных не указано имя базы данных, будет использоваться база данных по умолчанию.

Создание и модификация схемы базы данных

Схема базы данных — это логический контейнер, который включает в себя другие объекты базы данных. Имена объектов базы данных должны быть уникальными в рамках схемы. Схемы данных удобно использовать в больших базах данных для удобного логического разбиения всей базы данных на отдельные части, а также для разграничения доступа к объектам базы данных для различных пользователей.

Любая база данных Microsoft SQL Server, начиная с версии 2005, содержит схему *dbo*, которая используется по умолчанию. При необходимости, можно создать новую схему данных с помощью следующего запроса:

```
CREATE SCHEMA schema_name
```

где *schema_name* — имя схемы.

Удаление созданной ранее схемы данных выполняется с помощью запроса:

```
DROP SCHEMA schema_name
```

Создание и модификация таблицы данных

Любая таблица базы данных содержит один или несколько столбцов, а также может содержать *ограничения целостности*. Имена столбцов должны быть уникальными для таблицы, а имена ограничений уникальны для текущей схемы базы данных.

Для создания таблицы в базе данных используется следующий запрос:

```
CREATE TABLE table_name (column[, column, ...])
```

где *table_name* — имя таблицы в полном или сокращённом виде, *column* — описание столбца.

Полный формат имени таблицы имеет вид:

```
[[database_name.]schema_name.]t_name
```

где *t_name* — имя таблицы. Если не указано имя базы данных или схемы данных, используются значения по умолчанию.

Описание столбца состоит из нескольких секций, разделённых пробелами, которые обычно располагаются в следующем порядке:

- имя столбца, обязательно;
- тип данных, обязательно;
- допускает ли столбец пустые значения: если допускает — значение *NULL*, если не допускает — значение *NOT NULL*. Если значение не указано, применяется *NULL*;
- *UNIQUE* — если указано, значения столбца являются уникальными;

- *PRIMARY KEY* — если указано, столбец является первичным ключом. *PRIMARY KEY* включает в себя ограничения *NOT NULL* и *UNIQUE*;
- *IDENTITY(seed,increment)* — если указано, столбец является автоинкрементным. Параметр *seed* задаёт значение, присваиваемое самой первой строке, добавляемой в таблицу, параметр *increment* — значение приращения, которое прибавляется к значению идентификатора предыдущей добавленной строки. Если значения параметров не указаны, применяются значения по умолчанию: *(1,1)*. Применяется к суррогатным ключам целочисленного типа;
- *DEFAULT value* — если указано, параметр *value* задаёт значение по умолчанию, которое используется, если при добавлении строки не указано значение данного столбца;
- *CHECK(condition)* — если указано, параметр *condition* задаёт условие, которое проверяется при добавлении и изменении строки. Если условие не выполняется, действие со строкой завершается ошибкой. Условие является логическим выражением, которое может содержать константные значения, имена столбцов таблицы, стандартные и пользовательские функции;
- связь — если указано, столбец является внешним ключом по отношению к другой таблице.

В Microsoft SQL Server определены следующие основные типы данных для столбцов:

- беззнаковый целочисленный тип: *tinyint* (1 байт);
- знаковые целочисленные типы: *smallint* (2 байта), *int* (4 байта), *bigint* (8 байтов);
- числовой тип с фиксированной точностью и масштабом: *decimal(p,s)* или *numeric(p,s)*, где *p* — максимальное количество десятичных разрядов от 1 до 38, *s* — максимальное количество десятичных разрядов дробной части числа от 1 до *p*.
- вещественные типы: *real* (4 байта), *float* (размер зависит от значения);
- денежные типы: *smallmoney* (4 байта, максимальное значение — 214748,3647), *money* (8 байтов);
- строковые типы фиксированной длины: *char(n)* (для символов в заданной кодировке), *nchar(n)* (для символов в кодировке Unicode), где *n* — максимальное количество символов до 8000;
- строковые типы переменной длины: *varchar(n)* (для символов в заданной кодировке) и *nvarchar(n)* (для символов в кодировке Unicode), где *n* — максимальное количество символов до 4000

или значение *max*, которое определяет максимально возможное количество символов. Максимальный размер строки составляет 2 Гб;
- тип *bit* — может принимать значения 0 или 1;
- типы даты и времени: *date* — дата, *time* — время, *datetime2* — дата и время, *datetimeoffset* — дата и время с учётом часового пояса;
- массив байт фиксированной длины: *binary(n)*, где *n* — размер массива до 8000;
- массив байт переменной длины: *varbinary(n)*, где *n* — максимальный размер массива до 8000 или значение *max*, которое определяет максимально возможное количество байт. Максимальный размер массива составляет 2 Гб;
- тип GUID: *uniqueidentifier* (16 байт). Может использоваться в качестве идентификатора для обеспечения уникальности записи при добавлении в таблицу данных из внешних источников.

Описание связи для столбца внешнего ключа имеет следующий формат:

```
[CONSTRAINT constraint_name] [FOREIGN KEY]
   REFERENCES ref_table_name [(ref_column)]
   [ON DELETE on_delete_action] [ON UPDATE on_update_action]
```

где *constraint_name* — имя связи (если не задано, генерируется автоматически), *ref_table_name* — имя главной таблицы, *ref_column* — имя столбца главной таблицы, участвующего в связи (если не задано, используется столбец первичного ключа главной таблицы), *on_delete_action* — действие, которое выполняется над строками текущей таблицы при попытке удалить соответствующую строку главной таблицы, *on_update_action* — действие, которое выполняется над значениями внешнего ключа записей текущей таблицы при попытке изменить значение первичного ключа соответствующей записи главной таблицы. Значение *on_delete_action* и *on_update_action* может быть одним из следующих:
- *NO ACTION* — операция изменения записи главной таблицы завершается ошибкой, если на неё ссылается хотя бы одна запись подчинённой таблицы. Используется по умолчанию, если соответствующая секция отсутствует;
- *SET NULL* — полю внешнего ключа для затрагиваемых записей текущей таблицы присваивается значение *NULL*. Нельзя использовать, если столбец имеет ограничение *NOT NULL*;

– *SET DEFAULT* — полю внешнего ключа для затрагиваемых записей текущей таблицы присваивается значение, заданное по умолчанию данного столбца. Можно использовать, если столбец имеет ограничение *DEFAULT*;
– *CASCADE* — соответствующая операция изменения выполняется также над затрагиваемыми записями главной таблицы.

Если требуется задать ограничение *UNIQUE*, *PRIMARY KEY*, *FOREIGN KEY* для нескольких столбцов таблицы, их указывают не в описании столбца, а после описания столбцов таблицы через запятую. Имена столбцов ограничения указываются в круглых скобках через запятую.

Все создаваемые ограничения являются объектами таблицы и имеют имя. Если имя ограничения не задаётся, оно генерируется автоматически. Также можно создавать ограничения для таблицы отдельными запросами после создания самой таблицы.

Удаление из базы данных созданной ранее таблицы выполняется с помощью запроса:

```
DROP TABLE table_name
```

Удаление таблицы также приводит к удалению всех созданных для неё ограничений.

Изменение существующей таблицы выполняется с помощью следующего запроса:

```
ALTER TABLE table_name operation
```

где *operation* — операция модификации объекта таблицы: *ALTER COLUMN* — изменение столбца, *ADD* — добавления столбца или ограничения, *DROP* — удаление столбца или ограничения.

Вставка строк данных в таблицу

Для добавления данных в таблицу базы данных используется следующий запрос:

```
INSERT [INTO] table_name
[(column_name[, column_name, ...])]
VALUES
(value[, value, ...])[, (value[, value, ...]), ...]
```

где *column_name* — имя столбца, *value* — значение соответствующего поля добавляемой записи.

В списке столбцов указываются имена столбцов, для которых указываются значения добавляемой записи, в произвольном порядке. При этом, количество, порядок и типы значений должны соответствовать количеству, порядку и типам столбцов. Если список столбцов не указан, значения полей добавляемой записи должны быть порядке следования столбцов в таблице.

Для тех столбцов, которые не указаны в тексте запроса, значения полей формируются следующим образом:

1) если для столбца таблицы задано ограничение *IDENTITY*, генерируется новое значение поля;
2) если для столбца таблицы задано ограничение *DEFAULT*, используется значение по умолчанию для столбца;
3) если столбец таблицы допускает пустые значения, используется значение *NULL*;
4) в противном случае — операция добавления записи завершается ошибкой.

В результате запроса в таблицу добавляется одна или несколько строк с соответствующими значениями столбцов.

Изменение данных в таблице

Для изменения полей существующих записей таблицы базы данных используется следующий запрос:

```
UPDATE table_name
SET column_name = new_value[, column_name = new_value, ...]
[WHERE condition]
```

где *new_value* — новое значение поля, *condition* — условие выборки записей, данные в которых требуется изменить.

Новое значение может являться константным значением или выражением, которое может использовать старое значение полей записи. Запрос изменяет данные только тех строк таблицы, которые соответствуют заданному условию. Если условие не задано, изменения применяются ко всем строкам таблицы.

Удаление записей из таблицы

Для удаления существующих записей из таблицы базы данных используется следующий запрос:

```
DELETE [FROM] table_name
[WHERE condition]
```

где *condition* — условие выборки записей, которые требуется удалить.

Если условие не задано, удаляются все строки указанной таблицы.

Задание к работе

1. Составить SQL-запросы для создания структуры базы данных, полученной в результате лабораторной работы №1. Указать используемые типы данных, ограничения значений полей; для связей: действия с записями подчинённой таблицы при удалении и изменении соответствующей записи главной таблицы.
2. С помощью SQL-запросов выполнить добавление 3–4 записей в каждую таблицу, изменение и удаление нескольких записей.

Содержание отчёта

1. Листинги SQL-запросов для создания новой базы данных, содержащей все таблицы и связи.
2. Описание используемых ограничений в каждой таблице.

Лабораторная работа № 3
Средства языка SQL для выборки данных

Цель работы: изучить основные принципы создания SQL-запросов для выборки данных из таблиц базы данных и представления данных в требуемом виде.

Основные теоретические сведения

Одной из основных задач, выполняемых на сервере баз данных, является выборка данных, их первичная обработка и представление в удобном виде.

Для получения данных из таблиц базы данных используются SELECT-запросы, которые состоят из нескольких секций, большинство из которых могут отсутствовать.

SELECT-запросы в общем случае имеют следующий вид:

```
SELECT [TOP top_N] [DISTINCT] field_value[, field_value, ...]
[FROM table_source]
[WHERE condition]
[GROUP BY field_name[, field_name, ...]
[HAVING condition]]
[ORDER BY field_name [sort_type][, field_name [sort_type], ...]
```

Фактический порядок выполнения секций SELECT-запроса отличается от порядка их следования в тексте запроса

Секция FROM

В секции *FROM* указывается таблица данных, из которых выбирается необходимая информация. Это может быть физическая таблица базы данных, *соединение* нескольких таблиц или *табличное выражение* (будет рассмотрено в разделе «Вложенные подзапросы»).

В языке SQL существуют операции *соединения* двух таблиц, которые имеют следующий вид:

```
table_1 [[AS] alias_1] [join_type] table_2 [[AS] alias_2]
  [ON condition]
```

где *table_1* и *table_2* — таблицы или табличные структуры, *alias_1* и *alias_2* — псевдонимы таблиц, которые будут использованы в текущем запросе вместо имени таблицы или в качестве имени табличного выражения, если они указаны, *join_type* — тип соединения, *condition* — условие соединения таблиц. Результатом операции соединения двух таблиц является набор данных — табличная структура, содержащая все столбцы обеих таблиц.

Псевдонимы таблиц обязательны для табличных выражений, а также в случае, если выполняется операция соединения таблицы данных с этой же таблицей — для устранения неоднозначности их использования. В остальных случаях использование коротких имён в качестве псевдонимов позволяет сократить объём текста SQL-запроса.

Существуют следующие типы соединения таблиц:

- *CROSS JOIN* (перекрёстное соединение) — выполняется декартово произведение строк таблиц. Размер результирующего набора данных равен произведению количества строк заданных таблиц;
- *INNER JOIN* (внутреннее соединение) — в результирующий набор данных попадают пары строк таблиц, для которых выполняется условие соединения таблиц. Результирующий набор может не содержать строк;
- *LEFT JOIN* или *LEFT OUTER JOIN* («левое» соединение) — в результирующий набор данных попадают пары строк таблиц, для которых выполняется условие соединения таблиц, а также строки таблицы *table_1*, которым не найдено соответствующих по условию строк из таблицы *table_2*. В последнем случае в этих строках всем полям, соответствующих столбцам таблицы *table_2*, будет присвоено значение *NULL*. Результирующий набор гарантированно содержит все строки таблицы *table_1*;

- *RIGHT JOIN* или *RIGHT OUTER JOIN* («правое» соединение) — в результирующий набор данных попадают пары строк таблиц, для которых выполняется условие соединения таблиц, а также строки таблицы *table_2*, которым не найдено соответствующих по условию строк из таблицы *table_1*. В последнем случае в этих строках всем полям, соответствующих столбцам таблицы *table_1*, будет присвоено значение *NULL*. Результирующий набор гарантированно содержит все строки таблицы *table_2*;
- *FULL JOIN* или *FULL OUTER JOIN* (полное соединение) — в результирующий набор данных попадают пары строк таблиц, для которых выполняется условие соединения таблиц, а также строки таблицы *table_1*, которым не найдено соответствующих по условию строк из таблицы *table_2*, и строки таблицы *table_2*, которым не найдено соответствующих по условию строк из таблицы *table_1*. Результирующий набор аналогичен объединению результатов выполнения «левого» и «правого» соединения заданных таблиц.

Перекрёстное соединение является единственным типом соединения, для которого не задаётся секция *ON*. Для всех остальных типов условие соединения таблиц является обязательным.

В результате выполнения секции *FROM* создаётся временная таблица данных, в которой имена и типы столбцов соответствуют столбцам заданных таблиц и табличных структур, а строки представляют результирующий набор данных.

Секция WHERE

В секции *WHERE* указывается условие выборки данных. Каждая строка таблицы, полученной в секции *FROM*, проверяется на соответствие условию независимо от других строк. В результате выполнения секции *WHERE* из результирующего набора данных удаляются строки, для которых не выполняется условие выборки данных. Если секция *WHERE* не задана, все полученные на предыдущем этапе строки сохраняются.

Секция GROUP BY

Секция *GROUP BY* группирует выбранный набор данных для получения набора сводных строк по значениям одного или нескольких столбцов. В одну группу объединяются строки набора данных с одинаковыми значениями полей, столбцы которых перечислены в секции *GROUP BY*. В результирующий набор данных заносится по одной строке для каждой группы.

Секция HAVING

В секции *HAVING* указывается условие выборки результата группировки, указанной в секции *GROUP BY*. Каждая группа, проверяется на соответствие условию независимо от других групп. Условие секции *HAVING* обычно содержит *агрегатные функции* (будут рассмотрены в разделе «Агрегатные функции»). В результате выполнения секции *HAVING* из результирующего набора данных удаляются строки для групп, для которых не выполняется условие.

Секция SELECT

В секции *SELECT* содержится описание столбцов результирующей выборки данных в указанном порядке. Синтаксис описания столбца следующий:

```
column_value [[AS] column_name]
```

где *column_value* — выражение, определяющее значение поля, *column_name* — имя столбца.

Выражение, в соответствии с которым формируются значения поля столбца, может содержать константные значения, имена столбцов таблицы из секции *FROM*, вызовы функций, значения которых будут вычисляться для каждой записи, агрегатные функции. Если имя столбца содержится в нескольких таблицах или табличных выражениях в секции *FROM*, перед именем столбца обязательно указывать имя или псевдоним соответствующей таблицы или табличного выражения.

Если требуется включить все столбцы результирующей выборки, можно использовать символ «*». Если требуется включить все поля только одной таблицы или табличного выражения из секции *FROM*, то перед символом «*» нужно указать имя или псевдоним соответствующей таблицы или табличного выражения.

Если в *column_value* указано имя столбца таблицы из секции *FROM*, это же имя будет назначено данному столбцу в результирующем наборе данных, а *column_name* позволяет задать столбцу другое имя. В остальных случаях, если не указано *column_name*, столбец не имеет имени.

Имена столбцов результирующей выборки данных могут повторяться. Если требуется обеспечить их уникальность, необходимо задавать для столбцов уникальные значения *column_name*.

Если в секции *SELECT* указано ключевое слово *DISTINCT*, из результирующей выборки данных удаляются все одинаковые строки. Далее выполняется секция *ORDER BY*. Если указана секция TOP, из

результирующей выборки данных остаётся выбираются не более *top_N* строк, а все остальные строки удаляются.

Секция ORDER BY

В секции *ORDER BY* указываются столбцы секции *SELECT* или *FROM*, по которым выполняется сортировка. Если указано несколько столбцов, то сначала выполняется сортировка по первому столбцу, затем группы записей с одинаковыми значениями соответствующих полей упорядочиваются по значениям второго поля, затем группы записей с одинаковыми значениями полей первых двух столбцов упорядочиваются по значениям третьего столбца и т.д.

Для каждого столбца указывается тип сортировки: *ASC* — по неубыванию значений, *DESC* — по невозрастанию значений. Если тип не указан, используется значение *ASC*.

Агрегатные функции

Агрегатная функция — это статистическая функция, которая принимает в качестве входного параметра имя столбца или выражения, выполняет вычисление на наборе значений и возвращает одиночное значение в качестве результата.

В языке SQL определены следующие основные агрегатные функции:

- *COUNT* — возвращает количество непустых значений. Если перед именем столбца или выражением указать ключевое слово *DISTINCT*, возвращается количество уникальных значений. Если в качестве аргумента указан символ «*», возвращается количество записей;
- *SUM* — возвращает сумму непустых значений. Если среди значений нет ни одного непустого, функция возвращает *NULL*. Если перед именем столбца или выражением указать ключевое слово *DISTINCT*, возвращается сумма уникальных значений. Функция *SUM* может быть использована только для числовых значений;
- *AVG* — возвращает среднее значение непустых значений. Если среди значений нет ни одного непустого, функция возвращает *NULL*. Если перед именем столбца или выражением указать ключевое слово *DISTINCT*, возвращается среднее значение уникальных значений. Функция *AVG* может быть использована только для числовых значений;
- *MIN* — возвращает минимальное значение. Если среди значений нет ни одного непустого, функция возвращает *NULL*;

– *MAX* — возвращает минимальное значение. Если среди значений нет ни одного непустого, функция возвращает *NULL*.

Агрегатные функции могут использоваться в секциях *SELECT* и *HAVING*. Если в секции *SELECT* указаны только агрегатные функции, результирующий набор данных содержит одну строку.

Если в секции *SELECT* встречаются как агрегатные функции, так и столбцы таблицы секции *FROM*, секция *GROUP BY* обязательна, и агрегатные функции вычисляются для каждой группы.

Вложенные подзапросы

Один SELECT-запрос может быть помещён в другой SELECT-запрос. В этом случае первый запрос является вложенным, а второй — главным. Вложенные подзапросы заключаются в круглые скобки.

Вложенные подзапросы можно разделить на три типа:

– скалярные подзапросы — возвращают одно значение, могут использоваться в любом месте главного запроса, где ожидается простое значение соответствующего типа данных;
– векторные подзапросы — возвращают табличное выражение, которое состоит из одного столбца. Его называют *вектором данных*. Векторные подзапросы могут использоваться в операции *IN*, которая проверяет наличие значения в массиве (векторе) данных;
– табличные подзапросы — возвращают табличное выражение, содержащее произвольное количество строк и столбцов. Могут использоваться в любом месте главного запроса, где ожидается таблица или табличное выражение.

Столбцы табличного подзапроса должны иметь уникальные имена, тип столбцов определяется по значениям полей записей.

Вложенные подзапросы могут использовать столбцы таблиц из главного запроса. Доступ к ним осуществляется указанием во вложенном подзапросе имён или псевдонимов таблиц главного запроса.

Для временного хранения результатов вложенных подзапросов сервер баз данных выделяет дополнительную память и другие ресурсы, что оказывает влияние на производительность системы. Поэтому вложенные подзапросы рекомендуется использовать только в том случае, когда их нельзя заменить другими средствами языка SQL. Например, в некоторых случаях вместо вложенного подзапроса можно использовать операцию соединения таблиц, что будет выполняться быстрее и эффективнее.

Объединение результатов нескольких SELECT-запросов

В некоторых случаях требуется выбрать данные из разных таблиц или других источников данных, имеющих одинаковую структуру, и объединить все строки полученных результатов в один результирующий набор данных. В этом случае записи можно рассматривать как элементы множеств.

Из двух SELECT-запросов можно получить один результирующий набор данных, указав между ними одну из следующих операций:

- *UNION* — в результирующий набор данных попадают все записи из обоих запросов. При этом, из результатов исключаются записи результатов второго запроса, которые есть в результатах первого запроса;
- *UNION ALL* — в результирующий набор данных попадают все записи из обоих запросов. Количество записей результата объединения равно сумме количеств записей каждого запроса по отдельности;
- *EXCEPT* — в результирующий набор данных попадают все записи из первого запроса, которых нет во втором запросе;
- *INTERSECT* — в результирующий набор данных попадают те записи, которые есть как в первом, так и во втором запросе.

Результатом объединения запросов является табличное выражение, которое также может участвовать в другой операции объединения результатов SELECT-запросов. Поскольку в этом случае не используются вложенные подзапросы, SELECT-запросы не заключаются в круглые скобки.

SELECT-запросы, участвующие в операции объединения результатов запросов, должны соответствовать следующим требованиям:

- количество, типы и порядок столбцов должны совпадать в обоих запросах;
- имена столбцов результирующего набора данных определяются в первом запросе;
- сортировка результатов применяется только ко всему результирующему набору данных, а не к отдельным запросам, поэтому секция *ORDER BY* может быть использована только после последнего SELECT-запроса.

Задание к работе

1. Решить задачи на составление SELECT-запросов, предложенные преподавателем.

2. Составить различные SELECT-запросы к своей базе данных, созданной в результате лабораторной работы №2. Включить в отчёт формулировки задач, которые решаются с помощью полученных SQL-запросов.

Лабораторная работа № 4
Использование базы данных в приложениях .NET Framework

Цель работы: изучить программные средства платформы Microsoft .NET Framework для использования баз данных в приложениях.

Основные теоретические сведения

Независимо от вида приложений, создание которых поддерживает платформа Microsoft .NET Framework, организацию их доступа к базам данных отвечают одни и те же классы, предназначенные для взаимодействия с конкретной СУБД.

В состав платформы входят классы для поддержки Microsoft SQL Server. Для использования других СУБД можно использовать технологию ODBC или подключить к проекту приложения соответствующие библиотеки, которые имеют схожую структуру и принципы использования.

Классы для доступа к Microsoft SQL Server находятся в пространстве имён *System.Data.SqlClient*. Для их использования требуется создать объект (экземпляр) класса, присвоить необходимые значения свойствам, после чего вызвать соответствующие методы. Для удобства значения некоторых свойств можно задать в одной из перегрузок конструктора класса.

Если необходим доступ к полям, использующим тип данных, не имеющий соответствия среди классов .NET Framework, соответствующие типы описаны в пространстве имён *System.Data.SqlTypes*. Также некоторые методы используют классы из пространства имён *System.Data*.

Подключение к серверу баз данных

Для подключения к серверу баз данных используется *строка подключения*, которая представляет собой набор секций вида «параметр=значение», разделённых символом «;».

Основные параметры для Microsoft SQL Server:

- *Data Source* — имя или адрес сервера баз данных;
- *User ID* — имя пользователя;
- *Password* — пароль;
- *Initial Catalog* — база данных по умолчанию.

Если требуется использовать системные данные текущего пользователя, то вместо имени пользователя и пароля задаётся параметр *Integrated Security* со значением *true*.

Пример строки подключения:

```
string connectionString = "Data Source=DBServer;"
    + "Initial Catalog=MyDatabase;Integrated Security=False;"
    + "User ID=sa;Password=masterkey";
```

Строку подключения можно создать с помощью объекта *SqlConnectionStringBuilder*, задав необходимые значения одноимённым свойствам. Сформированная строка подключения будет содержаться в свойстве *ConnectionString*:

```
SqlConnectionStringBuilder csb = new SqlConnectionStringBuilder();
csb.DataSource = "DBServer";
csb.InitialCatalog = "MyDatabase";
csb.UserID = "sa";
csb.Password = "masterkey";

string connectionString = csb.ConnectionString;
```

Для возможности изменения параметров целевого сервера баз данных без перекомпиляции приложения удобно размещать строку подключения в файле конфигурации *app.config*. Для этого предназначен раздел *connectionStrings*, в котором размещаются инструкции по управлению коллекцией строк подключения. Каждая строка подключения должна иметь уникальное имя. Код для добавления строки подключения с именем *MyConnectionString* выглядит так:

```
<configuration>
  <connectionStrings>
    <add name="MyConnectionString"
         connectionString="Data Source=DBServer;
             Initial Catalog=MyDatabase;Integrated Security=False;
             User ID=sa;Password=masterkey" />
  </connectionStrings>
</configuration>
```

В коде приложения для доступа к коллекции строк подключения используется класс *ConfigurationManager*, который находится в сборке *System.Configuration* в одноимённом пространстве имён. Этот класс содержит статической свойство *ConnectionStrings*, которое является именованной коллекцией объектов *ConnectionStringSettings*. Доступ к этим объектам осуществляется по индексу или по имени. Строка подключения доступна в свойстве *ConnectionString*:

```
string connectionString = ConfigurationManager
    .ConnectionStrings["MyConnectionString"].ConnectionString;
```

Для установления связи с сервером баз данных используется класс *SqlConnection*. Строка подключения задаётся в свойстве *ConnectionString*. Метод *Open* устанавливает подключение к серверу баз данных, метод *Close* закрывает подключение. Любое взаимодействие с базой данных — отправка команд и получение результатов, должно выполняться при открытом подключении к серверу баз данных:

```
SqlConnection conn = new SqlConnection(connectionString);
conn.Open();

// вызов SQL-запросов и обработка результатов

conn.Close();
```

В процессе работы с базой данных возможно возникновение исключительных ситуаций, которые необходимо обрабатывать в приложении. В этом случае необходимо гарантировать, чтобы метод Close был вызван для закрытия установленного подключения. В языке C# удобно использовать синтаксическую конструкцию *using*, которая уничтожает указанный в ней объект и освобождает выделенные ресурсы во всех случаях, когда выполнение кода выходит за пределы выделенного блока:

```
using (SqlConnection conn = new SqlConnection(connectionString))
{
    conn.Open();

    // вызов SQL-запросов и обработка результатов
}
```

Здесь вызов метода *Close* происходит неявно: при выходе выполнения программы за пределы блока *using* для объекта *conn* вызывается метод *Dispose*, который в свою очередь вызывает метод *Close*, если подключение к серверу баз данных было установлено. Блок *using* можно создавать для объектов, созданных на основе классов, реализующих интерфейс *IDisposable*.

Вызов SQL-запросов к базе данных

Для выполнения SQL-запросов к базе данных используется класс *SqlCommand*, который содержит следующие основные свойства:

- *Connection* — объект *SqlConnection*, который выполняет подключение к серверу базы данных;
- *CommandType* — определяет, как будет интерпретироваться значение свойства *CommandText*. Для выполнения SQL-запроса используется значение *CommandType.Text*, которое является значением по умолчанию;
- *CommandText* — текст SQL-запроса;
- *Parameters* — коллекция параметров, используемых в SQL-запросе.

SQL-запрос может быть задан в одной строке или в нескольких строках. Если в запросе используются параметры, их имена начинаются с символа «@». Перед выполнением запроса для всех параметров, указанных в тексте запроса, в коллекцию параметров должны быть добавлены соответствующие объекта класса *SqlParameter*, который содержит следующие основные свойства:

- *ParameterName* — имя параметра;
- *DbType* — тип данных .NET Framework;
- *SqlDbType* — тип данных Microsoft SQL Server;
- *IsNullable* — может ли значение параметра быть пустым. По умолчанию содержит значение *false*;
- *Value* — значение параметра одного из типов данных .NET Framework;
- *SqlValue* — значение параметра одного из типов данных Microsoft SQL Server;
- *Direction* — определяет, является параметр входным и/или выходным. Для параметризованных SQL-запросов значением свойства является значение по умолчанию: *ParameterDirection.Input*.

Свойства *DbType* и *SqlDbType* связаны друг с другом: при изменении значения одного из этих свойств соответствующим образом изменится значение другого свойства. Аналогичным образом связаны свойства *Value* и *SqlValue*. Если присвоить свойству *Value* или *SqlValue* значение, не соответствующее типу параметра, также автоматически изменятся значения свойств *DbType* и *SqlDbType*.

Выполнение SQL-запроса осуществляется вызовом одного из следующих методов:

- *ExecuteNonQuery* — предназначен для выполнения SQL-запросов, не возвращающих данные. К таким запросам относятся инструкции DDL и DML. Для запросов DML метод возвращает количество обработанных строк данных;

- *ExecuteScalar* — предназначен для выполнения SELECT-запросов, возвращающих одно значение (скалярные запросы). Возвращает значение первого поля первой записи результирующего набора данных, все остальные строки и поля игнорируются;
- *ExecuteReader* — предназначен для выполнения SELECT-запросов, возвращающих произвольную таблицу данных. Возвращает объект *SqlDataReader*, который осуществляет доступ к полученным данным.

При вызове метода *ExecuteReader* данные из базы не передаются в приложение все сразу. При создании объекта *SqlDataReader* создаётся *программный курсор*, который позволяет перемещаться между записями результирующего набора данных в прямом направлении.

Объект *SqlDataReader* реализует интерфейс *IDisposable* и содержит следующие основные свойства и методы:

- *HasRows* — свойство, имеет значение *true*, если в результирующем наборе данных есть хотя бы одна строка, или значение *false*, если результат не содержит строк;
- *Read* — метод, выполняет переход к следующей записи результирующего набора данных. При первом вызове переходит к первой записи. Если переход выполнен, метод возвращает значение *true*, если следующей записи нет — значение *false*;
- *GetXXX* — группа методов, возвращающих для текущей записи значение поля, номер которого передаётся в качестве аргумента. Первое поле имеет номер 0. Каждый из методов возвращает значение, приведённое к соответствующему типу. Если значение поля пустое, генерируется исключение;
- *IsDBNull* — метод, возвращает значение *true*, если для текущей записи значение поля, номер которого передаётся в качестве аргумента, пустое, иначе — значение *false*;
- *NextResult* — метод, выполняет переход к следующему результирующему набору данных. Если в результате запроса возвращается один результирующий набор данных, вызывать данный метод не требуется. Если переход выполнен, метод возвращает значение *true*, если следующего набора данных нет — значение *false*.

Также доступ к значению поля текущей записи можно получить с помощью индексатора. В качестве индекса можно использовать номер (начинается с нуля) или имя столбца результирующего набора данных (если оно задано). Полученный результат необходимо привести к со-

ответствующему типу данных. Если значение поля пустое, оно равно *DBNull.Value*.

В следующем примере выполняется получение данных из таблицы Books, содержащей столбцы Title, Pages, SubjectId, и таблицы Subjects, содержащей столбцы Id, Name. Выбираются только те записи, у которых значение поля Year таблицы Books больше 2000.

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
cmd.CommandText = @"SELECT b.Title, b.Pages, s.Name AS Subject
FROM Books b INNER JOIN Subjects s ON b.SubjectId = s.Id
WHERE Year > @year";
cmd.Parameters.Add(new SqlParameter("@year", 2000));

using (SqlDataReader reader = cmd.ExecuteReader())
{
    if (reader.HasRows)
    {
        while (reader.Read())
        {
            string t = reader.GetString(0);
            int p = reader.IsDBNull(1) ? reader.GetInt32(1) : 0;
            string s = reader["Subject"] == DBNull.Value
                ? (string)reader[2] : "Другие книги";

            Console.WriteLine($"{s}: \"{t}\" ({p} страниц)");
        }
    }
    else
        Console.WriteLine("Запрос вернул пустой набор данных");
}
```

Приложения Windows Forms для работы с базой данных

При создании приложений с оконным интерфейсом в первую очередь необходимо решить, в какой момент будет устанавливаться подключение к серверу баз данных, и в какой момент оно должно закрываться. Существуют следующие основные стратегии:

– постоянное подключение — подключение устанавливается при запуске приложения и закрывается при завершении работы. В этом случае объект *SqlConnection* должен быть доступен во всём коде приложения. При аварийном закрытии подключения необходимо предусмотреть возможность повторного подключения к серверу баз данных и блокировать выполнение всех запросов к нему до тех пор, пока подключение не будет восстановлено.

Данная стратегия применяется, если приложение должно постоянно вызывать большое количество запросов к базе данных;

- подключение по запросу — подключение устанавливается при попытке выполнить запрос к базе данных и закрывается сразу же по окончании получения результатов. Данная стратегия применяется, если приложение обращается к базе данных сравнительно редко, что позволяет сэкономить ресурсы приложения и устраняет необходимость отслеживать аварийное закрытие подключения между запросами;
- поддержание активного подключения по запросу — подключение устанавливается при попытке выполнить запрос к базе данных и некоторое время остаётся активным. Следующие запросы могут использовать это подключение вместо создания нового, что позволяет сэкономить ресурсы приложения. Подключение закрывается по истечении определённого интервала времени после выполнения последнего запроса. В случае аварийного закрытия подключения его можно не восстанавливать до выполнения следующего запроса. Данная стратегия применяется, если приложение периодически выполняет запросы к базе данных, но в течение продолжительных промежутков времени обращение к базе данных не происходит;
- работа с данными в отсоединённой среде — приложение загружает из базы данных в память всю необходимую информацию, после чего подключение закрывается. Далее приложение использует загруженные данные и фиксирует все сделанные изменения в специальный журнал. При следующем подключении к серверу баз данных приложение выполняет все произведённые изменения к базе данных и синхронизирует данные в памяти с базой данных. Данная стратегия применяется, если ограничена возможность поддержания активного подключения к базе данных, и вероятность модификации одних и тех же данных несколькими пользователями низкая или такая ситуация не является критичной.

В приложениях Windows Forms для ввода информации пользователем и вывода полученных после выполнения запроса данных можно использовать стандартные элементы пользовательского интерфейса, используя техники, описанные в предыдущем разделе.

Для вывода данных в виде таблицы предназначен элемент *DataGridView*, который находится в разделе *Данные* панели элементов Visual Studio. В свойстве *Columns* хранится информация о столбцах таблицы, в свойстве *Rows* — строки с данными. Если структура таблицы

не будет изменяться во время работы приложения, создать столбцы можно с помощью встроенного в Visual Studio редактора столбцов.

Работа с данными в отсоединённой среде

При реализации стратегии, которая предполагает хранение данных в памяти приложения, базовым является элемент источника данных *DataSet*, который находится в пространстве имён *System.Data*. Элемент *DataSet* позволяет создать автономную структуру базы данных, включая таблицы и их взаимосвязи, путём создания и добавления объектов следующих классов:

- *DataTable* — представляет таблицу данных в памяти. Объекты этого класса размещаются в свойстве *Tables* объекта *DataSet*;
- *DataColumn* — представляет описание столбца таблицы данных. Объекты этого класса размещаются в свойстве *Columns* объекта *DataTable*;
- *DataRow* — представляет строку таблицы данных. Объекты этого класса размещаются в свойстве *Rows* объекта *DataTable*;
- *Constraint* — представляет описание ограничений ссылочной целостности для таблиц данных. Объекты этого класса размещаются в свойстве *Constraints* объекта *DataTable*. От класса *Constraint* наследуются классы *ForeignKeyConstraint* (описывает внешний ключ для столбца) и *UniqueConstraint* (описывает ограничение уникальности значений столбца или группы столбцов).

Указанные классы используют типы данных .NET Framework, и их можно использовать отдельно от СУБД.

Для синхронизации данных в объекте *DataSet* с соответствующими таблицами базы данных используется класс *SqlDataAdapter*, который содержит следующие методы:

- *Fill* — добавляет или обновляет таблицы источника данных, который передаётся в качестве параметра, данными из базы данных, полученными с помощью объекта *SqlCommand*, указанного в свойстве *SelectCommand*;
- *Update* — выполняет изменения, сделанные в источнике данных, к базе данных с помощью объектов *SqlCommand*, указанных в свойствах *InsertCommand*, *UpdateCommand* и *DeleteCommand*.

При вызове указанных методов, если подключение к базе данных, указанное в свойстве *Connection* соответствующего объекта *SqlCommand*, было открыто до вызова метода, по окончании его выполнения остаётся открытым, иначе — автоматически закрывается.

Если при вызове метода *Fill* в целевом источнике данных нет соответствующих таблиц, они автоматически создаются. Если нужно заполнить данными объекты источника данных, структура которого отличается от базы данных, необходимо задать сопоставление столбцов таблиц источника данных и столбцов таблиц базы данных с помощью объектов *DataTableMapping*, которые нужно добавить в коллекцию *TableMappings* объекта *SqlDataAdapter*.

Создание объекта источника данных можно выполнить с помощью мастера, который вызывается в пункте *Проект→Добавить новый источник данных…* главного меню Visual Studio. Мастер создаст типизированный объект-наследник класса *DataSet*, содержащий дополнительные свойства методы и внутренние типизированные объекты, которые облегчают доступ к данным из программного кода.

Привязка данных к элементам управления Windows Forms

Объекты *DataSet* могут быть использованы в качестве источника данных для других элементов пользовательского интерфейса Windows Forms. Связывание свойства элемента управления с таблицей источника данных называют ***привязкой данных***. Каждая таблица источника данных имеет программный курсор, который позволяет переходить от одной активной записи к другой. В этом случае, а также при изменении значений полей активной записи, автоматически изменяются значения свойств всех связанных с источником данных элементов управления.

Элемент *DataGridView* содержит следующие свойства для создания привязки данных:

- *DataSource* — источник данных, в качестве которого можно использовать объект *DataSet* или *BindingSource*;
- *DataMember* — имя таблицы данных, если в качестве источника данных выбран объект *DataSet*.

Другие элементы управления используют в качестве источника данных объект класса *BindingSource*, который выполняет привязку к таблице данных и содержит свойства *DataSource* и *DataMember*, аналогичные одноимённым свойствам класса *DataGridView*. Также класс *BindingSource* содержит свойства *Filter* и *Sort*, которые позволяют выполнить фильтрацию и сортировку данных, полученных из источника данных.

Привязка данных практически во всех стандартных элементах пользовательского интерфейса Windows Forms задаётся путём заполнения коллекции *DataBindings*. Это можно сделать как в программном коде на C#, так и средствами конструктора формы Visual Studio.

Для перехода между записями таблицы источника данных можно использовать элементы *DataGridView* и *BindingNavigator*. Также это можно сделать программно, используя методы *MoveFirst*, *MoveLast*, *MoveNext*, *MovePrevious* класса *BindingSource*, а также метод *PerformClick* объекта-значения свойства *MoveFirstItem*, *MoveLastItem*, *MoveNextItem*, *MovePreviousItem* класса *BindingNavigator*.

Задание к работе

1. Создать приложение Windows Forms, которое позволяет для каждой таблицы базы данных, созданной в лабораторной работе №2, добавлять, изменять и удалять записи.
2. Включить в отчёт описание используемых элементов Windows Forms.

Лабораторная работа № 5
Представления и хранимые процедуры

Цель работы: изучить основные возможности и средства создания представлений и хранимых процедур и их использования в приложениях .NET Framework.

Основные теоретические сведения

Представление (*view*) — это объект базы данных, который создаётся на основе одной или нескольких базовых таблиц или других представлений. Фактически представление является именованным SELECT-запросом, и его можно использовать в качестве таблицы в других SQL-запросах.

При вызове представления выполняется соответствующий SELECT-запрос, который возвращает таблицу данных из физических таблиц, указанных в тексте запроса, и представляет данные в виде табличного набора данных. Поэтому представления также называют *виртуальной таблицей*.

Хранимая процедура (*stored procedure*) — это объект базы данных, который содержит последовательность инструкций на языке SQL и процедурных расширений. В Microsoft SQL Server процедурные расширения реализованы в языке Transact-SQL, который не является частью стандарта SQL и является ориентированным на данную СУБД.

Хранимые процедуры является аналогом подпрограмм в языках программирования высокого уровня. Они позволяют выполнять на сервере базы данных алгоритмические действия, могут иметь входные и выходные параметры, также могут возвращать один или несколько результирующих наборов данных.

Аналогично таблицам, представления и хранимые процедуры создаются в схеме базы данных, имя которой указывается перед именем соответствующего объекта. Если схема не указана, используется схема по умолчанию.

Создание и модификация представления

Для создания представления в базе данных используется следующий запрос:

```
CREATE VIEW view_name
AS select_statement
```

где *view_name* — имя представления, *select_statement* — SELECT-запрос для выборки данных.

Поскольку результат представления рассматривается как таблица, для SELECT-запроса предъявляются требования для табличных подзапросов. Таблицы базы данных в SELECT-запросе называют *базовыми таблицами* представления.

Представления можно использовать как таблицу в любых запросах выборки данных. Получение данных представления аналогично вызову вложенного подзапроса.

Некоторые представления могут использоваться для вставки, изменения и удаления данных соответствующей таблицы. Такие представления называются *обновляемыми*, и они должны соответствовать следующим требованиям:

- изменения должны ссылаться на столбцы только одной базовой таблицы;
- представление не должно содержать агрегатные функции и вычисляемые столбцы;
- для вставки данных столбцы базовой таблицы, которые не указаны в представлении, должны иметь возможность генерации значения полей.

Изменение существующего представления выполняется с помощью следующего запроса:

```
ALTER VIEW view_name
AS select_statement
```

Удаление из базы данных созданного ранее представления выполняется с помощью запроса:

```
DROP VIEW view_name
```

Создание и модификация хранимой процедуры

Для создания хранимой процедуры в базе данных используется следующий запрос:

```
CREATE PROCEDURE proc_name [param[, param, ...]]
AS
BEGIN
...
END
```

где *proc_name* — имя хранимой процедуры, *param* — описание параметра.

Описание параметра имеет следующий вид:

```
@param_name data_type [= default] [OUTPUT]
```

где *param_name* — имя параметра, *data_type* — тип данных, *default* — значение по умолчанию, которое будет использоваться, если параметр не указан при вызове хранимой процедуры.

Если для параметра указано ключевое слово *OUTPUT*, он является *выходным*. Выходные параметры используются для возврата значений в вызвавший хранимую процедуру код.

Изменение существующей хранимой процедуры выполняется с помощью следующего запроса:

```
ALTER PROCEDURE proc_name
AS
BEGIN
...
END
```

Удаление из базы данных созданной ранее хранимой процедуры выполняется с помощью запроса:

```
DROP PROCEDURE proc_name
```

Основные инструкции Transact-SQL

Вызов хранимой процедуры можно выполнить с помощью следующей инструкции Transact-SQL:

```
EXECUTE proc_name [[param = ]value [OUTPUT], ...]
```

где *value* — значение параметра.

Ключевое слово *OUTPUT* задаётся для выходных параметров. В этом случае в качестве значения параметра должна передаваться переменная. Для входных параметров значением может быть произвольное выражение. Имя параметра указывается или не указывается для всех параметров. В последнем случае значения параметров передаются строго в порядке их следования при создании хранимой процедуры.

Помимо SQL-запросов, в хранимой процедуре часто используют следующие инструкции Transact-SQL:

– *DECLARE* — объявление переменной:

```
DECLARE @var_name data_type [= default]
```

где *var_name* — имя переменной;

– *SET* — присваивание значения переменной:

```
SET @var_name = value
```

где *value* — выражение, значение которого присваивается переменной. Также для присваивания переменной значения можно использовать SELECT-запрос:

```
SELECT @var_name = value [FROM ...]
```

В этом случае выполнение присваивания переменной значения выполняется для каждой строки результирующего набора данных, а сам результирующий набор данных не возвращается;

– *IF* — оператор ветвления:

```
IF condition
    operator1
[ELSE
    operator2]
```

где *condition* — логическое условие, *operator1* — инструкция, которая выполняется при выполнении условия, *operator2* — инструкция, которая выполняется при невыполнении условия;

– *WHILE* — оператор цикла:

```
WHILE condition
    operator
```

где *condition* — условие выполнения следующего цикла, *operator* — инструкция, которая выполняется в цикле.

Если в инструкциях *IF* и *WHILE* необходимо выполнить несколько инструкций, их нужно заключить в операторные скобки *BEGIN..END*.

Временные таблицы и табличные типы данных

В хранимой процедуре можно создавать таблицы данных, которые располагаются в памяти сервера баз данных, доступны только внутри хранимой процедуры и автоматически удаляются при завершении выполнения хранимой процедуры. Такие таблицы называются *временными*. Имя временной таблицы начинается с символа «#».

Для работы с временными таблицами можно использовать любые запросы и инструкции, предназначенные для работы с физическими таблицами базы данных.

Кроме этого, можно создавать переменные с типами данных, которые являются таблицами с заданным набором столбцов. Синтаксис объявления переменной табличного типа следующий:

```
DECLARE @var_name TABLE (column[, column, ...])
```

где *var_name* — имя переменной, *column* — описание столбца, синтаксис которого аналогичен описанию столбца при создании физических таблиц.

Переменные табличного типа, как и временные таблицы, доступны только в текущей хранимой процедуре до окончания её выполнения.

Табличные типы данных можно сохранить в виде объекта базы данных, используя следующий синтаксис:

```
CREATE TYPE type_name AS TABLE (column[, column, ...])
```

где *type_name* — имя типа данных, *column* — описание столбца, синтаксис которого аналогичен описанию столбца при создании физических таблиц.

Табличные типы данных помогают избежать дублирования описания столбцов таблицы для нескольких однотипных табличных переменных, а также позволяют передавать таблицы данных в качестве параметров хранимых процедур.

Табличные курсоры

Transact-SQL содержит тип данных *CURSOR* и инструкции, которые позволяют построчно обрабатывать результаты SELECT-запросов. Как правило, обработка данных выполняется в цикле.

Работа с программным курсором для табличного набора данных выполняется следующим образом:

1) создаётся курсор для указанного SELECT-запроса:

```
DECLARE cursor_name FOR select_statement
```

где *cursor_name* — имя курсора, *select_statement* — SELECT-запрос, по записям которого будет проходить курсор.

2) объявляются переменные, в которые будут записываться значения полей записей табличного набора данных, для которого создан курсор. Количество и тип переменных должны соответствовать количеству и типам столбцов результирующего набора данных SELECT-запроса;
3) открывается курсор и выполняется связанный с ним SELECT-запрос:

```
OPEN cursor_name
```

4) выполняется чтение первой записи и запись значений полей в переменные, подготовленные в пункте 2:

```
FETCH NEXT FROM cursor_name
INTO @var_name[, @var_name, ...]
```

5) проверяется результат последней операции чтения данных курсором: если запись считана успешно, значение глобальной переменной *@@FETCH_STATUS* равно 0. Эта проверка обычно выполняется в цикле:

```
WHILE @@FETCH_STATUS = 0
BEGIN
...
END
```

6) в цикле выполняется обработка полученных значений и чтение следующей записи аналогично пункту 4;
7) после завершения цикла курсор закрывается и освобождает все ресурсы:

```
CLOSE cursor_name
DEALLOCATE cursor_name
```

Использование хранимых процедур в приложениях .NET Framework

Для вызова хранимой процедуры в приложении .NET Framework используется объект *SqlCommand*, описанный в лабораторной работе №4, со следующими значениями свойств:

– свойству *CommandType* должно быть присвоено значение *CommandType.StoredProcedure*;

– в свойство *CommandText* вместо текста SQL-запроса заносится имя хранимой процедуры;
– в коллекцию *Parameters* добавляются все параметры хранимой процедуры. Для входных параметров задаются значения, для выходных параметров свойству *Direction* должно быть присвоено значение *ParameterDirection.Output*.

С объектом *SqlCommand*, настроенным на вызов хранимой процедуры, можно выполнять те же действия, что и при выполнении SQL-запросов.

Функции Transact-SQL

Хранимые процедуры обычно вызываются отдельно от других инструкций SQL и не могут использоваться непосредственно в качестве источника данных SELECT-запросов.

В Transact-SQL можно создать *функцию*, которая аналогична хранимой процедуре, но имеет ряд отличий:

– функцию можно использовать в других SQL-запросах;
– все параметры функции являются входными;
– функция возвращает только одно значение указанного типа данных.

Для создания функции в базе данных используется следующий запрос:

```
CREATE FUNCTION func_name ([param[, param, ...]])
RETURNS [@return_name] data_type
AS
BEGIN
...
END
```

где *func_name* — имя функции, *param* — описание параметра, *data_type* — тип возвращаемого значения, *return_name* — имя переменной, в которую заносится возвращаемое значение функции.

Описание параметров аналогично параметрам хранимой процедуры за исключением ключевого слова *OUTPUT*.

Возврат значения осуществляется одним из двух способов:

1) если указано имя переменной *return_name*, возвращаемое значение присваивается этой переменной;
2) возвращаемое значение указывается после ключевого слова *RETURN*.

Если функция возвращает таблицу данных, значением data_type должно быть ключевое слово TABLE или имя одного из табличных

типов данных. В остальных случаях функция возвращает одиночное значение.

В зависимости от типа функции её можно вызывать в SQL-запросах в том месте, где ожидается соответствующий тип данных. После имени функции должны следовать скобки, в которых перечисляются значения параметров, если они есть.

Изменение существующей функции выполняется с помощью следующего запроса:

```
ALTER FUNCTION func_name ([param[, param, ...]])
RETURNS [@return_name] data_type
AS
BEGIN
...
END
```

Удаление из базы данных созданной ранее хранимой процедуры выполняется с помощью запроса:

```
DROP FUNCTION func_name
```

Задание к работе

1. Создать два представления, выполняющие запрос данных из одной и нескольких таблиц.
2. Создать две хранимые процедуры для решения алгоритмических задач.
3. Добавить в приложение, созданное в лабораторной работе №4, возможность вызова созданных представлений и хранимых процедур с возможностью ввода значений параметров и вывода результатов.
4. Включить в отчёт описание созданных объектов базы данных и их использования в приложении .NET Framework.

Лабораторная работа № 6
Триггеры и транзакции

Цель работы: изучить основные возможности и средства создания триггеров и организации создания и управления транзакциями.

Основные теоретические сведения

Триггер (*trigger*) — это механизм, который вызывается при возникновении в заданной таблице определённых действий. По сути триггер

аналогичен хранимой процедуре, но вызывается не по запросу, а при возникновении события на сервере базы данных.

Триггеры можно создавать для таблиц перед или после возникновения события добавления, изменения и удаления записей.

Транзакция (*transaction*) — это механизм объединения нескольких операций с базой данных в единую атомарную логическую единицу. В рамках транзакции либо успешно выполняются все операции, либо при возникновении ошибки или исключения все изменения, сделанные с момента начала транзакции, отменяются.

Создание и модификация триггера

Для создания триггера в базе данных используется следующий запрос:

```
CREATE TRIGGER trigger_name
ON table_name
event_type
AS
BEGIN
...
END
```

где *trigger_name* — имя триггера, *table_name* — имя таблицы, *event_type* — событие, при возникновении которого должен срабатывать триггер.

Описание события состоит из двух секций:

1) одно из следующих ключевых слов: *FOR* — триггер срабатывает перед указанной во второй секции операцией модификации, *AFTER* — триггер срабатывает после указанной операцией модификации, *INSTEAD OF* — триггер срабатывает вместо указанной операцией модификации;
2) операция модификации указанной таблицы: *INSERT* — вставка записи, *UPDATE* — изменение записи, *DELETE* — удаление записи.

В триггере можно использовать те же инструкции Transact-SQL, что и в хранимой процедуре. Кроме этого, в триггерах доступны две особые таблицы:

– *inserted* — содержит копии строк, с которыми работали инструкция *INSERT* или *UPDATE*;
– *deleted* — содержит копии строк, с которыми работали инструкция *DELETE* или *UPDATE*.

Для отмены в триггере операции модификации данных соответствующей таблицы требуется выполнить одно их следующих действий:

- в триггере *FOR* выполнить инструкцию *ROLLBACK*;
- использовать триггер *INSTEAD OF*, в котором задать условие выполнения операции модификации.

Для одного типа события одной таблицы базы данных можно создать несколько триггеров. В этом случае при возникновении события выполняются последовательно все соответствующие триггеры. При этом, не гарантируется порядок выполнения триггеров.

Изменение существующего триггера выполняется с помощью следующего запроса:

```
ALTER TRIGGER trigger_name
ON table_name
event_type
AS
BEGIN
...
END
```

Удаление из базы данных созданного ранее триггера выполняется с помощью запроса:

```
DROP TRIGGER trigger_name
```

Создание и управление транзакцией

В Transact-SQL есть следующие инструкции по управлению транзакциями:

- *BEGIN TRANSACTION* — создание новой транзакции;
- *COMMIT TRANSACTION* — завершение транзакции с сохранением сделанных изменений;
- *ROLLBACK TRANSACTION* — завершение транзакции с отменой всех сделанных с момента начала транзакции изменений.

Все перечисленные инструкции могут принимать в качестве параметра имя транзакции или строковую переменную, содержащую имя транзакции. Если имя транзакции не задано, управлять можно только одной транзакцией.

В рамках транзакции выполняются все инструкции, вызываемые после команды *BEGIN TRANSACTION* и до вызова команды *COMMIT TRANSACTION* или *ROLLBACK TRANSACTION*. Если ни одна из последних команд не выполнена, по истечении определённого проме-

жутка времени автоматически вызывается команда *ROLLBACK TRANSACTION*, и все выполненные в рамках транзакции изменения отменяются.

При одновременном выполнении нескольких транзакций к одним и тем же данным необходимо задать требуемый уровень изоляции для каждой транзакции. ***Уровень изоляции*** задаёт степень защищённости данных, обрабатываемых в транзакции, от возможности изменения другими транзакциями.

Уровень изоляции транзакции задаётся следующей инструкцией:

```
SET TRANSACTION ISOLATION LEVEL isolation_level
```

где *isolation_level* — уровень изоляции транзакции.

В Microsoft SQL Server поддерживаются следующие уровни изоляции транзакций:

- *READ UNCOMMITED* — текущая транзакция может считывать изменения, которые были сделаны другими транзакциями, но ещё не были подтверждены;
- *READ COMMITED* — текущая транзакция не может считывать изменения, которые были сделаны другими транзакциями, но ещё не были подтверждены. Другие транзакции могут изменять данные, считанные текущей транзакцией, до её завершения;
- *REPEATABLE READ* — текущая транзакция не может считывать изменения, которые были сделаны другими транзакциями, но ещё не были подтверждены. Другие транзакции не могут изменять данные, считанные текущей транзакцией, до её завершения;
- *SNAPSHOT* — данные, считанные текущей транзакцией, будут согласованы с версией данных, существовавших в её начале. Транзакция видит только те изменения, которые были зафиксированы до её начала, и не видит изменений, сделанных после этого другими транзакциями;
- *SERIALIZABLE* — текущая транзакция не может считывать изменения, которые были сделаны другими транзакциями, но ещё не были подтверждены. Другие транзакции не могут изменять данные, считанные текущей транзакцией, до её завершения, а также не могут добавлять новые строки со значением ключа, которые входят в диапазон ключей, считываемых инструкциями текущей транзакции, до её завершения.

Установленный уровень изоляции продолжает действовать для текущего подключения к базе данных до тех пор, пока не будет явно изменён. Уровнем изоляции по умолчанию является *READ COMMITED*.

Управление транзакциями в приложениях .NET Framework

Для создания и управления транзакциями в приложении .NET Framework используется объект *SqlTransaction*, который создаётся вызовом метода *BeginTransaction* объекта *SqlConnection*. Метод *BeginTransaction* имеет несколько перегрузок, которые позволяют задавать имя и уровень изоляции транзакции.

Для того чтобы запрос к базе данных выполнялся в рамках транзакции, объект *SqlTransaction* необходимо указать в качестве значения свойства *Transaction* объекта *SqlCommand*.

Объект *SqlTransaction* содержит следующие методы для управления транзакцией:

- *Commit* — выполняет завершение транзакции с сохранением сделанных изменений;
- *Rollback* — выполняет завершение транзакции с отменой сделанных изменений.

Метод *Commit* следует вызывать после выполнения всех запросов в рамках транзакции для подтверждения сделанных изменений. Метод *Rollback* следует вызывать, если какая-либо команда, которая должна выполняться в рамках транзакции, завершается с ошибкой, а также при необходимости отменить сделанные в рамках транзакции изменения.

Задание к работе

1. Создать два триггера для таблиц базы данных, которые выполняют изменения в других таблицах.
2. Добавить в приложение, созданное в лабораторной работе №4, возможность выполнения нескольких операций модификации данных в рамках транзакции. Предусмотреть возможность отмены транзакции по решению пользователя или при возникновении определённой ситуации.
3. Включить в отчёт описание триггеров и транзакции с обоснованием выбранного уровня изоляции данных.

Лабораторная работа № 7
Формат XML для представления данных

Цель работы: изучить основные особенности использования формата XML для представления структурированных данных и программные средства для работы с данными в формате XML.

Основные теоретические сведения

XML (*eXtensible Markup Language*) — это язык с простым формальным синтаксисом, предназначенный для создания и обработки документов программами, а также для представления структурированных данных.

Формат XML является текстовым. Содержимое XML-файлов можно редактировать с помощью любого текстового редактора. Практически все современные технологии и языки программирования имеют библиотеки и классы, облегчающие обработку данных в формате XML с использованием объектно-ориентированного подхода.

XML-файлы состоят из *элементов*, которые могут вкладываться в другие элементы, образуя иерархию. Элементы состоят из *тегов*. Тегом называется идентификатор, заключённый в угловые скобки. Теги бывают следующих видов:

– открывающий:

```
<tag>
```

– закрывающий:

```
</tag>
```

– самозакрывающийся — является сокращённой формой записи рядом стоящих открывающего и закрывающего тегов с одинаковым именем:

```
<tag />
```

где *tag* — имя тега, которое является чувствительным к регистру символов.

Любой элемент состоит из открывающего и закрывающего тегов с одинаковым именем или одного самозакрывающегося тега. Имя тега является также именем элемента.

Между тегами одного элемента может содержаться текст или другие элементы. При этом, должна соблюдаться вложенность тегов: открывающий и закрывающий теги должны быть на одном уровне вложенности.

Если элемент содержит другие элементы, он является *составным*, иначе — *простым*. Элемент, состоящий из самозакрывающегося тега, аналогичен простому элементу, не имеющему значения между открывающим и закрывающим тегами.

Содержимое между тегами элемента имеет ряд особенностей:

– все рядом стоящие пробелы и другие символы-разделители трактуются как один пробел;
– начальные и конечные пустые символы игнорируются;
– некоторые значимые символы (не являются частью XML-разметки) являются *специальными* и должны заменяться символьными комбинациями.

В XML часто используются следующие основные комбинации:

– < — знак «меньше»;
– > — знак «больше»;
– & — амперсанд;
– ' — апостроф или одинарная кавычка;
– " — двойная кавычка;
– &#*value*; — символ, код которого задан десятичным значением *value*;
– &#x*value*; — символ, код которого задан шестнадцатеричным значением *value*.

Элементы также могут содержать *атрибуты* — наборы пар вида «ключ=значение». Атрибуты задаются в открывающем (или в самозакрывающемся) теге элемента после имени тега и разделяются любым количеством символов-разделителей. Имена атрибутов для каждого элемента должны быть уникальными. Значение атрибута является строкой, заключённой в кавычки. В значении атрибута вместо специальных символов должны использоваться символьные комбинации, аналогичные используемым в тексте элементов.

Данные в формате XML называются *XML-документом* и обычно хранятся в текстовом файле с расширением XML. Документ состоит из *декларации* и *корневого элемента*. Декларация не является обязательной, но, если она присутствует, она должна располагаться в самом начале. Пример декларации выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
```

где *version* — номер версии (на текущий момент существует только версия 1.0, однако в будущем она может измениться), *encoding* — кодировка символов в XML-файле (может отсутствовать), *standalone* — значение *yes*, если в XML-файле используются сведения из внешних источников, иначе — значение *no* (эта секция может отсутствовать).

Декларацию следует использовать, если в документе используется кодировка, отличная от Unicode (UTF-8 или UTF-16), для её правильного определения другими приложениями.

Документ должен обязательно содержать один корневой элемент. Другие элементы (если они есть) образуют иерархию внутри корневого элемента.

Представление табличной структуры в формате XML

Поскольку формат XML позволяет хранить структурированные данные, его можно использовать для хранения данных из базы данных. Это может быть полезным для обмена данными между различными приложениями, а также при необходимости перенести данные из одной базы данных в другую. При этом, целевая база данных может находиться на другом сервере баз данных, в том числе под управлением другой СУБД.

XML позволяет хранить данные всей базы данных или её части. Формат представления данных может быть различным.

В общем случае базу данных можно представить в виде иерархии:

- уровень 0 — база данных;
- уровень 1 — таблицы;
- уровень 2 — записи таблицы.

Универсальный формат данных, который позволяет хранить данные любой базы данных, может выглядеть следующим образом:

```
<Database name="MyDatabase">
  <Table name="Table1">
    <Row>
      <Column1>...</Column1>
      <Column2>...</Column2>
      ...
    </Row>
    <Row>
      <Column1>...</Column1>
      <Column2>...</Column2>
      ...
    </Row>
    ...
  </Table>
  <Table name="Table2">
    ...
  </Table>
  ...
</Database>
```

Здесь имена базы данных и таблиц задаются в атрибутах, значения полей записей — в тексте элементов. Элементы *Database*, *Table* и *Row* описывают представленную выше иерархию.

Аналогичную структуру можно представить иначе:

```
<MyDatabase>
  <Table1>
    <Row Column1="..." Column2="..." ... />
    <Row Column1="..." Column2="..." ... />
    ...
  </Table1>
  <Table2>
    ...
  </Table2>
</MyDatabase>
```

Данный формат использует меньше символов для описания структуры базы данных и, следовательно, меньше памяти для хранения документа.

Формат представления данных может разрабатываться с учётом особенностей структуры базы данных. Используя возможность формата XML создавать многоуровневые иерархии, можно представлять связи «один-ко-многим» в виде вложенности элементов.

Пусть таблицы Table1 и Table2 имеют связь «один-ко-многим». Соответственно, таблица Table2 содержит внешний ключ для реализации данной связи в базе данных. В XML элементы, представляющие записи таблицы Table2 можно вкладывать в элементы, соответствующие связанным записям таблицы Table1:

```
<Table1>
  <Row Column1="..." Column2="...">
    <Table2>
      <Row Column1="..." Column2="..." ... />
      <Row Column1="..." Column2="..." ... />
      ...
    </Table2>
  </Row>
  <Row Column1="..." Column2="...">
    <Table2>
      <Row Column1="..." Column2="..." ... />
      <Row Column1="..." Column2="..." ... />
      ...
    </Table2>
  </Row>
  ...
</Table1>
```

В этом случае значения внешнего ключа таблицы Table2 можно не хранить в документе — связь между таблицами полностью реализуется вложенностью элементов. Это справедливо для связей «один-ко-многим» и «один-к-одному».

Реализация связи «многие-ко-многим» может быть выполнено аналогичным образом, однако это приводит к множественному дублированию элементов. Для устранения этого недостатка можно использовать следующий подход:

1) таблицы Table1 и Table2 представить на одном уровне иерархии XML;
2) в элементах, представляющих записи таблицы Table2, вложенные в элементы записей таблицы Table1, хранить только значения полей, по которым можно однозначно идентифицировать элемент, соответствующий записи из таблицы Table2, представленной в пункте 1.

В документе XML не требуется хранить суррогатные ключи таблиц базы данных, поскольку каждый элемент документа можно считать уникальным, и значения суррогатных ключей не является характеристикой сущности. Среди остальных столбцов таблицы базы данных можно выделить набор столбцов, совокупность значений полей которых будет уникальной для каждой записи этой таблицы.

Синтаксическая и логическая правильность документа

Одно из основных преимуществ технологии XML для представления данных состоит в том, что программные средства для работы с данными в формате XML имеют возможность проверить *правильность* документа перед его использованием.

Выделяют два уровня правильности документов:

- синтаксический — документ полностью соответствует всем синтаксическим правилам, принятым для всех документов XML;
- логический — документ соответствует формату, принятому для использования в данном конкретном случае.

Основные синтаксические требования для документов следующие:

- каждый элемент содержит открывающий и закрывающий теги или является самозакрывающимся;
- содержимое одного элемента полностью вкладывается в другой элемент между его открывающим и закрывающим тегами;
- имена атрибутов уникальны в пределах одного элемента;
- значения атрибутов заключены в кавычки;
- текст внутри элементов и значения атрибутов не содержат специальных символов — их необходимо заменить на специальные комбинации символов;
- имена элементов и атрибутов задаются с учётом регистра.

Логический уровень правильности документа подразумевает, что одни и те же данные могут быть представлены в XML различным образом (см. предыдущие примеры), однако для возможности корректной обработки документа в конкретной программе требуется определить *формат структуры* XML. Перед обработкой файла данных XML программа может выполнить проверку, соответствует ли структура данных заданному формату, и в противном случае такой файл обрабатываться не будет.

Для описания формата структуры XML создаётся схема, которая представляет собой данные XML, имеющие определённую структуру, и сохраняется в файле с расширением XSD (*XML Schema Definition*). Схема содержит следующий корневой элемент:

```
<xs:schema targetNamespace="namespace" xmlns="namespace"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  ...
</xs:schema>
```

где *namespace* — пространство имён для связи элемента данных XML и всех его вложенных элементов с текущей схемой.

Пространство имён представляет собой URI с URL-адресом организации и путём к файлу схемы. Этот адрес в действительности может не существовать — в качестве пространства имён можно использовать любой допустимый адрес ресурса, указывающий на файл с заданным именем и расширением XSD, например, http://tempuri.org/MyXSD.xsd. Элемент данных XML, содержимое которого требуется проверить на соответствие схеме данных, должен иметь атрибут *xmlns* со значением, совпадающим со значением атрибута *targetNamespace* схемы. Если атрибут *xmlns* указан в корневом элементе данных, проверка на соответствие схеме будет выполнена для всего файла.

Описание элемента в схеме задаётся в элементе *xs:element,* который содержит следующие основные атрибуты:

- *name* — имя элемента;
- *type* — тип данных, задаётся для простых элементов или для составных элементов, если тип данных определён в этой же схеме;
- *minOccurs* — минимальное количество появлений элемента в текущем месте. Может задаваться для всех элементов, кроме корневого. Если значение не задано, используется значение по умолчанию, равное 1 — такой элемент является *обязательным*. Если задано значение 0 — элемент является *необязательным*;
- *maxOccurs* — максимальное количество появлений элемента в текущем месте. Может задаваться для всех элементов, кроме

корневого. Если значение не задано, используется значение по умолчанию, равное 1 — такой элемент является *уникальным*. Если задано значение *unbounded* — количество появлений элемента не ограничено.

В схеме XML определены следующие основные типы данных:

- беззнаковые целочисленные типы: *xs:unsignedByte* (1 байт), *xs:unsignedShort* (2 байта), *xs:unsignedInt* (4 байта), *xs:unsignedLong* (8 байтов), *xs:nonNegativeInteger* (произвольного размера, включая 0), *xs:positiveInteger* (произвольного размера, большие 0);
- знаковые целочисленные типы: *xs:byte* (1 байт), *xs:short* (2 байта), *xs:int* (4 байта), *xs:long* (8 байтов), *xs:integer* (произвольного размера), *xs:nonPositiveInteger* (произвольного размера, меньшие или равные 0), *xs:negativeInteger* (произвольного размера, меньшие 0);
- числовой тип с фиксированной точкой: *xs:decimal*;
- вещественные типы: *xs:float* (4 байта), *xs:double* (8 байтов);
- строковый тип: *xs:string*;
- логический тип: *xs:boolean* — может принимать значения 0, 1, *true*, *false*;
- типы даты и времени: *xs:date* — дата, *xs:time* — время, *xs:dateTime* — дата и время;
- массив шестнадцатеричных значений: *xs:hexBinary*.

Также можно объявлять новые типы данных на основе существующих с помощью элементов *xs:simpleType* и *xs:complexType*.

Содержимое составного элемента данных XML и атрибуты элемента описываются в элементе *xs:complexType*, в котором могут содержаться следующие элементы:

- *xs:sequence* — содержит описание вложенных элементов, которые должны объявляться в заданной последовательности;
- *xs:all* — содержит описание вложенных элементов, которые могут объявляться в любой последовательности;
- *xs:choice* — содержит описание элементов, из которых должен быть объявлен только один элемент;
- *xs:attribute* — содержит описание атрибута.

Элемент *xs:attribute* содержит следующие основные атрибуты:

- *name* — имя элемента;
- *type* — тип данных;

– *use* — значение *required*, если атрибут является обязательным, значение *optional*, если атрибут не является обязательным. Если атрибут не задан, используется значение *optional*.

Обработка данных XML в приложениях .NET Framework

Несмотря на то, что данные в формате XML можно обрабатывать как обычный текст, значительно удобнее и эффективнее использовать для этого объектную модель, которую в .NET Framework реализуют классы в пространстве имён *System.Xml*.

Основным является класс *XmlDocument*, который представляет документ XML в памяти приложения и содержит следующие основные методы:

– *CreateXXX* — набор методов для создания *узлов*, представляющих часть данных XML соответствующего типа. Для каждого объекта должно быть задано имя, и может быть определён префикс и пространство имён;
– *Load* — загружает документ из указанного файла или потока в память в виде объектной модели;
– *LoadXml* — загружает документ из указанной строки;
– *Save* — сохраняет сформированный документ в файл или поток.

Доступ к корневому элементу документа осуществляется через свойство *DocumentElement*. Если данные не загружены, корневой элемент можно создать с помощью метода *CreateElement* и добавить его к документу с помощью метода *AppendChild*.

Для представления элементов используется класс *XmlElement*, для атрибутов — *XmlAttribute*. Оба класса являются наследниками абстрактного класса *XmlNode* и содержат следующие основные свойства:

– *InnerText* — содержит все значения текущего узла и всех его дочерних узлов. Используется для доступа к значениям атрибутов и простых элементов;
– *InnerXml* — содержит XML-разметку дочерних узлов текущего узла. Для элементов возвращает полное содержимое элемента, для атрибутов — значение атрибута с заменой специальных символов на символьные комбинации;
– *OuterXml* — содержит XML-разметку текущего узла и его дочерних узлов;
– *OwnerDocument* — содержит ссылку на объект документа, которому принадлежит текущий узел;
– *ParentNode* — содержит ссылку на родительский элемент текущего узла;

– *ChildNodes* — представляет коллекцию дочерних узлов текущего элемента. Не используется для атрибутов;
– *Attributes* — представляет коллекцию атрибутов текущего элемента. Не используется для атрибутов.

Для проверки документа на соответствие схеме требуется загрузить схему документа в коллекцию *Schemas* объекта *XmlDocument* с помощью одной из перегрузок метода *Add*. Для использования специализированных классов, представляющих схему данных, требуется ссылка на пространство имён *System.Xml.Schema*.

Для проверки всего документа XML на соответствие схеме значение атрибута *xmlns* корневого элемента должно соответствовать пространству имён одной из загруженных в объект *XmlDocument* схем данных. Это пространство имён доступно в свойстве *NamespaceURI* корневого элемента документа. Если атрибут *xmlns* не задан, значением свойства *NamespaceURI* будет пустая строка. В этом случае можно присвоить нужное значение атрибуту *xmlns* корневого элемента документа в коде программы, после чего необходимо перезагрузить данные в объекте *XmlDocument*:

```
XmlDocument doc = new XmlDocument();
doc.Load(dataFilename);
doc.Schemas.Add(schemaNamespace, schemaFilename);
if (string.IsNullOrEmpty(doc.DocumentElement.NamespaceURI))
{
    doc.DocumentElement.SetAttribute("xmlns", schemaNamespace);
    doc.LoadXml(doc.InnerXml);
}
```

Проверка на соответствие документа XML схеме выполняется вызовом метода *Validate*, который принимает в качестве параметра метод, имеющий сигнатуру делегата *ValidationEventHandler*. Этот метод будет вызываться, если при проверке документа будут найдены ошибки. Если в качестве значения параметра метода *Validate* передать значение *null*, при нахождении ошибок будет сгенерировано исключение *XmlSchemaValidationException*.

Если проверка прошла успешно, документ полностью соответствует схеме, и его можно использовать для получения данных.

Задание к работе

1. Выбрать в базе данных, полученной в результате лабораторной работы №1, 2–3 связанные таблицы и составить XSD-схему для представления данных из этих таблиц.

2. Добавить в приложение, созданное в лабораторной работе №4, возможность экспорта данных из выбранных таблиц в XML-файл, соответствующий составленной схеме.
3. Добавить в приложение возможность импорта данных из указанного пользователем XML-файла с предварительным выполнением проверки на соответствие данных составленной XSD-схеме.
4. Включить в отчёт описание процессов экспорта и импорта данных и решения сопутствующих задач.

Библиографический список

1. *Дейт, К. Дж.* Введение в базы данных, 7-е издание. : Пер. с англ. — М.: Издательский дом «Вильямс», 2001. — 1072 с. : ил. — Парал. тит. англ. — ISBN 5-8459-0138-3 (рус.)
2. *Крёнке, Д.* Теория и практика построения баз данных. 8-е изд. / Д. Крёнке. — СПб.: Питер, 2003. — 800 с.: ил. — (Серия «Классика computer science») — ISBN 5-94723-275-8
3. *Бен-Ган, Ицик.* Microsoft SQL Server 2012. Основы T-SQL / Ицик Бен-Ган : [пер. с англ. М. А. Райтмана]. — Москва : Эксмо, 2015. — 400 с. — (Мировой компьютерный бестселлер). — ISBN 978-5-699-73617-1
4. *Молинаро, Э.* SQL. Сборник рецептов. — Пер. с англ. — СПб.: Символ-Плюс, 2009. — 672 с., ил. — ISBN: 978-5-93286-125-7
5. *Тернстрем, Т.* Microsoft SQL Server 2008. Разработка баз данных. Учебный курс Microsoft : Пер. с англ. / Т. Тернстрем, Э. Вебер, М. Хотек совместно с компанией GrandMasters. — М.: Издательство «Русская Редакция», 2010. — 496 с.: ил. + CD-ROM. — ISBN 978-5-7502-0394-9
6. *Бен-Ган, И.* Microsoft SQL Server 2012. Создание запросов. Учебный курс Microsoft : Пер. с англ. / И. Бен-Ган, Д. Сарка, Р. Талмейдж. — М.: Издательство «Русская Редакция», 2014. — 720 с.: ил. + CD-ROM. — ISBN 978-5-7502-0432-8
7. *Стэкер Мэтью А, Стэйн Стивен Дж., Нортроп Тони*. Разработка клиентских Windows-приложений на платформе Microsoft .NET Framework: Учебный курс Microsoft / Пер. с англ. — М.: Издательство «Русская Редакция»; СПб.: Питер, 2008. — 624 стр.: ил. — ISBN 978-5-7502-0313-0 («Русская Редакция»), ISBN 978-5-388-00015-6 («Питер»)

Учебное издание

БАЗЫ ДАННЫХ

Методические указания к выполнению лабораторных работ
для студентов, обучающихся по направлениям бакалавриата
09.03.01 — Информатика и вычислительная техника и
09.03.04 — Программная инженерия

Составители: **Гарибов** Александр Искендерович
Бондаренко Татьяна Владимировна

Подписано в печать X.XX.18. Формат 60×84/16. Усл.печ.л. 3,0. Уч.-изд.л. 3,3.
Тираж 40 экз. Заказ Цена
Отпечатано в Белгородском государственном технологическом университете
им. В. Г. Шухова
308012, г. Белгород, ул. Костюкова, 46