

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»



МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по выполнению лабораторных работ
по дисциплине «Базы данных»
для студентов направления 09.03.03 «Прикладная информатика»
(профиль «Прикладная информатика в экономике»)

Ставрополь
2020

Оглавление

Лабораторная работа №1. Создание файла данных и журнала транзакций ..	3
Лабораторная работа №2. Создание и заполнение таблиц	24
Лабораторная работа №3. Создание запросов и фильтров	64
Лабораторная работа №4. Освоение программирования с помощью встроенного языка Transact SQL.....	84
Лабораторная работа №5 Оператор Select.....	92
Лабораторная работа № 6 Инструкция JOIN.....	113
Лабораторная работа №7. Создание хранимых процедур в Microsoft SQL Server.....	123
Лабораторная работа №8. Определение триггера в стандарте языка SQL	130
Лабораторная работа №9. Управление временем в реляционных базах данных	141
Лабораторная работа №10. Пользовательские функции.....	142
Лабораторная работа №11 Транзакции.....	150
Лабораторная работа №12. Блокировки.....	158
Лабораторная работа №13. Администрирование баз данных.....	166

Лабораторная работа №1. Создание файла данных и журнала транзакций

Цель: научиться создавать файлы данных и журнал транзакций
Система управления базами данных (СУБД) - это общий набор различных программных компонентов баз данных и собственно баз данных, содержащий следующие составляющие:

- прикладные программы баз данных;
- клиентские компоненты;
- серверы баз данных;
- собственно базы данных.

Прикладная программа баз данных представляет собой программное обеспечение специального назначения, разработанное и реализованное пользователями или сторонними компаниями-разработчиками ПО. В противоположность, *клиентские компоненты* - это программное обеспечение баз данных общего назначения, разработанное и реализованное компанией-разработчиком базы данных. С помощью клиентских компонентов пользователи могут получить доступ к данным, хранящимся на локальном или удаленном компьютере.

Сервер баз данных выполняет задачу управления данными, хранящимися в базе данных. Клиенты взаимодействуют с сервером баз данных, отправляя ему запросы. Сервер обрабатывает каждый полученный запрос и отправляет результаты соответствующему клиенту.

Возможности СУБД

В общих чертах, базу данных можно рассматривать с двух точек зрения - пользователя и системы базы данных. Пользователи видят базу данных как набор логически связанных данных, а для системы баз данных это просто последовательность байтов, которые обычно хранятся на диске. Хотя это два полностью разных взгляда, между ними есть что-то общее: система баз данных должна предоставлять не только интерфейс, позволяющий пользователям создавать базы данных и извлекать или модифицировать данные, но также системные компоненты для управления хранимыми данными. Поэтому система баз данных должна предоставлять следующие возможности:

- разнообразные пользовательские интерфейсы;
- физическую независимость данных;
- логическую независимость данных;
- оптимизацию запросов;
- целостность данных;
- управление параллелизмом;
- резервное копирование и восстановление;
- безопасность баз данных.

Все эти возможности вкратце описываются в следующих далее разделах.

Разнообразные пользовательские интерфейсы

Большинство баз данных проектируются и реализовываются для работы с ними разных типов пользователей, имеющих разные уровни знаний. По этой причине система баз данных должна предоставлять несколько отдельных пользовательских интерфейсов. Пользовательский интерфейс может быть графическим или текстовым.

В графических интерфейсах ввод осуществляется посредством клавиатуры или мыши, а вывод реализуется в графическом виде на монитор. Разновидностью текстового интерфейса, часто используемого в системах баз данных, является интерфейс командной строки, с помощью которого пользователь осуществляет ввод посредством набора команд на клавиатуре, а система отображает вывод в текстовом формате на мониторе.

Физическая независимость данных

Физическая независимость данных означает, что прикладные программы базы данных не зависят от физической структуры данных, хранимых в базе данных. Эта важная особенность позволяет изменять хранимые данные без необходимости вносить какие-либо изменения в прикладные программы баз данных.

Например, если данные были сначала упорядочены по одному критерию, а потом этот порядок был изменен по другому критерию, изменение физических данных не должно влиять на существующие приложения баз данных или ее схему (описание базы данных, созданное языком определения данных системы базы данных).

Логическая независимость данных

При обработке файлов, используя традиционные языки программирования, файлы объявляются прикладными программами, поэтому любые изменения в структуре файла обычно требуют внесения соответствующих изменений во все использующие его программы.

Системы баз данных предоставляют логическую независимость файлов, т.е., иными словами, логическую структуру базы данных можно изменять без необходимости внесения каких-либо изменений в прикладные программы базы данных. Например, добавление атрибута к уже существующей в системе баз данных структуре объекта с именем Person (например, адрес) вызывает необходимость модифицировать только логическую структуру базы данных, а не существующие прикладные программы. (Однако приложения потребуют модифицирования для использования нового столбца.)

Оптимизация запросов

Большинство систем баз данных содержат подкомпонент, называющийся **оптимизатором**, который рассматривает несколько возможных стратегий исполнения запроса данных и выбирает из них наиболее эффективную. Выбранная стратегия называется *планом исполнения запроса*. Оптимизатор принимает решение, принимая во внимание такие факторы, как размер таблиц, к которым направлен запрос, существующие

индексы и логические операторы (AND, OR или NOT), используемые в предложении WHERE.

Целостность данных

Одной из стоящих перед системой баз данных задач является идентифицировать логически противоречивые данные и не допустить их помещения в базу данных. (Примером таких данных будет дата "30 февраля" или время "5:77:00".) Кроме этого, для большинства реальных задач, которые реализовываются с помощью систем баз данных, существуют *ограничения для обеспечения целостности (integrity constraints)*, которые должны выполняться для данных. (В качестве примера ограничения для обеспечения целостности можно назвать требование, чтобы табельный номер сотрудника был пятизначным целым числом.)

Обеспечение целостности данных может осуществляться пользователем в прикладной программе или же системой управления базами данных. До максимально возможной степени эта задача должна осуществляться посредством СУБД.

Управление параллелизмом

Система баз данных представляет собой многопользовательскую систему программного обеспечения, что означает одновременное обращение к базе данных множественных пользовательских приложений. Поэтому каждая система баз данных должна обладать каким-либо типом механизма, обеспечивающим управление попытками модифицировать данные несколькими приложениями одновременно. Далее приводится пример проблемы, которая может возникнуть, если система баз данных не оснащена таким механизмом управления:

1. На общем банковском счете № 3811 в банке X имеется \$1500.
2. Владельцы этого счета, госпожа А и господин Б, идут в разные отделения банка и одновременно снимают со счета по \$750 каждый.
3. Сумма, оставшаяся на счету № 3811 после этих транзакций, должна быть \$0, и ни в коем случае не \$750.

Все системы баз данных должны иметь необходимые механизмы для обработки подобных ситуаций, обеспечивая управление параллелизмом.

Резервное копирование и восстановление

Система баз данных должна быть оснащена подсистемой для восстановления после ошибок в программном и аппаратном обеспечении. Например, если в процессе обновления 100 строк таблицы базы данных происходит сбой, то подсистема восстановления должна выполнить откат всех выполненных обновлений, чтобы обеспечить непротиворечивость данных.

Безопасность баз данных

Наиболее важными понятиями безопасности баз данных являются аутентификация и авторизация. *Аутентификация* - это процесс проверки подлинности учетных данных пользователя, чтобы не допустить использования системы несанкционированными пользователями. Аутентификация наиболее часто реализуется, требуя, чтобы пользователь

вводил свое имя пользователя и пароль. Система проверяет достоверность этой информации, чтобы решить, имеет ли данный пользователь право входа в систему или нет. Этот процесс можно усилить применением шифрования.

Авторизация - это процесс, применяемый к пользователям, уже получившим доступ к системе, чтобы определить их права на использование определенных ресурсов. Например, доступ к информации о структуре базы данных и системному каталогу определенной сущности могут получить только администраторы.

Системы реляционных баз данных

Компонент Database Engine сервера Microsoft SQL Server является системой реляционных баз данных. Понятие систем реляционных баз данных было впервые введено в 1970 г. Эдгаром Ф. Коддом в статье "A Relational Model of Data for Large Shared Data Banks". В отличие от предшествующих систем баз данных (сетевых и иерархических), реляционные системы баз данных основаны на реляционной модели данных, обладающей мощной математической теорией.

Модель данных - это набор концепций, взаимосвязей между ними и их ограничений, которые используются для представления данных в реальной задаче. Центральным понятием реляционной модели данных является таблица. Поэтому, с точки зрения пользователя, реляционная база данных содержит только таблицы и ничего больше. Таблицы состоят из столбцов (одного или нескольких) и строк (ни одной или нескольких). Каждое пересечение строки и столбца таблицы всегда содержит ровно одно значение данных.

Работа с демонстрационной базой данных

Используемая база данных SampleDb представляет некую компанию, состоящую из отделов (department) и сотрудников (employee). Каждый сотрудник принадлежит только одному отделу, а отдел может содержать одного или нескольких сотрудников. Сотрудники работают над проектами (project): в любое время каждый сотрудник занят одновременно в одном или нескольких проектах, а над каждым проектом может работать один или несколько сотрудников.

Эта информация представлена в базе данных SampleDb посредством четырех таблиц:

Department

Employee

Project

Works_on

Организация этих таблиц показана на рисунках ниже. Таблица Department представляет все отделы компании. Каждый отдел обладает следующими атрибутами (столбцами):

Department (Number, DepartmentName, Location)

	Number	DepartmentName	Location
1	d1	Исследования	Москва
2	d2	Бух. учет	Санкт-Петербург
3	d3	Маркетинг	Москва

Атрибут Number представляет однозначный номер каждого отдела, атрибут DepartmentName - его название, а атрибут Location - расположение. Таблица Employee представляет всех работающих в компании сотрудников. Каждый сотрудник обладает следующими атрибутами (столбцами):

Employee (Id, FirstName, LastName, DepartmentNumber)

	Id	FirstName	LastName	DepartmentNumber
1	25348	Дмитрий	Волков	d3
2	10102	Анна	Иванова	d3
3	18316	Игорь	Соловьев	d1
4	29346	Олег	Маменко	d2
5	9031	Елена	Лебедеенко	d2
6	2581	Василий	Фролов	d2
7	28559	Наталья	Вершинина	d1

Атрибут Id представляет однозначный табельный номер каждого сотрудника, атрибуты FirstName и LastName - имя и фамилию сотрудника соответственно, а атрибут DepartmentNumber - номер отдела, в котором работает сотрудник.

Все проекты компании представлены в таблице проектов Project, состоящей из следующих столбцов (атрибутов):

Project (ProjectNumber, ProjectName, Budget)

	Number	ProjectName	Budget
1	p1	Apollo	120000
2	p2	Gemini	95000
3	p3	Mercury	186500

В столбце ProjectNumber указывается однозначный номер проекта, а в столбцах ProjectName и Budget - название и бюджет проекта соответственно.

В таблице Works_on указывается связь между сотрудниками и проектами:

Works_on (EmpId, ProjectNumber, Job, EnterDate)

Results		Messages		
	Empld	ProjectNumber	Job	EnterDate
1	10102	p1	Аналитик	2006-10-01
2	10102	p3	Менеджер	2008-01-01
3	25348	p2	Консультант	2007-02-15
4	18316	p2	NULL	2007-06-01
5	29346	p2	NULL	2006-12-15
6	2581	p3	Аналитик	2007-10-15
7	9031	p1	Менеджер	2007-04-15
8	28559	p1	NULL	2007-08-01
9	28559	p2	Консультант	2008-02-01
10	9031	p3	Консультант	2006-11-15
11	29346	p1	Консультант	2007-01-04

В столбце Empld указывается табельный номер сотрудника, а в столбце ProjectNumber - номер проекта, в котором он принимает участие. Комбинация значений этих двух столбцов всегда однозначна. В столбцах Job и EnterDate указывается должность и начало работы сотрудника в данном проекте соответственно.

На примере базы данных SampleDb можно описать некоторые основные свойства реляционных систем баз данных:

- Строки таблицы не организованы в каком-либо определенном порядке.
- Также не организованы в каком-либо определенном порядке столбцы таблицы.
- Каждый столбец таблицы должен иметь однозначное имя в любой данной таблице. Но разные таблицы могут содержать столбцы с одним и тем же именем. Например, таблица Department содержит столбец Number и столбец с таким же именем имеется в таблице Project.
- Каждый элемент данных таблицы должен содержать одно значение. Это означает, что любая ячейка на пересечении строк и столбцов таблицы никогда не содержит какого-либо набора значений.
- Каждая таблица содержит, по крайней мере, один столбец, значения которого определяют такое свойство, что никакие две строки не содержат одинаковой комбинации значений для всех столбцов таблицы. В реляционной модели данных такой столбец называется *потенциальным ключом (candidate key)*. Если таблица содержит несколько потенциальных ключей, разработчик указывает один из них, как *первичный ключ (primary key)* данной таблицы. Например, первичным ключом таблицы Department будет столбец Number, а первичными ключами таблиц Employee будет Id. Наконец, первичным ключом таблицы Works_on будет комбинация столбцов Empld и ProjectNumber.
- Таблица никогда не содержит одинаковых строк. Но это свойство существует только в теории, т.к. компонент Database Engine и все

другие реляционные системы баз данных допускают существование в таблице одинаковых строк.

SQL - язык реляционной базы данных

Язык реляционной базы данных в системе SQL Server называется **Transact-SQL**. Это разновидность самого значимого на сегодняшний день языка базы данных - ***языка SQL (Structured Query Language - язык структурированных запросов)***. Происхождение языка SQL тесно связано с проектом, называемым System R, разработанным и реализованным компанией IBM еще в начале 80-х годов прошлого столетия. Посредством этого проекта было продемонстрировано, что, используя теоретические основы работы Эдгара Ф. Кодда, возможно создание системы реляционных баз данных.

В отличие от традиционных языков программирования, таких как C#, C++ и Java, язык SQL является *множество-ориентированным (set-oriented)*. Разработчики языка также называют его *запись-ориентированным (record-oriented)*. Это означает, что в языке SQL можно запрашивать данные из нескольких строк одной или нескольких таблиц, используя всего лишь одну инструкцию. Это одно из наиболее важных преимуществ языка SQL, позволяющее использовать этот язык на логически более высоком уровне, чем традиционные языки программирования.

Другим важным свойством языка SQL является его непроцедурность. Любая программа, написанная на процедурном языке (C#, C++, Java), пошагово описывает, как выполнять определенную задачу. В противоположность этому, язык SQL, как и любой другой непроцедурный язык, описывает, что хочет пользователь. Таким образом, ответственность за нахождение подходящего способа для удовлетворения запроса пользователя лежит на системе.

Язык SQL содержит два подязыка: *язык описания данных DDL (Data Definition Language)* и *язык обработки данных DML (Data Manipulation Language)*. Инструкции языка DDL также применяются для описания схем таблиц баз данных. Язык DDL содержит три общие инструкции SQL: CREATE, ALTER и DROP. Эти инструкции используются для создания, изменения и удаления, соответственно, объектов баз данных, таких как базы данных, таблицы, столбцы и индексы.

В отличие от языка DDL, язык DML охватывает все операции по манипулированию данными. Для манипулирования базами данных всегда применяются четыре общие операции: извлечение, вставка, удаление и модифицирование данных (SELECT, INSERT, DELETE, UPDATE).

Создание любой БД начинается с создания файла данных. Рассмотрим этот процесс в "Microsoft SQL Server 2014" на примере создания простой БД.

Для начала необходимо запустить среду разработки "SQL Server Management Studio". Для этого в меню "Пуск" выбираем пункт "Программы-Microsoft SQL Server 2014/SQL Server Management Studio" (рисунок. 1.1).

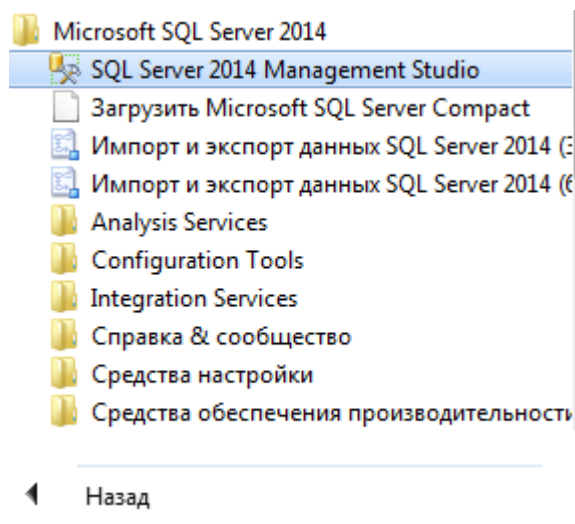


Рисунок. 1.1.

После запуска среды разработки появится окно подключения к серверу "Соединение с сервером" (рисунок. 1.2).

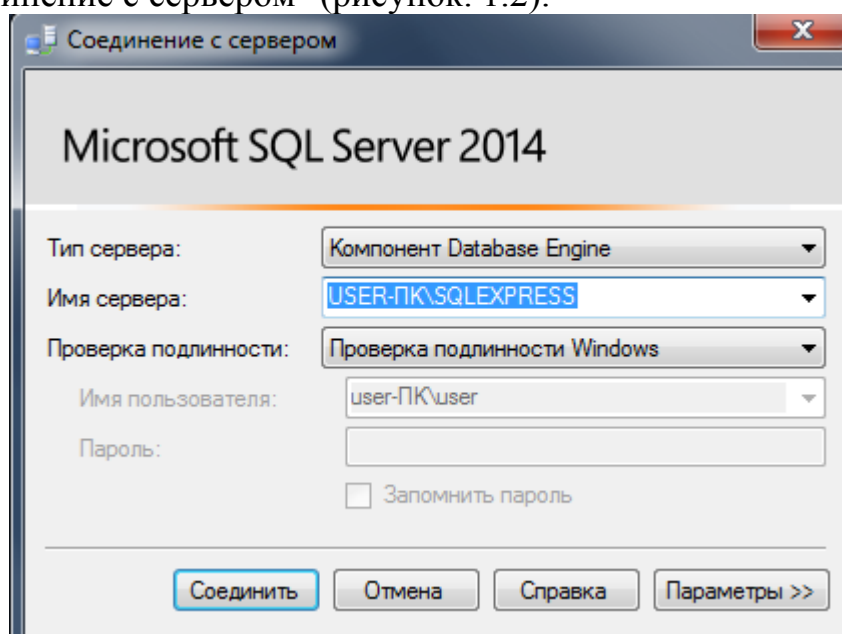


Рисунок. 1.2.

В этом окне необходимо нажать кнопку "Connect"

Замечание: Если при установке "Microsoft SQL Server 2014" был задан логин и пароль подключения к серверу, то перед нажатием кнопки "Соединить", в выпадающем списке нужно выбрать "Проверка подлинности Windows".

После нажатия кнопки "Соединить" появится окно среды разработки "SQL Server Management Studio" (рисунок. 1.3).

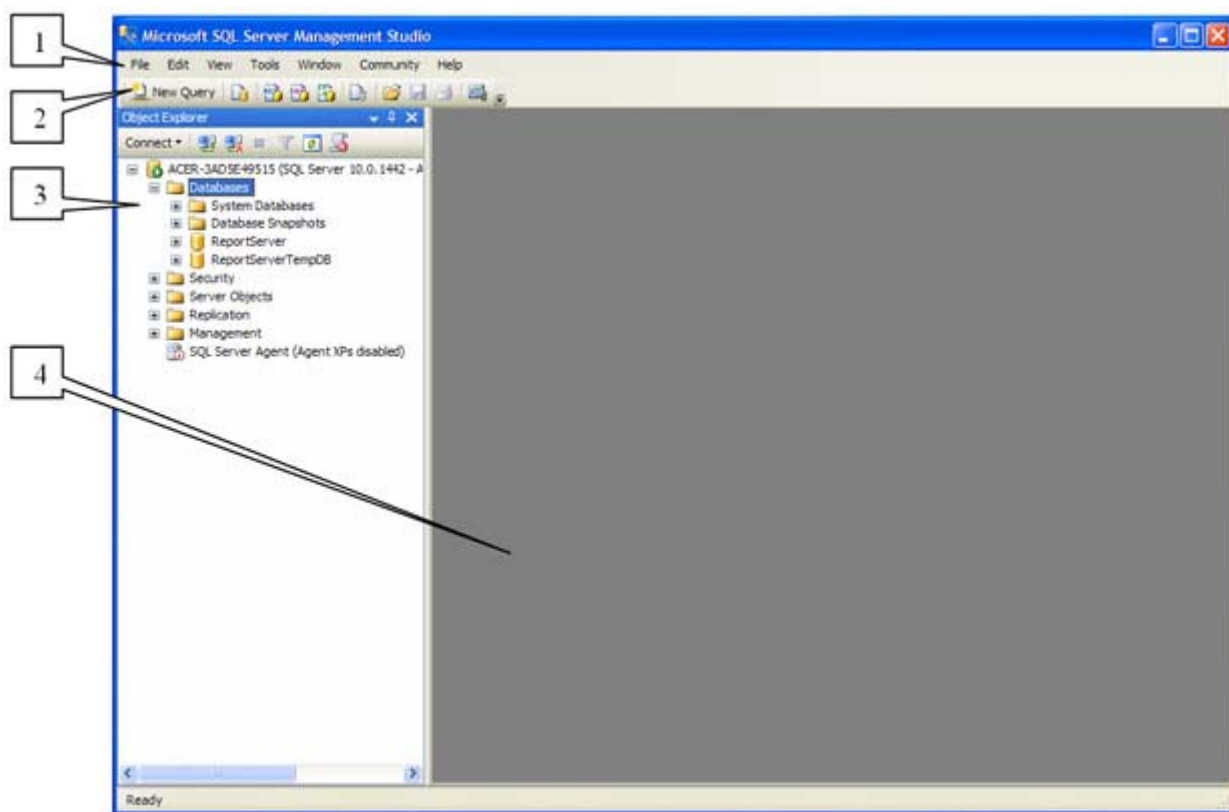


Рисунок. 1.3.

Данное окно имеет следующую структуру (рисунок. 1.3):

1. Оконное меню - содержит полный набор команд для управления сервером и выполнения различных операций.
2. Панель инструментов - содержит кнопки для выполнения наиболее часто производимых операций. Внешний вид данной панели зависит от выполняемой операции.
3. Панель "Object Explorer" - обозреватель объектов. Обозреватель объектов - это панель с древовидной структурой, отображающая все объекты сервера, а также позволяющая производить различные операции, как с самим сервером, так и с БД. Обозреватель объектов является основным инструментом для разработки БД.
4. Рабочая область. В рабочей области производятся все действия с БД, а также отображается ее содержимое.

Замечание: В обозревателе объектов сами объекты находятся в папках. Чтобы открыть папку необходимо щелкнуть по знаку "+" слева от изображения папки.

Теперь перейдем непосредственно к созданию файла данных. Для этого в обозревателе объектов щелкните ПКМ на папке "Databases" (Базы данных) и в появившемся меню выберите пункт "New Database" (Новая БД). Появится окно настроек параметров файла данных новой БД "New Database" (рисунок. 1.4). В левой части окна настроек имеется список "Select a page". Этот список позволяет переключаться между группами настроек.

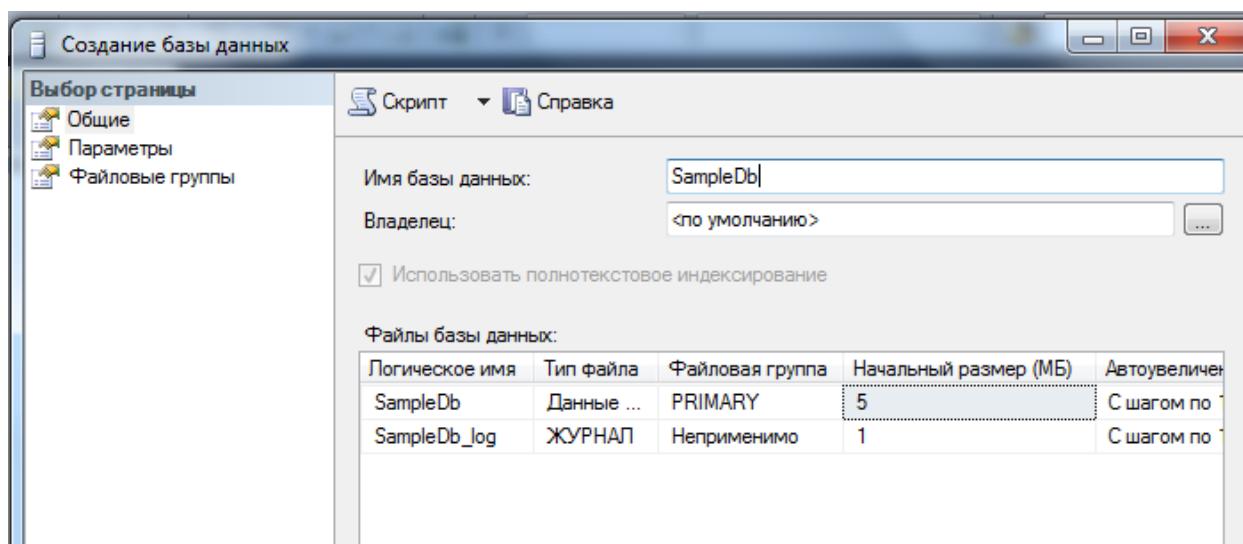


Рисунок. 1.4.

Для начала настроим основные настройки "Параметры". Для выбора основных настроек нужно просто щелкнуть мышью по пункту "Параметры" в списке "Выбор страницы". В правой части окна "New Database" появятся основные настройки.

Рассмотрим их более подробно. В верхней части окна расположено два параметра: "Database name" (Имя БД) и "Owner" (Владелец). Задайте параметр "Database name" равным "SampleDb". Параметр "Owner" оставьте без изменений.

Под вышеприведенными параметрами в виде таблицы располагаются настройки файла данных и журнала транзакций. Таблица имеет следующие столбцы:

- Logical Name - логическое имя файла данных и журнала транзакций. По этим именам будет происходить обращение к вышеприведенным файлам в БД. Можно заметить, что файл данных имеет то же имя что и БД, а имя файла журнала транзакций составлено из имени БД и суффикса "_log".
- File Type - тип файла. Этот параметр показывает, является ли файл файлом данных или журналом транзакций.
- Filegroup - группа файлов, показывает к какой группе файлов относится файл. Группы файлов настраиваются в группе настроек "Filegroups".
- Initial Size (MB) - начальный размер файла данных и журнала транзакций в мегабайтах.
- Autogrowth - автоувеличение размера файла. Как только файл заполняется информацией его размер автоматически увеличивается на величину, указанную в параметре "Autogrowth". Увеличение можно задавать как в мегабайтах так и в процентах. Здесь же можно задать максимальный размер файлов. Для изменения этого параметра надо нажать кнопку "...". В нашем случае (рисунок. 1.4) размер файлов не ограничен. Файл данных увеличивается на 1 мегабайт, а файл журнала транзакций на 10%.

- Path - путь к папке, где хранятся файлы. Для изменения этого параметра также надо нажать кнопку "...".

- File Name - имена файлов. По умолчанию имена файлов аналогичны логическим именам. Однако файл данных имеет расширение "mdf", а файл журнала транзакций - расширение "ldf".

Замечание: Для добавления новых файлов данных или журналов транзакций используется кнопка "Add", а для удаления кнопка "Remove".

В нашем случае мы оставим все основные настройки без изменений.

Теперь перейдем к другим второстепенным настройкам файла данных. Для доступа к этим настройкам необходимо щелкнуть мышью по пункту "Параметры" в списке "Выбор страницы". Появится следующее окно (рисунок. 1.5).

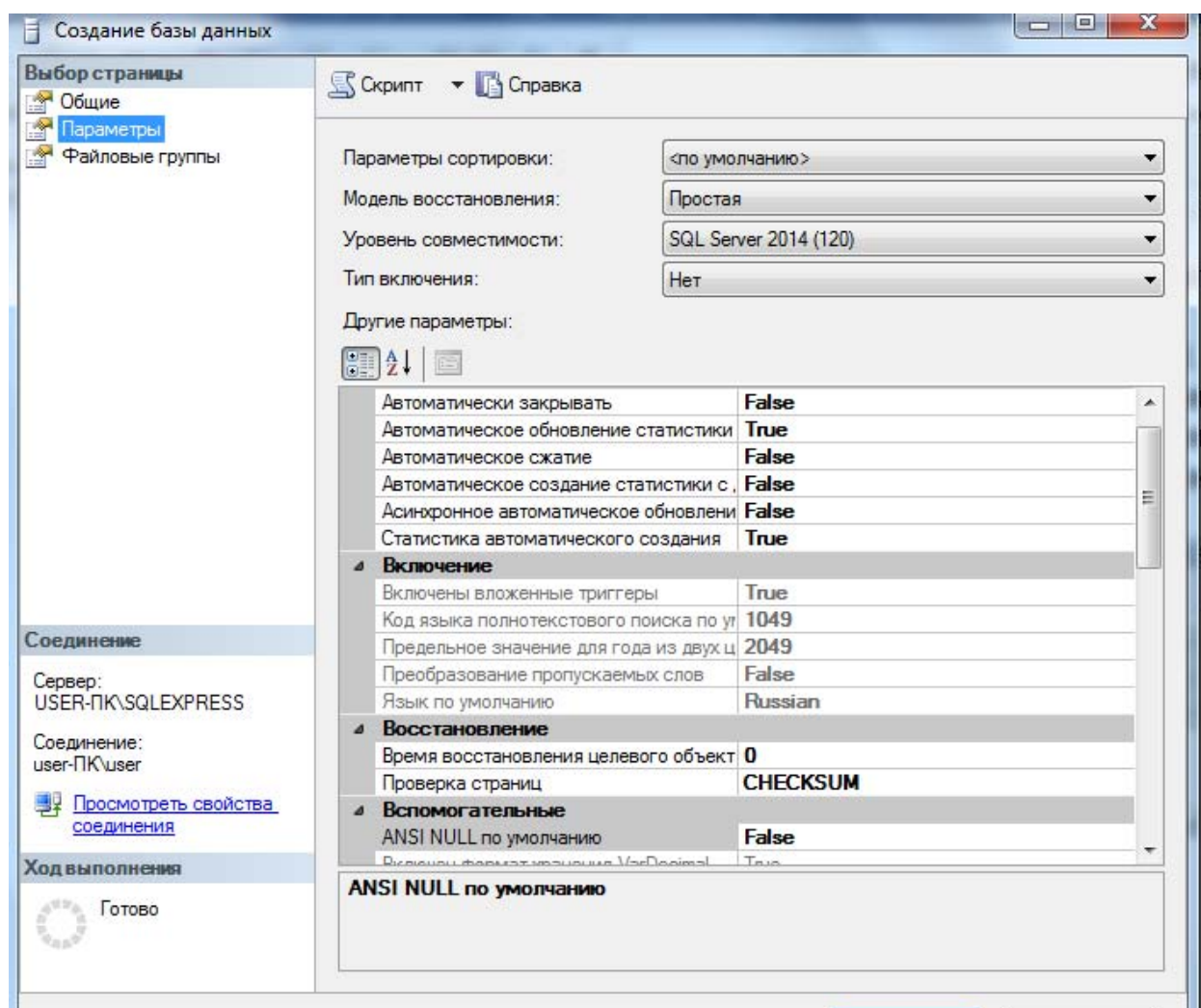


Рисунок. 1.5.

В правой части окна мы видим следующие настройки:

- Collation - этот параметр отвечает за обработку текстовых строк, их сравнение, текстовый поиск и т.д. Рекомендуется оставить его как "<server default>". При этом данный параметр будет равен значению, заданному на вкладке "Collation", при установке сервера.

- Recovery Model - модель восстановления. Данный параметр отвечает за информацию, предназначенную для восстановления БД,

хранящуюся в файле транзакций. Чем полнее модель восстановления, тем больше вероятность восстановления данных при сбое системы или ошибках пользователей, но и больше размер файла журнала транзакций. При наличии места на диске, рекомендуется оставить этот параметр в значении "Full".

- **Compatibility level** - уровень совместимости, определяет совместимость файла данных с более ранними версиями сервера. Если планируется перенос данных на другую, более раннюю версию сервера, то ее необходимо указать в этом параметре.

- **Other options** - второстепенные параметры. Данные параметры являются необязательными для изменения.

В нашем случае все параметры в разделе "Options", рекомендуется оставить как на рисунок. 1.5.

Наконец рассмотрим последнюю группу настроек "Файловые группы". Данная группа настроек отвечает за группы файлов. Для ее отображения в списке "Выбор страницы" необходимо щелкнуть мышью по пункту "Файловые группы". Отобразятся настройки групп файлов (рисунок. 1.6).

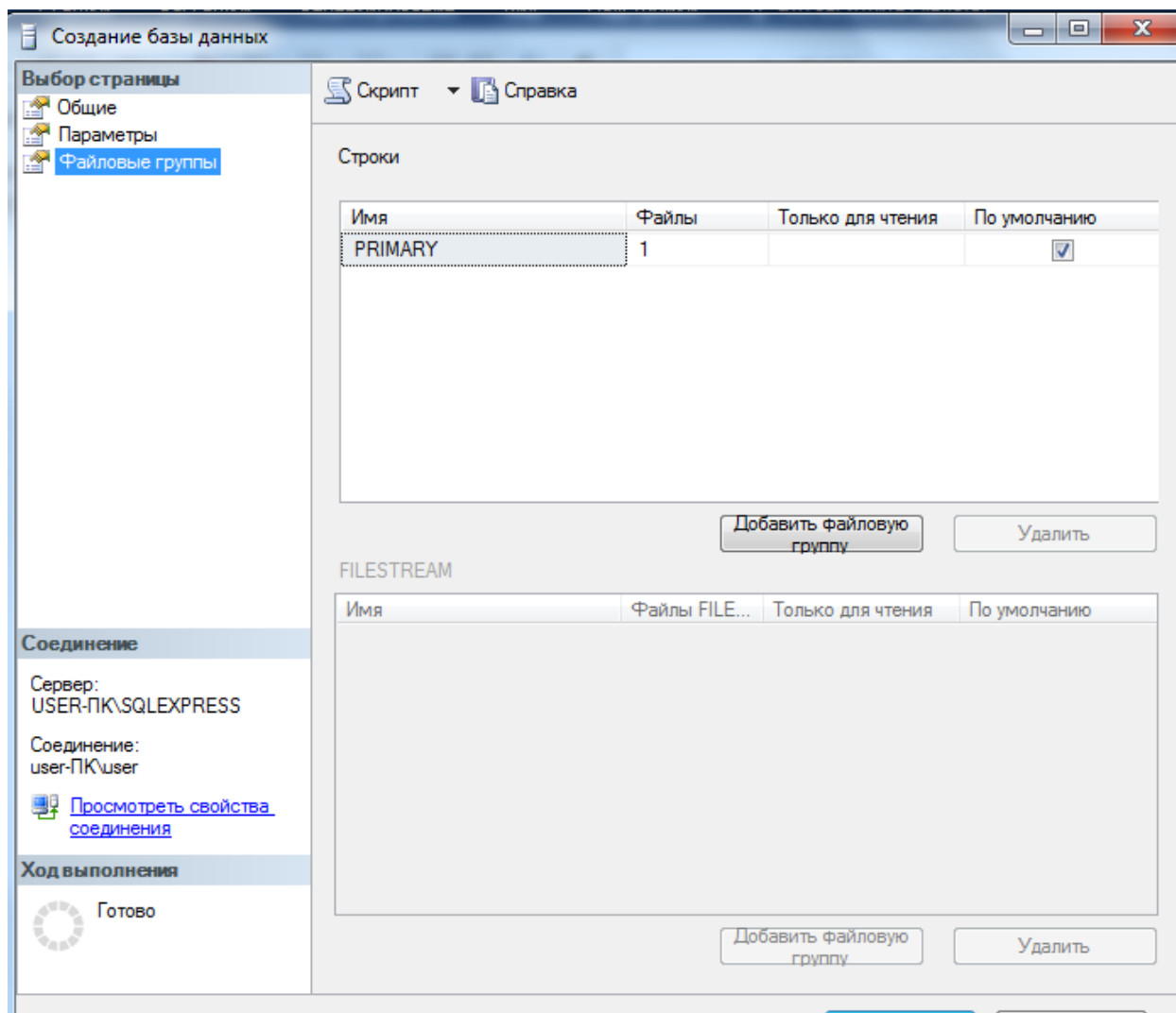


Рисунок. 1.6.

Группы файлов представлены в таблице "Строки" в правой части окна (рисунок. 1.6). Данная таблица имеет следующие столбцы:

- Name - имя группы файлов.
- Files - количество файлов входящих в группу.
- Read only - файлы в группе будут только для чтения. То есть, их можно только просматривать, но нельзя изменять.
- Default - группа по умолчанию. Все новые файлы данных будут входить в эту группу.

Замечание: Как и в случае с файлами данных, для добавления новых групп используется кнопка "Add", а для удаления кнопка "Remove".

В рассматриваемой БД нет необходимости добавлять новые группы файлов. Поэтому оставим группу настроек "Файловые группы" без изменений.

На этом мы заканчиваем настройку свойств наших файлов. Для принятия всех настроек и создание файла данных и журнала транзакций нашей БД в окне "Создание базы данных" нажмем кнопку "Ok".

Произойдет возврат в окно среды разработки "SQL Server Management Studio". На панели обозревателя объектов в папке "Databases" появится новая БД "SampleDb" (рисунок. 1.7).

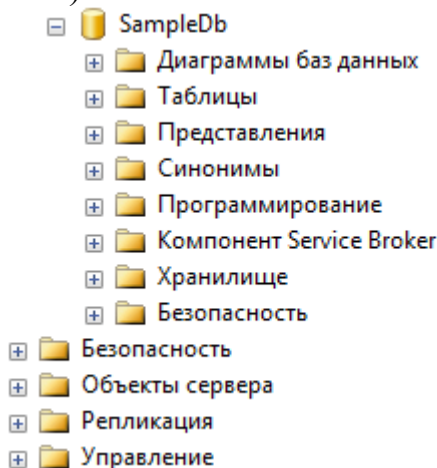


Рисунок. 1.7.

Замечание: Для переименования БД необходимо в обозревателе объектов щелкнуть по ней ПКМ и в появившемся меню выбрать пункт "Переименовать". Для удаления в это же меню выбираем пункт "Удалить", для обновления - пункт "Обновить", а для изменения свойств описанных выше - пункт "Свойства".

Задание к лабораторной работе 1.

В организации базы данных задействуется большое число различных объектов. Все объекты базы данных являются либо физическими, либо логическими. Физические объекты связаны с организацией данных на физических устройствах (дисках). Физическими объектами компонента Database Engine являются файлы и файловые группы. Логические объекты являются пользовательскими представлениями базы данных. В качестве примера логических объектов можно назвать таблицы, столбцы и представления (виртуальные таблицы).

Объектом базы данных, который требуется создать в первую очередь, является сама база данных. Компонент Database Engine управляет как системными, так и пользовательскими базами данных. Пользовательские базы данных могут создаваться авторизованными пользователями, тогда как системные базы данных создаются при установке СУБД.

Для создания базы данных используется два основных метода. В первом методе задействуется обозреватель объектов среды SQL Server Management Studio, как было показано ранее, а во втором применяется инструкция языка Transact-SQL **CREATE DATABASE**. Далее приводится общая форма этой инструкции, а затем подробно рассматриваются ее составляющие:

CREATE DATABASE db_name

[**ON** [**PRIMARY**] { file_spec1 } ,...]
[**LOG ON** {file_spec2} ,...]
[**COLLATE** collation_name]
[**FOR** {ATTACH | ATTACH_REBUILD_LOG }]

Параметр db_name - это имя базы данных. Имя базы данных может содержать максимум 128 символов. Одна система может управлять до 32 767 базами данных. Все базы данных хранятся в файлах, которые могут быть указаны явно администратором или предоставлены неявно системой. Если инструкция **CREATE DATABASE** содержит *параметр ON*, все файлы базы данных указываются явно.

Компонент Database Engine хранит файлы данных на диске. Каждый файл содержит данные одной базы данных. Эти файлы можно организовать в файловые группы. Файловые группы предоставляют возможность распределять данные по разным приводам дисков и выполнять резервное копирование и восстановление частей базы данных. Это полезная функциональность для очень больших баз данных.

Параметр file_spec1 представляет спецификацию файла и сам может содержать дополнительные опции, такие как логическое имя файла, физическое имя и размер. *Параметр PRIMARY* указывает первый (и наиболее важный) файл, который содержит системные таблицы и другую важную внутреннюю информацию о базе данных. Если параметр **PRIMARY** отсутствует, то в качестве первичного файла используется первый файл, указанный в спецификации.

Учетная запись компонента Database Engine, применяемая для создания базы данных, называется *владельцем базы данных*. База данных может иметь только одного владельца, который всегда соответствует учетной записи. Учетная запись, принадлежащая владельцу базы данных, имеет специальное **имя dbo**. Это имя всегда используется в отношении базы данных, которой владеет пользователь.

Опция **LOG ON** параметра **dbo** определяет один или более файлов в качестве физического хранилища журнала транзакций базы данных. Если

опция LOG ON отсутствует, то журнал транзакций базы данных все равно будет создан, поскольку каждая база данных должна иметь, по крайней мере, один журнал транзакций. (Компонент Database Engine ведет учет всем изменениям, которые он выполняет с базой данных. Система сохраняет все эти записи, в особенности значения до и после транзакции, в одном или более файлов, которые называются журналами транзакций. Для каждой базы данных системы ведется ее собственный журнал транзакций.)

В опции *COLLATE* указывается порядок сортировки по умолчанию для базы данных. Если опция *COLLATE* не указана, базе данных присваивается порядок сортировки по умолчанию, совершенно такой же, как и порядок сортировки по умолчанию системы баз данных.

В опции *FOR ATTACH* указывается, что база данных создается за счет подключения существующего набора файлов. При использовании этой опции требуется явно указать первый первичный файл. В опции *FOR ATTACH_REBUILD_LOG* указывается, что база данных создается методом присоединения существующего набора файлов операционной системы.

Компонент Database Engine создает новую базу данных по шаблону образцовой базы данных *model*. Свойства базы данных *model* можно настраивать для удовлетворения персональных концепций системного администратора. Если определенный объект базы данных должен присутствовать в каждой пользовательской базе данных, то этот объект следует сначала создать в базе данных *model*.

В примере ниже показан код для создания простой базы данных, без указания дополнительных подробностей. Чтобы исполнить этот код, введите его в редактор запросов среды Management Studio и нажмите клавишу <F5>.

USE master;

CREATE DATABASE SampleDb;

Код, приведенный в примере, создает базу данных, которая называется SampleDb. Такая сокращенная форма инструкции CREATE DATABASE возможна благодаря тому, что почти все ее параметры имеют значения по умолчанию. По умолчанию система создает два файла. Файл данных имеет логическое имя SampleDb и исходный размер 2 Мбайта. А файл журнала транзакций имеет логическое имя SampleDb_log и исходный размер 1 Мбайт. (Значения размеров обоих файлов, а также другие свойства новой базы данных зависят от соответствующих спецификаций базы данных *model*.)

Задание.

Создать новую базу данных можно с помощью соответствующего программного кода. Для этого в утилите «Создать запрос» («New query») с помощью команды:

Create database SampleDb

Для выполнения команды нажать F5.

Открыть утилиту **SQL Server Management Studio**. Проверить наличие БД, если ее не видите в разделе «Базы данных», то нажмите F5 для обновления.

Создайте новую базу данных в соответствии с вариантом и изучите ее свойства.

Варианты заданий к лабораторной работе №1

Общие положения

В утилите «Создать запрос» создать новую базу данных с помощью оператора **Create Database**, название БД определить, исходя из предметной области.

Вариант 1. БД «Учет выданных подарков несовершеннолетним детям сотрудников предприятия»

Код сотрудника	Код сотрудника	Код ребенка
Фамилия	Имя ребенка	Стоимость подарка
Имя	Дата рождения	Дата выдачи подарка
Отчество	Код ребенка	Код выдачи
Должность		
Подразделение		
Дата приема на работу		

Вариант 2. БД «Учет выполненных ремонтных работ»

Код прибора в ремонте	Код прибора	Код мастера
Название прибора	Код мастера	Фамилия мастера
Тип прибора	ФИО владельца прибора	Имя мастера
Дата производства	Дата приема в ремонт	Отчество мастера
	Вид поломки	Разряд мастера
	Стоимость ремонта	Дата приема на работу
	Код ремонта	

Вариант 3. БД «Продажа цветов»

Код цветка	Код цветка	Код продавца
Название цветка	Дата продажи	Фамилия
Сорт цветка	Цена продажи	Имя
Средняя высота	Код продавца	Отчество
Тип листа	Код продажи	Разряд
Цветущий		Оклад
Дополнительные сведения		Дата приема на работу

Вариант 4. БД «Поступление лекарственных средств»

Код лекарства	Код лекарства	Код поставщика
Название лекарства	Код	Сокращенное название

Показания к применению
Единица измерения
Количество в упаковке
Название производителя

поставщика
Дата поставки
Цена за единицу
Количество
Код поступления

Полное название
Юридический адрес
Телефон
ФИО руководителя

Вариант 5. БД «Списание оборудования»

Код оборудования
Название оборудования
Тип оборудования
Дата поступления
ФИО ответственного
Место установки

Код оборудования
Причина списания
Дата списания
Код сотрудника
Код списания

Код сотрудника
Фамилия
Имя
Отчество
Должность
Подразделение
Дата приема на работу

Вариант 6. БД «Поваренная книга»

Код блюда
Тип блюда
Вес блюда
Порядок приготовления
Количество калорий
Количество углеводов

Код блюда
Код продукта
Объем продукта

Код продукта
Название продукта
Ед измерения

Вариант 7. БД «Регистрация входящей документации»

Код регистратора
Фамилия
Имя
Отчество
Должность
Дата приема на работу

Код документа
Номер документа
Дата регистрации
Краткое содержание документа
Тип документа
Код организации-отправителя
Код регистратора

Код организации-отправителя
Сокращенное название
Полное название
Юридический адрес
Телефон
ФИО руководителя

Вариант 9. БД «Увольнение сотрудника»

Код сотрудника
Фамилия
Имя
Отчество
Должность
Подразделение
Дата приема на работу

Код документа
Номер документа
Дата регистрации
Дата увольнения
Код статьи увольнения
Код сотрудника
Денежная компенсация

Код статьи увольнения
Название статьи увольнения
Причина увольнения
Номер статьи увольнения
Номер пункта/подпункта увольнения

Вариант 9. БД «Приказ на отпуск»

Код сотрудника
Фамилия
Имя
Отчество
Должность
Подразделение
Дата приема на работу

Код документа
Номер документа
Дата регистрации
Дата начала отпуска
Дата окончания отпуска
Код сотрудника
Код отпуска

Код отпуска
Тип отпуска
Оплата отпуска
Льготы по опуску

Вариант 10. БД «Регистрация выходящей документации»

Код отправителя
Фамилия
Имя
Отчество
Должность
Дата приема на работу

Код документа
Номер документа
Дата регистрации
Краткое содержание документа
Тип документа
Код организации-получателя
Код отправителя

Код организации-получателя
Сокращенное название
Полное название
Юридический адрес
Телефон
ФИО руководителя

Вариант 11. БД «Назначение на должность»

Код сотрудника
Фамилия
Имя
Отчество
Дата приема на работу
Дата рождения

Код документа
Номер документа
Дата регистрации
Дата назначения
Код сотрудника
Код должности

Код должности
Название должности
Льготы по должности
Требования к квалификации

Пол		
-----	--	--

Вариант 12. БД «Выдача оборудования в прокат»

Код клиента	Код выдачи	Код оборудования
Фамилия	Номер документа	Название оборудования
Имя	Дата начала проката	Тип оборудования
Отчество	Дата окончания проката	Дата поступления в прокат
Адрес	Код оборудования	
Телефон	Код клиента	
Серия и номер паспорта	Стоимость	

Вариант 13. БД «Списание оборудования из проката»

Код оборудования	Код оборудования	Код сотрудника
Название оборудования	Причина списания	Фамилия
Тип оборудования	Дата списания	Имя
Дата поступления в прокат	Код сотрудника	Отчество
	Номер документа	Должность
	Дата регистрации	Дата приема на работу
	Код списания	

Вариант 14. БД «Прием цветов в магазин»

Код цветка	Код цветка	Код поставщика
Название цветка	Дата поступления	Сокращенное название
Сорт цветка	Цена за единицу	Полное название
Средняя высота	Код поставщика	Юридический адрес
Тип листа	Код поступления	Телефон
Цветущий	Количество	ФИО руководителя
Дополнительные сведения		

Вариант 15. БД «Регистрация клиентов гостиницы»

Код номера	Код регистрации	Код клиента
Тип номера	Код номера	Фамилия

Перечень удобств
Цена за сутки

Дата заезда
Дата выезда
Стоимость
Код клиента

Имя
Отчество
Адрес
Телефон
Серия и номер паспорта

Вариант 16. БД «Возврат оборудования в службу проката»

Код клиента
Фамилия
Имя
Отчество
Адрес
Телефон
Серия и номер паспорта

Код возврата
Номер документа
Дата возврата
Состояние оборудования
Код оборудования
Код клиента
Штраф

Код оборудования
Название оборудования
Тип оборудования
Дата поступления в прокат

Вариант 17. БД «Учет материальных ценностей на предприятии»

Код ценности
Название ценности
Тип ценности
Закупочная стоимость
Срок гарантии
Дата начала гарантии

Код постановки на учет
Код ценности
Код материально ответственного
Дата постановки на учет
Место нахождения ценности

Код материально ответственного
Фамилия
Имя
Отчество
Должность
Дата приема на работу
Подразделение

Вариант 18. БД «Состав ремонтных работ»

Код ремонтной работы
Код этапа работы
Название этапа работы
Стоимость этапа

Код ремонтной работы
Код мастера
Стоимость ремонта
Количество дней ремонта
Название ремонтной работы

Код мастера
Фамилия мастера
Имя мастера
Отчество мастера
Разряд мастера
Дата приема на работу

Вариант 19. БД «Продажа лекарственных средств»

Код лекарства
Название лекарства
Показания к применению
Единица измерения
Количество в упаковке
Название производителя

Номер чека
Цена за единицу
Количество
Код лекарства
Код записи в чеках

Номер чека
Дата продажи
Сумма
ФИО кассира

Вариант 20. БД «Учет исполнения по входящей документации»

Код исполнителя
Фамилия
Имя
Отчество
Должность
Подразделение
Дата приема на работу

Код документа
Дата назначения на исполнения
Срок выполнения в днях
Тип результата
Код исполнителя
Факт исполнения

Код документа
Номер документа
Дата регистрации
Краткое содержание документа
Тип документа
Организация-отправитель
Код исполнителя

Лабораторная работа №2. Создание и заполнение таблиц

Цель: научиться создавать и заполнять таблицы

Среда SQL Server Management Studio имеет два основных назначения: администрирование серверов баз данных и управление объектами баз данных. Эти функции рассматриваются далее.

Администрирование серверов баз данных

Задачи администрирования, которые можно выполнять с помощью среды SQL Server Management Studio, включают, среди прочих, следующие:

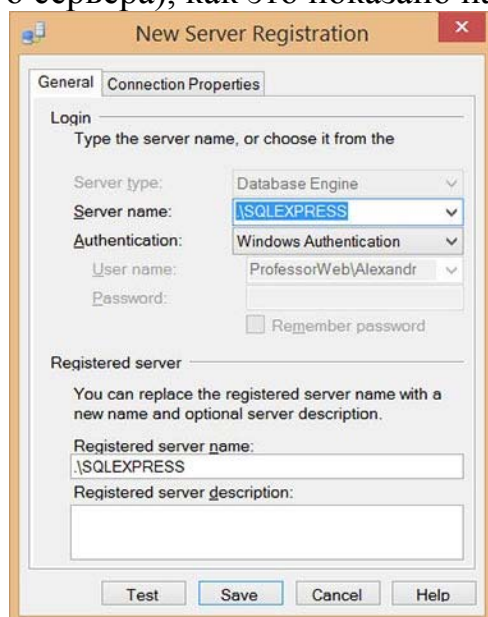
- регистрация серверов;
- подключение к серверу;
- создание новых групп серверов;
- управление множественными серверами;
- пуск и остановка серверов.

Эти задачи администрирования описываются в следующих подразделах.

Регистрация серверов

Среда SQL Server Management Studio отделяет деятельность по регистрации серверов от деятельности по исследованию баз данных и их объектов. (Действия этих обоих типов можно выполнять посредством обозревателя объектов.) Прежде чем можно использовать базы данных и объекты любого сервера, будь то локального или удаленного, его нужно зарегистрировать.

Сервер можно зарегистрировать при первом запуске среды SQL Server Management Studio или позже. Чтобы зарегистрировать сервер базы данных, щелкните правой кнопкой требуемый сервер в обозревателе объектов и в контекстном меню выберите пункт Register. Если панель обозревателя объектов скрыта, то откройте ее, выполнив команду меню View --> Object Explorer. Откроется диалоговое окно New Server Registration (Регистрация нового сервера), как это показано на рисунке ниже:



Выберите имя сервера, который нужно зарегистрировать, и тип проверки подлинности для этого сервера (т.е. проверка подлинности Windows или проверка подлинности SQL Server), после чего нажмите кнопку Save.

Подключение к серверу

Среда SQL Server Management Studio также разделяет задачи регистрации сервера и подключения к серверу. Это означает, что при регистрации сервера автоматического подключения этого сервера не происходит. Чтобы подключиться к зарегистрированному серверу, нужно щелкнуть правой кнопкой требуемый сервер в окне инспектора объектов и в появившемся контекстном меню выбрать пункт Connect (Подключиться).

Создание новой группы серверов

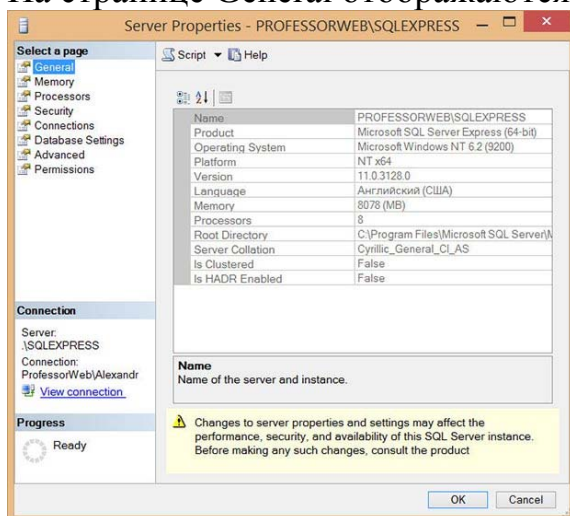
Чтобы создать новую группу серверов в панели зарегистрированных серверов, щелкните правой кнопкой узел Local Server Groups (Группы локальных серверов) в окне Registered Server и в контекстном меню выберите пункт New Server Group (Создание группы серверов). В открывшемся диалоговом окне New Server Group Properties (Свойства новой группа серверов) введите однозначное имя группы и, по выбору, ее описание.

Управление множественными серверами

Посредством обозревателя объектов среда SQL Server Management Studio позволяет администрировать множественные серверы баз данных (называемые экземплярами) на одном компьютере. Каждый экземпляр компонента Database Server имеет свой собственный набор объектов баз данных (системные и пользовательские базы данных), который не разделяется между экземплярами.

Для управления сервером и его конфигурацией щелкните правой кнопкой имя сервера в обозревателе объектов и в появившемся контекстном меню выберите пункт Properties (Свойства). Откроется диалоговое окно Server Properties (Свойства сервера), содержащее несколько страниц, таких как General (Общие), Security (Безопасность), Permissions (Разрешения) и т.п.

На странице General отображаются общие свойства сервера:



Страница Security содержит информацию о режиме аутентификации сервера и методе аудита входа. На странице Permissions воспроизводятся все

учетные записи и роли, которые имеют доступ к серверу. В нижней части страницы отображаются все разрешения, которые можно предоставлять этим учетным записям и ролям.

Можно изменить имя сервера, присвоив ему новое имя. Для этого щелкните правой кнопкой требуемый сервер в окне обозревателя объектов и в контекстном меню выберите пункт Register. Теперь можно присвоить серверу новое имя и изменить его описание. Серверы не следует переименовывать без особой на это надобности, поскольку это может повлиять на другие серверы, которые ссылаются на них.

Запуск и останов серверов

Сервер Database Engine по умолчанию запускается автоматически при запуске операционной системы Windows. Чтобы запустить сервер с помощью среды SQL Server Management Studio, щелкните правой кнопкой требуемый сервер в инспекторе объектов и в контекстном меню выберите пункт Start (Запустить). Это меню также содержит пункты Stop (Остановить) и Pause (Приостановить) для выполнения соответствующих действий с сервером.

Управление базами данных посредством обозревателя объектов Object Explorer

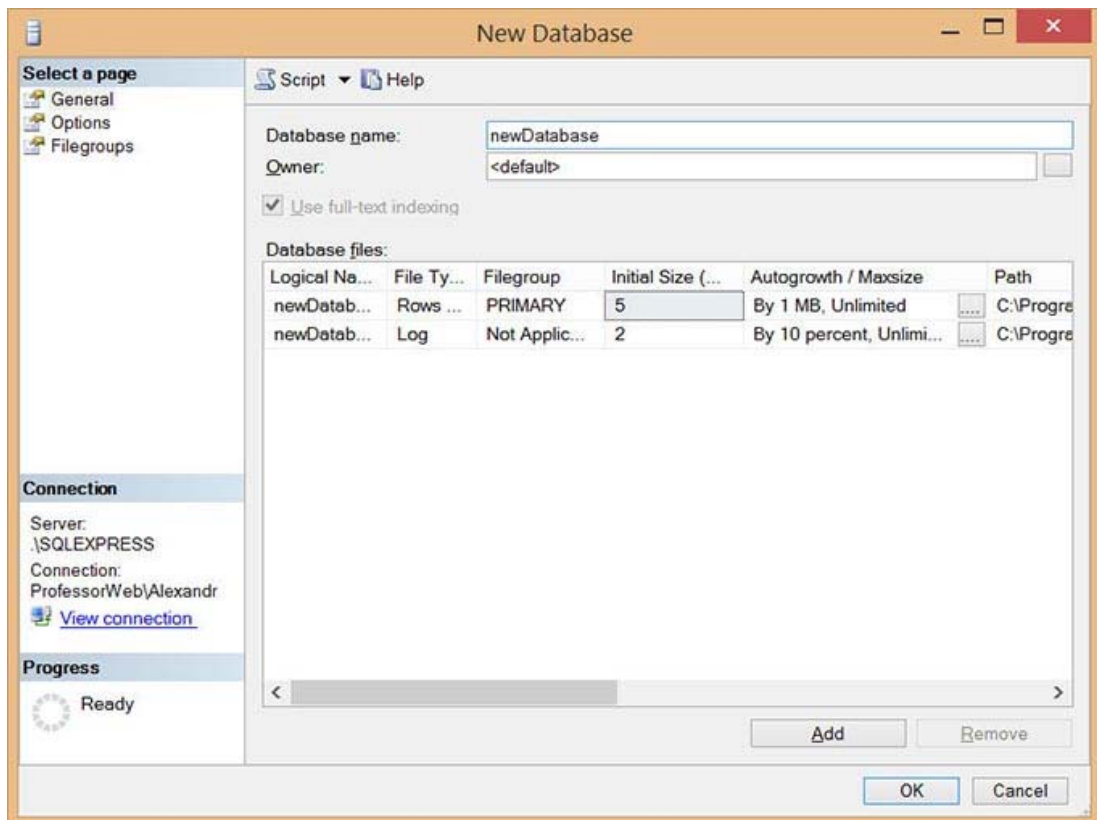
Задачи администрирования, которые можно выполнять с помощью среды SQL Server Management Studio, включают:

- создание баз данных, не прибегая к использованию языка Transact-SQL;
- модифицирование баз данных, не прибегая к использованию языка Transact-SQL;
- управление таблицами, не прибегая к использованию языка Transact-SQL;
- создание и исполнение инструкций SQL.

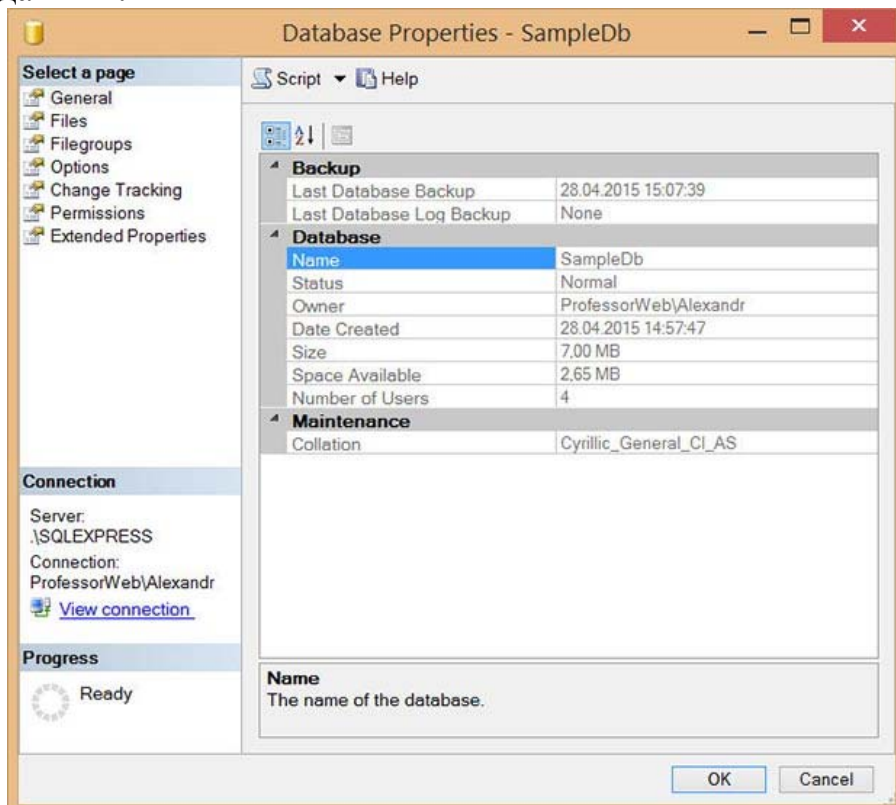
Создание баз данных без использования T-SQL

Новую базу данных можно создать посредством обозревателя объектов **Object Explorer**. Как можно судить по его названию, обозреватель объектов также можно использовать для исследования объектов сервера. С панели этого инструмента можно просматривать все объекты сервера и управлять сервером и базами данных. Дерево иерархии объектов сервера содержит, среди прочих папок, папку Databases (Базы данных). Эта папка, в свою очередь, содержит несколько подпапок, включая папку для системных баз данных, и по папке для каждой базы данных, созданной пользователем.

Чтобы создать базу данных посредством обозревателя объектов, щелкните правой кнопкой узел Databases и выберите пункт меню New Database (Создать базу данных). В открывшемся диалоговом окне New Database в поле Database name введите имя новой базы данных, после чего нажмите кнопку ОК.



Каждая база данных обладает несколькими свойствами, такими как тип файла, начальный размер и т.п. Список страниц свойств базы данных расположен в левой панели диалогового окна New Database. Страница General (Общие) диалогового окна Database Properties содержит, среди прочего, такую информацию, как имя, владелец и параметры сортировки базы данных:



Свойства файлов данных определенной базы данных перечисляются на странице Files (Файлы) и содержат такую информацию, как имя и начальный размер файла, расположение базы данных, а также тип файла (например, primary). База данных может храниться в нескольких файлах. В SQL Server применяется динамическое управление дисковым пространством. Это означает, что можно сконфигурировать размер базы данных для автоматического увеличения и уменьшения по мере надобности.

Чтобы изменить *свойство Autogrowth* (Автоувеличение) на странице Files, в столбце Autogrowth нажмите значок троеточия (...) и внесите соответствующие изменения в диалоговом окне Change Autogrowth. Чтобы позволить автоматическое увеличение размера базы данных, нужно установить **флажок Enable Autogrowth**. Каждый раз, когда существующий размер файла недостаточен для хранения добавляемых данных, сервер автоматически запрашивает систему выделить файлу дополнительное дисковое пространство. Объем дополнительного дискового пространства (в процентах или мегабайтах) указывается в **поле File Growth (Увеличение размера файла)** в том же диалоговом окне. А в разделе **Maximum File Size (Максимальный размер файла)** можно или ограничить максимальный размер файла, установив переключатель Limited to (MB) (Ограничение (Мбайт)), или снять ограничения на размер, установив переключатель Unlimited (Без ограничений) (это настройка по умолчанию). При ограниченном размере файла нужно указать его допустимый максимальный размер.

На странице Filegroups (Файловые группы) диалогового окна Database Properties отображаются имена файловых групп, к которым принадлежит файл базы данных, раздел файловой группы (по умолчанию или заданный явно), а также операции, разрешенные для выполнения с файловой группой (чтение и запись или только чтение).

На странице Options (Параметры) диалогового окна Database Properties можно просмотреть и модифицировать все параметры уровня базы данных. Существуют следующие группы параметров: Automatic (Автоматически), Containment (Включение), Cursor (Курсор), Miscellaneous (Вспомогательные), Recovery (Восстановление), Service Broker (Компонент Service Broker) и State (Состояние). Группа State содержит, например, следующие четыре параметра:

Database Read-Only (База данных доступна только для чтения)

Позволяет установить доступ к базе данных полный доступ или доступ только для чтения. В последнем случае пользователи не могут модифицировать данные. Значение по умолчанию этого параметра - False.

Restrict Access (Ограничение доступа)

Устанавливает количество пользователей, которые могут одновременно использовать базу данных. Значение по умолчанию - MULTI_USER.

Database State (Состояние базы данных)

Описывает состояние базы данных. Значение по умолчанию этого параметра - Normal.

Encryption Enabled (Шифрование включено)

Определяет режим шифрования базы данных. Значение по умолчанию этого параметра - False.

На странице Extended Properties (Расширенные свойства) отображаются дополнительные свойства текущей базы данных. На этой странице можно удалять существующие свойства и добавлять новые.

На странице Permissions (Разрешения) отображаются все пользователи, роли и соответствующие разрешения.

Остальные страницы Change Tracking (Отслеживание изменений), Mirroring (Зеркальное отображение) и Transaction Log Shipping (Доставка журналов транзакций) описывают возможности, связанные с доступностью данных.

Модифицирование баз данных

С помощью обозревателя объектов можно модифицировать существующие базы данных, изменяя файлы и файловые группы базы данных. Чтобы добавить новые файлы в базу данных, щелкните правой кнопкой требуемую базу данных и в контекстном меню выберите пункт Properties. В открывшемся диалоговом окне Database Properties выберите страницу Files и нажмите кнопку Add, расположенную внизу раздела Database files. В раздел будет добавлена новая строка, в поле Logical Name которой следует ввести имя добавляемого файла базы данных, а в других полях задать необходимые свойства этого файла. Также можно добавить и вторичную файловую группу для базы данных, выбрав страницу Filegroups (Файловые группы) и нажав кнопку Add.

Упомянутые ранее свойства базы данных может модифицировать только системный администратор или владелец базы данных.

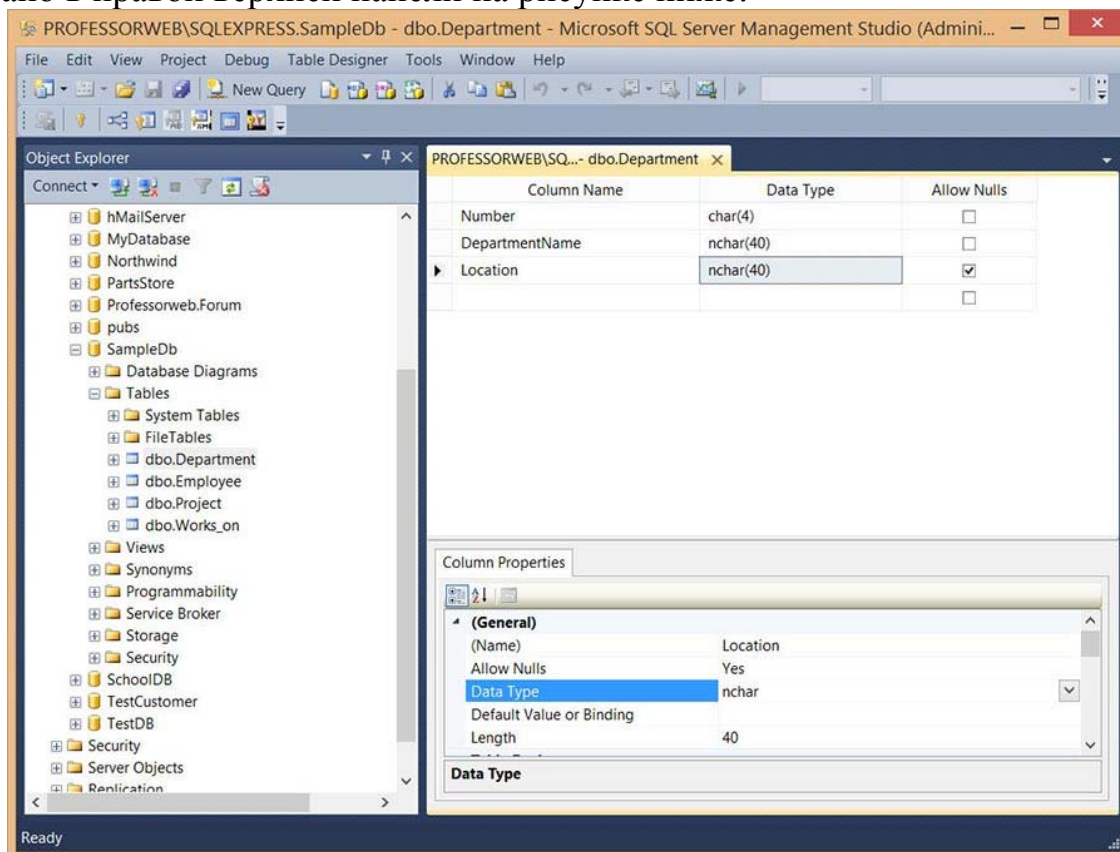
Чтобы удалить базы данных с помощью обозревателя объектов, щелкните правой кнопкой имя требуемой базы данных и в открывшемся контекстном меню выберите пункт Delete (Удалить).

Управление таблицами

Следующей задачей после создания базы данных является создание всех необходимых таблиц. Подобно созданию базы данных, таблицы в ней также можно создать либо с помощью языка Transact-SQL, либо посредством обозревателя объектов. Как и в случае с созданием базы данных, здесь мы рассмотрим создание таблиц только с помощью обозревателя объектов.

Для практики создания таблиц, в базе данных SampleDb создадим таблицу Department. Чтобы создать таблицу базы данных с помощью обозревателя объектов, разверните в нем узел Databases, а потом узел требуемой базы данных, щелкните правой кнопкой папку Tables и в открывшемся контекстном меню выберите пункт New Table. В верхней части с правой стороны окна средства Management Studio откроется окно для создания столбцов новой таблицы. Введите имена столбцов таблицы, их

типы данных и разрешение значений null для каждого столбца, как это показано в правой верхней панели на рисунке ниже:



Чтобы выбрать для столбца один из поддерживаемых системой типов данных, в столбце Data Type (Тип данных) выберите, а затем нажмите направленный вниз треугольник у правого края поля (этот треугольник появляется после того, как будет выбрана ячейка). В результате в открывшемся раскрывающемся списке выберите требуемый тип данных для столбца.

Тип данных существующего столбца можно изменить на вкладке Column Properties (Свойства столбца) (нижняя панель на рисунке). Для одних типов данных, таких как char, требуется указать длину в строке Length, а для других, таких как decimal, на вкладке Column Properties требуется указать масштаб и точность в соответствующих строках Scale (Масштаб) и Precision (Точность). Для некоторых других, таких как int, не требуется указывать ни одно из этих свойств. (Недействительные значения для конкретного типа данных выделены затененным шрифтом в списке всех возможных свойств столбца.)

Чтобы разрешить значения null для данного столбца, следует установить для него соответствующий флажок поля. Также, если для столбца требуется значение по умолчанию, его следует ввести в строку Default Value or Binding (Значение по умолчанию или привязка) панели Column Properties. Значение по умолчанию присваивается ячейке столбца автоматически, если для нее явно не введено значение.

Столбец Number является первичным ключом таблицы Department. Чтобы сделать столбец первичным ключом таблицы, щелкните его правой

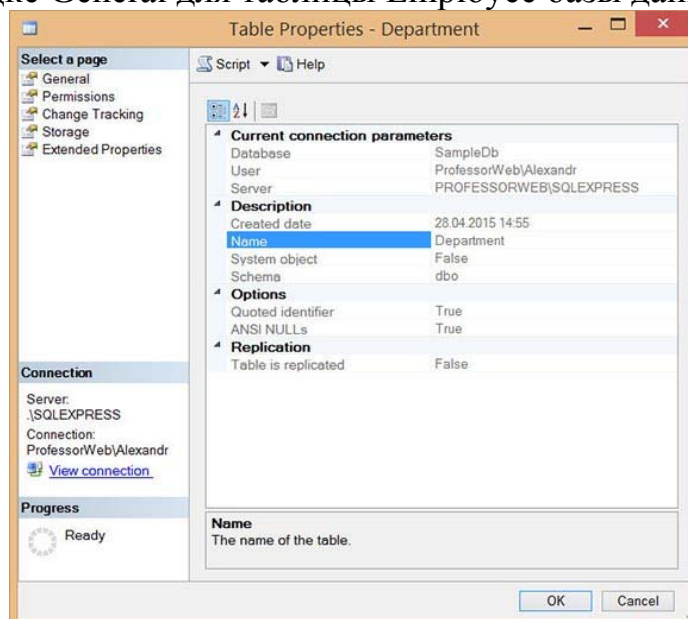
кнопкой и в контекстном меню выберите пункт Set Primary Key (Задать первичный ключ).

Так как, поле будет являться первичным полем связи в запросе, то мы должны сделать его числовым счетчиком. То есть данное поле должно автоматически заполняться числовыми значениями. Более того, оно должно быть ключевым.

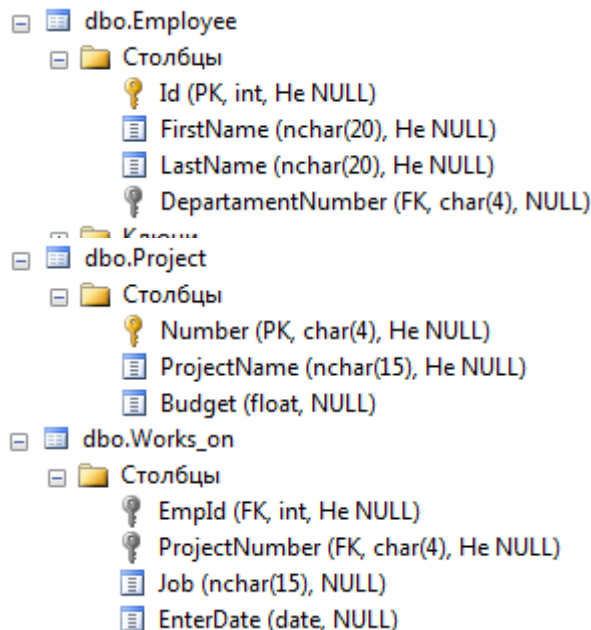
Сделаем поле счетчиком. Для этого выделите поле, просто щелкнув по нему мышкой в таблице определения полей. В таблице свойств поля отобразятся свойства поля. Разверните группу свойств "Identity Specification" (Настройка особенности). Свойство "(Is Identity)" (Особенное) установите в значение "Yes" (Да). Задайте свойства "Identity Increment" (Увеличение особенности, шаг счетчика) и "Identity Seed" (Начало особенности, начальное значение счетчика) равными 1. Эти настройки показывают, что значение поля у первой записи в таблице будет равным 1, у второй - 2, у третьей 3 и т.д.

Завершив все работы по созданию таблицы, щелкните крестик вкладки конструктора таблиц. Откроется диалоговое окно с запросом, сохранить ли сделанные изменения. Нажмите кнопку Yes, после чего откроется диалоговое окно Choose Name (Выбор имени) с запросом ввести имя таблицы. Введите требуемое имя таблицы и нажмите кнопку ОК. Таблица будет сохранена под указанным именем. Чтобы отобразить новую таблицу в иерархии базы данных, в панели инструментов обозревателя объектов щелкните значок Renew (Обновить).

Для просмотра и изменения свойств существующей таблицы разверните узел базы данных, содержащей требуемую таблицу, разверните узел Tables в этой базе данных и щелкните правой кнопкой требуемую таблицу, а затем в контекстном меню выберите пункт Properties. В результате для данной таблицы откроется диалоговое окно Table Properties. Для примера, на рисунке ниже показано диалоговое окно Table Properties на вкладке General для таблицы Employee базы данных SampleDb.



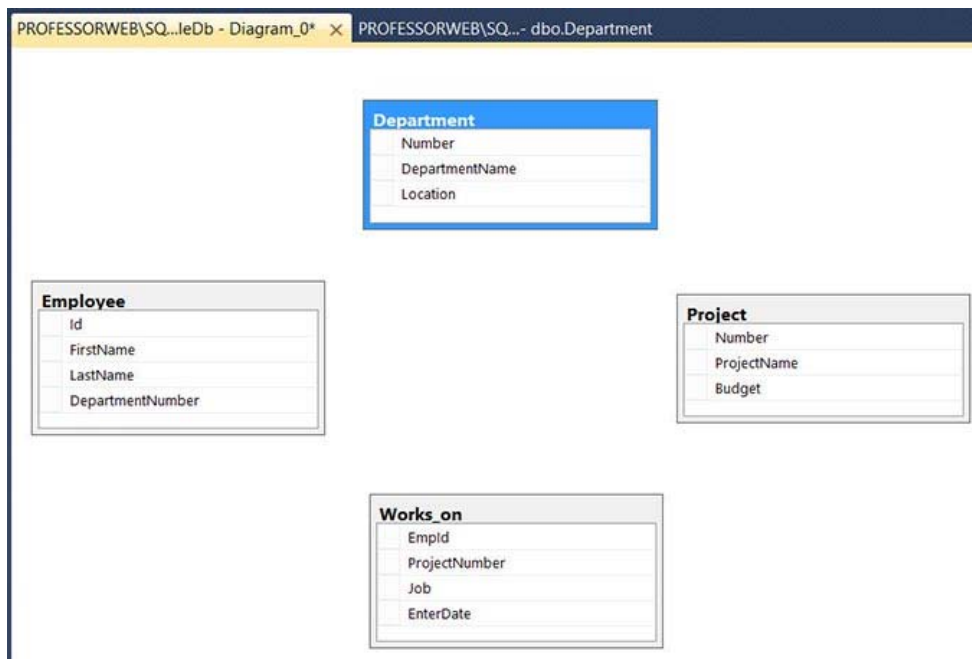
Чтобы переименовать таблицу, в папке Tables щелкните ее правой кнопкой в списке таблиц и в контекстном меню выберите пункт Rename. А чтобы удалить таблицу, щелкните ее правой кнопкой и выберите пункт Delete.



Создав все четыре таблицы базы данных SampleDb (Employee, Department, Project и Works_on), можно использовать еще одну возможность среды SQL Server Management Studio, чтобы отобразить диаграмму типа "сущность - отношение" - *диаграмму (ER) (entity-relationship)* этой базы данных. (Процесс преобразования таблиц базы данных в диаграмму "сущность - отношение" (ER) называется обратным проектированием.)

Чтобы создать диаграмму ER для базы данных SampleDb, щелкните правой кнопкой ее подпапку Database Diagrams (Диаграммы баз данных) и в контекстном меню выберите пункт New Database Diagram (Создать диаграмму базы данных). Если откроется диалоговое окно, в котором спрашивается, создавать ли вспомогательные объекты, выберите ответ Yes.

После этого откроется диалоговое окно Add Table, в котором нужно выбрать таблицы для добавления в диаграмму. Добавив все необходимые таблицы (в данном случае все четыре), нажмите кнопку Close, и мастер создаст диаграмму, подобную показанной на рисунке ниже:



На рисунке показана только промежуточная, а не конечная диаграмма ER базы данных SampleDb, поскольку, хотя на ней и показаны все четыре таблицы с их столбцами (и соответствующими первичными ключами), на ней все же отсутствуют отношения между таблицами. Отношение между двумя таблицами основывается на первичном ключе одной из таблиц и возможным соответствующим столбцом (или столбцами) другой таблицы.

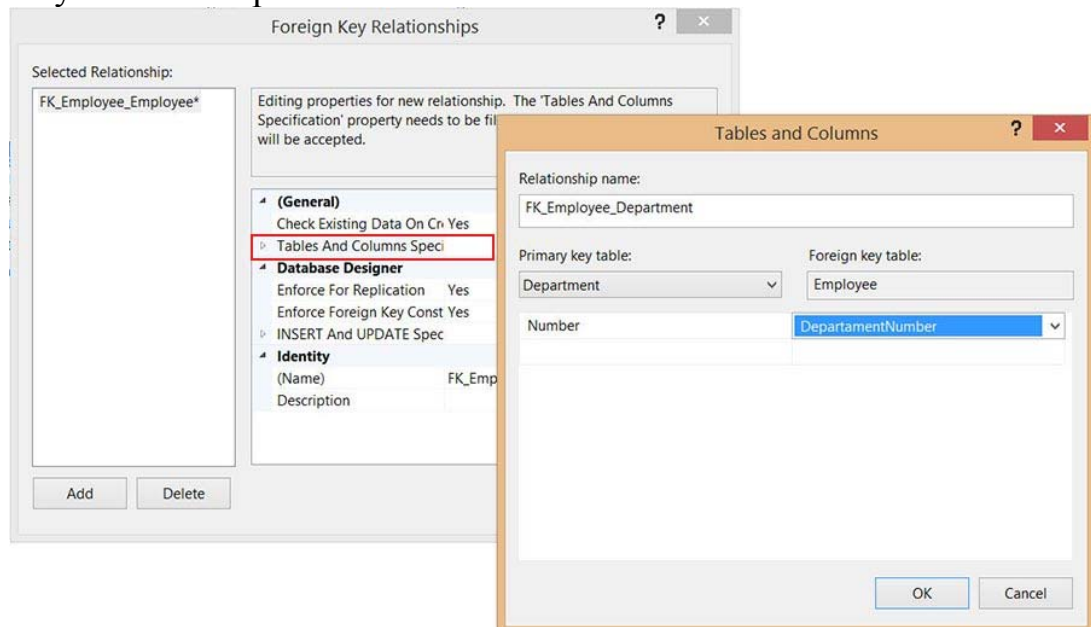
Между таблицами базы данных SampleDb существует три отношения. Таблица Department имеет отношение типа 1:N с таблицей Employee, поскольку каждому значению первичного ключа таблицы Department (столбец Number) соответствует одно или более значений столбца DepartmentNumber таблицы Employee (в одном отделе может работать несколько сотрудников).

Аналогично существует отношение между таблицами Employee и Works_on, поскольку только значения, которые присутствуют в столбце первичного ключа таблицы Employee (Id) также имеются в столбце EmpId таблицы Works_on. Третье отношение существует между таблицами Project и Works_on, т.к. только значения, которые присутствуют в первичном ключе таблицы Project (Number) также присутствуют в столбце ProjectNumber таблицы Works_on.

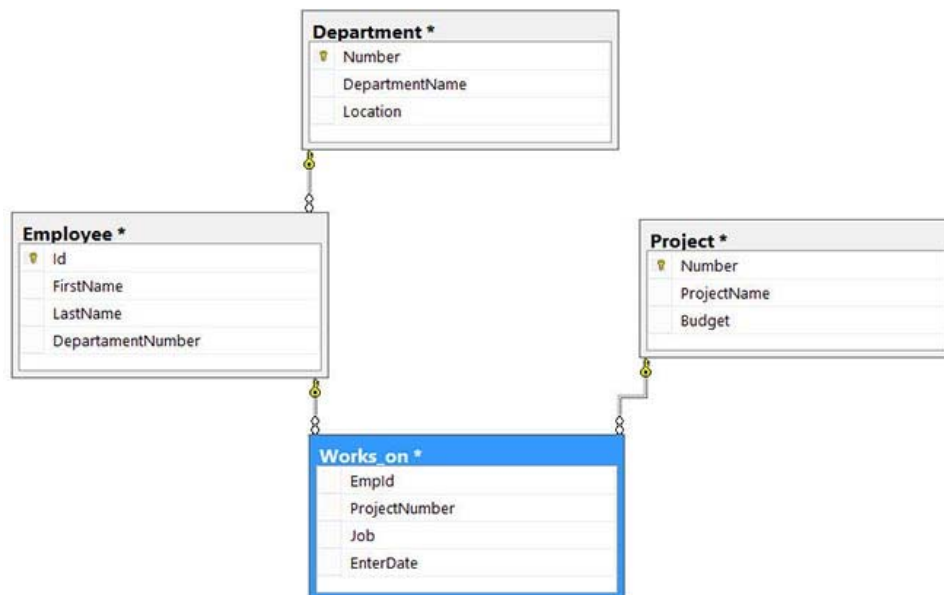
Чтобы создать эти три отношения, диаграмму ER нужно реконструировать, указав для каждой таблицы столбцы, которые соответствуют ключевым столбцам других таблиц. Такой столбец называется *внешним ключом (foreign key)*. Чтобы увидеть, как это делается, определим столбец DepartmentNumber таблицы Employee, как внешний ключ таблицы Department. Для этого выполним следующие действия:

1. В созданной диаграмме щелкните правой кнопкой графическое представление таблицы Employee и в контекстном меню выберите пункт Relationships (Отношения). В открывшемся диалоговом окне

- Foreign Key Relationships (Связи по внешнему ключу) нажмите кнопку Add.
- В правой панели диалогового окна расширьте первый столбец, выберите в нем строку Table and Columns Specification (Спецификация таблиц и столбцов) и нажмите кнопку с троеточием во втором столбце этой строки.
 - В открывшемся диалоговом окне Tables and Columns в раскрывающемся списке Primary key table (Таблица первичного ключа) выберите таблицу с соответствующим первичным ключом. В данном случае это будет таблица Department.
 - Выберите для этой таблицы столбец Number в качестве первичного ключа и столбец DepartmentNumber для таблицы Employee в качестве внешнего ключа, после чего нажмите кнопку OK чтобы закрыть окно Tables and Columns. Нажмите кнопку Close, чтобы закрыть окно Foreign Key Relationships.



Подобным образом создаются и другие два отношения. На рисунке ниже показана диаграмма ER, отображающая все три отношения между таблицами базы данных SampleDb:



При создании полей таблиц необходимо воспользоваться правилом CamelCase. CamelCase (рус. ВерблюжийРегистр, также ГорбатыйРегистр, СтилЬВерблюда) — стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово пишется с заглавной буквы.

Все значения в столбце должны быть одного типа данных. (Единственным исключением из этого правила являются значения типа данных SQL_VARIANT.) Используемые в Transact-SQL типы данных можно разбить на следующие категории:

- числовые типы;
- символьные типы;
- временные типы (даты и/или времени);
- прочие типы данных.

Все эти категории данных рассматриваются далее в последующих разделах.

Числовые типы данных

Как и следовало ожидать по их названию, числовые типы данных применяются для представления чисел. Эти типы и их краткое описание приводятся в таблице ниже:

Числовые типы данных T-SQL

Тип данных	Описание
<i>INTEGER</i>	Представляет целочисленные значения длиной в 4 байта в диапазоне от -2^{32} до $2^{32} - 1$. INT - сокращенная форма от INTEGER.

<i>SMALLINT</i>	Представляет целочисленные значения длиной в 2 байта в диапазоне от -32 768 до 32 767
<i>TINYINT</i>	Представляет целочисленные значения длиной в 1 байт в диапазоне от 0 до 255
<i>BIGINT</i>	Представляет целочисленные значения длиной в 8 байт в диапазоне от -2^{63} до $2^{63} - 1$
<i>DECIMAL(p,[s])</i>	Представляет значения с фиксированной точкой. Аргумент p (precision - точность) указывает общее количество разрядов, а аргумент s (scale - степень) - количество разрядов справа от полагаемой десятичной точки. В зависимости от значения аргумента p, значения decimal сохраняются в 5 до 17 байтах. DEC - сокращенная форма от DECIMAL.
<i>NUMERIC(p,[s])</i>	Синоним DECIMAL.
<i>REAL</i>	Применяется для представления значений с плавающей точкой. Диапазон положительных значений простирается приблизительно от $2,23E-308$ до $-1,18E-38$. Также может быть представлено и нулевое значение.
<i>FLOAT(p)</i>	Подобно типу REAL, представляет значения с плавающей точкой [(p)]. Аргумент p определяет точность. При значении $p < 25$ представляемые значения имеют одинарную точность (требуют 4 байта для хранения), а при значении $p \geq 25$ - двойную точность (требуют 8 байтов для хранения).
<i>MONEY</i>	Используется для представления денежных значений. Значения типа MONEY соответствуют 8-байтовым значениям типа DECIMAL, округленным до четырех разрядов после десятичной точки
<i>SMALLMONEY</i>	Представляет такие же значения, что и тип MONEY, но длиной в 4 байта

Символьные типы данных

Существует два общих вида символьных типов данных. Строки могут представляться однобайтовыми символами или же символами в кодировке Unicode. (В кодировке Unicode для представления одного символа применяется несколько байтов.) Кроме этого, строки могут быть разной длины. В таблице ниже перечислены категории символьных типов данных с их кратким описанием.

Символьные типы данных T-SQL

Тип данных	Описание
<i>CHAR</i> [(n)]	Применяется для представления строк фиксированной длины, состоящих из n однобайтовых символов. Максимальное значение n равно 8000. CHARACTER(n) - альтернативная эквивалентная форма CHAR(n). Если n явно не указано, то его значение полагается равным 1.
<i>VARCHAR</i> [(n)]	Используется для представления строки однобайтовых символов переменной длины (0 < n < 8 000). В отличие от типа данных CHAR, количество байтов для хранения значений типа данных VARCHAR равно их действительной длине. Этот тип данных имеет два синонима: CHAR VARYING и CHARACTER VARYING.
<i>NCHAR</i> [(n)]	Используется для хранения строк фиксированной длины, состоящих из символов в кодировке Unicode. Основная разница между типами данных CHAR и NCHAR состоит в том, что для хранения каждого символа строки типа NCHAR требуется 2 байта, а строки типа CHAR - 1 байт. Поэтому строка типа данных NCHAR может содержать самое большее 4000 символов. Тип NCHAR можно использовать для хранения, например, символов русского алфавита, т.к. однобайтовые кодировки не позволяют делать этого.
<i>NVARCHAR</i> [(n)]	Используется для хранения строк переменной длины, состоящих из символов в кодировке Unicode. Для хранения каждого символа строки типа NVARCHAR требуется 2 байта, поэтому строка типа данных NVARCHAR может содержать самое большее 4000 символов.

Тип данных **VARCHAR** идентичен типу данных **CHAR**, за исключением одного различия: если содержимое строки **CHAR(n)** короче, чем *n* символов, остаток строки заполняется пробелами. А количество байтов, занимаемых строкой типа **VARCHAR**, всегда равно количеству символов в ней.

Типы данных времени

В языке Transact-SQL поддерживаются следующие временные типы данных:

- **DATETIME**;
- **SMALLDATETIME**;
- **DATE**;
- **TIME**;
- **DATETIME2**;
- **DATETIMEOFFSET**.

Типы данных **DATETIME** и **SMALLDATETIME** применяются для хранения даты и времени в виде целочисленных значений длиной в 4 и 2 байта соответственно. Значения типа **DATETIME** и **SMALLDATETIME** сохраняются внутренне как два отдельных числовых значения. Составляющая даты значений типа **DATETIME** хранится в диапазоне от 01/01/1753 до 31/12/9999, а соответствующая составляющая значений типа **SMALLDATETIME** - в диапазоне от 01/01/1900 до 06/06/2079. Составляющая времени хранится во втором 4-байтовом (2-байтовом для значений типа **SMALLDATETIME**) поле в виде числа трехсотых долей секунды (для **DATETIME**) или числа минут (для **SMALLDATETIME**), истекших после полуночи.

Если нужно сохранить только составляющую даты или времени, использование значений типа **DATETIME** или **SMALLDATETIME** несколько неудобно. По этой причине в SQL Server были введены типы данных **DATE** и **TIME**, в которых хранятся только составляющие даты и времени значений типа **DATETIME**, соответственно. Значения типа **DATE** занимают 3 байта, представляя диапазон дат от 01/01/0001 до 31/12/9999. Значения типа **TIME** занимают 3-5 байт и представляют время с точностью до 100 нс.

Тип данных **DATETIME2** используется для представления значений дат и времени с высокой точностью. В зависимости от требований, значения этого типа можно определять разной длины, и занимают они от 6 до 8 байтов. Составляющая времени представляет время с точностью до 100 нс. Этот тип данных не поддерживает переход на летнее время.

Все рассмотренные на данный момент временные типы данных не поддерживают часовые пояса. Тип данных **DATETIMEOFFSET** имеет составляющую для хранения смещения часового пояса. По этой причине значения этого типа занимают от 6 до 8 байтов. Все другие свойства этого типа данных аналогичны соответствующим свойствам типа данных **DATETIME2**.

Значения дат в Transact-SQL по умолчанию определены в виде строки формата 'ммм дд гггг' (например, 'Jan 10 1993'), заключенной в одинарные

или двойные кавычки. (Но относительный порядок составляющих месяца, дня и года можно изменять с помощью инструкции *SET DATEFORMAT*. Кроме этого, система поддерживает числовые значения для составляющей месяца и разделители / и -.) Подобным образом, значение времени указывается в 24-часовом формате в виде 'чч:мм' (например, '22:24').

Язык Transact-SQL поддерживает различные форматы ввода значений типа DATETIME. Как уже упоминалось, каждая составляющая определяется отдельно, поэтому значения дат и времени можно указать в любом порядке или отдельно. Если одна из составляющих не указывается, система использует для него значение по умолчанию. (Значение по умолчанию для времени - 12:00 AM (до полудня).)

Двоичные и битовые типы данных

К двоичным типам данным принадлежат два типа: BINARY и VARBINARY. Эти типы данных описывают объекты данных во внутреннем формате системы и используются для хранения битовых строк. По этой причине значения этих типов вводятся, используя шестнадцатеричные числа.

Значения битового типа bit содержат лишь один бит, вследствие чего в одном байте можно сохранить до восьми значений этого типа. Краткое описание свойств двоичных и битовых типов данных приводится в таблице ниже:

Двоичные и битовые типы данных T-SQL

Тип данных	Описание
<i>BINARY</i> [(n)]	Определяет строку битов фиксированной длины, содержащую ровно n байтов (0 < n < 8000)
<i>VARBINARY</i> [(n)]	Определяет строку битов переменной длины, содержащую до n байтов (0 < n < 8000)
<i>BIT</i>	Применяется для хранения логических значений, которые могут иметь три возможных состояния: false, true и null

Тип данных больших объектов

Тип данных LOB (Large Object - большой объект) используется для хранения объектов данных размером до 2 Гбайт. Такие объекты обычно применяются для хранения больших объемов текстовых данных и для загрузки подключаемых модулей и аудио- и видеофайлов. В языке Transact-SQL поддерживаются следующие типы данных LOB:

- VARCHAR(MAX);
- NVARCHAR(MAX);
- VARBINARY(MAX).

Начиная с версии SQL Server 2005, для обращения к значениям стандартных типов данных и к значениям типов данных LOB применяется одна и та же модель программирования. Иными словами, для работы с объектами LOB можно использовать удобные системные функции и строковые операторы.

В компоненте Database Engine **параметр MAX** применяется с типами данных VARCHAR, NVARCHAR и VARBINARY для определения значений столбцов переменной длины. Когда вместо явного указания длины значения используется значение длины по умолчанию MAX, система анализирует длину конкретной строки и принимает решение, сохранять ли эту строку как обычное значение или как значение LOB. Параметр MAX указывает, что размер значений столбца может достигать максимального размера LOB данной системы.

Хотя решение о способе хранения объектов LOB принимается системой, настройки по умолчанию можно переопределить, используя системную процедуру `sp_tableoption` с аргументом `LARGE_VALUE_TYPES_OUT_OF_ROW`. Если значение этого аргумента равно 1, то данные в столбцах, объявленных с использованием параметра MAX, будут сохраняться отдельно от остальных данных. Если же значение аргумента равно 0, то компонент Database Engine сохраняет все значения размером до 8 060 байт в строке таблицы, как обычные данные, а значения большего размера хранятся вне строки в области хранения объектов LOB.

Начиная с версии SQL Server 2008, для столбцов типа VARBINARY(MAX) можно применять **атрибут FILESTREAM**, чтобы сохранять данные *BLOB (Binary Large Object - большой двоичный объект)* непосредственно в файловой системе NTFS. Основным достоинством этого атрибута является то, что размер соответствующего объекта LOB ограничивается только размером тома файловой системы.

Тип данных UNIQUEIDENTIFIER

Как можно судить по его названию, тип данных UNIQUEIDENTIFIER является однозначным идентификационным номером, который сохраняется в виде 16-байтовой двоичной строки. Этот тип данных тесно связан с идентификатором *GUID (Globally Unique Identifier - глобально уникальный идентификатор)*, который гарантирует однозначность в мировом масштабе. Таким образом, этот тип данных позволяет однозначно идентифицировать данные и объекты в распределенных системах.

Инициализировать столбец или переменную типа UNIQUEIDENTIFIER можно посредством функции NEWID или NEWSEQUENTIALID, а также с помощью строковой константы особого формата, состоящей из шестнадцатеричных цифр и дефисов.

К столбцу со значениями типа данных UNIQUEIDENTIFIER можно обращаться, используя в запросе **ключевое слово ROWGUIDCOL**, чтобы указать, что столбец содержит значения идентификаторов. (Это ключевое слово не генерирует никаких значений.) Таблица может содержать несколько

столбцов типа UNIQUEIDENTIFIER, но только один из них может иметь ключевое слово ROWGUIDCOL.

Тип данных SQL_VARIANT

Тип данных SQL_VARIANT можно использовать для хранения значений разных типов одновременно, таких как числовые значения, строки и даты. (Исключением являются значения типа TIMESTAMP.) Каждое значение столбца типа SQL_VARIANT состоит из двух частей: собственно значения и информации, описывающей это значение. Эта информация содержит все свойства действительного типа данных значения, такие как длина, масштаб и точность.

Для доступа и отображения информации о значениях столбца типа SQL_VARIANT применяется функция SQL_VARIANT_PROPERTY.

Объявлять тип столбца как SQL_VARIANT следует только в том случае, если это действительно необходимо. Например, если столбец предназначается для хранения значений разных типов данных или если при создании таблицы тип данных, которые будут храниться в данном столбце, неизвестен.

Тип данных HIERARCHYID

Тип данных HIERARCHYID используется для хранения полной иерархии. Например, в значении этого типа можно сохранить иерархию всех сотрудников или иерархию папок. Этот тип реализован в виде определяемого пользователем типа CLR, который охватывает несколько системных функций для создания узлов иерархии и работы с ними. Следующие функции, среди прочих, принадлежат к методам этого типа данных: GetLevel(), GetAncestor(), GetDescendant(), Read() и Write().

Тип данных TIMESTAMP

Тип данных TIMESTAMP указывает столбец, определяемый как VARBINARY(8) или BINARY(8), в зависимости от свойства столбца принимать значения null. Для каждой базы данных система содержит счетчик, значение которого увеличивается всякий раз, когда вставляется или обновляется любая строка, содержащая ячейку типа TIMESTAMP, и присваивает этой ячейке данное значение. Таким образом, с помощью ячеек типа TIMESTAMP можно определить относительное время последнего изменения соответствующих строк таблицы. (**ROWVERSION** является синонимом TIMESTAMP.)

Само по себе значение, сохраняемое в столбце типа TIMESTAMP, не представляет никакой важности. Этот столбец обычно используется для определения, изменилась ли определенная строка таблицы со времени последнего обращения к ней.

Варианты хранения

Начиная с версии SQL Server 2008, существует два разных варианта хранения, каждый из которых позволяет сохранять объекты LOB и экономить дисковое пространство. Это следующие варианты:

- хранение данных типа FILESTREAM;
- хранение с использованием разреженных столбцов (sparse columns).

Эти варианты хранения рассматриваются в следующих подразделах.

Хранение данных типа FILESTREAM

Как уже упоминалось ранее, SQL Server поддерживает хранение больших объектов (LOB) посредством типа данных VARBINARY(MAX). Свойство этого типа данных таково, что большие двоичные объекты (BLOB) сохраняются в базе данных. Это обстоятельство может вызвать проблемы с производительностью в случае хранения очень больших файлов, таких как аудио- или видеофайлов. В таких случаях эти данные сохраняются вне базы данных во внешних файлах.

Хранение данных типа FILESTREAM поддерживает управление объектами LOB, которые сохраняются в файловой системе NTFS. Основным преимуществом этого типа хранения является то, что хотя данные хранятся вне базы данных, управляются они базой данных. Таким образом, этот тип хранения имеет следующие свойства:

- данные типа FILESTREAM можно сохранять с помощью инструкции CREATE TABLE, а для работы с этими данными можно использовать инструкции для модифицирования данных (SELECT, INSERT, UPDATE и DELETE);
- система управления базой данных обеспечивает такой же самый уровень безопасности для данных типа FILESTREAM, как и для данных, хранящихся внутри базы данных.

Разреженные столбцы (sparse columns)

Цель варианта хранения, предоставляемого разреженными столбцами, значительно отличается от цели хранения типа FILESTREAM. Тогда как целью хранения типа FILESTREAM является хранение объектов LOB вне базы данных, целью разреженных столбцов является минимизировать дисковое пространство, занимаемое базой данных.

Столбцы этого типа позволяют оптимизировать хранение столбцов, большинство значений которых равны null. При использовании разреженных столбцов для хранения значений null дисковое пространство не требуется, но, с другой стороны, для хранения значений, отличных от null, требуется дополнительно от 2 до 4 байтов, в зависимости от их типа. По этой причине разработчики Microsoft рекомендуют использовать разреженные столбцы только в тех случаях, когда ожидается, по крайней мере, 20% общей экономии дискового пространства.

Разреженные столбцы определяются таким же образом, как и прочие столбцы таблицы; аналогично осуществляется и обращение к ним. Это означает, что для обращения к разреженным столбцам можно использовать инструкции SELECT, INSERT, UPDATE и DELETE таким же образом, как и при обращении к обычным столбцам. Единственная разница касается создания разреженных столбцов: для определения конкретного столбца разреженным применяется **аргумент SPARSE** после названия столбца, как это показано в данном примере:

имя_столбца тип_данных SPARSE

Несколько разреженных столбцов таблицы можно сгруппировать в набор столбцов. Такой набор будет альтернативным способом сохранять значения во всех разреженных столбцах таблицы и обращаться к ним.

Значение NULL

Значение null - это специальное значение, которое можно присвоить ячейке таблицы. Это значение обычно применяется, когда информация в ячейке неизвестна или неприменима. Например, если неизвестен номер домашнего телефона служащего компании, рекомендуется присвоить соответствующей ячейке столбца `home_telephone` значение null.

Если значение любого операнда любого арифметического выражения равно null, значение результата вычисления этого выражения также будет null. Поэтому в унарных арифметических операциях, если значение выражения A равно null, тогда как +A, так и -A возвращает null. В бинарных выражениях, если значение одного или обоих операндов A и B равно null, тогда результат операции сложения, вычитания, умножения, деления и деления по модулю этих операндов также будет null.

Если выражение содержит операцию сравнения и значение одного или обоих операндов этой операции равно null, результат этой операции также будет null.

Значение null должно отличаться от всех других значений. Для числовых типов данных значение 0 и значение null не являются одинаковыми. То же самое относится и к пустой строке и значению null для символьных типов данных.

Значения null можно сохранять в столбце таблицы только в том случае, если это явно разрешено в определении данного столбца. С другой стороны, значения null не разрешаются для столбца, если в его определении явно указано NOT NULL. Если для столбца с типом данных (за исключением типа TIMESTAMP) не указано явно NULL или NOT NULL, то присваиваются следующие значения:

- NULL, если значение параметра *ANSI_NULL_DFLT_ON* инструкции SET равно on.
- NOT NULL, если значение параметра *ANSI_NULL_DFLT_OFF* инструкции SET равно on.

Если инструкцию set не активировать, то столбец по умолчанию будет содержать значение NOT NULL. (Для столбцов типа TIMESTAMP значения null не разрешаются.)

Заполнение значениями.

Теперь рассмотрим операцию заполнения таблиц начальными данными. В первую очередь заполняем справочники нашей базы данных.

Для начала заполним таблицу "Departnem". Для заполнения этой таблицы в обозревателе объектов щелкните правой кнопкой мыши по таблице «Departnem» в появившемся меню выберите пункт «Изменить первые 200 строк». В рабочей области "Microsoft SQL Server Management Studio" проявится окно заполнения таблиц. Заполните таблицу " Departnem", как показано на рисунок. 2.14.

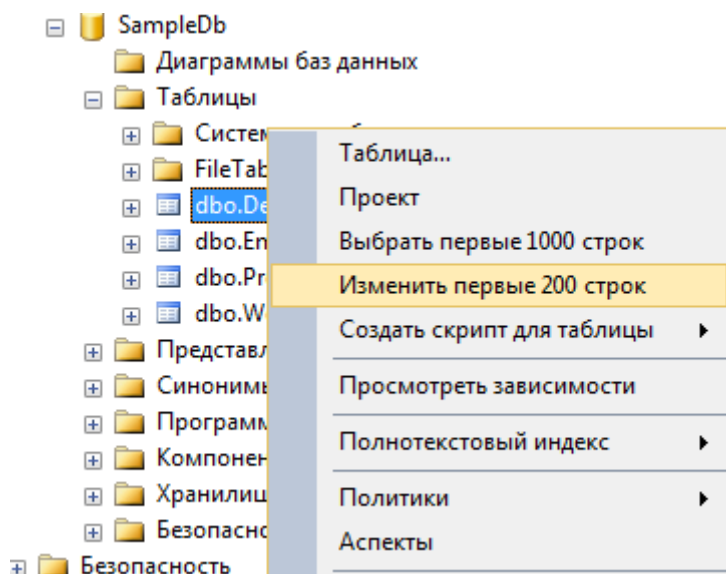


Рисунок. 2.14.

Замечание: Заполнение таблиц происходит полностью аналогично табличному процессору "Microsoft Excel 2000".

Задание к лабораторной работе 2.

Инструкция **CREATE TABLE** создает новую таблицу базы данных со всеми соответствующими столбцами требуемого типа данных. Далее приводится базовая форма инструкции CREATE TABLE:

```
CREATE TABLE table_name (
    col_name1 type1 [NOT NULL | NULL]
    [{, col_name2 type2 [NOT NULL | NULL]} ...]
)
```

Параметр table_name - имя создаваемой базовой таблицы. Максимальное количество таблиц, которое может содержать одна база данных, ограничивается количеством объектов базы данных, число которых не может быть более 2 миллиардов, включая таблицы, представления, хранимые процедуры, триггеры и ограничения. В параметрах col_name1, col_name2, ... указываются имена столбцов таблицы, а в параметрах type1, type2, ... - типы данных соответствующих столбцов.

Имя объекта базы данных может обычно состоять из четырех частей, в форме:

```
[server_name.[db_name.[schema_name.]]]object_name
```

Здесь object_name - это имя объекта базы данных, schema_name - имя схемы, к которой принадлежит объект, а server_name и db_name - имена сервера и базы данных, к которым принадлежит объект. Имена таблиц, сгруппированные с именем схемы, должны быть однозначными в рамках базы данных. Подобным образом имена столбцов должны быть однозначными в рамках таблицы.

Рассмотрим теперь ограничение, связанное с присутствием или отсутствием значений NULL в столбце. Если для столбца не указано, что значения NULL разрешены (NOT NULL), то данный столбец не может

содержать значения NULL, и при попытке вставить такое значение система возвратит сообщение об ошибке.

Как уже упоминалось, объект базы данных (в данном случае таблица) всегда создается в схеме базы данных. Пользователь может создавать таблицы только в такой схеме, для которой у него есть полномочия на выполнение инструкции ALTER. Любой пользователь с ролью sysadmin, db_ddladmin или db_owner может создавать таблицы в любой схеме.

Создатель таблицы не обязательно должен быть ее владельцем. Это означает, что один пользователь может создавать таблицы, которые принадлежат другим пользователям. Подобным образом таблица, создаваемая с помощью инструкции CREATE TABLE, не обязательно должна принадлежать к текущей базе данных, если в префиксе имени таблицы указать другую (существующую) базу данных и имя схемы.

Схема, к которой принадлежит таблица, может иметь два возможных имени по умолчанию. Если таблица указывается без явного имени схемы, то система выполняет поиск имени таблицы в соответствующей схеме по умолчанию. Если имя объекта найти в схеме по умолчанию не удастся, то система выполняет поиск в схеме dbo. Имена таблиц всегда следует указывать вместе с именем соответствующей схемы. Это позволит избежать возможных неопределенностей.

В примере ниже показано создание всех таблиц базы данных SampleDb. (База данных SampleDb должна быть установлена в качестве текущей базы данных.)

```
USE SampleDb;
```

```
CREATE TABLE Department (  
    Number      CHAR (4) NOT NULL,  
    DepartmentName NCHAR (40) NOT NULL,  
    Location     NCHAR (40) NULL  
);
```

```
CREATE TABLE [dbo].[Project] (  
    [Number]      CHAR (4) NOT NULL,  
    [ProjectName] NCHAR (15) NOT NULL,  
    [Budget]      FLOAT (53) NULL  
);
```

```
CREATE TABLE dbo.Employee (  
    Id           INT      NOT NULL,  
    FirstName     NCHAR (20) NOT NULL,  
    LastName      NCHAR (20) NOT NULL,  
    DepartamentNumber CHAR (4) NULL  
);
```

```
CREATE TABLE dbo.Works_on (  

```

```

EmpId      INT      NOT NULL,
ProjectNumber CHAR (4) NOT NULL,
Job        NCHAR (15) NULL,
EnterDate  DATE      NULL

```

);

Кроме типа данных и наличия значения NULL, в спецификации столбца можно указать следующие параметры:

- предложение DEFAULT;
- свойство IDENTITY.

Предложение DEFAULT в спецификации столбца указывает значение столбца по умолчанию, т.е. когда в таблицу вставляется новая строка, ячейка этого столбца будет содержать указанное значение, которое останется в ячейке, если в нее не будет введено другое значение. В качестве значения по умолчанию можно использовать константу, например одну из системных функций, таких как, USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, CURRENT_TIMESTAMP и NULL.

Столбец идентификаторов, создаваемый указанием свойства **IDENTITY**, может иметь только целочисленные значения, которые системой присваиваются обычно неявно. Каждое следующее значение, вставляемое в такой столбец, вычисляется, увеличивая последнее, вставленное в этот столбец, значение. Поэтому определение столбца со свойством IDENTITY содержит (явно или неявно) начальное значение и шаг инкремента (такой столбец еще называют столбцом с автоинкрементом).

Ниже показан пример использования этих инструкций:

```
USE SampleDb;
```

```

CREATE TABLE UserInfo (
    -- Для столбца Id будет использоваться автоинкремент
    IDENTITY(10,5),
    -- т.е. при вставке данных первому элементу будет присвоено
    -- значение 10, второму 15, третьему 20 и т.д.
    Id INT NOT NULL PRIMARY KEY IDENTITY (10,5),
    Login VARCHAR(40) NOT NULL,

    -- Для поля BirthDate будет указана дата по умолчанию
    -- (если это поле не задано явно при вставке данных)
    BirthDate DATETIME DEFAULT (
        -- По умолчанию -30 лет от текущей даты
        DATEADD(year, -30, GETDATE())
    )
)

```

Инструкция CREATE TABLE и ограничения декларативной целостности

Одной из самых важных особенностей, которую должна предоставлять СУБД, является способ обеспечения целостности данных. Ограничения,

которые используются для проверки данных при их модификации или вставке, называются **ограничениями для обеспечения целостности (integrity CONSTRAINTs)**. Обеспечение целостности данных может осуществляться пользователем в прикладной программе или же системой управления базами данных. Наиболее важными преимуществами предоставления ограничений целостности системой управления базами данных являются следующие:

- повышается надежность данных;
- сокращается время на программирование;
- упрощается техническое обслуживание.

Определение ограничений для обеспечения целостности посредством СУБД повышает надежность данных, поскольку устраняется возможность, что программист прикладного приложения может забыть реализовать их. Если ограничения целостности предоставляются прикладными программами, то все приложения, затрагиваемые этими ограничениями, должны содержать соответствующий код. Если код отсутствует хоть в одном приложении, то целостность данных будет поставлена под сомнение.

Если ограничения для обеспечения целостности не предоставляются системой управления базами данных, то их необходимо определить в каждой программе приложения, которая использует данные, включенные в это ограничение. В противоположность этому, если ограничения для обеспечения целостности предоставляются системой управления базами данных, то их требуется определить только один раз. Кроме этого, код для ограничений, предоставляемых приложениями, обычно более сложный, чем в случае таких же ограничений, предоставляемых СУБД.

Если ограничения для обеспечения целостности предоставляются СУБД, то в случае изменений ограничений, соответствующие изменения в коде необходимо реализовать только один раз - в системе управления базами данных. А если ограничения предоставляются приложениями, то модификацию для отражения изменений в ограничениях необходимо выполнить в каждом из этих приложений.

Системами управления базами данных предоставляются два типа ограничений для обеспечения целостности:

- декларативные ограничения для обеспечения целостности;
- процедурные ограничения для обеспечения целостности, реализуемые посредством триггеров.

Декларативные ограничения определяются с помощью инструкций языка DDL CREATE TABLE и ALTER TABLE. Эти ограничения могут быть уровня столбцов или уровня таблицы. Ограничения уровня столбцов определяются наряду с типом данных и другими свойствами столбца в объявлении столбца, тогда как ограничения уровня таблицы всегда определяются в конце инструкции CREATE TABLE или ALTER TABLE после определения всех столбцов.

Между ограничениями уровня столбцов и ограничениями уровня таблицы есть лишь одно различие: ограничения уровня столбцов можно

применять только к одному столбцу, в то время как ограничения уровня таблицы могут охватывать больше, чем один столбец таблицы.

Каждому декларативному ограничению присваивается имя. Это имя может быть присвоено явно посредством использования опции **CONSTRAINT** в инструкции **CREATE TABLE** или **ALTER TABLE**. Если опция **CONSTRAINT** не указывается, то имя ограничению присваивается неявно компонентом Database Engine. Настоятельно рекомендуется использовать явные имена ограничений, поскольку это может значительно улучшить поиск этих ограничений.

Декларативные ограничения можно сгруппировать в следующие категории:

- предложение **DEFAULT**;
- предложение **UNIQUE**;
- предложение **PRIMARY KEY**;
- предложение **CHECK**;
- ссылочная целостность и предложение **FOREIGN KEY**.

Использование предложения **DEFAULT** для определения ограничения по умолчанию было показано ранее. Все другие ограничения рассматриваются в последующих разделах.

Предложение UNIQUE

Иногда несколько столбцов или группа столбцов таблицы имеет уникальные значения, что позволяет использовать их в качестве первичного ключа. Столбцы или группы столбцов, которые можно использовать в качестве первичного ключа, называются *потенциальными ключами (candidate key)*. Каждый потенциальный ключ определяется, используя предложение **UNIQUE** в инструкции **CREATE TABLE** или **ALTER TABLE**. Синтаксис предложения **UNIQUE** следующий:

[**CONSTRAINT** *c_name*]

UNIQUE [**CLUSTERED** | **NONCLUSTERED**] ({ *col_name1* } ,...)

Опция **CONSTRAINT** в предложении **UNIQUE** присваивает явное имя потенциальному ключу. Опция **CLUSTERED** или **NONCLUSTERED** связана с тем обстоятельством, что компонент Database Engine создает индекс для каждого потенциального ключа таблицы. Этот индекс может быть кластеризованным, когда физический порядок строк определяется посредством индексированного порядка значений столбца. Если порядок строк не указывается, индекс является некластеризованным. По умолчанию применяется опция **NONCLUSTERED**. Параметр *col_name1* обозначает имя столбца, который создает потенциальный ключ. (Потенциальный ключ может иметь до 16 столбцов.)

Применение предложения **UNIQUE** показано в примере ниже. (Прежде чем выполнять этот пример, в базе данных SampleDb нужно удалить таблицу Projects, используя для этого инструкцию **DROP TABLE Projects**.)

USE SampleDb;

CREATE TABLE Projects (


```

        Number    CHAR(4) DEFAULT 'p1',
        ProjectName NCHAR (15) NOT NULL,
        Budget     FLOAT (53) NULL,
        CONSTRAINT unique_number UNIQUE (Number)
    );

```

);
 Каждое значение столбца Number таблицы Projects является уникальным, включая значение NULL. (Точно так же, как и для любого другого значения с ограничением UNIQUE, если значения NULL разрешены для соответствующего столбца, этот столбец может содержать не более одной строки со значением NULL.) Попытка вставить в столбец Number уже имеющееся в нем значение будет неуспешной, т.к. система не примет его. Явное имя ограничения, определяемого в примере - unique_number.

Предложение PRIMARY KEY

Первичным ключом таблицы является столбец или группа столбцов, значения которого разные в каждой строке. Каждый первичный ключ определяется, используя предложение **PRIMARY KEY** в инструкции CREATE TABLE или ALTER TABLE. Синтаксис предложения PRIMARY KEY следующий:

```

    [CONSTRAINT c_name]
    PRIMARY KEY [CLUSTERED | NONCLUSTERED] ({col_name1}
    ,....)

```

Все параметры предложения PRIMARY KEY имеют такие же значения, как и соответствующие одноименные параметры предложения UNIQUE. Но в отличие от столбца UNIQUE, столбец PRIMARY KEY не разрешает значений NULL и имеет значение по умолчанию CLUSTERED.

В примере ниже показано объявление первичного ключа для таблицы Employee базы данных SampleDb. Прежде чем выполнять этот пример, в базе данных SampleDb нужно удалить таблицу Employee, используя для этого инструкцию DROP TABLE Employee.

```

USE SampleDb;

```

```

CREATE TABLE Employee (
    Id          INT    NOT NULL,
    FirstName   NCHAR (20) NOT NULL,
    LastName    NCHAR (20) NOT NULL,
    DepartamentNumber CHAR (4) NULL,
    CONSTRAINT primary_id PRIMARY KEY (Id)
);

```

В результате выполнения этого кода снова создается таблица Employee, в которой определен первичный ключ. Первичный ключ таблицы определяется посредством декларативного ограничения для обеспечения целостности с именем primary_id. Это ограничение для обеспечения целостности является ограничением уровня таблицы, поскольку оно указывается после определения всех столбцов таблицы Employee.

Следующий пример эквивалентен предыдущему, за исключением того, что первичный ключ таблицы Employee определяется как ограничение уровня столбца.

```
USE SampleDb;
```

```
DROP TABLE Employee;
```

```
CREATE TABLE Employee (  
    Id INT NOT NULL CONSTRAINT primary_id PRIMARY KEY,  
    FirstName NCHAR (20) NOT NULL,  
    LastName NCHAR (20) NOT NULL,  
    DepartamentNumber CHAR (4) NULL  
);
```

В примере предложение PRIMARY KEY принадлежит к объявлению соответствующего столбца, наряду с объявлением его типа данных и свойства содержать значения NULL. По этой причине это ограничение называется *ограничением на уровне столбца*.

Предложение CHECK

Проверочное ограничение (CHECK CONSTRAINT) определяет условия для вставляемых в столбец данных. Каждая вставляемая в таблицу строка или каждое значение, которым обновляется значение столбца, должно отвечать этим условиям. Проверочные ограничения устанавливаются посредством предложения **CHECK**, определяемого в инструкции CREATE TABLE или ALTER TABLE. Синтаксис предложения CHECK следующий:

```
[CONSTRAINT c_name]
```

```
CHECK [NOT FOR REPLICATION] expression
```

Параметр expression должен иметь логическое значение (true или false) и может ссылаться на любые столбцы в текущей таблице (или только на текущий столбец, если определен как ограничение уровня столбца), но не на другие таблицы. Предложение CHECK не применяется принудительно при репликации данных, если присутствует параметр NOT FOR REPLICATION. (При репликации база данных, или ее часть, хранится в нескольких местах. С помощью репликации можно повысить уровень доступности данных.)

В примере ниже показано применение предложения CHECK:

```
USE SampleDb;
```

```
CREATE TABLE Customer (  
    CustomerId INTEGER NOT NULL,  
    CustomerRole VARCHAR(100) NULL,  
    CHECK (CustomerRole IN ('admin', 'moderator', 'user'))  
);
```

Создаваемая в примере таблица Customer включает столбец CustomerRole, содержащий соответствующее проверочное ограничение. При вставке нового значения, отличающегося от значений в наборе ('admin', 'moderator', 'user'), или при попытке изменения существующего значения на

значение, отличающегося от этих значений, система управления базой данных возвращает сообщение об ошибке.

Предложение FOREIGN KEY

Внешний ключ (foreign key) - это столбец (или группа столбцов таблицы), содержащий значения, совпадающие со значениями первичного ключа в этой же или другой таблице. Внешний ключ определяется с помощью предложения **FOREIGN KEY** в комбинации с предложением **REFERENCES**. Синтаксис предложения **FOREIGN KEY** следующий:

```
[CONSTRAINT c_name]
  [[FOREIGN KEY] ({col_name1} ,...)]
  REFERENCES table_name ({col_name2},...)
  [ON DELETE {NO ACTION | CASCADE | SET
NULL | SET DEFAULT}]
  [ON UPDATE {NO ACTION | CASCADE | SET
NULL | SET DEFAULT}]
```

Предложение **FOREIGN KEY** явно определяет все столбцы, входящие во внешний ключ. В предложении **REFERENCES** указывается имя таблицы, содержащей столбцы, создающие соответствующий первичный ключ. Количество столбцов и их тип данных в предложении **FOREIGN KEY** должны совпадать с количеством соответствующих столбцов и их типом данных в предложении **REFERENCES** (и, конечно же, они должны совпадать с количеством столбцов и типами данных в первичном ключе таблицы, на которую они ссылаются).

Таблица, содержащая внешний ключ, называется *ссылающейся (или дочерней) таблицей (referencing table)*, а таблица, содержащая соответствующий первичный ключ, называется *ссылочной (referenced table) или родительской (parent table) таблицей*. В примере ниже показано объявление внешнего ключа для таблицы **Works_on** базы данных **SampleDb**:

```
USE SampleDb;
```

```
CREATE TABLE Works_on (
  EmpId      INT      NOT NULL,
  ProjectNumber CHAR (4) NOT NULL,
  Job        NCHAR (15) NULL,
  EnterDate  DATE      NULL,
  CONSTRAINT primary_works PRIMARY KEY (EmpId,
ProjectNumber),

  CONSTRAINT foreign_employee FOREIGN KEY (EmpId)
REFERENCES Employee (Id),

  CONSTRAINT foreign_project FOREIGN KEY (ProjectNumber)
REFERENCES Projects (Number)
);
```

Таблица Works_on в этом примере задается с тремя декларативными ограничениями для обеспечения целостности: primary_works, foreign_employee и foreign_project. Эти ограничения являются ограничением уровня таблицы, где первое указывает первичный ключ, а второе и третье - внешний ключ таблицы Works_on. Кроме этого, внешние ключи определяют таблицы Employee и Projects, как ссылочные таблицы, а их столбцы Id и Number, как соответствующий первичный ключ столбца с таким же именем в таблице Works_on.

Предложение FOREIGN KEY можно пропустить, если внешний ключ определяется, как ограничение уровня таблицы, поскольку столбец, к которому применяется ограничение, является неявным "списком" столбцов внешнего ключа, и ключевого слова REFERENCES достаточно для указания того, какого типа является это ограничение. Таблица может содержать самое большее 63 ограничения FOREIGN KEY.

Определение внешних ключей в таблицах базы данных налагает определение другого важного ограничения для обеспечения целостности: ссылочной целостности.

Ссылочная целостность (referential integrity) обеспечивает выполнение правил для вставок и обновлений таблиц, содержащих внешний ключ и соответствующее ограничение первичного ключа. Пример выше имеет два таких ограничения: foreign_employee и foreign_project. Предложение REFERENCES в примере определяет таблицы Employee и Projects в качестве ссылочных (родительских) таблиц.

Если для двух таблиц указана ссылочная целостность, модифицирование значений в первичном ключе и соответствующем внешнем ключе будет не всегда возможным. В последующих разделах рассматривается, когда это возможно, а когда нет.

Модификация значений внешнего или первичного ключа может создавать проблемы в четырех случаях. Все эти случаи будут продемонстрированы с использованием базы данных SampleDb. В первых двух случаях затрагиваются модификации ссылающейся таблицы, а в последних двух - родительской.

Возможные проблемы со ссылочной целостностью - случай 1

Вставка новой строки в таблицу Works_on с номером сотрудника 11111. Соответствующая инструкция Transact-SQL выглядит таким образом:
USE SampleDb;

```
INSERT INTO Works_on VALUES (11111, 'p1', 'qwe', GETDATE())
```

При вставке новой строки в дочернюю таблицу Works_on используется новый номер сотрудника EmpId, для которого нет совпадающего сотрудника (и номера) в родительской таблице Employee. Если для обеих таблиц определена ссылочная целостность, как это сделано ранее, то компонент Database Engine не допустит вставки новой строки с таким номером EmpId.

Возможные проблемы со ссылочной целостностью - случай 2

Изменение номера сотрудника 9502 во всех строках таблицы Works_on на номер 11111. Соответствующая инструкция Transact-SQL выглядит таким образом:

```
USE SampleDb;
```

```
UPDATE Works_on
```

```
SET EmpId = 11111 WHERE EmpId = 9502;
```

В данном случае существующее значение внешнего ключа в ссылающейся таблице Works_on заменяется новым значением, для которого нет совпадающего значения в родительской таблице Employee. Если для обеих таблиц определена ссылочная целостность, то система управления базой данных не допустит модификацию строки с таким номером EmpId в таблице Works_on.

Возможные проблемы со ссылочной целостностью - случай 3

Замена значения 9502 номера сотрудника Id на значение 22222 в таблице Employee. Соответствующая инструкция Transact-SQL будет выглядеть таким образом:

```
USE SampleDb;
```

```
UPDATE Employee
```

```
SET Id = 22222 WHERE Id = 9502;
```

В данном случае предпринимается попытка заменить существующее значение 9502 номера сотрудника Id значением 22222 только в родительской таблице Employee, не меняя соответствующие значения Id в ссылающейся таблице Works_on. Система не разрешает выполнения этой операции. Ссылочная целостность не допускает существования в ссылающейся таблице (таблице, для которой предложением FOREIGN KEY определен внешний ключ) таких значений, для которых в родительской таблице (таблице, для которой предложением PRIMARY KEY определен первичный ключ) не существует соответствующего значения. В противном случае такие строки в ссылающейся таблице были бы "сиротами". Если бы описанная выше модификация таблицы Employee была разрешена, тогда строки в таблице Works_on со значением Id равным 9502 были бы сиротами. Поэтому система и не разрешает выполнения такой модификации.

Возможные проблемы со ссылочной целостностью - случай 4

Удаление строки в таблице Employee со значением Id равным 9502.

Этот случай похожий на случай 3. В случае выполнения этой операции, из таблицы Employee была бы удалена строка со значением Id, для которого существуют совпадающие значения в ссылающейся (дочерней) таблице Works_on.

Опции ON DELETE и ON UPDATE

Компонент Database Engine на попытку удаления и модифицирования первичного ключа может реагировать по-разному. Если попытаться обновить значения внешнего ключа, то все эти обновления будут несогласованы с

соответствующим первичным ключом, база данных откажется выполнять эти обновления и выведет сообщение об ошибке.

Но при попытке внести обновления в значения первичного ключа, вызывающие несогласованность в соответствующем внешнем ключе, система базы данных может реагировать достаточно гибко. В целом, существует четыре опции, определяющих то, как система базы данных может реагировать:

NO ACTION

Модифицируются (обновляются или удаляются) только те значения в родительской таблице, для которых нет соответствующих значений во внешнем ключе дочерней (ссылающейся) таблицы.

CASCADE

Разрешается модификация (обновление или удаление) любых значений в родительской таблице. При обновлении значения первичного ключа в родительской таблице или при удалении всей строки, содержащей данное значение, в дочерней (ссылающейся) таблице обновляются (т.е. удаляются) все строки с соответствующими значениями внешнего ключа.

SET NULL

Разрешается модификация (обновление или удаление) любых значений в родительской таблице. Если обновление значения в родительской таблице вызывает несогласованность в дочерней таблице, система базы данных присваивает внешнему ключу всех соответствующих строк в дочерней таблице значение NULL. То же самое происходит и в случае удаления строки в родительской таблице, вызывающего несогласованность в дочерней таблице. Таким образом, все несогласованности данных пропускаются.

SET DEFAULT

Аналогично опции SET NULL, но с одним исключением: всем внешним ключам, соответствующим модифицируемому первичному ключу, присваивается значение по умолчанию. Само собой разумеется, что после модификации первичный ключ родительской таблицы все равно должен содержать значение по умолчанию.

В языке Transact-SQL поддерживаются первые две из этих опций. Использование опций ON DELETE и ON UPDATE показано в примере ниже:

```
USE SampleDb;
```

```
CREATE TABLE Works_on (  
    EmpId      INT      NOT NULL,  
    ProjectNumber CHAR (4) NOT NULL,  
    Job        NCHAR (15) NULL,  
    EnterDate  DATE     NULL,  
    CONSTRAINT primary_works PRIMARY KEY (EmpId,  
ProjectNumber),  
  
    CONSTRAINT foreign_employee FOREIGN KEY (EmpId)  
REFERENCES Employee (Id) ON DELETE CASCADE,
```

**CONSTRAINT foreign_project FOREIGN KEY (ProjectNumber)
REFERENCES Projects (Number) ON UPDATE CASCADE**

);

В этом примере создается таблица Works_on с использованием опций ON DELETE CASCADE и ON UPDATE CASCADE. Если таблицу Works_on загрузить значениями, каждое удаление строки в таблице Employee будет вызывать каскадное удаление всех строк в таблице Works_on, которые имеют значения внешнего ключа, соответствующие значениям первичного ключа строк, удаляемых в таблице Employee. Подобным образом каждое обновление значения столбца Number таблицы Project будет вызывать такое же обновление всех соответствующих значений столбца ProjectNumber таблицы Works_on.

Изменение таблиц

Для модифицирования схемы таблицы применяется инструкция **ALTER TABLE**. Язык Transact-SQL позволяет осуществлять следующие виды изменений таблиц:

- добавлять и удалять столбцы;
- изменять свойства столбцов;
- добавлять и удалять ограничения для обеспечения целостности;
- разрешать или отключать ограничения;
- переименовывать таблицы и другие объекты базы данных.

Эти типы изменений рассматриваются в последующих далее разделах.

Добавление и удаление столбцов

Чтобы добавить новый столбец в существующую таблицу, в инструкции ALTER TABLE используется предложение **ADD**. В одной инструкции ALTER TABLE можно добавить только один столбец. Применение предложения ADD показано в примере ниже:

USE SampleDb;

ALTER TABLE Employee

ADD PhoneNumber CHAR(12) NULL;

В этом примере инструкция ALTER TABLE добавляет в таблицу Employee столбец PhoneNumber. Компонент Database Engine заполняет новый столбец значениями NULL или IDENTITY или указанными значениями по умолчанию. По этой причине новый столбец должен или поддерживать значения NULL, или для него должно быть указано значение по умолчанию.

Новый столбец нельзя вставить в таблицу в какой-либо конкретной позиции. Столбец, добавляемый предложением ADD, всегда вставляется в конец таблицы.

Столбцы из таблицы удаляются посредством предложения **DROP COLUMN**. Применение этого предложения показано в примере ниже:

USE SampleDb;

```
ALTER TABLE Employee  
    DROP COLUMN PhoneNumber;
```

В этом коде инструкция **ALTER TABLE** удаляет в таблице **Employee** столбец **PhoneNumber**, который был добавлен в эту таблицу предложением **ADD** ранее.

Изменение свойств столбцов

Для изменения свойств существующего столбца применяется предложение **ALTER COLUMN** инструкции **ALTER TABLE**. Изменению поддаются следующие свойства столбца:

- тип данных;
- поддержка значения **NULL**.

Применение предложения **ALTER COLUMN** показано в примере ниже:
USE SampleDb;

```
ALTER TABLE Department  
    ALTER COLUMN Location NCHAR(25) NOT NULL;
```

Инструкция **ALTER TABLE** в этом примере изменяет начальные свойства (**nchar**(40), значения **NULL** разрешены) столбца **Location** таблицы **Department** на новые (**nchar**(25), значения **NULL** не разрешены).

Добавление и удаления ограничений для обеспечения целостности (ключей и проверок)

Для добавления в таблицу новых ограничений для обеспечения целостности используется параметр **ADD CONSTRAINT** инструкции **ALTER TABLE**. В примере ниже показано использование параметра **ADD CONSTRAINT** для добавления проверочного ограничения и определения первичного ключа таблицы:

```
USE SampleDb;
```

```
CREATE TABLE Sales (  
    Id INTEGER NOT NULL,  
    OrderDate DATE NOT NULL,  
    ShipDate DATE NOT NULL);
```

```
-- Добавляем ограничение для дат (поля OrderDate и ShipDate)
```

```
ALTER TABLE Sales  
    ADD CONSTRAINT order_check CHECK(OrderDate <=  
ShipDate);
```

```
-- Добавляем первичный ключ для столбца Id
```

```
ALTER TABLE Sales  
    ADD CONSTRAINT pkey_sales PRIMARY KEY(Id);
```

В этом примере сначала инструкцией **CREATE TABLE** создается таблица **Sales**, содержащая два столбца с типом данных **DATE**: **OrderDate** и **ShipDate**. Далее, инструкция **ALTER TABLE** определяет ограничение для обеспечения целостности **order_check**, которое сравнивает значения обоих

этих столбцов и выводит сообщение об ошибке, если дата отправки ShipDate более ранняя, чем дата заказа OrderDate. Далее инструкция ALTER TABLE используется для определения первичного ключа таблицы в столбце Id.

Ограничения для обеспечения целостности можно удалить посредством предложения **DROP CONSTRAINT** инструкции ALTER TABLE, как это показано в примере ниже:

```
ALTER TABLE Sales
```

```
DROP CONSTRAINT order_check, pkey_sales;
```

Определения существующих ограничений нельзя модифицировать. Чтобы изменить ограничение, его сначала нужно удалить, а потом создать новое, содержащее требуемые модификации.

Разрешение и запрещение ограничений

Как упоминалось ранее, ограничение для обеспечения целостности всегда имеет имя, которое может быть объявленным или явно посредством опции CONSTRAINT, или неявно посредством системы. Имена всех ограничений таблицы (объявленных как явно, так и неявно) можно просмотреть с помощью системной процедуры **sp_helpconstraint**.

В последующих операциях вставки или обновлений значений в соответствующий столбец ограничение по умолчанию обеспечивается принудительно. Кроме этого, при объявлении ограничения все существующие значения соответствующего столбца проверяются на удовлетворение условий ограничения. Начальная проверка не выполняется, если ограничение создается с параметром WITH NOCHECK. В таком случае ограничение будет проверяться только при последующих операциях вставки и обновлений значений соответствующего столбца. (Оба параметра - WITH CHECK и WITH NOCHECK - можно применять только с ограничениями проверки целостности CHECK и проверки внешнего ключа FOREIGN KEY.)

В примере ниже показано, как отключить все существующие ограничения таблицы:

```
USE SampleDb;
```

```
ALTER TABLE Sales NOCHECK CONSTRAINT ALL;
```

Все ограничения таблицы Sales отключаются посредством ключевого слова ALL. Применять опцию NOCHECK не рекомендуется, поскольку любые подавленные нарушения условий ограничения могут вызвать ошибки при будущих обновлениях.

Переименование таблиц и других объектов баз данных

Для изменения имени существующей таблицы (и любых других объектов базы данных, таких как база данных, представление или хранимая процедура) применяется системная процедура **sp_rename**. В примере ниже показано использование этой системной процедуры:

```
USE SampleDb;
```

```
-- Переименование таблицы Sales в BigSales
```

```
EXEC sp_rename @objname = Sales, @newname = BigSales;
```

-- Переименование столбца OrderDate в таблице BigSales
EXEC sp_rename @objname = 'BigSales.OrderDate', @newname =
date_order;

Использовать системную процедуру sp_rename настоятельно не рекомендуется, поскольку изменение имен объектов может повлиять на другие объекты базы данных, которые ссылаются на них. Вместо этого следует удалить объект и воссоздать его с новым именем.

Удаление объектов баз данных

Все инструкции Transact-SQL для удаления объектов базы данных имеют следующий общий вид:

DROP тип_объекта имя_объекта;

Для каждой инструкции CREATE object для создания объекта имеется соответствующая инструкция DROP object для удаления. Инструкция для удаления одной или нескольких баз данных имеет следующий вид:

DROP DATABASE database1 {, database2, ...}

Эта инструкция безвозвратно удаляет базу данных из системы баз данных. Для удаления одной или нескольких таблиц применяется следующая инструкция:

DROP TABLE table_name1 {, table_name2, ...}

При удалении таблицы удаляются все ее данные, индексы и триггеры. Но представления, созданные по удаленной таблице, не удаляются. Таблицу может удалить только пользователь, имеющий соответствующие разрешения.

Кроме объектов DATABASE и TABLE, в параметре objects инструкции DROP можно указывать, среди прочих, следующие объекты:

- TYPE;
- VIEW;
- SYNONYM;
- PROCEDURE;
- SCHEMA;
- INDEX;
- TRIGGER.

Задание к лабораторной работе 2.

1. В соответствии с вариантом необходимо создать соответствующие таблицы «Создать запрос» с помощью следующих команд:

use [БД]

CREATE TABLE [Таблица 1] (поля таблицы с указанием типов данных)

CREATE TABLE [Таблица 2] (поля таблицы с указанием типов данных)...

В SQL Server Enterprise MANAGER в разделе диаграмм созданной БД сгенерировать новую диаграмму и установите связи между таблицами.

2. Заполняем созданные таблицы данными.

Язык манипуляции данными DML (Data Manipulation Language) содержит инструкции: SELECT, INSERT, UPDATE и DELETE.

Инструкция INSERT вставляет строки (или части строк) в таблицу. Существует две разные формы этой инструкции:

```
INSERT [INTO] tab_name [(col_list)]  
    DEFAULT VALUES | VALUES ( { DEFAULT | NULL | expression } [  
    ,...n] )
```

```
INSERT INTO tab_name | view_name [(col_list)]  
    {select_statement | execute_statement}
```

Первая форма инструкции позволяет вставить в таблицу одну строку (или часть ее). А вторая форма инструкции INSERT позволяет вставить в таблицу результирующий набор инструкции SELECT или хранимой процедуры, выполняемой посредством инструкции EXECUTE. Хранимая процедура должна возвращать данные для вставки в таблицу. Применяемая с инструкцией INSERT инструкция SELECT может выбирать значения из другой или той же самой таблицы, в которую вставляются данные, при условии совместимости типов данных соответствующих столбцов.

Для обеих форм тип данных каждого вставляемого значения должен быть совместимым с типом данных соответствующего столбца таблицы. Все строковые и временные данные должны быть заключены в кавычки; численные значения заключать в кавычки не требуется.

Вставка одной строки

Для обеих форм инструкции INSERT явное указание списка столбцов не является обязательным. Отсутствие списка столбцов равнозначно указанию всех столбцов таблицы.

Параметр DEFAULT VALUES вставляет значения по умолчанию для всех столбцов. В столбцы с типом данных TIMESTAMP или свойством IDENTITY по умолчанию вставляются значения, автоматически создаваемые системой. Для столбцов других типов данных вставляется соответствующее ненулевое значение по умолчанию, если таково имеется, или NULL в противном случае. Если для столбца значения NULL не разрешены и для него не определено значение по умолчанию, выполнение инструкции INSERT завершается ошибкой и выводится соответствующее сообщение.

В примере ниже показана вставка строк в таблицу Employee базы данных SampleDb, демонстрируя использование инструкции INSERT для вставки небольшого объема данных в базу данных:

```
USE SampleDb;  
INSERT INTO Employee VALUES (34990, 'Андрей', 'Батонов', 'd1');  
INSERT INTO Employee VALUES (38640, 'Алексей', 'Васин', 'd3');
```

Существует два разных способа вставки значений в новую строку. Инструкция INSERT в примере ниже явно использует ключевое слово NULL и вставляет значение NULL в соответствующий столбец:

```
USE SampleDb;
```

```
INSERT INTO Employee VALUES (34991, 'Андрей', 'Батонов', NULL);
```

Чтобы вставить значения в некоторые (но не во все) столбцы таблицы, обычно необходимо явно указать эти столбцы. Не указанные столбцы должны или разрешать значения **NULL**, или для них должно быть определено значение по умолчанию.

```
USE SampleDb;
```

```
INSERT INTO Employee(Id, FirstName, LastName)  
VALUES (34992, 'Андрей', 'Батонов');
```

Предыдущие два примера равнозначны. В таблице Employee единственным столбцом, разрешающим значения **NULL**, является столбец DepartmentNumber, а для всех прочих столбцов это значение было запрещено предложением **NOT NULL** в инструкции **CREATE TABLE**.

Порядок значений в предложении **VALUES** инструкции **INSERT** может отличаться от порядка, указанного в инструкции **CREATE TABLE**. В таком случае их порядок должен совпадать с порядком, в котором соответствующие столбцы перечислены в списке столбцов. Ниже показан пример вставки данных в порядке, отличающемся от исходного:

```
USE SampleDb;
```

```
INSERT INTO Employee(DepartamentNumber, LastName, Id, FirstName)  
VALUES ('d1', 'Батонов', 34993, 'Андрей');
```

Конструкторы значений таблицы и инструкция INSERT

Конструктор значений таблицы или строки (table (row) value constructor) позволяет вставить в таблицу несколько записей (строк) посредством инструкции языка DML, такой как, например, **INSERT** или **UPDATE**. В примере ниже показана вставка в таблицу нескольких строк, используя такой конструктор с помощью инструкции **INSERT**:

```
USE SampleDb;
```

```
INSERT INTO Employee  
VALUES (34995, 'Андрей', 'Батонов', 'd1'),  
        (38641, 'Алексей', 'Васин', 'd3'),  
        (12590, 'Светлана', 'Рыжова', 'd3');
```

В этом примере ниже инструкция **INSERT** одновременно вставляет три строки в таблицу Department, используя конструктор значений таблицы. Как можно видеть, синтаксис этого конструктора довольно простой. Для вставки в таблицу строк с данными посредством конструктора значений таблицы нужно в круглых скобках перечислить значения каждой строки, разделяя как значения каждого списка, так и отдельные списки запятыми.

Инструкция UPDATE

Инструкция UPDATE используется для модифицирования строк таблицы. Эта инструкция имеет следующую общую форму:

```
UPDATE tab_name
```

```
{ SET column_1 = {expression | DEFAULT | NULL} [...n]  
[FROM tab_name1 [...n]]  
[WHERE condition]
```

Строки таблицы `tab_name` выбираются для изменения в соответствии с условием в предложении `WHERE`. Значения столбцов каждой модифицируемой строки изменяются с помощью *предложения SET* инструкции `UPDATE`, которое соответствующему столбцу присваивает выражение (обычно) или константу. Если предложение `WHERE` отсутствует, то инструкция `UPDATE` модифицирует все строки таблицы. С помощью инструкции `UPDATE` данные можно модифицировать только в одной таблице.

В примере ниже инструкция `UPDATE` изменяет всего лишь одну строку таблицы `Works_on`, поскольку комбинация столбцов `EmpId` и `ProjectNumber` является первичным ключом этой таблицы и, следовательно, она однозначна. В данном примере изменяется должность сотрудника, значение которого было ранее неизвестно или имело значение `NULL`:

```
USE SampleDb;  
UPDATE Works_on  
SET Job = 'Менеджер'  
WHERE EmpId = 9031 AND ProjectNumber = 'p3';
```

В примере ниже значения строкам таблицы присваиваются посредством выражения. Запрос пересчитывает бюджеты всех проектов с долларов на евро:

```
USE SampleDb;  
  
UPDATE Project  
SET Budget = Budget * 0.9;
```

В данном примере изменяются все строки таблицы `Project`, поскольку в запросе отсутствует предложение `WHERE`.

В примере ниже в предложении `WHERE` инструкции `UPDATE` используется вложенный запрос. Поскольку применяется оператор `IN`, то этот запрос может вернуть более одной строки:

```
USE SampleDb;  
  
UPDATE Works_on  
SET Job = NULL  
WHERE EmpId IN (SELECT Id  
FROM Employee  
WHERE LastName = 'Вершинина');
```

Согласно этому запросу, для сотрудницы Вершининой Натальи во всех ее проектах в столбце ее должности присваивается значение `NULL`. Запрос в этом примере можно также выполнить посредством предложения `FROM` инструкции `UPDATE`. В предложении `FROM` указываются имена таблиц, которые обрабатываются инструкцией `UPDATE`. Все эти таблицы должны

быть в дальнейшем соединены. Применение предложения FROM показано в примере ниже. Логически, этот пример идентичен предыдущему:

```
USE SampleDb;
```

```
UPDATE Works_on
SET Job = NULL
FROM Works_on, Employee
WHERE LastName = 'Вершинина'
AND Works_on.EmpId = Employee.Id;
```

В примере ниже показано использование выражения CASE в инструкции UPDATE. (Подробное рассмотрение этого выражения описывалось ранее.) В данном примере нужно увеличить бюджет всех проектов на определенное число процентов (20, 10 или 5), в зависимости от исходной суммы бюджета: чем меньше бюджет, тем больше должно быть его процентное увеличение:

```
USE SampleDb;
```

```
UPDATE Project
SET Budget = CASE
WHEN Budget > 0 AND Budget <= 100000 THEN Budget * 1.2
WHEN Budget > 100000 AND Budget < 150000 THEN Budget * 1.1
ELSE Budget * 1.05
END;
```

Инструкция DELETE

Инструкция DELETE удаляет строки из таблицы. Подобно инструкции INSERT, эта инструкция также имеет две различные формы:

```
DELETE FROM table_name
[WHERE predicate];
```

```
DELETE table_name
FROM table_name [...n]
[WHERE condition];
```

Удаляются все строки, которые удовлетворяют условию в предложении WHERE. Явно перечислять столбцы в инструкции DELETE не то чтобы нет необходимости, а даже не разрешается, поскольку эта инструкция оперирует строками, а не столбцами. Использование первой формы инструкции DELETE показано в примере ниже, в котором происходит удаление из таблицы Works_on всех сотрудников с должностью 'Менеджер':

```
USE SampleDb;
```

```
DELETE FROM Works_on
WHERE Job = 'Менеджер';
```

Предложение WHERE инструкции DELETE может содержать вложенный запрос, как это показано в примере ниже:

```

USE SampleDb;
DELETE FROM Works_on
  WHERE EmpId IN
    (SELECT Id
     FROM Employee
     WHERE LastName = 'Вершинина');
DELETE FROM Employee
  WHERE LastName = 'Вершинина';

```

Поскольку сотрудница Вершинина уволилась, из базы данных удаляются все записи, связанные с ней. Запрос из этого примера можно также выполнить с помощью предложения FROM, как это показано ниже. В данном случае семантика этого предложения такая же, как и предложения FROM в инструкции UPDATE.

```

USE SampleDb;

DELETE Works_on
  FROM Works_on w, Employee e
  WHERE w.EmpId = e.Id
        AND LastName = 'Вершинина';

```

```

DELETE FROM Employee
  WHERE LastName = 'Вершинина';

```

Использование предложения WHERE в инструкции DELETE не является обязательным. Если это предложение отсутствует, то из таблицы удаляются все строки:

```

USE SampleDb;
-- Удаление всех строк таблицы
DELETE FROM Works_on;

```

Инструкции DELETE и DROP TABLE существенно отличаются друг от друга. Инструкция DELETE удаляет (частично или полностью) содержимое таблицы, тогда как инструкция DROP TABLE удаляет как содержимое, так и схему таблицы. Таким образом, после удаления всех строк посредством инструкции DELETE таблица продолжает существовать в базе данных, а после выполнения инструкции DROP TABLE таблица больше не существует.

Задание для лабораторной работы №2.

Реализовать правила ссылочной целостности в базе данных, в том числе целостность ключей, целостность отношений. Разработать и реализовать корпоративные правила целостности данных в виде triggers (не менее 1 шт.) и constraints (не менее 1 шт.).

Лабораторная работа №3. Создание запросов и фильтров

Цель: научиться создавать запросы и фильтры

В отличие от базовых таблиц, представления по умолчанию не существуют физически, т.е. их содержимое не сохраняется на диске. Это не относится к так называемым индексированным представлениям, которые рассматриваются позже.

Представления (views) - это объекты базы данных, которые всегда создаются на основе одной или более базовых таблиц (или других представлений), используя информацию метаданных. Эта информация (включая имя представления и способ получения строк из базовых таблиц) - все, что сохраняется физически для представления. По этой причине представления также называются виртуальными таблицами.

Перейдем к созданию статических запросов. В обозревателе объектов "Microsoft SQL Server 2014" все запросы БД находятся в папке "Представления" (рис. 3.1).

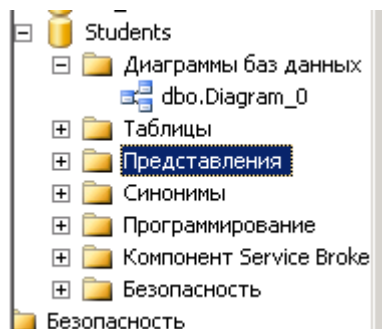


Рис. 3.1.

Создадим запрос "Запрос Сотрудники+отделы", связывающий таблицы "Department" и "Employee" по полю связи "ID". Для создания нового запроса необходимо в обозревателе объектов в БД "SampleDb" щелкнуть ПКМ по папке "Представления", затем в появившемся меню выбрать пункт "Новое представление". Появится окно "Добавление таблицы", предназначенное для выбора таблиц и запросов, участвующих в новом запросе (рис. 3.2).

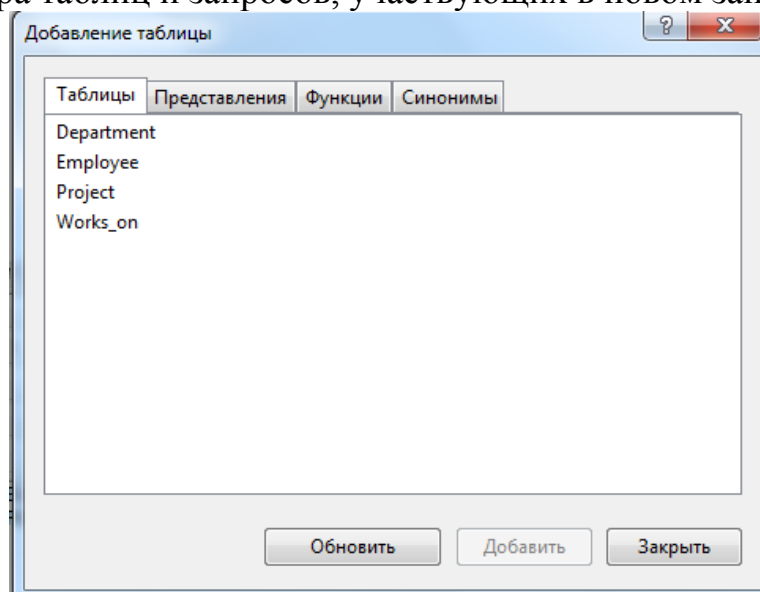


Рис. 3.2.

Добавим в новый запрос таблицы " Department" и "Employee". Для этого в окне "Добавление таблицы" выделите таблицу " Department " и нажмите кнопку "Добавить". Аналогично добавьте таблицу " Employee ". После добавления таблиц, участвующих в запросе, закройте окно "Добавление таблицы" нажав кнопку "Закорыть". Появится окно конструктора запросов (рис. 3.3).

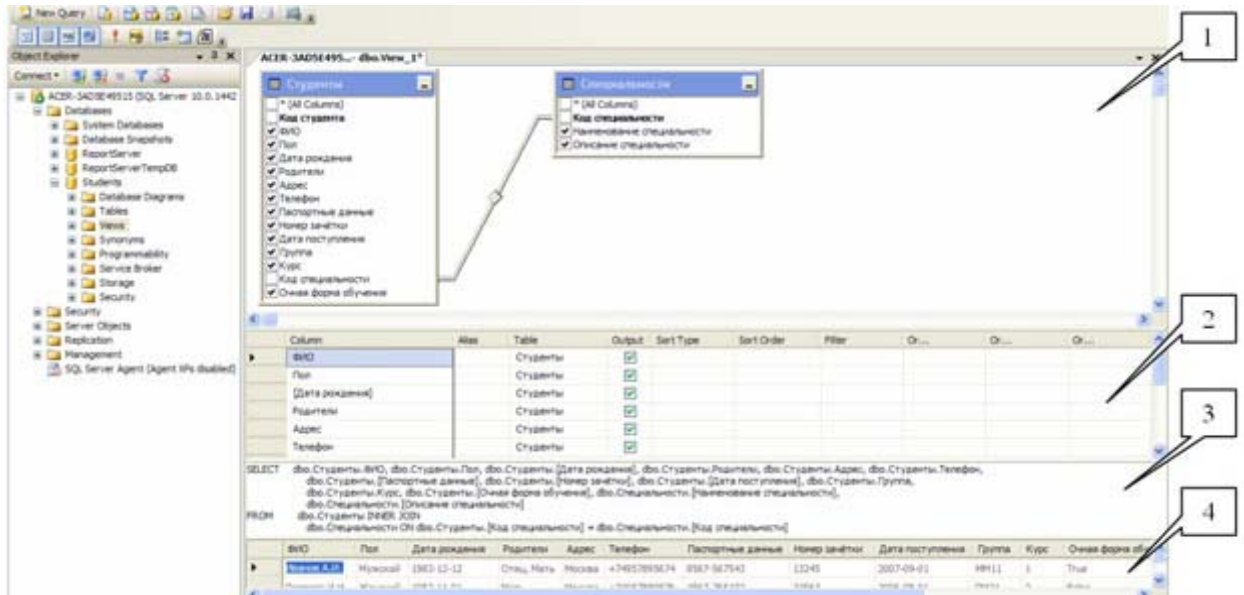


Рис. 3.3.

Замечание: Окно конструктора запросов состоит из следующих панелей:

1. Схема данных - отображает поля таблиц и запросов, участвующих в запросе, позволяет выбирать отображаемые поля, позволяет устанавливать связи между участниками запроса по специальным полям связи. Эта панель включается и выключается следующей кнопкой на панели инструментов

2. Таблица отображаемых полей - показывает отображаемые поля (столбец "Column"), позволяет задавать им псевдонимы (столбец "Alias"), позволяет устанавливать тип сортировки записей по одному или нескольким полям (столбец "Sort Type"), позволяет задавать порядок сортировки (столбец "Sort Order"), позволяет задавать условия отбора записей в фильтрах (столбцы "Filter" и "Or..."). Также эта таблица позволяет менять порядок отображения полей в запросе. Эта панель включается и выключается следующей кнопкой на панели инструментов

3. Код SQL - код создаваемого запроса на языке T-SQL. Эта панель включается и выключается следующей кнопкой на панели инструментов

4. Результат - показывает результат запроса после его выполнения. Эта панель включается и выключается следующей кнопкой на панели инструментов

Замечание: Если необходимо снова отобразить окно "Add Table" для добавления новых таблиц или запросов, то для этого на панели инструментов "Microsoft SQL Server 2014" нужно нажать кнопку

Замечание: Если необходимо удалить таблицу или запрос из схемы данных, то для этого нужно щелкнуть ПКМ и в появившемся меню выбрать пункт "Удалить".

Теперь перейдем к связыванию таблиц "Department" и "Employee" по полям связи "ID". Чтобы создать связь необходимо в схеме данных перетащить мышью поле "ID" таблицы "Employee" на такое же поле таблицы "Department". Связь отобразится в виде ломаной линии соединяющей эти два поля связи.

Замечание: Если необходимо удалить связь, то для этого необходимо щелкнуть по ней ПКМ и в появившемся меню выбрать пункт "Удалить".


Замечание: После связывания таблиц (а также при любых изменениях в запросе) в области кода T-SQL будет отображаться T-SQL код редактируемого запроса.

Теперь определим поля, отображаемые при выполнении запроса. Отображаемые поля обозначаются галочкой (слева от имени поля) на схеме данных, а также отображаются в таблице отображаемых полей. Чтобы сделать поле отображаемым при выполнении запроса необходимо щелкнуть мышью по пустому квадрату (слева от имени поля) на схеме данных, в квадрате появится галочка.

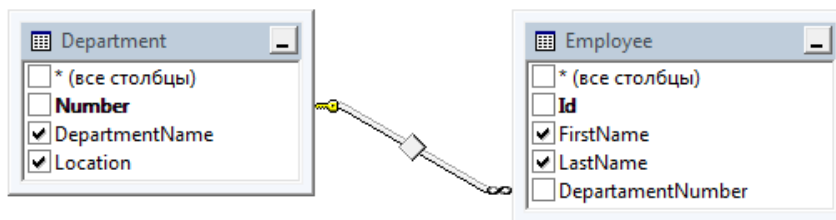
Замечание: Если необходимо сделать поле невидимым при выполнении запроса, то нужно убрать галочку, расположенную слева от имени поля на схеме данных. Для этого просто щелкните мышью по галочке.

Замечание: Если необходимо отобразить все поля таблицы, то необходимо установить галочку слева от пункта "*" (Все поля)", принадлежащего соответствующей таблице на схеме данных.

Определите отображаемые поля нашего запроса. Отображаются все поля кроме полей с кодами, то есть полей связи.

На этом настройку нового запроса можно считать законченной. Перед сохранением запроса проверим его работоспособность, выполнив его. Для запуска запроса на панели инструментов нажмите кнопку 

Либо щелкните ПКМ в любом месте окна конструктора запросов и в появившемся меню выберите пункт "Выполнить SQL". Результат выполнения запроса появится в виде таблицы в области результата (рис. 3.3).



Столбец	Псевдо...	Таблица	Выход	Тип сортиро...	Порядок сор...	Фильтр	Или
FirstName		Employee	<input checked="" type="checkbox"/>				
DepartmentN...		Department	<input checked="" type="checkbox"/>				
Location		Department	<input checked="" type="checkbox"/>				
LastName		Employee	<input checked="" type="checkbox"/>				

ECT
M

dbo.Employee.FirstName, dbo.Department.DepartmentName, dbo.Department.Location, dbo.Employee.Las
dbo.Department INNER JOIN
dbo.Employee ON dbo.Department.Number = dbo.Employee.DepartmentNumber

FirstName	DepartmentName
Василий	Бух. учет
Елена	Бух. учет
Анна	Маркетинг
Игорь	Исследования
Дмитрий	Маркетинг

1 для 7 Ячейка доступна только для чтения.

Замечание: Если после выполнения запроса результат не появился, а появилось сообщение об ошибке, то в этом случае проверьте, правильно ли создана связь.

Если запрос выполняется правильно, то необходимо сохранить. Для сохранения запроса закройте окно конструктора запросов, щелкнув мышью по кнопке закрытия, расположенной в верхнем правом углу окна конструктора (над схемой данных). Появится окно с вопросом о сохранении запроса.

В данном окне необходимо нажать кнопку "Yes" (Да). Появится окно "Choose Name" (Выберите имя) (рис. 3.4).

Рис. 3.4.

В данном окне зададим имя нового запроса "Запрос Сотрудники+Отделы" и нажмем кнопку "Ок". Запрос появится в папке "Представления" в обозревателе объектов (рис. 3.5).

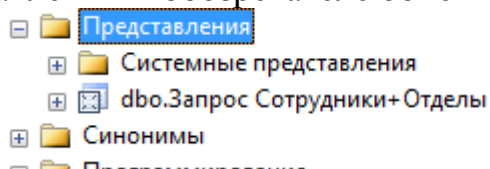


Рис. 3.5.

Проверим работоспособность созданного запроса вне конструктора запросов. Запустим вновь созданный запрос "Запрос Сотрудники+Отделы" без использования конструктора запросов. Для выполнения уже сохраненного запроса необходимо щелкнуть ПКМ по запросу и в появившемся меню выбрать пункт "Отобразить первые 1000 записей". Выполните эту операцию для запроса "Запрос Сотрудники+Отделы".

Перейдем к созданию запроса "Запрос Сотрудники+Проекты". В обозревателе объектов щелкните ПКМ по папке "Представления", затем в появившемся меню выберите пункт "New View". Появится окно "Add Table" (рис. 3.2).

В запросе "Запрос Сотрудники+Проекты" мы связываем таблицы "Employee" "Project" и «Works_on» по полям связи "Id". Следовательно, в окне "Добавление таблицы" в новый запрос добавляем соответствующие таблицы. После добавления таблиц закройте окно "Добавление таблицы", появится окно конструктора запросов.

В окне конструктора запросов установите связи между таблицами и определите отображаемые поля, как показано на рис. 3.7.

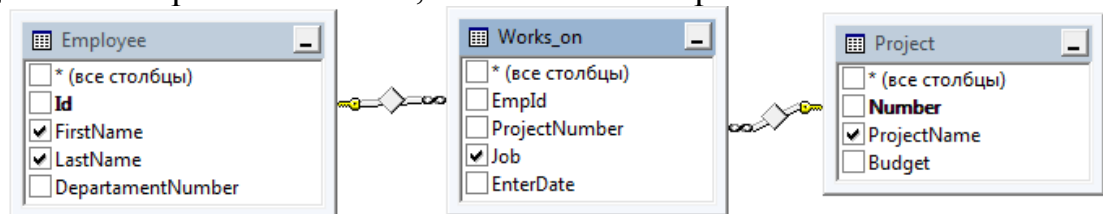


Рис. 3.7.

В поле «Псевдоним» опишем для пользователя более внятное представление полей таблиц. Расположите отображаемые поля в таблице отображаемых полей как показано на рис. 3.8.

The diagram illustrates the relationships between three tables: Employee, Works_on, and Project. Employee is connected to Works_on, which is connected to Project. Fields shown include Id, FirstName, LastName, DepartmentNumber, EmpId, ProjectNumber, Job, EnterDate, Number, ProjectName, and Budget.

Столбец	Псевдоним	Таблица	Выход	Тип сортиро...	Порядок сор...	Фильтр	Или...	Или...
FirstName	Имя	Employee	<input checked="" type="checkbox"/>					
LastName	Фамилия	Employee	<input checked="" type="checkbox"/>					
Job	Должность	Works_on	<input checked="" type="checkbox"/>					

Рис. 3.8.

Проверьте работоспособность нового запроса, выполнив его. Обратите внимание на то, что реальные названия полей были заменены их псевдонимами. Закройте окно конструктора запросов. В появившемся окне "Choose Name" задайте имя нового запроса "Запрос Сотрудники+Проекты".

Проверьте работоспособность нового запроса вне конструктора. Для этого запустите запрос. Результат выполнения запроса "Запрос Запрос Сотрудники+Проекты " должен выглядеть как на рис. 3.10.

Результаты		Сообщения		
	Имя	Фамилия	Должность	Название проекта
1	Анна	Иванова	Аналитик	Apollo
2	Анна	Иванова	Менеджер	Mercury
3	Дмитрий	Волков	Консультант	Gemini
4	Игорь	Соловьев	NULL	Gemini
5	Олег	Маменко	NULL	Gemini
6	Василий	Фролов	Аналитик	Mercury
7	Елена	Лебеденко	Менеджер	Apollo
8	Наталья	Вершинина	NULL	Apollo
9	Наталья	Вершинина	Консультант	Gemini
10	Елена	Лебеденко	Консультант	Mercury
11	Олег	Маменко	Консультант	Apollo

Рис. 3.10.

На этом мы заканчиваем рассмотрение обычных запросов и переходим к созданию фильтров.

На основе запроса " Запрос Сотрудники+Отделы" создадим фильтры, отображающие отдельных сотрудников. Создайте новый запрос. Так как он будет основан на запросе " Запрос Сотрудники+Отделы ", то в окне "Добавления таблиц" перейдите на вкладку "Представления" и добавьте в новый запрос " Запрос Сотрудники+Отделы " (рис. 3.11). Затем закройте окно "Добавление таблиц".

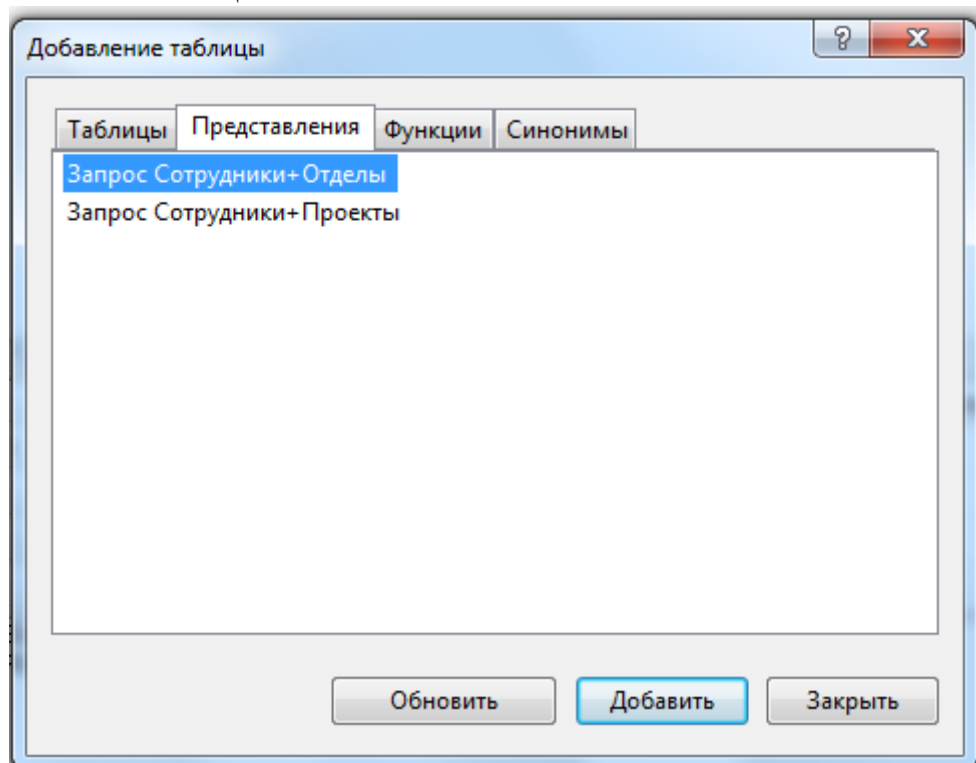


Рис. 3.11.

В появившемся окне конструктора запросов определите в качестве отображаемых полей все поля запроса "Запрос Сотрудники+Отделы" (рис. 3.12).

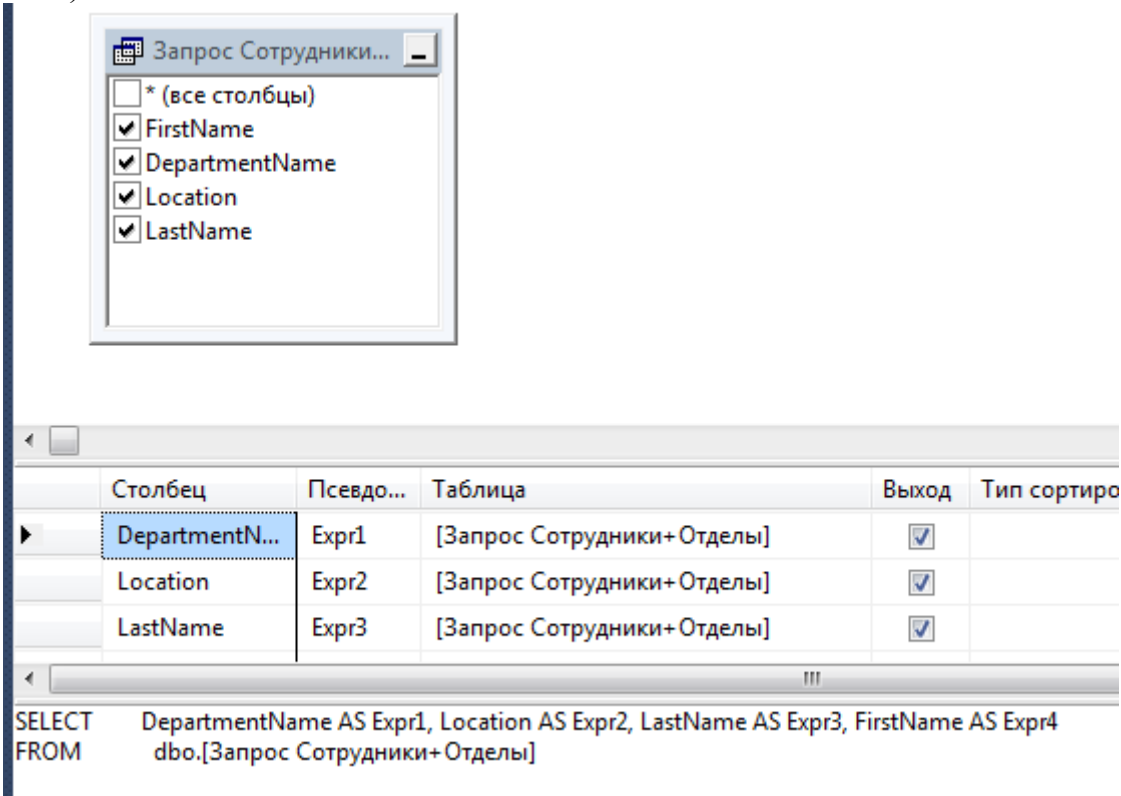


Рис. 3.12.

Замечание: Для отображения всех полей запроса, в данном случае, мы не можем использовать пункт "* Все поля". Так как в этом случае мы не можем устанавливать критерий отбора записей в фильтре, а также невозможно установить сортировку записей.

Теперь установим критерий отбора записей в фильтре. Пусть наш фильтр отображает только сотрудника Фролова. Для определения условия отбора записей в таблице отображаемых полей в строке, соответствующей полю, на которое накладывается условие, в столбце "Filter", необходимо задать условие. В нашем случае условие накладывается на поле "LastName". Следовательно, в строке " LastName", в столбце "Filter" нужно задать следующее условие отбора "='Фролов'" (рис. 3.12).

В заключение настроим сортировку записей в фильтре. Пусть при выполнении фильтра сначала происходит сортировка записей по возрастанию по полю " DepartmentName ", а затем по убыванию по полю " LastName ". Для установки сортировки записей по возрастанию, в таблице определяемых полей, в строке для поля "DepartmentNam", в столбце "Sort Type" (Тип сортировки), задайте "Ascending" (По возрастанию), а в строке для поля " LastName " - задайте "Descending" (По убыванию). Для определения порядка сортировки для поля " DepartmentName " в столбце "Sort Order" (Порядок сортировки) поставьте 1, а для поля " LastName " поставьте 2 (рис. 3.12). То есть, при выполнении запроса записи сначала сортируются по полю " DepartmentName ", а затем по полю " LastName".

Замечание: После установки условий отбора и сортировки записей на схеме данных напротив соответствующих полей появятся специальные значки. Значки



и



обозначают сортировку по возрастанию и убыванию, а значок



показывает наличие условия отбора.

После установки сортировки записей в фильтре проверим его работоспособность, выполнив его. Результат выполнения фильтра должен выглядеть как на рис. 3.12. Закройте окно конструктора запросов. В качестве имени нового фильтра в окне "Выбор имени" задайте "Фильтр Фролов" (рис. 3.13) и нажмите кнопку "Ok".

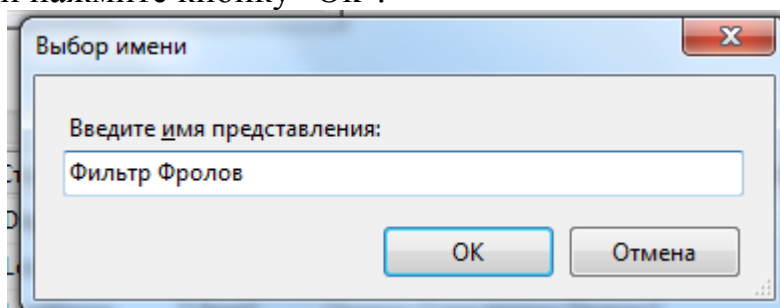


Рис. 3.13.

Фильтр "Фильтр Фролов" появится в обозревателе объектов. Выполните созданный фильтр вне окна конструктора запросов.

Задания для лабораторной работы 3.

В языке Transact-SQL имеется одна основная инструкция для выборки информации из базы данных - **инструкция SELECT**. Эта инструкция позволяет извлекать информацию из одной или нескольких таблиц базы данных и даже из нескольких баз данных. Результаты выполнения инструкции **SELECT** помещаются в еще одну таблицу, называемую *результатирующим набором (result set)*.

Самая простая форма инструкции **SELECT** состоит из списка столбцов выборки и **предложения FROM**. (Все прочие предложения являются необязательными.) Эта форма инструкции **SELECT** имеет следующий синтаксис:

```
SELECT [ ALL | DISTINCT ] column_list  
FROM {table1 [tab_alias1] } ,...
```

В параметре table1 указывается имя таблицы, из которой извлекается информация, а в параметре tab_alias1 - псевдоним имени соответствующей таблицы. Псевдоним - это другое, сокращенное, имя таблицы, посредством которого можно обращаться к этой таблице или к двум логическим

экземплярам одной физической таблицы. Если вы испытываете трудности с пониманием этого концепта, не волнуйтесь, поскольку все станет ясно в процессе рассмотрения примеров.

В параметре `column_list` указывается один или несколько из следующих спецификаторов:

- символ звездочка (*) указывает все столбцы таблиц, перечисленных в предложении FROM (или с одной таблицы, если задано указателем в виде `table2.*`);
- явное указание имен столбцов, из которых нужно извлечь значения;
- спецификатор в виде `column_name [as] column_heading`, что позволяет заменить имя столбца при чтении данных (не в базе) или присвоить новое имя выражению;
- выражение;
- системная или агрегатная функция.

Инструкция SELECT может извлекать из таблицы как отдельные столбцы, так и строки. Первая операция называется *списком выбора* (или проекцией), а вторая - *выборкой*. Инструкция SELECT также позволяет выполнять комбинацию обеих этих операций.

Перед тем как можно выполнять примеры запросов, вам необходимо заново создать базу данных SampleDb. В следующем примере показана самая простая форма выборки данных посредством инструкции SELECT:

```
USE SampleDb;
```

```
SELECT Number, DepartmentName, Location
```

```
FROM Department;
```

Результат выполнения этого запроса:

Results		Messages	
	Numb...	DepartmentName	Location
1	d1	Исследования	Москва
2	d2	Бух. учет	Санкт-Петербург
3	d3	Маркетинг	Москва

В примере инструкция SELECT извлекает все строки всех столбцов таблицы Department. Если список выбора инструкции SELECT содержит все столбцы таблицы (как это показано в примере), их можно указать с помощью звездочки (*), но использовать этот способ не рекомендуется. Имена столбцов служат в качестве заголовков столбцов в результирующем выводе.

Только что рассмотренная простейшая форма инструкции SELECT не очень полезна для практических запросов. На практике в запросах с инструкцией SELECT приходится применять намного больше предложений, чем в запросе, приведенном в примере выше. Далее показан синтаксис инструкции SELECT, содержащий почти все возможные предложения:

```
SELECT select_list
```

```
[INTO new_table_]
```

```
FROM table
```

```
[WHERE search_condition]
```


[**GROUP BY** group_by_expression]
 [**HAVING** search_condition]
 [**ORDER BY** order_expression [**ASC** | **DESC**]];

Порядок предложений в инструкции **SELECT** должен быть таким, как показано в приведенном синтаксисе. Например, предложение **GROUP BY** должно следовать за предложением **WHERE** и предшествовать предложению **HAVING**.

Далее мы рассмотрим эти предложения в том порядке, в каком они следуют в запросе, а также рассмотрим свойство **IDENTITY**, возможность упорядочивания результатов, операторы над множествами и выражение **CASE**. Но так как первое в списке предложение **INTO** менее важно, чем остальные, оно будет рассматриваться позже, после всех других предложений.

Предложение **WHERE**

Часто при выборке данных из таблицы нужны данные только из определенных строк, для чего в запросе определяется одно или несколько соответствующих условий. В предложении **WHERE** определяется логическое выражение (т.е. выражение, возвращающее одно из двух значений: **true** или **false**), которое проверяется для каждой из строк, кандидатов на выборку. Если строка удовлетворяет условию выражения, т.е. выражение возвращает значение **true**, она включается в выборку; в противном случае строка пропускается.

Применение предложения **WHERE** показано в примере ниже, в котором происходит выборка имен и номеров отделов, расположенных в Москве:

```
USE SampleDb;
SELECT Number, DepartmentName
FROM Department
WHERE Location = N'Москва';
```

Кроме знака равенства, в предложении **WHERE** могут применяться другие операторы сравнения, включая следующие:

Операторы сравнения в T-SQL

Оператор	Значение
<> (или !=)	не равно
<	меньше чем
>	больше чем
>=	больше чем или равно

<=	меньше чем или равно
!>	не больше чем
!<	не меньше чем

В примере ниже показано использование в предложении WHERE одного из этих операторов сравнения:

```
USE SampleDb;
SELECT FirstName, LastName
FROM Employee
WHERE Id >= 15000;
```

В этом примере происходит выборка имен и фамилий всех сотрудников, чей табельный номер больше или равен 15 000. Частью условия в предложении WHERE может быть выражение, показанное в примере ниже:

```
USE SampleDb;
SELECT ProjectName
FROM Project
WHERE Budget * 1.1 > 110000;
```

В этом примере происходит выборка проектов с бюджетом свыше 60 000 евро (предполагая, что в поле Budget хранится информация в долларах). Валютный курс: 1.1\$ за €1.

Сравнение строк (т.е. значений с типами данных CHAR, VARCHAR, NCHAR и NVARCHAR) выполняется в действующем порядке сортировки, а именно в порядке сортировки, указанном при установке компонента Database Engine. При сравнении строк в кодировке ASCII (или в любой другой кодировке) сравниваются соответствующие символы каждой строки (т.е. первый символ первой строки с первым символом второй строки, второй символ первой строки со вторым символом второй строки и т.д.). Старшинство символа определяется его позицией в кодовой таблице: символ, чей код стоит в таблице перед кодом другого, считается меньше этого символа. При сравнении строк разной длины, более короткая строка дополняется в конце пробелами до длины более длинной строки.

Числа сравниваются алгебраически. Значения временных типов данных (таких как DATE, TIME и DATETIME) сравниваются в хронологическом порядке.

Логические операторы

Условия предложения WHERE могут быть простыми или составными, т.е. содержащими несколько простых условий. Множественные условия можно создавать посредством логических операторов **AND**, **OR** и **NOT**.

При соединении условий оператором **AND** возвращаются только те строки, которые удовлетворяют обоим условиям. При соединении двух условий оператором **OR** возвращаются все строки таблицы, которые

удовлетворяют одному или обоим этим условиям, как показано в примере ниже:

```
USE SampleDb;  
SELECT EmpId, ProjectNumber  
FROM Works_on  
WHERE  
    ProjectNumber = 'p1'  
OR  
    ProjectNumber = 'p2'
```

В этом примере происходит выборка номеров сотрудников, которые работают над проектом p1 или p2 (или над обоими).

Результаты выполнения этого примера содержат дубликаты значений столбца Id. Эту избыточную информацию можно устранить с помощью ключевого слова **DISTINCT**, как показано в следующем примере:

```
USE SampleDb;  
SELECT DISTINCT EmpId  
FROM Works_on  
WHERE  
    ProjectNumber = 'p1'  
OR  
    ProjectNumber = 'p2'
```

Предложение WHERE может содержать любое число одинаковых или разных логических операций. Следует помнить, что логические операторы имеют разный приоритет выполнения: оператор NOT имеет самый высший приоритет, далее идет оператор AND, а оператор OR имеет самый низший приоритет. Неправильное размещение логических операторов может дать непредвиденные результаты, как это показано в примере ниже:

```
USE SampleDb;  
  
SELECT Id, FirstName, LastName  
FROM Employee  
WHERE Id = 2581 AND LastName = 'Фролов'  
OR FirstName = 'Василий' AND DepartamentNumber = 'd1';
```

```
SELECT Id, FirstName, LastName  
FROM Employee  
WHERE ((Id = 2581 AND LastName = 'Фролов')  
OR FirstName = 'Василий') AND DepartamentNumber = 'd1';
```

Результат выполнения этих запросов:

Results		Messages	
	Id	FirstNa...	LastNa...
1	2581	Василий	Фролов

Id	FirstNa...	LastNa...
----	------------	-----------

Как можно видеть, эти два кажущиеся одинаковыми запроса SELECT выдают два разных результирующих набора данных. В первой инструкции SELECT система сначала вычисляет оба оператора AND (слева направо), а потом вычисляет оператор OR. Во второй же инструкции SELECT порядок выполнения операторов изменен вследствие использования скобок, операторы в скобках выполняются первыми, в порядке слева направо. Как можно видеть, первая инструкция возвратила одну строку, тогда как вторая не возвратила ни одной.

Наличие логических операторов в предложении WHERE усложняет содержащую его инструкцию SELECT и способствует появлению в ней ошибок. В таких случаях настоятельно рекомендуется применять скобки, даже если они не являются необходимыми. Применение скобок значительно улучшает читаемость инструкции SELECT и уменьшает возможность появления в ней ошибок. Далее приводится первый вариант инструкции SELECT из примера выше, модифицированной в соответствии с этой рекомендацией:

```
USE SampleDb;
```

```
SELECT Id, FirstName, LastName
FROM Employee
WHERE (Id = 2581 AND LastName = 'Фролов')
OR (FirstName = 'Василий' AND DepartmentNumber = 'd1');
```

Третий логический оператор NOT изменяет логическое значение, к которому он применяется, на противоположное. Это означает, что отрицание истинного значения (true) дает ложь (false) и наоборот. Отрицание значения NULL также дает NULL. Ниже демонстрируется использование оператора отрицания NOT:

```
USE SampleDb;
```

```
SELECT Id, FirstName
FROM Employee
WHERE NOT DepartmentNumber = 'd2';
```

В этом примере происходит выборка табельных номеров и имен сотрудников, не принадлежащих к отделу d2. В данном случае логический оператор NOT можно заменить логическим оператором сравнения <> (не равно).

Операторы IN и BETWEEN

Оператор IN позволяет указать одно или несколько выражений, по которым следует выполнять поиск в запросе. Результатом выражения будет истина (true), если значение соответствующего столбца равно одному из условий, указанных в предикате IN. В примере ниже демонстрируется использование оператора IN:

USE SampleDb;

```
SELECT Id, FirstName, LastName  
FROM Employee  
WHERE Id IN (10102, 25348, 28559);
```

В этом примере происходит выборка всех столбцов сотрудников, чей табельный номер равен 10102, 28559 или 25348. Результат выполнения этого запроса будет следующим:

	Id	FirstName	LastName
1	10102	Анна	Иванова
2	25348	Дмитрий	Волков
3	28559	Наталья	Вершинина

Оператор IN равнозначен последовательности условий, соединенных операторами OR. (Число операторов OR на один меньше, чем количество выражений в списке оператора IN.)

Оператор IN можно использовать совместно с логическим оператором NOT, как показано в примере ниже. В данном случае запрос выбирает все строки, не содержащие ни одного из указанных значений в соответствующих столбцах:

USE SampleDb;

```
SELECT Id, FirstName, LastName  
FROM Employee  
WHERE NOT Id IN (10102, 25348, 28559);
```

Результат выполнения этого запроса:

	Id	FirstName	LastName
1	2581	Василий	Фролов
2	9031	Елена	Лебедеенко
3	18316	Игорь	Соловьев
4	29346	Олег	Маменко

В отличие от оператора IN, для которого указываются отдельные значения, для оператора **BETWEEN** указывается диапазон значений, чьи границы определяются нижним и верхним значениями. Использование оператора BETWEEN показано в примере ниже:

USE SampleDb;

```
SELECT ProjectName, Budget
FROM Project
WHERE Budget BETWEEN 95000 AND 120000;
```

В этом примере происходит выборка наименований проектов и бюджетов всех проектов с бюджетом, находящимся в диапазоне от \$95 000 и до \$120 000 включительно.

Оператор BETWEEN возвращает все значения в указанном диапазоне, включая значения для границ; т.е. приемлемые значения могут быть между значениями указанных границ диапазона или быть равными значениям этих границ.

Оператор BETWEEN логически эквивалентен двум отдельным сравнениям, соединенным логическим оператором AND. Поэтому запрос, приведенный в примере выше, эквивалентен запросу:

```
USE SampleDb;
```

```
SELECT ProjectName, Budget
FROM Project
WHERE Budget >= 95000 AND Budget <= 120000;
```

Подобно оператору BETWEEN, оператор NOT BETWEEN можно использовать для выборки значений, находящихся за пределами указанного диапазона значений. Оператор BETWEEN также можно применять со значениями, которые имеют символьный или временной тип данных.

В примере ниже показаны две разные формы запроса SELECT, которые дают одинаковые результаты:

```
USE SampleDb;
```

```
SELECT ProjectName, Budget
FROM Project
WHERE Budget NOT BETWEEN 99000 AND 150000;
```

-- Аналог предыдущей конструкции

```
SELECT ProjectName, Budget
FROM Project
WHERE Budget < 99000 OR Budget > 150000;
```

В этом примере происходит выборка всех проектов с бюджетом меньшим, чем \$99 000 и большим, чем \$150 000. Формулировка требования запроса: "Выбрать наименования всех проектов с бюджетом меньшим, чем \$99 000 и имена всех проектов с бюджетом большим, чем \$150 000" может навести на мысль, что во втором запросе SELECT требуется применить логический оператор AND. Но логический смысл запроса требует применения оператора OR, т.к. использование оператора AND не даст никаких результатов вообще. Это потому, что не может бюджет быть одновременно и меньшим, чем \$99 000 и большим, чем \$150 000. Но это и является ответом, почему используется оператор OR, а не AND, поскольку

мы выбираем все проекты, бюджет которых меньше \$99 000 или больше \$150 000.

Запросы, связанные со значением NULL

Параметр NULL в инструкции CREATE TABLE указывает, что соответствующий столбец может содержать специальное значение NULL (которое обычно представляет неизвестное или неприменимое значение). Значения NULL отличаются от всех других значений базы данных. Предложение WHERE инструкции SELECT обычно возвращает строки, удовлетворяющие указанным в нем условиям сравнения. Но здесь возникает вопрос, как будут оцениваться в этих сравнениях значения NULL?

Все сравнения со значением NULL возвращают false, даже если им предшествует оператор NOT. Для выборки строк, содержащих значения NULL, в языке Transact-SQL применяется **оператор IS NULL**. Указание в предложении WHERE строк, содержащих (или не содержащих) значение NULL, имеет следующую общую форму:

column_name IS [NOT] NULL

Использование оператора IS NULL демонстрируется в примере ниже:

USE SampleDb;

```
SELECT EmpId, ProjectNumber
FROM Works_on
WHERE ProjectNumber = 'p2' AND job IS NULL;
```

В этом примере происходит выборка табельных номеров служащих и соответствующих номеров проектов для служащих, чья должность неизвестна и которые работают над проектом p2. В следующем примере демонстрируется синтаксически правильное, но логически неправильное использование сравнения с NULL. Причиной ошибки является то обстоятельство, что сравнение любого значения, включая NULL, с NULL возвращает false:

USE SampleDb;

```
SELECT EmpId, ProjectNumber
FROM Works_on
WHERE job <> NULL;
```

Выполнение этого запроса не возвращает никаких строк. Условие "column IS NOT NULL" эквивалентно условию "NOT (column IS NULL)". **Системная функция ISNULL** позволяет отображать указанное значение вместо значения NULL:

USE SampleDb;

```
SELECT EmpId, ProjectNumber, ISNULL(job, N'Работа не определена')
AS task
FROM Works_on
WHERE ProjectNumber = 'p1';
```

Результат выполнения этого запроса:

Results		Messages	
	Empld	ProjectNumber	task
1	10102	p1	Аналитик
2	9031	p1	Менеджер
3	28559	p1	Работа не опр...
4	29346	p1	Консультант

В примере выше для столбца должностей Job в результате запроса используется заголовок 'task'.

Оператор LIKE

Оператор LIKE используется для сопоставления с образцом, т.е. он сравнивает значения столбца с указанным шаблоном. Столбец может быть любого символьного типа данных или типа дата. Общая форма оператора LIKE выглядит таким образом:

column [NOT] LIKE 'pattern'

Параметр 'pattern' может быть строковой константой, или константой даты, или выражением (включая столбцы таблицы), и должен быть совместимым с типом данных соответствующего столбца. Для указанного столбца сравнение значения строки и шаблона возвращает true, если значение совпадает с выражением шаблона.

Определенные применяемые в шаблоне символы, называемые *подстановочными символами (wildcard characters)*, имеют специальное значение. Рассмотрим два из этих символов:

- % (знак процента) - обозначает последовательность любых символов любой длины;
- _ (символ подчеркивания) - обозначает любой один символ.

Использование подстановочных символов % и _ показано в примере ниже:

USE SampleDb;

```
SELECT Id, FirstName, LastName
FROM Employee
WHERE FirstName LIKE '_a%';
```

В этом примере происходит выборка имен, фамилий и табельных номеров сотрудников, у которых второй буквой имени является буква "а". Результат выполнения этого запроса:

Results		Messages	
	Id	FirstNa...	LastName
1	2581	Василий	Фролов
2	28559	Наталья	Вершинина

Кроме знака процентов и символа подчеркивания, поддерживает другие специальные символы, применяемые с оператором LIKE. Использование этих символов ([,] и ^) демонстрируется в примерах ниже:

USE SampleDb;


```

SELECT Id, FirstName, LastName
FROM Employee
WHERE FirstName LIKE '[B-И]>';

```

В этом примере происходит выборка данных сотрудников, чье имя начинается с символа в диапазоне от "B" до "И". Результат выполнения этого запроса:

Results		Messages	
	Id	FirstNa...	LastName
1	2581	Василий	Фролов
2	9031	Елена	Лебедеко
3	18316	Игорь	Соловьев
4	25348	Дмитрий	Волков

Как можно видеть по результатам примера, квадратные скобки [] ограничивают диапазон или список символов. Порядок отображения символов диапазона определяется порядком сортировки, указанным при установке системы.

Символ ^ обозначает отрицание диапазона или списка символов. Но такое значение этот символ имеет только тогда, когда находится внутри квадратных скобок, как показано в примере ниже. В данном примере запроса осуществляется выборка имен и фамилий тех сотрудников, чьи имена начинаются с буквы отличной от B-И, П-Я:

```

USE SampleDb;

```

```

SELECT Id, FirstName, LastName
FROM Employee
WHERE FirstName LIKE '[^B-И]%'
AND FirstName LIKE '[^П-Я]';

```

В примере ниже демонстрируется использование оператора LIKE совместно с отрицанием NOT:

```

USE SampleDb;

```

```

SELECT Id, FirstName, LastName
FROM Employee
WHERE FirstName NOT LIKE '[A-И]';

```

Здесь выбираются все столбцы сотрудников, чьи имена начинаются на буквы, отличные от букв в диапазоне 'A-И'.

Любой подстановочный символ (% , _ , [,] или ^), заключенный в квадратные скобки, остается обычным символом и представляет сам себя. Такая же возможность существует при использовании параметра **ESCAPE**. Поэтому оба варианта применения инструкции **SELECT**, показанные в примере ниже, эквивалентны:

```

USE SampleDb;

```

```

SELECT Id, FirstName, LastName

```

```
FROM Employee  
WHERE FirstName LIKE '%[_]%';
```

```
SELECT Id, FirstName, LastName  
FROM Employee  
WHERE FirstName LIKE '%!_%' ESCAPE '!';
```

Результат выполнения этих двух инструкций SELECT будет одинаковым. В этом примере обе инструкции SELECT рассматривают символ подчеркивания в значениях столбца FirstName, как таковой, а не как подстановочный. В первой инструкции SELECT это достигается заключением символа подчеркивания в квадратные скобки. А во второй инструкции SELECT этот же эффект достигается за счет применения *символа перехода (escape character)*, каковым в данном случае является символ восклицательного знака. Символ перехода переопределяет значение символа подчеркивания, делая его из подстановочного символа обычным. (Результат выполнения этих инструкций содержит ноль строк, потому что ни одно имя сотрудника не содержит символов подчеркивания.)

Стандарт SQL поддерживает только подстановочные символы %, _ и оператор ESCAPE. Поэтому если требуется представить подстановочный символ как обычный символ, то вместо квадратных скобок рекомендуется применять оператор ESCAPE.

Задание к лабораторной работе №3.

Для соответствующего варианта базы данных выполнить запросы из соответствующего блока.

Сортировка

Раздел ORDER BY – определяет поле для сортировки записей в запросе. Если указан параметр ASC, то будет производиться сортировка по возрастанию, если DESC – по убыванию. По умолчанию используется сортировка по возрастанию.

Изменение порядка следования полей

Выбор некоторых полей из двух таблиц

WHERE – предложение WHERE показывает, что в результаты запроса следует включать только некоторые строки. Для отбора строк, включаемых в результаты запроса, используется условие поиска, которое строится как логическое выражение, состоящее из одного или нескольких условий, объединенных логическими конструкциями типа AND или OR

Условие неточного совпадения

Точное несовпадение значений одного из полей

Выбор записей по диапазону значений (Between)

BETWEEN – проверка на принадлежность диапазону значений. При этом проверяется, находится ли значение поля между двумя определенными значениями. В MS SQL Server можно задать как строку в двойных кавычках в формате дд-мес-гггг (где месяц может быть оформлен как: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC).

Выбор записей по диапазону значений (In)

IN() – проверка на членство в множестве. Вывести только те строки, у которых значение указанного поля принадлежит указанному множеству, т.е. равно одному из значений, перечисленных в IN()

Выбор записей с использованием Like

(LIKE() – проверка на соответствие шаблону, где шаблон записывается в двойных кавычках. % или * – подстановочный знак в шаблоне, совпадающий с любой последовательностью из нуля и более символов. _ или ? – подстановочный знак в шаблоне, совпадающий с одним любым символом на указанном месте.

Выбор записей по нескольким условиям

Изменение наименований полей

Оператор обработки данных Update

Оператор обработки данных Insert

Оператор обработки данных Delete

Лабораторная работа №4. Освоение программирования с помощью встроенного языка Transact SQL

Цель работы – знакомство с основными принципами программирования в MS SQL Server средствами встроенного языка Transact SQL.

В предыдущих работах мы познакомились с инструкциями Transact-SQL языка описания данных DDL и языка манипуляции данными DML. Большинство этих инструкций можно сгруппировать в пакет. **Пакет (batch)** – это последовательность инструкций Transact-SQL и процедурных расширений, которые направляются системе базы данных для совместного их выполнения. Количество инструкций в пакете ограничивается допустимым размером скомпилированного пакетного объекта. Преимущество пакета над группой отдельных инструкций состоит в том, что одновременное исполнение всех инструкций позволяет получить значительное улучшение производительности.

Существует несколько ограничений на включение разных инструкций языка Transact-SQL в пакет. Наиболее важным из них является то обстоятельство, что если пакет содержит инструкцию описания данных CREATE VIEW, CREATE PROCEDURE или CREATE TRIGGER, то он не может содержать никаких других инструкций. Иными словами, такая инструкция должна быть единственной инструкцией пакета. Инструкции языка описания данных разделяются с помощью **инструкции GO**.

Каждое процедурное расширение языка Transact-SQL рассматривается по отдельности в следующих разделах.

Блок инструкций

Блок инструкций может состоять из одной или нескольких инструкций языка Transact-SQL. Каждый блок начинается с инструкции **BEGIN** и заканчивается инструкцией **END**, как это показано далее:

BEGIN

оператор 1

оператор 2

...

END

Блок можно разместить внутри инструкции IF, чтобы в зависимости от определенного условия разрешить исполнение одной или нескольких инструкций.

Инструкция IF

Инструкция IF языка Transact-SQL соответствует одноименной инструкции, поддерживаемой почти всеми языками программирования. Инструкция IF выполняет одну или несколько составляющих блок инструкций, если логическое выражение, следующее после ключевого слова IF, возвращает значение true (истина). Если же инструкция IF содержит **оператор ELSE**, то при условии, что логическое выражение возвращает значение false (ложь), выполняется вторая группа инструкций.

Ниже показан пример использования условной инструкции IF:
USE SampleDb;

```
IF (SELECT COUNT(*)
    FROM Works_on
    WHERE ProjectNumber = 'p1'
    GROUP BY ProjectNumber ) > 3
PRINT 'Число сотрудников работающих над проектом p1 больше 3'
ELSE BEGIN
    PRINT 'Следующие сотрудники работают над проектом p1:'
    SELECT FirstName, LastName
    FROM Employee, Works_on
    WHERE Employee.Id = Works_on.EmpId
    AND ProjectNumber = 'p1'
END
```

В этом примере демонстрируется использование блока инструкций внутри инструкции IF. Следующее далее логическое выражение инструкции IF:

```
(SELECT COUNT(*)
    FROM Works_on
    WHERE ProjectNumber = 'p1'
    GROUP BY ProjectNumber ) > 3
```

возвращает значение true (истина) для базы данных SampleDb. Поэтому будет выполняться инструкция PRINT, входящая в часть инструкции IF. Обратите внимание на то обстоятельство, что в этом примере используется подзапрос, чтобы вернуть число строк (посредством агрегатной функции COUNT), удовлетворяющих условию предложения WHERE (ProjectNumber='p1').

Оператор ELSE инструкции IF в примере содержит две инструкции: PRINT и SELECT. Поэтому для выполнения этих инструкций их необходимо заключить в блок между ключевыми словами BEGIN и END. (**Инструкция PRINT** является процедурным расширением и возвращает определяемое пользователем сообщение.)

Инструкция WHILE

Инструкция WHILE выполняет одну или несколько заключенных в блок инструкций, на протяжении времени, пока (WHILE) логическое выражение возвращает значение true (истина), т.е. позволяет создать цикл. Иными словами, если выражение возвращает true, выполняется инструкция или блок инструкций, после чего снова осуществляется проверка выражения. Этот процесс повторяется до тех пор, пока выражение не возвратит значение FALSE (ложь).

Блок внутри инструкции WHILE может содержать одну или две необязательных инструкций, применяемых для управления выполнением инструкций внутри блока: BREAK или CONTINUE. *Инструкция BREAK* останавливает выполнение цикла и начинает исполнение инструкций,

следующих сразу же после этого блока. А инструкция *CONTINUE* останавливает выполнение только текущей итерации цикла и начинает выполнять следующую итерацию.

В примере ниже показано использование инструкции WHILE:
USE SampleDb;

```
WHILE (SELECT SUM(Budget)
      FROM Project) < 500000
BEGIN
    UPDATE Project SET Budget = Budget * 1.1
    IF (SELECT MAX(budget)
        FROM project) > 240000
        BREAK
    ELSE CONTINUE
END
```

В этом примере бюджеты всех проектов увеличиваются на 10% до тех пор, пока общая сумма бюджетов не превысит \$500 000. Но выполнение блока прекратится, даже если общая сумма бюджетов будет меньше \$500 000, если только бюджет одного из проектов превысит \$240 000.

Локальные переменные

Локальные переменные являются важным процедурным расширением языка Transact-SQL. Они применяются для хранения значений любого типа в пакетах и подпрограммах. Локальными они называются по той причине, что они могут быть использованы только в том пакете, в котором они были объявлены. (Компонент Database Engine также поддерживает глобальные переменные, которые уже были рассмотрены ранее.)

Все локальные переменные пакета объявляются, используя инструкцию **DECLARE**. (Синтаксис этой инструкции приводится в примере ниже) Определение переменной состоит из имени переменной и ее типа данных. Имена локальных переменных в пакете всегда начинаются с префикса **@**. Присвоение значений локальной переменной осуществляется:

- используя специальную форму инструкции **SELECT**;
- используя инструкцию **SET**;
- непосредственно в инструкции **DECLARE** посредством знака **=** (например, **@extra_budget MONEY = 1500**).

Использование первых двух способов присвоения значения локальным переменным показано в примере ниже:

USE SampleDb;

-- Объявление и инициализация переменных

```
DECLARE @avg_budget MONEY,
        @extra_budget MONEY SET @extra_budget = 15000
SELECT @avg_budget = AVG(Budget) FROM Project;
```


```
IF (SELECT Budget
```

```

FROM Project
WHERE Number = 'p1') < @avg_budget
BEGIN
UPDATE Project
SET Budget = Budget + @extra_budget
WHERE Number = 'p1'
PRINT N'Бюджет проекта p1 увеличен на ' + convert(nvarchar(30),
@extra_budget)
END
ELSE PRINT N'Бюджет проекта p1 не изменился'

```

Результат выполнения:

 Messages

(1 row(s) affected)

Бюджет проекта p1 увеличен на 15000.00

Пакет инструкций в этом примере вычисляет среднее значение бюджетов всех проектов и сравнивает полученное значение с бюджетом проекта p1. Если бюджет проекта p1 меньше среднего значения всех бюджетов, его значение увеличивается на величину значения локальной переменной @extra_budget.

Смешанные процедурные инструкции

Процедурные расширения языка Transact-SQL также содержат следующие инструкции:

- RETURN;
- GOTO;
- RAISEERROR;
- WAITFOR

Инструкция RETURN выполняет ту же самую функцию внутри пакета, что и инструкция BREAK внутри цикла WHILE. Иными словами, инструкция RETURN останавливает выполнение пакета и начинает исполнение первой инструкции, следующей за пакетом.

Инструкция GOTO передает управление при выполнении пакета инструкции Transact-SQL внутри пакета, обозначенной маркером.

Инструкция RAISEERROR выводит определенное пользователем сообщение об ошибке и устанавливает флаг системной ошибки. Номер ошибки в определяемом пользователем сообщении должен быть больше, чем 50 000, т.к. все номера ошибок меньшие или равные 50 000 определены системой и зарезервированы компонентом Database Engine. Значения номеров ошибок сохраняются в *глобальной переменной @@error*.

Инструкция WAITFOR определяет задержку на период времени (с параметром DELAY) или определенное время (с параметром TIME), на протяжении которой система должна ожидать, прежде чем исполнять следующую инструкцию пакета. Синтаксис этой инструкции выглядит следующим образом:

WAITFOR

{DELAY 'time' | TIME 'time' | TIMEOUT 'timeout' }

Параметр DELAY указывает системе баз данных ожидать, пока не истечет указанный период времени, а *параметр TIME* указывает точку во времени, в одном из допустимых форматов, до которой ожидать. *Параметр TIMEOUT*, за которым следует аргумент *timeout*, задает период времени в миллисекундах, в течение которого надо ожидать прибытия сообщения в очередь.

Обработка исключений с помощью инструкций TRY, CATCH и THROW

В версиях SQL Server более ранних, чем SQL Server 2005 требовалось наличие кода для обработки ошибок после каждой инструкции Transact-SQL, которая могла бы вызвать ошибку. (Для обработки ошибок можно использовать глобальную переменную @@error.) Начиная с версии SQL Server 2005, исключения можно перехватывать для обработки с помощью **инструкций TRY и CATCH**. В этом разделе сначала объясняется значение понятия "исключение", после чего обсуждается работа этих двух инструкций.

Исключением (exception) называется проблема (обычно ошибка), которая не позволяет продолжать выполнение программы. Выполняющаяся программа не может продолжать исполнение по причине недостаточности информации, необходимой для обработки ошибки в данной части программы. Поэтому задача обработки ошибки передается другой части программы.

Роль инструкции TRY заключается в перехвате исключения. (Поскольку для реализации этого процесса обычно требуется несколько инструкций, то обычно применяется термин "блок TRY", а не "инструкция TRY".) Если же в блоке TRY возникает исключение, компонент системы, называемый *обработчиком исключений*, доставляет это исключение для обработки другой части программы. Эта часть программы обозначается ключевым словом CATCH и поэтому называется блоком CATCH.

Обработка исключений с использованием инструкций TRY и CATCH является общим методом обработки ошибок, применяемым в современных языках программирования, таких как C# и Java.

Обработка исключений с помощью блоков TRY и CATCH предоставляет программисту множество преимуществ, включая следующие:

- исключения предоставляют аккуратный способ определения ошибок без загромождения кода дополнительными инструкциями;
- исключения предоставляют механизм прямой индикации ошибок, вместо использования каких-либо побочных признаков;
- программист может видеть исключения и проверить их в процессе компиляции.

В SQL Server 2012 добавлена еще одна **инструкция THROW**, имеющая отношение к обработке ошибок. Эта инструкция позволяет вызвать исключение, которое улавливается в блоке обработки исключений. Иными

словами, инструкция THROW - это другой механизм возврата, который работает подобно рассмотренной ранее инструкции RAISEERROR.

Использование инструкций TRY, CATCH и THROW для обработки исключений показано в примере ниже:

```
USE SampleDb;
```

```
BEGIN TRY
```

```
    BEGIN TRANSACTION
```

```
        insert into Employee values(11111, 'Анна', 'Самойленко','d2');
```

```
        insert into Employee values(22222, 'Игорь', 'Васин','d4');
```

```
        -- Создаем ошибку ссылочной целостности
```

```
        insert into Employee values(33333, 'Петр', 'Сидоров', 'd2');
```

```
    COMMIT TRANSACTION
```

```
    PRINT 'Транзакция выполнена'
```

```
END TRY
```

```
BEGIN CATCH
```

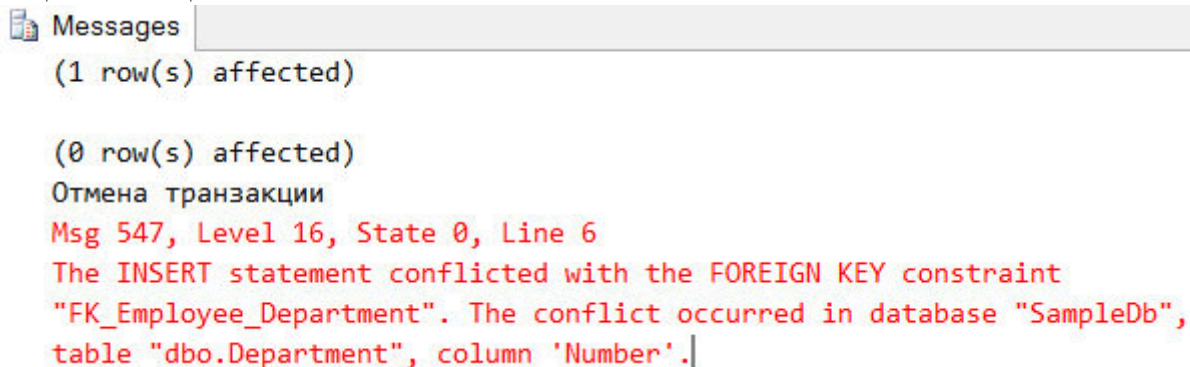
```
    ROLLBACK
```

```
    PRINT 'Отмена транзакции';
```

```
    THROW
```

```
END CATCH
```

В данном примере показано, как для обработки исключений оформлять инструкции в пакет и выполнять откат результатов исполнения всей группы инструкций при возникновении ошибки. Попытка выполнить пакет, показанный в примере, будет неудачной, а в результате будет выведено следующее сообщение:



The screenshot shows the 'Messages' window in SQL Server. It displays the following text:

```
(1 row(s) affected)

(0 row(s) affected)
Отмена транзакции
Msg 547, Level 16, State 0, Line 6
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_Employee_Department". The conflict occurred in database "SampleDb",
table "dbo.Department", column 'Number'.
```

Выполнение кода в примере осуществляется следующим образом. После успешного выполнения первой инструкции INSERT попытка исполнения второй инструкции вызывает ошибку нарушения ссылочной целостности. Так как все три инструкции заключены в блок TRY, возникает исключение для всего блока и обработчик исключений начинает исполнение блока CATCH. Выполнение кода в этом блоке осуществляет откат исполнения всех инструкций в блоке TRY и выводит соответствующее сообщение. После этого инструкция THROW возвращает управление исполнением вызывающему объекту. Вследствие всего этого содержимое таблицы Employee не будет изменено.

Инструкции Transact-SQL - BEGIN TRANSACTION, COMMIT TRANSACTION и ROLLBACK начинают, фиксируют и выполняют откат транзакций соответственно.

Задание для лабораторной работы №4

Использование функций для работы со строковыми переменными

Базовый текст дан в отдельном файле по вариантам. Для выполнения этого блока заданий в Query Analyzer объявите переменную типа varchar и присвойте ей в качестве значения строку с базовым текстом, который будет анализироваться и/или исправляться в заданиях.

1. Удалить в тексте лишние пробелы. Лишними считаются те, которые идут непосредственно за пробелом. Подсчитать количество исправлений.

2. Подсчитать количество встреч каждой из следующих букв: "а", "в", "и", "п" в базовом тексте.

3. Подсчитать доли процентов встречи следующих букв: "е", "о", если суммарный процент встречаемости всех этих букв равен 100% или процент встречаемости е% + о% равен 100%.

4. По правилам оформления машинописных текстов перед знаками ., !?; пробелы не ставятся, но обязательно ставятся после этих знаков. Удалите лишние пробелы. Подсчитать количество исправлений.

5. По правилам оформления машинописных текстов перед знаками ., !?; пробелы не ставятся, но обязательно ставятся после этих знаков. Расставьте недостающие пробелы. Подсчитать количество исправлений.

6. Найти из исходного текста второе предложение и вернуть его в переменную Регем, а также вывести на экран весь исходный текст и найденное предложение.

7. Удалить из базового текста 2, 4, 6, 8 слова.

8. Удалить из базового текста 3, 5, 7, 10 слова.

9. Вставить в базовый текст вместо букв «а» - «АА».

10. Вставить в базовый текст вместо букв «е» и «о» - «ББ».

11. Поменять местами первое и последнее слова в базовом тексте.

Использование функций для работы с числами

12. Вывести значение формулы (1), переменные которой нужно описать и присвоить произвольные значения.

$$v = v_0 \cdot e^{\sqrt{\frac{R \cdot T}{45}}} \quad (1)$$

13. Подсчитать значение формулы (2), переменные которой нужно описать и присвоить произвольные значения.

$$y = 2^x \cdot \exp(\ln(x^2)) \quad (2)$$

14. Подсчитать значение формулы (3), переменные которой нужно описать и присвоить произвольные значения.

$$y = \frac{\sin(a)}{x^2 - b^3} \cdot a. \quad (3)$$

15. Подсчитать значение формулы (4), переменные которой нужно описать и присвоить произвольные значения.

$$y = \sum_{n=1}^{10} I_n \cdot a \quad (4)$$

16. Подсчитать значение формулы (5), переменные которой нужно описать и присвоить произвольные значения.

$$y = \frac{\operatorname{tg}(a)}{a + b - c} \cdot \sqrt{a \cdot b \cdot c}. \quad (5)$$

17. Подсчитать значение формулы (6), переменные которой нужно описать и присвоить произвольные значения.

$$y = \sqrt{\sin(a) \cdot \exp(b \cdot c)} \quad (6)$$

18. Подсчитать значение формулы (7), переменные которой нужно описать и присвоить произвольные значения.

$$y = x^4 \cdot \ln(a) - b \cdot c. \quad (7)$$

19. Подсчитать значение формулы (8), переменные которой нужно описать и присвоить произвольные значения.

$$y = \frac{\sqrt{x - a}}{b^3} \quad (8)$$

20. Подсчитать значение формулы (9), переменные которой нужно описать и присвоить произвольные значения.

$$y = \frac{a \cdot \cos(x)}{b^2 - a^2} \cdot \sin(x). \quad (9)$$

Использование функций для работы с типом дата/время

21. Вывести на экран название текущего месяца и текущее время. Записать в таблицу Purchases в поле Date_order одинаковую дату поступления, которая равна 12.03.2014.

22. Разобрать на отдельные составляющие текущую дату и время и вывести значения на экран в следующем порядке (вместо многоточий):

23. "Сегодня: День = ..., Месяц = ..., Год = ..., Часов = ..., Минут = ..., Секунд = ..."

24. В исходный текст, сохраненный в переменной Perem, после слова " время " вставить текущее время. Результат сохранить в той же переменной Perem и вывести на экран.

Лабораторная работа №5 Оператор Select

Следующие подразделы описывают другие предложения оператора SELECT, которые могут быть использованы в запросах, а также агрегатные функции и наборы операторов. Напомню, к данному моменту мы рассмотрели использование предложения WHERE, а в этой работе мы рассмотрим предложения GROUP BY, ORDER BY и HAVING, и предоставим некоторые примеры использования этих предложений в сочетании с агрегатными функциями, которые поддерживаются в Transact-SQL.

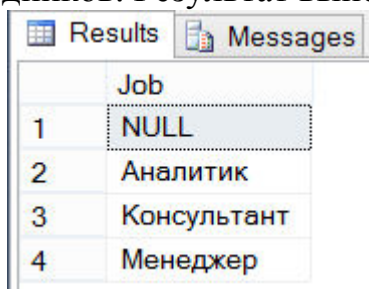
Предложение GROUP BY

Предложение **GROUP BY** группирует выбранный набор строк для получения набора сводных строк по значениям одного или нескольких столбцов или выражений. Простой случай применения предложения GROUP BY показан в примере ниже:

```
USE SampleDb;
```

```
SELECT Job  
FROM Works_On  
GROUP BY Job;
```

В этом примере происходит выборка и группирование должностей сотрудников. Результат выполнения этого запроса:



	Job
1	NULL
2	Аналитик
3	Консультант
4	Менеджер

В примере выше предложение GROUP BY создает отдельную группу для всех возможных значений (включая значение NULL) столбца Job.

Использование столбцов в предложении GROUP BY должно отвечать определенным условиям. В частности, каждый столбец в списке выборки запроса также должен присутствовать в предложении GROUP BY. Это требование не распространяется на константы и столбцы, являющиеся частью агрегатной функции. (Агрегатные функции рассматриваются в следующем подразделе.) Это имеет смысл, т.к. только для столбцов в предложении GROUP BY гарантируется одно значение для каждой группы.

Таблицу можно сгруппировать по любой комбинации ее столбцов. В примере ниже демонстрируется группирование строк таблицы Works_on по двум столбцам:

```
USE SampleDb;
```

```
SELECT ProjectNumber, Job  
FROM Works_On
```

GROUP BY ProjectNumber, Job;

Результат выполнения этого запроса:

	ProjectNum...	Job
1	p1	NULL
2	p1	Аналитик
3	p1	Консультант
4	p1	Менеджер
5	p2	NULL
6	p2	Консультант
7	p3	Аналитик
8	p3	Консультант
9	p3	Менеджер

По результатам выполнения запроса можно видеть, что существует девять групп с разными комбинациями номера проекта и должности. Последовательность имен столбцов в предложении **GROUP BY** не обязательно должна быть такой же, как и в списке столбцов выборки **SELECT**.

Агрегатные функции

Агрегатные функции используются для получения суммарных значений. Все агрегатные функции можно разделить на следующие категории:

- обычные агрегатные функции;
- статистические агрегатные функции;
- агрегатные функции, определяемые пользователем;
- аналитические агрегатные функции.

Здесь мы рассмотрим первые три типа агрегатных функций.

Обычные агрегатные функции

Язык Transact-SQL поддерживает следующие шесть агрегатных функций: **MIN**, **MAX**, **SUM**, **AVG**, **COUNT**, **COUNT_BIG**.

Все агрегатные функции выполняют вычисления над одним аргументом, который может быть или столбцом, или выражением. (Единственным исключением является вторая форма двух функций: **COUNT** и **COUNT_BIG**, а именно **COUNT(*)** и **COUNT_BIG(*)** соответственно.) Результатом вычислений любой агрегатной функции является константное значение, отображаемое в отдельном столбце результата.

Агрегатные функции указываются в списке столбцов инструкции **SELECT**, который также может содержать предложение **GROUP BY**. Если в инструкции **SELECT** отсутствует предложение **GROUP BY**, а список столбцов выборки содержит, по крайней мере, одну агрегатную функцию, тогда он не должен содержать простых столбцов (кроме как столбцов, служащих аргументами агрегатной функции). Поэтому код в примере ниже неправильный:

USE SampleDb;

```
SELECT LastName, MIN(Id)
FROM Employee;
```

Здесь столбец LastName таблицы Employee не должен быть в списке выборки столбцов, поскольку он не является аргументом агрегатной функции. С другой стороны, список выборки столбцов может содержать имена столбцов, которые не являются аргументами агрегатной функции, если эти столбцы служат аргументами предложения GROUP BY.

Аргументу агрегатной функции может предшествовать одно из двух возможных ключевых слов:

ALL

Указывает, что вычисления выполняются над всеми значениями столбца. Это значение по умолчанию.

DISTINCT

Указывает, что для вычислений применяются только уникальные значения столбца.

Агрегатные функции MIN и MAX

Агрегатные функции MIN и MAX вычисляют наименьшее и наибольшее значение столбца соответственно. Если запрос содержит предложение WHERE, функции MIN и MAX возвращают наименьшее и наибольшее значение строк, отвечающих указанным условиям. В примере ниже показано использование агрегатной функции MIN:

```
USE SampleDb;
```

-- Вернет 2581

```
SELECT MIN(Id) AS 'Минимальное значение Id'
FROM Employee;
```

Возвращенный в примере выше результат не очень информативный. Например, неизвестна фамилия сотрудника, которому принадлежит этот номер. Но получить эту фамилию обычным способом невозможно, потому что, как упоминалось ранее, явно указать столбец LastName не разрешается. Для того чтобы вместе с наименьшим табельным номером сотрудника также получить и фамилию этого сотрудника, используется подзапрос. В примере ниже показано использование такого подзапроса, где вложенный запрос содержит инструкцию SELECT из предыдущего примера:

```
USE SampleDb;
```

```
SELECT Id, LastName
FROM Employee
WHERE Id = (SELECT MIN(Id)
           FROM Employee);
```

Результат выполнения запроса:

Results		Messages
	Id	LastNa...
1	2581	Фролов

Использование агрегатной функции MAX показано в примере ниже:
USE SampleDb;

```
-- 29346
SELECT Id, LastName
FROM Employee
WHERE Id = (SELECT MAX(Id)
FROM Employee);
```

В качестве аргумента функции MIN и MAX также могут принимать строки и даты. В случае строкового аргумента значения сравниваются, используя фактический порядок сортировки. Для всех аргументов временных данных типа "дата" наименьшим значением столбца будет наиболее ранняя дата, а наибольшим - наиболее поздняя.

С функциями MIN и MAX можно применять ключевое слово DISTINCT. Перед применением агрегатных функций MIN и MAX из столбцов их аргументов исключаются все значения NULL.

Агрегатная функция SUM

Агрегатная функция SUM вычисляет общую сумму значений столбца. Аргумент этой агрегатной функции всегда должен иметь числовой тип данных. Использование агрегатной функции SUM показано в примере ниже:

USE SampleDb;

```
SELECT SUM (Budget) 'Суммарный бюджет'
FROM Project;
```

В этом примере происходит вычисление общей суммы бюджетов всех проектов. Результат выполнения запроса:

Results		Messages
	Суммарный бюджет	
1	401500	

В этом примере агрегатная функция группирует все значения бюджетов проектов и определяет их общую сумму. По этой причине запрос содержит неявную функцию группирования (как и все аналогичные запросы). Неявную функцию группирования из примера выше можно указать явно, как это показано в примере ниже:

USE SampleDb;

```
SELECT SUM (Budget) 'Суммарный бюджет'
FROM Project
GROUP BY();
```

Рекомендуется использовать этот синтаксис в предложении GROUP BY, поскольку таким образом группирование определяется явно.

Использование параметра DISTINCT устраняет все повторяющиеся значения в столбце перед применением функции SUM. Аналогично удаляются все значения NULL перед применением этой агрегатной функции.

Агрегатная функция AVG

Агрегатная функция **AVG** возвращает среднее арифметическое значение для всех значений столбца. Аргумент этой агрегатной функции всегда должен иметь числовой тип данных. Перед применением функции **AVG** все значения NULL удаляются из ее аргумента.

Использование агрегатной функции **AVG** показано в примере ниже:
USE SampleDb;

```
-- Вернем 133833
```

```
SELECT AVG (Budget) 'Средний бюджет на проект'  
FROM Project;
```

Здесь происходит вычисление среднего арифметического значения бюджета для всех бюджетов.

Агрегатные функции COUNT и COUNT_BIG

Агрегатная функция **COUNT** имеет две разные формы:

COUNT([DISTINCT] col_name)

COUNT(*)

Первая форма функции подсчитывает количество значений в столбце col_name. Если в запросе используется ключевое слово **DISTINCT**, перед применением функции **COUNT** удаляются все повторяющиеся значения столбца. При подсчете количества значений столбца эта форма функции **COUNT** не принимает во внимание значения NULL.

Использование первой формы агрегатной функции **COUNT** показано в примере ниже:

USE SampleDb;

```
SELECT ProjectNumber, COUNT(DISTINCT Job) 'Работ в проекте'  
FROM Works_on  
GROUP BY ProjectNumber;
```

Здесь происходит подсчет количества разных должностей для каждого проекта. Результат выполнения этого запроса:

	ProjectNumber	Работ в проекте
1	p1	3
2	p2	1
3	p3	3

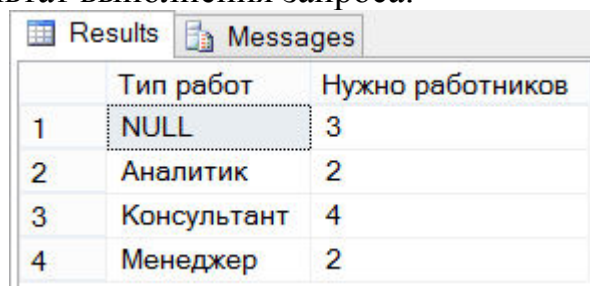
Как можно видеть в результате выполнения запроса, представленного в примере, значения NULL функцией **COUNT** не принимались во внимание. (Сумма всех значений столбца должностей получилась равной 7, а не 11, как должно быть.)

Вторая форма функции COUNT, т.е. функция COUNT(*) подсчитывает количество строк в таблице. А если инструкция SELECT запроса с функцией COUNT(*) содержит предложение WHERE с условием, функция возвращает количество строк, удовлетворяющих указанному условию. В отличие от первого варианта функции COUNT вторая форма не игнорирует значения NULL, поскольку эта функция оперирует строками, а не столбцами. В примере ниже демонстрируется использование функции COUNT(*):

```
USE SampleDb;
```

```
SELECT Job AS 'Тип работ', COUNT(*) 'Нужно работников'  
FROM Works_on  
GROUP BY Job;
```

Здесь происходит подсчет количества должностей во всех проектах. Результат выполнения запроса:



	Тип работ	Нужно работников
1	NULL	3
2	Аналитик	2
3	Консультант	4
4	Менеджер	2

Функция COUNT_BIG аналогична функции COUNT. Единственное различие между ними заключается в типе возвращаемого ими результата: функция COUNT_BIG всегда возвращает значения типа BIGINT, тогда как функция COUNT возвращает значения данных типа INTEGER.

Статистические агрегатные функции

Следующие функции составляют группу статистических агрегатных функций:

VAR

Вычисляет статистическую дисперсию всех значений, представленных в столбце или выражении.

VARP

Вычисляет статистическую дисперсию совокупности всех значений, представленных в столбце или выражении.

STDEV

Вычисляет среднее квадратическое отклонение (которое рассчитывается как квадратный корень из соответствующей дисперсии) всех значений столбца или выражения.

STDEVP

Вычисляет среднее квадратическое отклонение совокупности всех значений столбца или выражения.

Агрегатные функции, определяемые пользователем

Компонент Database Engine также поддерживает реализацию функций, определяемых пользователем. Эта возможность позволяет пользователям дополнить системные агрегатные функции функциями, которые они могут

реализовывать и устанавливать самостоятельно. Эти функции представляют специальный класс определяемых пользователем функций и подробно рассматриваются позже.

Предложение HAVING

В предложении **HAVING** определяется условие, которое применяется к группе строк. Таким образом, это предложение имеет такой же смысл для групп строк, что и предложение **WHERE** для содержимого соответствующей таблицы. Синтаксис предложения **HAVING** следующий:

HAVING condition

Здесь параметр **condition** представляет условие и содержит агрегатные функции или константы.

Использование предложения **HAVING** совместно с агрегатной функцией **COUNT(*)** показано в примере ниже:

USE SampleDb;

-- Вернет 'р3'

```
SELECT ProjectNumber  
  FROM Works_on  
  GROUP BY ProjectNumber  
    HAVING COUNT(*) < 4;
```

В этом примере посредством предложения **GROUP BY** система группирует все строки по значениям столбца **ProjectNumber**. После этого подсчитывается количество строк в каждой группе и выбираются группы, содержащие менее четырех строк (три или меньше).

Предложение **HAVING** можно также использовать без агрегатных функций, как это показано в примере ниже:

USE SampleDb;

-- Вернет 'Консультант'

```
SELECT Job  
  FROM Works_on  
  GROUP BY Job  
    HAVING Job LIKE 'K%';
```

В этом примере происходит группирование строк таблицы **Works_on** по должности и устранение тех должностей, которые не начинаются с буквы "К".

Предложение **HAVING** можно также использовать без предложения **GROUP BY**, хотя это не является распространенной практикой. В таком случае все строки таблицы возвращаются в одной группе.

Предложение ORDER BY

Предложение **ORDER BY** определяет порядок сортировки строк результирующего набора, возвращаемого запросом. Это предложение имеет следующий синтаксис:

ORDER BY {[col_name | col_number [ASC | DESC]]} , ...

Порядок сортировки задается в параметре `col_name`. Параметр `col_number` является альтернативным указателем порядка сортировки, который определяет столбцы по порядку их вхождения в список выборки инструкции `SELECT` (1 - первый столбец, 2 - второй столбец и т.д.). Параметр `ASC` определяет сортировку в восходящем порядке, а параметр `DESC` - в нисходящем. По умолчанию применяется параметр `ASC`.

Имена столбцов в предложении `ORDER BY` не обязательно должны быть указаны в списке столбцов выборки. Но это не относится к запросам типа `SELECT DISTINCT`, т.к. в таких запросах имена столбцов, указанные в предложении `ORDER BY`, также должны быть указаны в списке столбцов выборки. Кроме этого, это предложение не может содержать имен столбцов из таблиц, не указанных в предложении `FROM`.

Как можно видеть по синтаксису предложения `ORDER BY`, сортировка результирующего набора может выполняться по нескольким столбцам. Такая сортировка показана в примере ниже:

```
USE SampleDb;
```

```
SELECT *  
FROM Employee  
WHERE Id < 20000  
ORDER BY LastName, FirstName;
```

В этом примере происходит выборка номеров отделов и фамилий и имен сотрудников для сотрудников, чей табельный номер меньше чем 20 000, а также с сортировкой по фамилии и имени. Результат выполнения этого запроса:

	Id	FirstNa...	LastName	DepartamentNum...
1	10102	Анна	Иванова	d3
2	9031	Елена	Лебеденко	d2
3	18316	Игорь	Соловьев	d1
4	2581	Василий	Фролов	d2

Столбцы в предложении `ORDER BY` можно указывать не по их именам, а по порядку в списке выборки. Соответственно, предложение в примере выше можно переписать таким образом:

```
USE SampleDb;
```

```
SELECT *  
FROM Employee  
WHERE Id < 20000  
ORDER BY 3, 2;
```

Такой альтернативный способ указания столбцов по их позиции вместо имен применяется, если критерий упорядочивания содержит агрегатную функцию. (Другим способом является использование наименований

столбцов, которые тогда отображаются в предложении ORDER BY.) Однако в предложении ORDER BY рекомендуется указывать столбцы по их именам, а не по номерам, чтобы упростить обновление запроса, если в списке выборки придется добавить или удалить столбцы. Указание столбцов в предложении ORDER BY по их номерам показано в примере ниже:

```
USE SampleDb;
```

```
SELECT ProjectNumber, COUNT(*) 'Количество сотрудников'  
FROM Works_on  
GROUP BY ProjectNumber  
ORDER BY 2 DESC;
```

Здесь для каждого проекта выбирается номер проекта и количество участвующих в нем сотрудников, упорядочив результат в убывающем порядке по числу сотрудников.

Язык Transact-SQL при сортировке в возрастающем порядке помещает значения NULL в начале списка, и в конце списка - при убывающем.

Использование предложения ORDER BY для разбиения результатов на страницы

Отображение результатов запроса на текущей странице можно или реализовать в пользовательском приложении, или же дать указание осуществить это серверу базы данных. В первом случае все строки базы данных отправляются приложению, чьей задачей является отобрать требуемые строки и отобразить их. Во втором случае, со стороны сервера выбираются и отображаются только строки, требуемые для текущей страницы. Как можно предположить, создание страниц на стороне сервера обычно обеспечивает лучшую производительность, т.к. клиенту отправляются только строки, необходимые для отображения.

Для поддержки создания страниц на стороне сервера в SQL Server 2012 вводится два новых предложения инструкции SELECT: OFFSET и FETCH. Применение этих двух предложений демонстрируется в примере ниже. Здесь из базы данных AdventureWorks2012 извлекается идентификатор бизнеса, название должности и день рождения всех сотрудников женского пола с сортировкой результата по названию должности в возрастающем порядке. Результирующий набор строк разбивается на 10-строчные страницы и отображается третья страница:

```
USE AdventureWorks2012;
```

```
SELECT BusinessEntityID, JobTitle, BirthDate  
FROM HumanResources.Employee  
WHERE Gender = 'F'  
ORDER BY JobTitle  
OFFSET 20 ROWS  
FETCH NEXT 10 ROWS ONLY;
```

В предложении **OFFSET** указывается количество строк результата, которые нужно пропустить в отображаемом результате. Это количество

вычисляется после сортировки строк предложением **ORDER BY**. В предложении **FETCH NEXT** указывается количество удовлетворяющих условию **WHERE** и отсортированных строк, которое нужно возвратить. Параметром этого предложения может быть константа, выражение или результат другого запроса. Предложение **FETCH NEXT** аналогично предложению **FETCH FIRST**.

Основной целью при создании страниц на стороне сервера является возможность реализация общих страничных форм, используя переменные. Эту задачу можно выполнить посредством пакета SQL Server.

Операторы работы с наборами

Кроме операторов, рассмотренных ранее, язык Transact-SQL поддерживает еще три оператора работы с наборами: **UNION**, **INTERSECT** и **EXCEPT**.

Оператор UNION

Оператор UNION объединяет результаты двух или более запросов в один результирующий набор, в который входят все строки, принадлежащие всем запросам в объединении. Соответственно, результатом объединения двух таблиц является новая таблица, содержащая все строки, входящие в одну из исходных таблиц или в обе эти таблицы.

Общая форма оператора **UNION** выглядит таким образом:

```
select_1 UNION [ALL] select_2 {[UNION [ALL] select_3]}...
```

Параметры **select_1**, **select_2**, ... представляют собой инструкции **SELECT**, которые создают объединение. Если используется параметр **ALL**, отображаются все строки, включая дубликаты. В операторе **UNION** параметр **ALL** имеет то же самое значение, что и в списке выбора **SELECT**, но с одним отличием: для списка выбора **SELECT** этот параметр применяется по умолчанию, а для оператора **UNION** его нужно указывать явно.

В своей исходной форме база данных SampleDb не подходит для демонстрации применения оператора **UNION**. Поэтому в этом разделе создается новая таблица EmployeeEnh, которая идентична существующей таблице Employee, но имеет дополнительный столбец City. В этом столбце указывается место жительства сотрудников.

Создание таблицы EmployeeEnh предоставляет нам удобный случай продемонстрировать использование предложения **INTO** в инструкции **SELECT**. Инструкция **SELECT INTO** выполняет две операции. Сначала создается новая таблица со столбцами, перечисленными в списке выбора **SELECT**. Потом строки исходной таблицы вставляются в новую таблицу. Имя новой таблицы указывается в предложении **INTO**, а имя таблицы-источника указывается в предложении **FROM**.

В примере ниже показано создание таблицы EmployeeEnh из таблицы Employee:

```
USE SampleDb;
```

```
SELECT *  
  INTO EmployeeEnh
```

FROM Employee;

ALTER TABLE EmployeeEnh
ADD City NCHAR(40) NULL;

В этом примере инструкция **SELECT INTO** создает таблицу EmployeeEnh, вставляет в нее все строки из таблицы-источника Employee, после чего инструкция **ALTER TABLE** добавляет в новую таблицу столбец City. Но добавленный столбец City не содержит никаких значений. Значения в этот столбец можно вставить посредством среды Management Studio или же с помощью следующего кода:

USE SampleDb;

UPDATE EmployeeEnh **SET** City = 'Казань' **WHERE** Id = 2581;

UPDATE EmployeeEnh **SET** City = 'Москва' **WHERE** Id = 9031;

UPDATE EmployeeEnh **SET** City = 'Екатеринбург' **WHERE** Id = 10102;

UPDATE EmployeeEnh **SET** City = 'Санкт-Петербург' **WHERE** Id = 18316;

UPDATE EmployeeEnh **SET** City = 'Краснодар' **WHERE** Id = 25348;

UPDATE EmployeeEnh **SET** City = 'Казань' **WHERE** Id = 28559;

UPDATE EmployeeEnh **SET** City = 'Пермь' **WHERE** Id = 29346;

Теперь мы готовы продемонстрировать использование инструкции **UNION**. В примере ниже показан запрос для создания соединения таблиц EmployeeEnh и Department, используя эту инструкцию:

USE SampleDb;

SELECT City **AS** 'Город' **FROM** EmployeeEnh

UNION

SELECT Location

FROM Department;

Результат выполнения этого запроса:

Results		Messages
	Город	
1	Екатеринбург	
2	Казань	
3	Краснодар	
4	Москва	
5	Пермь	
6	Санкт-Петербург	

Объединять с помощью инструкции **UNION** можно только совместимые таблицы. Под совместимыми таблицами имеется в виду, что оба списка столбцов выборки должны содержать одинаковое число столбцов, а соответствующие столбцы должны иметь совместимые типы данных. (В отношении совместимости типы данных **INT** и **SMALLINT** не являются совместимыми.)

Результат объединения можно упорядочить, только используя предложение ORDER BY в последней инструкции SELECT, как это показано в примере ниже. Предложения GROUP BY и HAVING можно применять с отдельными инструкциями SELECT, но не в самом объединении.

```
USE SampleDb;
```

```
-- Вернет 18316, 28559
```

```
SELECT Id
FROM Employee
WHERE DepartamentNumber = 'd1'
UNION
SELECT EmpId
FROM Works_on
WHERE EnterDate > '01.01.2008'
ORDER BY 1;
```

Запрос в этом примере осуществляет выборку сотрудников, которые или работают в отделе d1, или начали работать над проектом до 1 января 2008 г.

Оператор UNION поддерживает параметр ALL. При использовании этого параметра дубликаты не удаляются из результирующего набора. Вместо оператора UNION можно применить оператор OR, если все инструкции SELECT, соединенные одним или несколькими операторами UNION, ссылаются на одну и ту же таблицу. В таком случае набор инструкций SELECT заменяется одной инструкцией SELECT с набором операторов OR.

Операторы INTERSECT и EXCEPT

Два других оператора для работы с наборами, **INTERSECT** и **EXCEPT**, определяют пересечение и разность соответственно. Под пересечением в данном контексте имеется набор строк, которые принадлежат к обеим таблицам. А разность двух таблиц определяется как все значения, которые принадлежат к первой таблице и не присутствуют во второй. В примере ниже показано использование оператора INTERSECT:

```
USE SampleDb;
```

```
-- Вернет только 28559
```

```
SELECT Id
FROM Employee
WHERE DepartamentNumber = 'd1'
INTERSECT
SELECT EmpId
FROM Works_on
WHERE EnterDate > '01.01.2008';
```

Язык Transact-SQL не поддерживает использование параметра ALL ни с оператором INTERSECT, ни с оператором EXCEPT. Использование оператора EXCEPT показано в примере ниже:

USE SampleDb;

-- Вернем 10102, 25348

```
SELECT Id
FROM Employee
WHERE DepartamentNumber = 'd3'
EXCEPT
SELECT EmpId
FROM Works_on
WHERE EnterDate > '01.01.2008';
```

Следует помнить, что эти три оператора над множествами имеют разный приоритет выполнения: оператор INTERSECT имеет наивысший приоритет, за ним следует оператор EXCEPT, а оператор UNION имеет самый низкий приоритет. Невнимательность к приоритету выполнения при использовании нескольких разных операторов для работы с наборами может повлечь неожиданные результаты.

Выражения CASE

В области прикладного программирования баз данных иногда требуется модифицировать представление данных. Например, людей можно подразделить, закодировав их по их социальной принадлежности, используя значения 1, 2 и 3, обозначив так мужчин, женщин и детей соответственно. Такой прием программирования может уменьшить время, необходимое для реализации программы. **Выражение CASE** языка Transact-SQL позволяет с легкостью реализовать такой тип кодировки.

В отличие от большинства языков программирования, CASE не является инструкцией, а выражением. Поэтому выражение CASE можно использовать почти везде, где язык Transact-SQL позволяет применять выражения. Выражение CASE имеет две формы:

- простое выражение CASE;
- поисковое выражение CASE.

Синтаксис простого выражения CASE следующий:

```
CASE expression_1
{WHEN expression_2 THEN result_1} ...
[ELSE result_n]
END
```

Инструкция с простым выражением CASE сначала ищет в списке всех выражений в предложении *WHEN* первое выражение, совпадающее с выражением *expression_1*, после чего выполняет соответствующее предложение *THEN*. В случае отсутствия в списке *WHEN* совпадающего выражения, выполняется предложение *ELSE*.

Синтаксис поискового выражения CASE следующий:

```
CASE
```



```

    {WHEN condition_1 THEN result_1} ...
    [ELSE result_n]
END

```

В данном случае выполняется поиск первого отвечающего требованиям условия, после чего выполняется соответствующее предложение THEN. Если ни одно из условий не отвечает требованиям, выполняется предложение ELSE. Применение поискового выражения CASE показано в примере ниже:

```
USE SampleDb;
```

```

SELECT ProjectName,
CASE
    WHEN Budget > 0 AND Budget <= 100000 THEN 1
    WHEN Budget > 100000 AND Budget <= 150000 THEN 2
    WHEN Budget > 150000 AND Budget <= 200000 THEN 3
    ELSE 4
END 'Категория бюджета'
FROM Project;

```

Результат выполнения этого запроса:

	ProjectName	Категория бюджета
1	Apollo	2
2	Gemini	1
3	Mercury	3

В этом примере взвешиваются бюджеты всех проектов, после чего отображаются вычисленные их весовые коэффициенты вместе с соответствующими наименованиями проектов.

В примере ниже показан другой способ применения выражения CASE, где предложение WHEN содержит вложенные запросы, составляющие часть выражения:

```
USE SampleDb;
```

```

SELECT ProjectName,
CASE
    WHEN p1.Budget < (SELECT AVG(p2.Budget) FROM Project p2)
    THEN 'ниже среднего'
    WHEN p1.Budget <=(SELECT AVG(p2.Budget) FROM Project p2)
    THEN 'равен среднему'
    WHEN p1.Budget > (SELECT AVG(p2.Budget) FROM Project p2)
    THEN 'выше среднего'
END 'Категория бюджета'
FROM Project p1;

```

Результат выполнения этого запроса следующий:

Results		Messages
	ProjectNa...	Категория бюдж...
1	Apollo	ниже среднего
2	Gemini	ниже среднего
3	Mercury	выше среднего

Подзапросы

Во всех рассмотренных ранее примерах значения столбцов сравниваются с выражением, константой или набором констант. Кроме таких возможностей сравнения язык Transact-SQL позволяет сравнивать значения столбца с результатом другой инструкции SELECT. Такая конструкция, где предложение WHERE инструкции SELECT содержит одну или больше вложенных инструкций SELECT, называется **подзапросом (subquery)**. Первая инструкция SELECT подзапроса называется *внешним запросом (outer query)*, а внутренняя инструкция (или инструкции) SELECT, используемая в сравнении, называется *вложенным запросом (inner query)*. Первым выполняется вложенный запрос, а его результат передается внешнему запросу. Вложенные запросы также могут содержать инструкции INSERT, UPDATE и DELETE.

Существует два типа подзапросов: независимые и связанные. В независимых подзапросах вложенный запрос логически выполняется ровно один раз. Связанный запрос отличается от независимого тем, что его значение зависит от переменной, получаемой от внешнего запроса. Таким образом, вложенный запрос связанного подзапроса выполняется каждый раз, когда система получает новую строку от внешнего запроса. В этом разделе приводятся несколько примеров независимых подзапросов.

Независимый подзапрос может применяться со следующими операторами:

- операторами сравнения;
- оператором IN;
- операторами ANY и ALL.

Подзапросы и операторы сравнения

Использование оператора равенства (=) в независимом подзапросе показано в примере ниже:

```
USE SampleDb;
```

```
SELECT FirstName, LastName
FROM Employee
WHERE DepartmentNumber = (SELECT Number
                           FROM Department
                           WHERE DepartmentName = 'Исследования');
```

В этом примере происходит выборка имен и фамилий сотрудников отдела 'Исследования'. Результат выполнения этого запроса:

Results		Messages
	FirstName	LastName
1	Игорь	Соловьев
2	Наталья	Вершинина

В примере выше сначала выполняется вложенный запрос, возвращая номер отдела разработки (d1). После выполнения внутреннего запроса подзапрос в примере можно представить следующим эквивалентным запросом:

```
USE SampleDb;
```

```
SELECT FirstName, LastName
FROM Employee
WHERE DepartmentNumber = 'd1';
```

В подзапросах можно также использовать любые другие операторы сравнения, при условии, что вложенный запрос возвращает в результате одну строку. Это очевидно, поскольку невозможно сравнить конкретные значения столбца, возвращаемые внешним запросом, с набором значений, возвращаемым вложенным запросом. В последующем разделе рассматривается, как можно решить проблему, когда результат вложенного запроса содержит набор значений.

Подзапросы и оператор IN

Оператор IN позволяет определить набор выражений (или констант), которые затем можно использовать в поисковом запросе. Этот оператор можно использовать в подзапросах при таких же обстоятельствах, т.е. когда вложенный запрос возвращает набор значений. Использование оператора IN в подзапросе показано в примере ниже:

```
USE SampleDb;
```

```
SELECT FirstName, LastName
FROM Employee
WHERE DepartmentNumber IN (SELECT Number
FROM Department
WHERE DepartmentName = 'Исследования')
```

Этот запрос аналогичен предыдущему. Каждый вложенный запрос может содержать свои вложенные запросы. Подзапросы такого типа называются *подзапросами с многоуровневым вложением*. Максимальная глубина вложения (т.е. количество вложенных запросов) зависит от объема памяти, которым компонент Database Engine располагает для каждой инструкции SELECT. В случае подзапросов с многоуровневым вложением система сначала выполняет самый глубокий вложенный запрос и возвращает полученный результат запросу следующего высшего уровня, который в свою очередь возвращает свой результат запросу следующего уровня над ним и т.д. Конечный результат выдается запросом самого высшего уровня.

Запрос с несколькими уровнями вложенности показан в примере ниже:

```
USE SampleDb;
```

```
SELECT FirstName, LastName
FROM Employee
WHERE ID IN
    (SELECT EmpId
     FROM Works_on
     WHERE ProjectNumber IN
        (SELECT Number
         FROM Project
         WHERE ProjectName = 'Apollo')
    )
```

В этом примере происходит выборка фамилий всех сотрудников, работающих над проектом Apollo. Самый глубокий вложенный запрос выбирает из таблицы ProjectNumber значение p1. Этот результат передается следующему вышестоящему запросу, который обрабатывает столбец ProjectNumber в таблице Works_on. Результатом этого запроса является набор табельных номеров сотрудников: (10102, 29346, 9031, 28559). Наконец, самый внешний запрос выводит фамилии сотрудников, чьи номера были выбраны предыдущим запросом.

Подзапросы и операторы ANY и ALL

Операторы ANY и ALL всегда используются в комбинации с одним из операторов сравнения. Оба оператора имеют одинаковый синтаксис:

```
column_name operator [ANY | ALL] query
```

Параметр operator обозначает оператор сравнения, а параметр query - вложенный запрос. Оператор ANY возвращает значение true (истина), если результат соответствующего вложенного запроса содержит хотя бы одну строку, удовлетворяющую условию сравнения. *Ключевое слово SOME* является синонимом ANY. Использование оператора ANY показано в примере ниже:

```
USE SampleDb;
```

```
SELECT DISTINCT EmpId, ProjectNumber, Job
FROM Works_on
WHERE EnterDate > ANY
    (SELECT EnterDate
     FROM Works_on);
```

В этом примере происходит выборка табельного номера, номера проекта и названия должности для сотрудников, которые не затратили большую часть своего времени при работе над одним из проектов. Каждое значение столбца EnterDate сравнивается со всеми другими значениями этого же столбца. Для всех дат этого столбца, за исключением самой ранней, сравнение возвращает значение true (истина), по крайней мере, один раз. Строка с самой ранней датой не попадает в результирующий набор,

поскольку сравнение ее даты со всеми другими датами никогда не возвращает значение true (истина). Иными словами, выражение "EnterDate > ANY (SELECT EnterDate FROM Works_on)" возвращает значение true, если в таблице Works_on имеется любое количество строк (одна или больше), для которых значение столбца EnterDate меньше, чем значение EnterDate текущей строки. Этому условию удовлетворяют все значения столбца EnterDate, за исключением наиболее раннего.

Оператор ALL возвращает значение true, если вложенный запрос возвращает все значения, обрабатываемого им столбца.

Настоятельно рекомендуется избегать использования операторов ANY и ALL. Любой запрос с применением этих операторов можно сформулировать лучшим образом посредством функции EXISTS. Кроме этого, семантическое значение оператора ANY можно легко принять за семантическое значение оператора ALL и наоборот.

Связанные подзапросы

Подзапрос называется *связанным (correlated)*, если любые значения вложенного запроса зависят от внешнего запроса. В примере ниже показано использование связанного подзапроса:

USE SampleDb;

```
SELECT LastName  
FROM Employee  
WHERE 'p3' IN  
    (SELECT ProjectNumber  
     FROM Works_on  
     WHERE Works_on.EmpId = Employee.Id);
```

В этом примере вложенный запрос должен логически выполняться несколько раз, поскольку он содержит столбец Id, который принадлежит таблице Employee во внешнем запросе, и значение столбца Id изменяется каждый раз, когда проверяется другая строка таблицы Employee во внешнем запросе.

Давайте проследим, как система может выполнять запрос в этом примере. Сначала система выбирает первую строку таблицы Employee (для внешнего запроса) и сравнивает табельный номер сотрудника в этом столбце (25348) со значениями столбца Works_on.EmpId вложенного запроса. Поскольку для этого сотрудника имеется только одно значение ProjectNumber равное p2, вложенный запрос возвращает значение p2. Это единственное значение результирующего набора вложенного запроса не равно значению p3 внешнего запроса, условие внешнего запроса (WHERE 'p3' IN...) не удовлетворяется и, следовательно, внешний запрос не возвращает никаких строк для этого сотрудника.

Далее система берет следующую строку таблицы Employee и снова сравнивает номера сотрудников в обеих таблицах. Для этой строки в таблице Works_on есть две строки, для которых значение ProjectNumber равно p1 и p3 соответственно. Следовательно, вложенный запрос возвращает результат p1

и р3. Значение одного из элементов этого результирующего набора равно константе р3, поэтому условие удовлетворяется, и отображается соответствующее значение второй строки столбца LastName ('Фролов'). Такой же обработке подвергаются все остальные строки таблицы Employee, и в конечном результате возвращается набор из трех строк.

В следующем разделе приводятся дополнительные примеры по связанным подзапросам.

Подзапросы и функция EXISTS

Функция EXISTS принимает вложенный запрос в качестве аргумента и возвращает значение false, если вложенный запрос не возвращает строк и значение true в противном случае. Рассмотрим работу этой функции на нескольких примерах, начиная со следующего примера:

USE SampleDb;

```
SELECT LastName
FROM Employee
WHERE EXISTS
  (SELECT *
   FROM Works_on
   WHERE Employee.Id = Works_on.EmpId
   AND ProjectNumber = 'p1');
```

В этом примере происходит выборка фамилий всех сотрудников, работающих над проектом p1. Вложенный запрос функции EXISTS почти всегда зависит от переменной с внешнего запроса. Поэтому функция EXISTS обычно определяет связанный подзапрос.

Давайте проследим, как Database Engine может обрабатывать запрос в этом примере. Сначала внешний запрос рассматривает первую строку таблицы Employee (сотрудник Фролов). Далее функция EXISTS определяет, есть ли в таблице Works_on строки, чьи номера сотрудников совпадают с номером сотрудника в текущей строке во внешнем запросе и чей ProjectNumber равен p1. Поскольку сотрудник Фролов не работает над проектом p1, вложенный запрос возвращает пустой набор, вследствие чего функция EXISTS возвращает значение false. Таким образом, сотрудник Фролов не включается в конечный результирующий набор. Этому процессу подвергаются все строки таблицы Employee, после чего выводится конечный результирующий набор.

В примере ниже показано использование функции **NOT EXISTS**:

USE SampleDb;

```
SELECT LastName
FROM Employee
WHERE NOT EXISTS
  (SELECT *
   FROM Department
   WHERE Employee.DepartmentNumber = Department.Number
```

AND Location = 'Санкт-Петербург');

В этом примере происходит выборка фамилий сотрудников, чей отдел не расположен в Санкт-Петербурге.

Список выбора инструкции **SELECT** во внешнем запросе с функцией **EXISTS** не обязательно должен быть в форме **SELECT ***, как в предыдущем примере. Можно использовать альтернативную форму **SELECT column_list**, где **column_list** представляет список из одного или нескольких столбцов таблицы. Обе формы равнозначны, потому что функция **EXISTS** только проверяет на наличие (или отсутствие) строк в результирующем наборе. По этой причине в данном случае правильнее использовать форму **SELECT ***.

Что использовать, соединения или подзапросы?

Почти все инструкции **SELECT** для соединения таблицы посредством оператора соединения **JOIN** можно заменить инструкциями подзапроса и наоборот. Конструкция инструкции **SELECT** с использованием оператора соединения часто более удобно читаемая и легче понимаемая, а также может помочь компоненту Database Engine найти более эффективную стратегию для выборки требуемых данных. Но некоторые задачи легче поддаются решению посредством подзапросов, а другие при помощи соединений.

Преимущества подзапросов

Подзапросы будет более выгодно использовать в таких случаях, когда требуется вычислить агрегатное значение "на лету" и использовать его в другом запросе для сравнения. Это показано в примере ниже:

USE SampleDb;

```
SELECT EmpId, EnterDate  
FROM Works_on  
WHERE EnterDate = (SELECT MIN(EnterDate)  
                   FROM Works_on);
```

В этом примере происходит выборка табельных номеров сотрудников и дат начала их работы над проектом (**EnterDate**) для всех сотрудников, у которых дата начала работы равна самой ранней дате. Решить эту задачу с помощью соединения будет нелегко, поскольку для этого нужно поместить агрегатную функцию в предложении **WHERE**, а это не разрешается. (Эту задачу можно решить, используя два отдельных запроса по отношению к таблице **Works_on**.)

Преимущества соединений

Использовать соединения вместо подзапросов выгоднее в тех случаях, когда список выбора инструкции **SELECT** в запросе содержит столбцы более чем из одной таблицы. Это показано в примере ниже:

USE SampleDb;

```
SELECT Employee.Id, Employee.LastName, Job  
FROM Employee, Works_on  
WHERE Employee.Id = Works_on.EmpId  
      AND EnterDate = '2007-04-15';
```

В этом примере происходит выборка информации о всех сотрудниках (табельный номер, фамилия и должность), которые начали участвовать в работе над проектом с 15 апреля 2007 г. Список выбора инструкции SELECT в запросе содержит столбцы Id и LastName из таблицы Employee и столбец Job из таблицы Works_on. По этой причине решение с применением подзапроса возвратило бы ошибку, поскольку подзапросы могут отображать информацию только из внешней таблицы.

Временные таблицы

Временная таблица - это объект базы данных, который хранится и управляется системой базы данных на временной основе. Временные таблицы могут быть локальными или глобальными. Локальные временные таблицы представлены физически, т.е. они хранятся в системной базе данных tempdb. Имена временных таблиц начинаются с префикса #, например #table_name.

Временная таблица принадлежит создавшему ее сеансу, и видима только этому сеансу. Временная таблица удаляется по завершению создавшего ее сеанса. (Также локальная временная таблица, определенная в хранимой процедуре, удаляется по завершению выполнения этой процедуры.)

Глобальные временные таблицы видимы любому пользователю и любому соединению и удаляются после отключения от сервера базы данных всех обращающихся к ним пользователей. В отличие от локальных временных таблиц имена глобальных временных таблиц начинаются с префикса ##. В примере ниже показано создание временной таблицы, называемой project_temp, используя две разные инструкции языка Transact-SQL:

```
USE SampleDb;
```

```
CREATE TABLE #project_temp (  
    Number NCHAR(4) NOT NULL,  
    Name NCHAR(25) NOT NULL  
);
```

```
-- Аналог предыдущей инструкции со вставкой  
-- данных во временную таблицу из существующей  
-- таблицы Project
```

```
SELECT Number, ProjectName  
    INTO #project_temp  
    FROM Project;
```

Два этих подхода похожи в том, что в обоих создается локальная временная таблица #project_temp. При этом таблица, созданная инструкцией CREATE TABLE, остается пустой, а созданная инструкцией SELECT заполняется данными из таблицы Project.

Лабораторная работа № 6 Инструкция JOIN

Ранее мы рассмотрели применение инструкции SELECT для выборки данных из одной таблицы базы данных. Если бы возможности языка Transact-SQL ограничивались поддержкой только таких простых инструкций SELECT, то присоединение в запросе двух или больше таблиц для выборки из них данных было бы невозможно. Следственно, все данные базы данных требовалось бы хранить в одной таблице. Хотя такой подход является вполне возможным, ему присущ один значительный недостаток - хранимые таким образом данные характеризуются высокой избыточностью.

Язык Transact-SQL устраняет этот недостаток, предоставляя для этого **оператор соединения JOIN**, который позволяет извлекать данные более чем из одной таблицы. Этот оператор, наверное, является наиболее важным оператором для реляционных систем баз данных, поскольку благодаря ему имеется возможность распределять данные по нескольким таблицам, обеспечивая, таким образом, важное свойство систем баз данных - отсутствие избыточности данных.

Оператор UNION, который мы рассмотрели ранее, также позволяет выполнять запрос по нескольким таблицам. Но этот оператор позволяет присоединить несколько инструкций SELECT, тогда как оператор соединения JOIN соединяет несколько таблиц с использованием всего лишь одной инструкции SELECT. Кроме этого, оператор UNION объединяет строки таблиц, в то время как оператор JOIN соединяет столбцы.

Оператор соединения также можно применять с базовыми таблицами и представлениями. Оператор соединения JOIN имеет несколько разных форм. Следующие основные формы этого оператора:

- естественное соединение;
- декартово произведение или перекрестное соединение;
- внешнее соединение;
- тета-соединение, самосоединение и полусоединение.

Прежде чем приступить к рассмотрению разных форм соединений, в этом разделе мы рассмотрим разные варианты оператора соединения JOIN.

Две синтаксические формы реализации соединений

Для соединения таблиц можно использовать две разные синтаксические формы оператора соединения:

- явный синтаксис соединения (синтаксис соединения ANSI SQL:1992);
- неявный синтаксис соединения (синтаксис соединения "старого стиля").

Синтаксис соединения ANSI SQL:1992 был введен стандартом SQL92 и определяет операции соединения явно, т.е. используя соответствующее имя для каждого типа операции соединения. При явном объявлении соединения используются следующие ключевые слова:

- CROSS JOIN;
- [INNER] JOIN;
- LEFT [OUTER] JOIN;

- RIGHT [OUTER] JOIN;
- FULL [OUTER] JOIN.

Ключевое слово CROSS JOIN определяет декартово произведение двух таблиц. Ключевое слово INNER JOIN определяет естественное соединение двух таблиц, а LEFT OUTER JOIN и RIGHT OUTER JOIN определяют одноименные операции соединения. Наконец, ключевое слово FULL OUTER JOIN определяет соединение правого и левого внешнего соединений. Все эти операции соединения рассматриваются в последующих разделах.

Неявный синтаксис оператора соединения является синтаксисом "старого стиля", где каждая операция соединения определяется неявно посредством предложения WHERE, используя так называемые столбцы соединения.

Для операций соединения рекомендуется использовать явный синтаксис, т.к. это повышает надежность запросов. По этой причине во всех примерах далее, связанных с операциями соединения, используются формы явного синтаксиса. Но в нескольких первых примерах также будет продемонстрирован и синтаксис "старого стиля".

Естественное соединение

Термины "естественное соединение" (natural join) и "соединение по эквивалентности" (equi-join) часто используют синонимично, но между ними есть небольшое различие. Операция соединения по эквивалентности всегда имеет одну или несколько пар столбцов с идентичными значениями в каждой строке. Операция, которая устраняет такие столбцы из результатов операции соединения по эквивалентности, называется естественным соединением. Наилучшим способом объяснить естественное соединение можно посредством примера:

USE SampleDb;

SELECT Employee.*, Department.*

FROM Employee **INNER JOIN** Department

ON Employee.DepartmentNumber = Department.Number;

Запрос возвращает всю информацию обо всех сотрудниках: имя и фамилию, табельный номер, а также имя, номер и местонахождение отдела, при этом для номера отдела отображаются дубликаты столбцов из разных таблиц.

В этом примере в инструкции SELECT для выборки указаны все столбцы таблиц для сотрудника Employee и отдела Department. Предложение FROM инструкции SELECT определяет соединяемые таблицы, а также явно указывает тип операции соединения - **INNER JOIN**. Предложение ON является частью предложения FROM и указывает соединяемые столбцы в обеих таблицах. Выражение "Employee.DepartmentNumber = Department.Number" определяет условие соединения, а оба столбца условия называются *столбцами соединения*.

Результат выполнения этого запроса:

Results		Messages					
	Id	FirstName	LastName	DepartmentNumber	Number	DepartmentName	Location
1	2581	Василий	Фролов	d2	d2	Бух. учет	Санкт-Петербург
2	9031	Елена	Лебеденко	d2	d2	Бух. учет	Санкт-Петербург
3	10102	Анна	Иванова	d3	d3	Маркетинг	Москва
4	18316	Игорь	Соловьев	d1	d1	Исследования	Москва
5	25348	Дмитрий	Волков	d3	d3	Маркетинг	Москва
6	28559	Наталья	Вершинина	d1	d1	Исследования	Москва
7	29346	Олег	Маменко	d2	d2	Бух. учет	Санкт-Петербург

Эквивалентный запрос с применением неявного синтаксиса ("старого стиля") будет выглядеть следующим образом:

```
USE SampleDb;
```

```
SELECT Employee.*, Department.*
FROM Employee, Department
WHERE Employee.DepartmentNumber = Department.Number;
```

Эта форма синтаксиса имеет два значительных различия с явной формой: список соединяемых таблиц указывается в предложении FROM, а соответствующее условие соединения указывается в предложении WHERE посредством соединяемых столбцов.

Настоятельно рекомендуется использовать подстановочный знак * в списке выбора SELECT только при работе с SQL в интерактивном режиме, и избегать его применения в прикладных программах.

На предыдущих примерах можно проиллюстрировать принцип работы операции соединения. Но при этом следует иметь в виду, что это всего лишь представление о процессе соединения, т.к. в действительности компонент Database Engine выбирает реализацию операции соединения из нескольких возможных стратегий. Представьте себе, что каждая строка таблицы Employee соединена с каждой строкой таблицы Department. В результате получится таблица с семью столбцами (4 столбца из таблицы Employee и 3 из таблицы Department) и 21 строкой.

Далее, из этой таблицы удаляются все строки, которые не удовлетворяют условию соединения "Employee.Number = Department.Number". Оставшиеся строки представляют результат первого примера выше. Соединяемые столбцы должны иметь идентичную семантику, т.е. оба столбца должны иметь одинаковое логическое значение. Соединяемые столбцы не обязательно должны иметь одинаковое имя (или даже одинаковый тип данных), хотя часто так и бывает.

Система базы данных не может определить логическое значение столбца. Например, она не может определить, что между столбцами номера проекта и табельного номера сотрудника нет ничего общего, хотя оба они имеют целочисленный тип данных. Поэтому система базы данных может только проверить тип данных и длину строк. Компонент Database Engine требует, что соединяемые столбцы имели совместимые типы данных, например INT и SMALLINT.

База данных SampleDb содержит три пары столбцов, где каждый столбец в паре имеет одинаковое логическое значение (а также одинаковые

имена). Таблицы Employee и Department можно соединить по столбцам Employee.DepartmentNumber и Department.Number. Столбцами соединения таблиц Employee и Works_on являются столбцы Employee.Id и Works_on.EmpId. Наконец, таблицы Project и Works_on можно соединить по столбцам Project.Number и Works_on.ProjectNumber.

Имена столбцов в инструкции SELECT можно уточнить. В данном контексте под уточнением имеется в виду, что во избежание неопределенности относительно того, какой таблице принадлежит столбец, в имя столбца включается имя его таблицы (или псевдоним таблицы), отделенное точкой:

table_name.column_name (имя_таблицы.имя_столбца)

В большинстве инструкций SELECT столбцы не требуют уточнения, хотя обычно рекомендуется применять уточнение столбцов с целью улучшения понимания кода. Если же имена столбцов в инструкции SELECT неоднозначны (как, например, столбцы Number в таблицах Project и Department) использование уточненных имен столбцов является обязательным.

В инструкции SELECT с операцией соединения, кроме условия соединения предложение WHERE может содержать и другие условия, как это показано в примере ниже:

USE SampleDb;

-- Явный синтаксис

```
SELECT EmpId, Project.Number, Job, EnterDate, ProjectName, Budget
FROM Works_on JOIN Project
ON Project.Number = Works_on.ProjectNumber
WHERE ProjectName = 'Gemini';
```

-- Старый стиль

```
SELECT EmpId, Project.Number, Job, EnterDate, ProjectName, Budget
FROM Works_on, Project
WHERE Project.Number = Works_on.ProjectNumber
AND ProjectName = 'Gemini';
```

Использование уточненного имени столбца Project.Number в примере выше не является обязательным, поскольку в данном случае нет никакой двусмысленности в отношении их имен. В дальнейшем во всех примерах будет использоваться только явный синтаксис соединения.

В примере ниже показано еще одно применение внутреннего соединения:

USE SampleDb;

-- Вернет 'd2'

```
SELECT DepartamentNumber
FROM Employee JOIN Works_on
ON Employee.Id = Works_on.EmpId
```

WHERE EnterDate = '2007-04-15';

Соединение более чем двух таблиц

Теоретически количество таблиц, которые можно соединить в инструкции **SELECT**, неограниченно. (Но одно условие соединения совмещает только две таблицы!) Однако для компонента Database Engine количество соединяемых таблиц в инструкции **SELECT** ограничено 64 таблицами.

В примере ниже показано соединение трех таблиц базы данных SampleDb:

USE SampleDb;

-- Вернет единственного сотрудника 'Василий Фролов'

SELECT FirstName, LastName

FROM Works_on

JOIN Employee **ON** Works_on.EmpId = Employee.Id

JOIN Department

ON Employee.DepartmentNumber = Department.Number

AND Location = 'Санкт-Петербург'

AND Job = 'Аналитик';

В этом примере происходит выборка имен и фамилий всех аналитиков (Job = 'Аналитик'), чей отдел находится в Санкт-Петербурге (Location = 'Санкт-Петербург'). Результат запроса, приведенного в примере выше, можно получить только в том случае, если соединить, по крайней мере, три таблицы: Works_on, Employee и Department. Эти таблицы можно соединить, используя две пары столбцов соединения:

(Works_on.EmpId, Employee.Id) (Employee.DepartmentNumber, Department.Number)

Обратите внимание, что для осуществления естественного соединения трех таблиц используется два условия соединения, каждое из которых соединяет по две таблицы. А при соединении четырех таблиц таких условий соединения требуется три. В общем, чтобы избежать получения декартового продукта при соединении n таблиц, требуется применять $n - 1$ условий соединения. Конечно же, допустимо использование более чем $n - 1$ условий соединения, а также других условий, для того чтобы еще больше уменьшить количество элементов в результирующем наборе данных.

Декартово произведение

В предшествующем разделе мы рассмотрели возможный способ создания естественного соединения. На первом шаге этого процесса каждая строка таблицы Employee соединяется с каждой строкой таблицы Department. Эта операция называется *декартовым произведением (cartesian product)*. Запрос для создания соединения таблиц Employee и Department, используя декартово произведение, показан в примере ниже:

USE SampleDb;

SELECT Employee.*, Department.*

FROM Employee **CROSS JOIN** Department;

Декартово произведение соединяет каждую строку первой таблицы с каждой строкой второй. В общем, результатом декартового произведения первой таблицы с n строками и второй таблицы с m строками будет таблица с $n*m$ строками. Таким образом, результирующий набор запроса в примере выше имеет $7 \times 3 = 21$ строку (эти строки содержат дублированные значения).

На практике декартово произведение применяется крайне редко. Иногда пользователи получают декартово произведение двух таблиц, когда они забывают включить условие соединения в предложении **WHERE** при использовании неявного синтаксиса соединения "старого стиля". В таком случае полученный результат не соответствует ожидаемому, т.к. содержит лишние строки. Наличие неожиданно большого количества строк в результате служит признаком того, что вместо требуемого естественного соединения двух таблиц было получено декартово произведение.

Внешнее соединение

В предшествующих примерах естественного соединения, результирующий набор содержал только те строки с одной таблицы, для которых имелись соответствующие строки в другой таблице. Но иногда кроме совпадающих строк бывает необходимым извлечь из одной или обеих таблиц строки без совпадений. Такая операция называется *внешним соединением (outer join)*.

В примере ниже показана выборка всей информации для сотрудников, которые проживают и работают в одном и том же городе.

USE SampleDb;

```
SELECT DISTINCT EmployeeEnh.*, Department.Location  
FROM EmployeeEnh JOIN Department  
ON City = Location;
```

Результат выполнения этого запроса:

Results		Messages				
	Id	FirstNa...	LastName	DepartamentNum...	City	Location
1	9031	Елена	Лебеденко	d2	Москва	Москва
2	18316	Игорь	Соловьев	d1	Санкт-Петербург	Санкт-Петербург

В этом примере получение требуемых строк осуществляется посредством естественного соединения. Если бы в этот результат потребовалось включить сотрудников, проживающих в других местах, то нужно было применить левое внешнее соединение. Данное внешнее соединение называется левым потому, что оно возвращает все строки из таблицы с левой стороны оператора сравнения, независимо от того, имеются ли совпадающие строки в таблице с правой стороны. Иными словами, данное внешнее соединение возвратит строку с левой таблицы, даже если для нее нет совпадения в правой таблице, со значением **NULL** соответствующего столбца для всех строк с несовпадающим значением столбца другой, правой,

таблицы. Для выполнения операции левого внешнего соединения компонент Database Engine использует оператор **LEFT OUTER JOIN**.

Операция правого внешнего соединения аналогична левому, но возвращаются все строки таблицы с правой части выражения. Для выполнения операции правого внешнего соединения компонент Database Engine использует оператор **RIGHT OUTER JOIN**.

USE SampleDb;

```
SELECT EmployeeEnh.*, Department.Location
FROM EmployeeEnh
LEFT OUTER JOIN Department
ON City = Location;
```

В этом примере происходит выборка сотрудников (с включением полной информации) для таких городов, в которых сотрудники или только проживают (столбец City в таблице EmployeeEnh), или проживают и работают. Результат выполнения этого запроса:

Results		Messages				
	Id	FirstNa...	LastName	DepartamentNum...	City	Location
1	2581	Василий	Фролов	d2	Казань	NULL
2	9031	Елена	Лебеденко	d2	Москва	Москва
3	10102	Анна	Иванова	d3	Екатеринбург	NULL
4	18316	Игорь	Соловьев	d1	Санкт-Петербург	Санкт-Петербург
5	25348	Дмитрий	Волков	d3	Краснодар	NULL
6	28559	Наталья	Вершинина	d1	Казань	NULL
7	29346	Олег	Маменко	d2	Пермь	NULL

Как можно видеть в результате выполнения запроса, когда для строки из левой таблицы (в данном случае EmployeeEnh) нет совпадающей строки в правой таблице (в данном случае Department), операция левого внешнего соединения все равно возвращает эту строку, заполняя значением NULL все ячейки соответствующего столбца для несовпадающего значения столбца правой таблицы. Применение правого внешнего соединения показано в примере ниже:

USE SampleDb;

```
SELECT EmployeeEnh.City, Department.*
FROM EmployeeEnh
RIGHT OUTER JOIN Department
ON City = Location;
```

В этом примере происходит выборка отделов (с включением полной информации о них) для таких городов, в которых сотрудники или только работают, или проживают и работают. Результат выполнения этого запроса:

Results		Messages		
	City	Number	DepartmentName	Location
1	Москва	d1	Исследования	Москва
2	Санкт-Петербург	d2	Бух. учет	Санкт-Петербург
3	Москва	d3	Маркетинг	Москва

Кроме левого и правого внешнего соединения, также существует полное внешнее соединение, которое является объединением левого и правого внешних соединений. Иными словами, результирующий набор такого соединения состоит из всех строк обеих таблиц. Если для строки одной из таблиц нет соответствующей строки в другой таблице, всем ячейкам строки второй таблицы присваивается значение NULL. Для выполнения операции полного внешнего соединения используется оператор **FULL OUTER JOIN**.

Любую операцию внешнего соединения можно эмулировать, используя оператор **UNION** совместно с функцией **NOT EXISTS**. Таким образом, запрос, показанный в примере ниже, эквивалентен запросу левого внешнего соединения, показанному ранее. В данном запросе осуществляется выборка сотрудников (с включением полной информации) для таких городов, в которых сотрудники или только проживают или проживают и работают:

USE SampleDb;

```
SELECT EmployeeEnh.*, Department.Location
FROM EmployeeEnh JOIN Department
ON City = Location
UNION
SELECT EmployeeEnh.*, 'NULL'
FROM EmployeeEnh
WHERE NOT EXISTS
  (SELECT *
   FROM Department
   WHERE Location = City);
```

Первая инструкция **SELECT** объединения определяет естественное соединение таблиц **EmployeeEnh** и **Department** по столбцам соединения **City** и **Location**. Эта инструкция возвращает все города для всех сотрудников, в которых сотрудники и проживают и работают. Дополнительно, вторая инструкция **SELECT** объединения возвращает все строки таблицы **EmployeeEnh**, которые не отвечают условию в естественном соединении.

Другие формы операций соединения

В предшествующих разделах мы рассмотрели наиболее важные формы соединения. Но существуют и другие формы этой операции, которые мы рассмотрим в следующих подразделах.

Тета-соединение

Условие сравнения столбцов соединения не обязательно должно быть равенством, но может быть любым другим сравнением. Соединение, в

котором используется общее условие сравнения столбцов соединения, называется *тета-соединением*. В примере ниже показана операция тета-соединения, в которой используется условие "меньше чем". Данный запрос возвращает все комбинации информации о сотрудниках и отделах для тех случаев, когда место проживания сотрудника по алфавиту идет перед месторасположением любого отдела, в котором работает этот служащий:

USE SampleDb;

```
SELECT FirstName, LastName, City, Location
FROM EmployeeEnh JOIN Department
ON City < Location
WHERE Location = 'Санкт-Петербург';
```

Результат выполнения этого запроса:

	FirstName	LastName	City	Location
1	Василий	Фролов	Казань	Санкт-Петербург
2	Елена	Лебеденко	Москва	Санкт-Петербург
3	Анна	Иванова	Екатеринбург	Санкт-Петербург
4	Дмитрий	Волков	Краснодар	Санкт-Петербург
5	Наталья	Вершинина	Казань	Санкт-Петербург
6	Олег	Маменко	Пермь	Санкт-Петербург

В этом примере сравниваются соответствующие значения столбцов City и Location. В каждой строке результата значение столбца City сравнивается в алфавитном порядке с соответствующим значением столбца Location.

Самосоединение, или соединение таблицы самой с собой

Кроме соединения двух или больше разных таблиц, операцию естественного соединения можно применить к одной таблице. В данной операции таблица соединяется сама с собой, при этом один столбец таблицы сравнивается сам с собой. Сравнение столбца с самим собой означает, что в предложении FROM инструкции SELECT имя таблицы употребляется дважды. Поэтому необходимо иметь возможность ссылаться на имя одной и той же таблицы дважды. Это можно осуществить, используя, по крайней мере, один псевдоним. То же самое относится и к именам столбцов в условии соединения в инструкции SELECT. Для того чтобы различить столбцы с одинаковыми именами, необходимо использовать уточненные имена.

Соединение таблицы с самой собой демонстрируется в примере ниже:

USE SampleDb;

```
SELECT t1.Number, t1.DepartmentName, t1.Location
FROM Department t1 JOIN Department t2
ON t1.Location = t2.Location
WHERE t1.Number <> t2.Number;
```

В этом примере происходит выборка всех отделов (с полной информацией), расположенных в том же самом месте, как и, по крайней мере, один другой отдел. Результат выполнения этого запроса:

Results		Messages	
	Numb...	DepartmentName	Location
1	d3	Маркетинг	Москва
2	d1	Исследования	Москва

Здесь предложение FROM содержит два псевдонима для таблицы Department: t1 и t2. Первое условие в предложении WHERE определяет столбцы соединения, а второе - удаляет ненужные дубликаты, обеспечивая сравнение каждого отдела с другими отделами.

Полусоединение

Полусоединение похоже на естественное соединение, но возвращает только набор всех строк из одной таблицы, для которой в другой таблице есть одно или несколько совпадений. Использование полусоединения показано в примере ниже:

USE SampleDb;

```
SELECT Id, LastName, FirstName
FROM Employee e JOIN Department d
ON e.DepartmentNumber = d.Number
WHERE Location = 'Москва';
```

Результат выполнения запроса:

Results		Messages	
	Id	LastName	FirstName
1	10102	Иванова	Анна
2	18316	Соловьев	Игорь
3	25348	Волков	Дмитрий
4	28559	Вершинина	Наталья

Как можно видеть, список выбора SELECT в полусоединении содержит только столбцы из таблицы Employee. Это и есть характерной особенностью операции полусоединения. Эта операция обычно применяется в распределенной обработке запросов, чтобы свести к минимуму объем передаваемых данных. Компонент Database Engine использует операцию полусоединения для реализации функциональности, называемой соединением типа "звезда".

Лабораторная работа №7. Создание хранимых процедур в Microsoft SQL Server

Цель работы – научиться создавать и использовать хранимые процедуры на сервере БД.

Содержание работы:

1. Проработка всех примеров, анализ результатов их выполнения в утилите Query Analyzer. Проверка наличия созданных процедур в текущей БД.
2. Выполнение всех примеров и заданий по ходу лабораторной работы.
3. Выполнение индивидуальных заданий по вариантам.

Пояснения к выполнению работы

Хранимая процедура - это специальный тип пакета инструкций Transact-SQL, созданный, используя язык SQL и процедурные расширения. Основное различие между пакетом и хранимой процедурой состоит в том, что последняя сохраняется в виде объекта базы данных. Иными словами, хранимые процедуры сохраняются на стороне сервера, чтобы улучшить производительность и постоянство выполнения повторяемых задач.

Компонент Database Engine поддерживает хранимые процедуры и системные процедуры. Хранимые процедуры создаются таким же образом, как и все другие объекты баз данных, т.е. при помощи языка DDL. *Системные процедуры* предоставляются компонентом Database Engine и могут применяться для доступа к информации в системном каталоге и ее модификации.

При создании хранимой процедуры можно определить необязательный список параметров. Таким образом, процедура будет принимать соответствующие аргументы при каждом ее вызове. Хранимые процедуры могут возвращать значение, содержащее определенную пользователем информацию или, в случае ошибки, соответствующее сообщение об ошибке.

Хранимая процедура предварительно компилируется перед тем, как она сохраняется в виде объекта в базе данных. Предварительно скомпилированная форма процедуры сохраняется в базе данных и используется при каждом ее вызове. Это свойство хранимых процедур предоставляет важную выгоду, заключающуюся в устранении (почти во всех случаях) повторных компиляций процедуры и получении соответствующего улучшения производительности. Это свойство хранимых процедур также оказывает положительный эффект на объем данных, участвующих в обмене между системой баз данных и приложениями. В частности, для вызова хранимой процедуры объемом в несколько тысяч байтов может потребоваться меньше, чем 50 байт. Когда множественные пользователи выполняют повторяющиеся задачи с применением хранимых процедур, накопительный эффект такой экономии может быть довольно значительным.

Хранимые процедуры можно также использовать для следующих целей:

- управления авторизацией доступа;
- для создания журнала логов о действиях с таблицами баз данных.

Использование хранимых процедур предоставляет возможность управления безопасностью на уровне, значительно превышающем уровень безопасности, предоставляемый использованием инструкций GRANT и REVOKE, с помощью которых пользователям предоставляются разные привилегии доступа. Это возможно вследствие того, что авторизация на выполнение хранимой процедуры не зависит от авторизации на модифицирование объектов, содержащихся в данной хранимой процедуре, как это описано в следующем разделе.

Хранимые процедуры, которые создают логи операций записи и/или чтения таблиц, предоставляют дополнительную возможность обеспечения безопасности базы данных. Используя такие процедуры, администратор базы данных может отслеживать модификации, вносимые в базу данных пользователями или прикладными программами.

Создание и исполнение хранимых процедур

Хранимые процедуры создаются посредством инструкции **CREATE PROCEDURE**, которая имеет следующий синтаксис:

```
CREATE PROC[EDURE] [schema_name.]proc_name  
  [({@param1} type1 [ VARYING] [= default1] [OUTPUT])] {, ...}  
  [WITH {RECOMPILE | ENCRYPTION | EXECUTE AS 'user_name'}]  
  [FOR REPLICATION]  
  AS batch | EXTERNAL NAME method_name
```

Параметр `schema_name` определяет имя схемы, которая назначается владельцем созданной хранимой процедуры. Параметр `proc_name` определяет имя хранимой процедуры. Параметр `@param1` является параметром процедуры (формальным аргументом), чей тип данных определяется параметром `type1`. Параметры процедуры являются локальными в пределах процедуры, подобно тому, как локальные переменные являются локальными в пределах пакета. Параметры процедуры - это значения, которые передаются вызывающим объектом процедуре для использования в ней. Параметр `default1` определяет значение по умолчанию для соответствующего параметра процедуры. (Значением по умолчанию также может быть NULL.)

Опция *OUTPUT* указывает, что параметр процедуры является возвращаемым, и с его помощью можно вернуть значение из хранимой процедуры вызывающей процедуре или системе.

Как уже упоминалось ранее, предварительно компилированная форма процедуры сохраняется в базе данных и используется при каждом ее вызове. Если же по каким-либо причинам хранимую процедуру требуется компилировать при каждом ее вызове, при объявлении процедуры используется опция *WITH RECOMPILE*. Использование опции *WITH RECOMPILE* сводит на нет одно из наиболее важных преимуществ

хранимых процедур: улучшение производительности благодаря одной компиляции. Поэтому опцию WITH RECOMPILE следует использовать только при частых изменениях используемых хранимой процедурой объектов базы данных.

Предложение EXECUTE AS определяет контекст безопасности, в котором должна исполняться хранимая процедура после ее вызова. Задавая этот контекст, с помощью Database Engine можно управлять выбором учетных записей пользователей для проверки полномочий доступа к объектам, на которые ссылается данная хранимая процедура.

По умолчанию использовать инструкцию CREATE PROCEDURE могут только члены предопределенной роли сервера sysadmin и предопределенной роли базы данных db_owner или db_ddladmin. Но члены этих ролей могут присваивать это право другим пользователям с помощью инструкции **GRANT CREATE PROCEDURE**.

В примере ниже показано создание простой хранимой процедуры для работы с таблицей Project:

```
USE SampleDb;
```

```
GO
```

```
CREATE PROCEDURE IncreaseBudget (@percent INT=5)
```

```
AS UPDATE Project
```

```
SET Budget = Budget + Budget * @percent/100;
```

Как говорилось ранее, для разделения двух пакетов используется **инструкция GO**. Инструкцию CREATE PROCEDURE нельзя объединять с другими инструкциями Transact-SQL в одном пакете. Хранимая процедура IncreaseBudget увеличивает бюджеты для всех проектов на определенное число процентов, определяемое посредством параметра @percent. В процедуре также определяется значение числа процентов по умолчанию (5), которое применяется, если во время выполнения процедуры этот аргумент отсутствует.

Хранимые процедуры могут обращаться к несуществующим таблицам. Это свойство позволяет выполнять отладку кода процедуры, не создавая сначала соответствующие таблицы и даже не подключаясь к конечному серверу.

В отличие от основных хранимых процедур, которые всегда сохраняются в текущей базе данных, возможно создание временных хранимых процедур, которые всегда помещаются во временную системную базу данных tempdb. Одним из поводов для создания временных хранимых процедур может быть желание избежать повторяющегося исполнения определенной группы инструкций при соединении с базой данных. Можно создавать локальные или глобальные временные процедуры. Для этого имя локальной процедуры задается с одинарным символом # (#proc_name), а имя глобальной процедуры - с двойным (##proc_name).

Локальную временную хранимую процедуру может выполнить только создавший ее пользователь и только в течение соединения с базой данных, в

которой она была создана. Глобальную временную процедуру могут выполнять все пользователи, но только до тех пор, пока не завершится последнее соединение, в котором она выполняется (обычно это соединение создателя процедуры).

Жизненный цикл хранимой процедуры состоит из двух этапов: ее создания и ее выполнения. Каждая процедура создается один раз, а выполняется многократно. Хранимая процедура выполняется посредством **инструкции EXECUTE** пользователем, который является владельцем процедуры или обладает правом EXECUTE для доступа к этой процедуре. Инструкция EXECUTE имеет следующий синтаксис:

```
[[EXEC[UTE]] [@return_status =] {proc_name | @proc_name_var}
  {[@parameter1 =] value
    | [@parameter1 =] @variable [OUTPUT]] | DEFAULT}..
[WITH RECOMPILE]
```

За исключением параметра return_status, все параметры инструкции EXECUTE имеют такое же логическое значение, как и одноименные параметры инструкции CREATE PROCEDURE. Параметр return_status определяет целочисленную переменную, в которой сохраняется состояние возврата процедуры. Значение параметру можно присвоить, используя или константу (value), или локальную переменную (@variable). Порядок значений именованных параметров не важен, но значения неименованных параметров должны предоставляться в том порядке, в каком они определены в инструкции CREATE PROCEDURE.

Предложение DEFAULT предоставляет значения по умолчанию для параметра процедуры, которое было указано в определении процедуры. Когда процедура ожидает значение для параметра, для которого не было определено значение по умолчанию и отсутствует параметр, либо указано ключевое слово DEFAULT, то происходит ошибка.

Когда инструкция EXECUTE является первой инструкцией пакета, ключевое слово EXECUTE можно опустить. Тем не менее будет надежнее включать это слово в каждый пакет. Использование инструкции EXECUTE показано в примере ниже:

```
USE SampleDb;
```

```
EXECUTE IncreaseBudget 10;
```

Инструкция EXECUTE в этом примере выполняет хранимую процедуру IncreaseBudget, которая увеличивает бюджет всех проектов на 10%.

В примере ниже показано создание хранимой процедуры для обработки данных в таблицах Employee и Works_on:

```
USE SampleDb;
```

```
GO
```

```
CREATE PROCEDURE ModifyEmpId (@oldId INTEGER, @newId  
INTEGER)
```

```
AS UPDATE Employee  
SET Id = @newId  
WHERE Id = @oldId;
```

```
UPDATE Works_on  
SET EmpId = @newId  
WHERE EmpId = @oldId;
```

Процедура ModifyEmpId в примере иллюстрирует использование хранимых процедур, как часть процесса обеспечения ссылочной целостности (в данном случае между таблицами Employee и Works_on). Подобную хранимую процедуру можно использовать внутри определения триггера, который собственно и обеспечивает ссылочную целостность.

В примере ниже показано использование в хранимой процедуре предложения OUTPUT:

```
USE SampleDb;
```

```
GO
```

```
CREATE PROCEDURE DeleteEmployee @empId INT, @counter INT  
OUTPUT
```

```
AS SELECT @counter = COUNT(*)  
FROM Works_on  
WHERE EmpId = @empId
```

```
DELETE FROM Employee  
WHERE Id = @empId
```

```
DELETE FROM Works_on  
WHERE EmpId = @empId;
```

Данную хранимую процедуру можно запустить на выполнение посредством следующих инструкций:

```
DECLARE @quantityDeleteEmployee INT;  
EXECUTE DeleteEmployee @empId=18316,  
@counter=@quantityDeleteEmployee OUTPUT;
```

```
PRINT N'Удалено сотрудников: ' + convert(nvarchar(30),  
@quantityDeleteEmployee);
```

Эта процедура подсчитывает количество проектов, над которыми занят сотрудник с табельным номером @empId, и присваивает полученное значение параметру @counter. После удаления всех строк для данного табельного номера из таблиц Employee и Works_on вычисленное значение присваивается переменной @quantityDeleteEmployee.

Значение параметра возвращается вызывающей процедуре только в том случае, если указана опция OUTPUT. В примере выше процедура DeleteEmployee передает вызывающей процедуре параметр @counter, следовательно, хранимая процедура возвращает значение системе. Поэтому параметр @counter необходимо указывать как в опции OUTPUT при объявлении процедуры, так и в инструкции EXECUTE при ее вызове.

Предложение WITH RESULTS SETS инструкции EXECUTE

В SQL Server 2012 для инструкции EXECUTE вводится **предложение WITH RESULTS SETS**, посредством которого при выполнении определенных условий можно изменять форму результирующего набора хранимой процедуры.

Следующие два примера помогут объяснить это предложение. Первый пример является вводным примером, который показывает, как может выглядеть результат, когда опущено предложение WITH RESULTS SETS:

```
USE SampleDb;
```

```
GO
```

```
CREATE PROCEDURE EmployeesInDept (@dept CHAR(4))
AS SELECT Id, LastName
FROM Employee
WHERE DepartmentNumber IN (
    SELECT @dept FROM Department
    GROUP BY Number);
```

Процедура EmployeesInDept - это простая процедура, которая отображает табельные номера и фамилии всех сотрудников, работающих в определенном отделе. Номер отдела является параметром процедуры, и его нужно указать при ее вызове. Выполнение этой процедуры выводит таблицу с двумя столбцами, заголовки которых совпадают с наименованиями соответствующих столбцов таблицы базы данных, т.е. Id и LastName. Чтобы изменить заголовки столбцов результата (а также их тип данных), в SQL Server 2012 применяется новое предложение WITH RESULTS SETS. Применение этого предложения показано в примере ниже:

```
USE SampleDb;
```

```
EXEC EmployeesInDept 'd1'
WITH RESULT SETS
    (([Id] INT NOT NULL,
    [Фамилия] CHAR(20) NOT NULL));
```

Результат выполнения хранимой процедуры, вызванной таким способом, будет следующим:

Results		
Messages		
	Id	Фамилия
1	18316	Соловьев
2	28559	Вершинина

Как можно видеть, запуск хранимой процедуры с использованием предложения **WITH RESULT SETS** в инструкции **EXECUTE** позволяет изменить наименования и тип данных столбцов результирующего набора, выдаваемого данной процедурой. Таким образом, эта новая функциональность предоставляет большую гибкость в исполнении хранимых процедур и помещении их результатов в новую таблицу.

Изменение структуры хранимых процедур

Компонент Database Engine также поддерживает инструкцию **ALTER PROCEDURE** для модификации структуры хранимых процедур. Инструкция **ALTER PROCEDURE** обычно применяется для изменения инструкций Transact-SQL внутри процедуры. Все параметры инструкции **ALTER PROCEDURE** имеют такое же значение, как и одноименные параметры инструкции **CREATE PROCEDURE**. Основной целью использования этой инструкции является избежание переопределения существующих прав хранимой процедуры.

Компонент Database Engine поддерживает *тип данных CURSOR*. Этот тип данных используется для объявления курсоров в хранимых процедурах. *Курсор* - это конструкция программирования, применяемая для хранения результатов запроса (обычно набора строк) и для предоставления пользователям возможности отображать этот результат построчно.

Для удаления одной или группы хранимых процедур используется **инструкция DROP PROCEDURE**. Удалить хранимую процедуру может только ее владелец или члены предопределенных ролей **db_owner** и **sysadmin**.

Задание для лабораторной работы №7.

Реализовать слой бизнес-логики на уровне хранимых процедур и функций СУБД:

- не менее 3 хранимых процедур для операторов Insert, Update, Delete;
- не менее 3 хранимых процедур для оператора Select с предложением **inner join**.

Лабораторная работа №8. Определение триггера в стандарте языка SQL

Триггер - это механизм, который вызывается, когда в указанной таблице происходит определенное действие. Каждый триггер имеет следующие основные составляющие: имя, действие и исполнение. Имя триггера может содержать максимум 128 символов. Действием триггера может быть или инструкция DML (INSERT, UPDATE или DELETE), или инструкция DDL. Таким образом, существует два типа триггеров: триггеры DML и триггеры DDL. Исполнительная составляющая триггера обычно состоит из хранимой процедуры или пакета.

Компонент Database Engine позволяет создавать триггеры, используя или язык Transact-SQL, или один из языков среды CLR, такой как C# или Visual Basic.

Создание триггера DML

Триггер создается с помощью инструкции **CREATE TRIGGER**, которая имеет следующий синтаксис:

```
CREATE TRIGGER [schema_name.]trigger_name  
  ON {table_name | view_name}  
  [WITH dml_trigger_option [...]]  
  {FOR | AFTER | INSTEAD OF} { [INSERT] [,] [UPDATE] [,]  
  [DELETE]}  
  [WITH APPEND]  
  {AS sql_statement | EXTERNAL NAME method_name}
```

Предшествующий синтаксис относится только к триггерам DML. Триггеры DDL имеют несколько иную форму синтаксиса, которая будет показана позже.

Здесь в параметре `schema_name` указывается имя схемы, к которой принадлежит триггер, а в параметре `trigger_name` - имя триггера. В параметре `table_name` задается имя таблицы, для которой создается триггер. (Также поддерживаются триггеры для представлений, на что указывает наличие параметра `view_name`.)

Также можно задать тип триггера с помощью двух дополнительных параметров: **AFTER** и **INSTEAD OF**. (Параметр **FOR** является синонимом параметра **AFTER**.) **Триггеры типа AFTER** вызываются после выполнения действия, запускающего триггер, а **триггеры типа INSTEAD OF** выполняются вместо действия, запускающего триггер. Триггеры **AFTER** можно создавать только для таблиц, а триггеры **INSTEAD OF** - как для таблиц, так и для представлений.

Параметры **INSERT**, **UPDATE** и **DELETE** задают действие триггера. Под действием триггера имеется в виду инструкция Transact-SQL, которая запускает триггер. Допускается любая комбинация этих трех инструкций. Инструкция **DELETE** не разрешается, если используется параметр **IF UPDATE**.

Как можно видеть в синтаксисе инструкции **CREATE TRIGGER**, действие (или действия) триггера указывается в спецификации **AS sql_statement**.

Компонент Database Engine позволяет создавать несколько триггеров для каждой таблицы и для каждого действия (**INSERT**, **UPDATE** и **DELETE**). По умолчанию определенного порядка исполнения нескольких триггеров для данного модифицирующего действия не имеется.

Только владелец базы данных, администраторы DDL и владелец таблицы, для которой определяется триггер, имеют право создавать триггеры для текущей базы данных. (В отличие от разрешений для других типов инструкции **CREATE** это разрешение не может передаваться.)

Изменение структуры триггера

Язык Transact-SQL также поддерживает инструкцию **ALTER TRIGGER**, которая модифицирует структуру триггера. Эта инструкция обычно применяется для изменения тела триггера. Все предложения и параметры инструкции **ALTER TRIGGER** имеют такое же значение, как и одноименные предложения и параметры инструкции **CREATE TRIGGER**.

Для удаления триггеров в текущей базе данных применяется инструкция **DROP TRIGGER**.

Использование виртуальных таблиц *deleted* и *inserted*

При создании действия триггера обычно требуется указать, ссылается ли он на значение столбца до или после его изменения действием, запускающим триггер. По этой причине, для тестирования следствия инструкции, запускающей триггер, используются две специально именованные виртуальные таблицы:

- *deleted* - содержит копии строк, удаленных из таблицы;
- *inserted* - содержит копии строк, вставленных в таблицу.

Структура этих таблиц эквивалентна структуре таблицы, для которой определен триггер.

Таблица *deleted* используется в том случае, если в инструкции **CREATE TRIGGER** указывается предложение **DELETE** или **UPDATE**, а если в этой инструкции указывается предложение **INSERT** или **UPDATE**, то используется таблица *inserted*. Это означает, что для каждой инструкции **DELETE**, выполненной в действии триггера, создается таблица *deleted*. Подобным образом для каждой инструкции **INSERT**, выполненной в действии триггера, создается таблица *inserted*.

Инструкция **UPDATE** рассматривается, как инструкция **DELETE**, за которой следует инструкция **INSERT**. Поэтому для каждой инструкции **UPDATE**, выполненной в действии триггера, создается как таблица *deleted*, так и таблица *inserted* (в указанной последовательности).

Таблицы *inserted* и *deleted* реализуются, используя управление версиями строк. Когда для таблицы с соответствующими триггерами выполняется инструкция DML (**INSERT**, **UPDATE** или **DELETE**), для всех изменений в этой таблице всегда создаются версии строк. Когда триггеру требуется информация из таблицы *deleted*, он обращается к данным в

хранилище версий строк. В случае таблицы inserted, триггер обращается к самым последним версиям строк.

В качестве хранилища версий строк механизм управления версиями строк использует системную базу данных tempdb. По этой причине, если база данных содержит большое число часто используемых триггеров, следует ожидать значительного увеличения объема этой системной базы данных.

Области применения DML-триггеров

Такие триггеры применяются для решения разнообразных задач. В этом разделе мы рассмотрим несколько областей применения триггеров DML, в частности триггеров AFTER и INSTEAD OF.

Триггеры AFTER

Как вы уже знаете, триггеры AFTER вызываются после того, как выполняется действие, запускающее триггер. Триггер AFTER задается с помощью ключевого слова AFTER или FOR. Триггеры AFTER можно создавать только для базовых таблиц. Триггеры этого типа можно использовать для выполнения, среди прочих, следующих операций:

- создания журнала логов действий в таблицах базы данных;
- реализации бизнес-правил;
- принудительного обеспечения ссылочной целостности.

Создание журнала логов

В SQL Server можно выполнять отслеживание изменения данных, используя систему перехвата изменения данных CDC (change data capture). Эту задачу можно также решить с помощью триггеров DML. В примере ниже показывается, как с помощью триггеров можно создать журнал логов действий в таблицах базы данных:

```
USE SampleDb;
```

```
/* Таблица AuditBudget используется в качестве  
журнала логов действий в таблице Project */  
GO
```

```
CREATE TABLE AuditBudget (  
    ProjectNumber CHAR(4) NULL,  
    UserName CHAR(16) NULL,  
    Date DATETIME NULL,  
    BudgetOld FLOAT NULL,  
    BudgetNew FLOAT NULL  
);
```

```
GO  
CREATE TRIGGER trigger_ModifyBudget  
    ON Project AFTER UPDATE  
    AS IF UPDATE(budget)  
BEGIN  
    DECLARE @budgetOld FLOAT  
    DECLARE @budgetNew FLOAT
```

```
DECLARE @projectNumber CHAR(4)
```

```
SELECT @budgetOld = (SELECT Budget FROM deleted)  
SELECT @budgetNew = (SELECT Budget FROM inserted)  
SELECT @projectNumber = (SELECT Number FROM deleted)
```

```
INSERT INTO AuditBudget VALUES  
    (@projectNumber,  USER_NAME(),  GETDATE(),  @budgetOld,  
    @budgetNew)  
END
```

В этом примере создается таблица AuditBudget, в которой сохраняются все изменения столбца Budget таблицы Project. Изменения этого столбца будут записываться в эту таблицу посредством триггера trigger_ModifyBudget.

Этот триггер активируется для каждого изменения столбца Budget с помощью инструкции UPDATE. При выполнении этого триггера значения строк таблиц deleted и inserted присваиваются соответствующим переменным @budgetOld, @budgetNew и @projectNumber. Эти присвоенные значения, совместно с именем пользователя и текущей датой, будут затем вставлены в таблицу AuditBudget.

В этом примере предполагается, что за один раз будет обновление только одной строки. Поэтому этот пример является упрощением общего случая, когда триггер обрабатывает многострочные обновления. Если выполнить следующие инструкции Transact-SQL:

```
USE SampleDb;
```

```
UPDATE Project  
    SET Budget = 200000  
    WHERE Number = 'p2';
```

то содержимое таблицы AuditBudget будет таким:

Results		Messages			
	ProjectNumber	UserName	Date	BudgetOld	BudgetNew
1	p2	dbo	2015-05-20 15:28:36.623	95000	200000

Реализация бизнес-правил

С помощью триггеров можно создавать бизнес-правила для приложений. Создание такого триггера показано в примере ниже:

```
USE SampleDb;
```

```
-- Триггер trigger_TotalBudget является примером использования  
-- триггера для реализации бизнес-правила  
GO
```

```
CREATE TRIGGER trigger_TotalBudget  
    ON Project AFTER UPDATE  
    AS IF UPDATE (Budget)
```

```

BEGIN
    DECLARE @sum_old1
    FLOAT DECLARE @sum_old2
    FLOAT DECLARE @sum_new FLOAT

    SELECT @sum_new = (SELECT SUM(Budget) FROM inserted)
    SELECT @sum_old1 = (SELECT SUM(p.Budget)
        FROM project p WHERE p.Number
            NOT IN (SELECT d.Number FROM deleted d))
    SELECT @sum_old2 = (SELECT SUM(Budget) FROM deleted)

    IF @sum_new > (@sum_old1 + @sum_old2) * 1.5
    BEGIN
        PRINT 'Бюджет не изменился'
        ROLLBACK TRANSACTION
    END
    ELSE
        PRINT 'Изменение бюджета выполнено'
    END

```

Здесь создается правило для управления модификацией бюджетов проектов. Триггер trigger_TotalBudget проверяет каждое изменение бюджетов и выполняет только такие инструкции UPDATE, которые увеличивают сумму всех бюджетов не более чем на 50%. В противном случае для инструкции UPDATE выполняется откат посредством инструкции ROLLBACK TRANSACTION.

Принудительное обеспечение ограничений целостности

В системах управления базами данных применяются два типа ограничений для обеспечения целостности данных: декларативные ограничения, которые определяются с помощью инструкций языка CREATE TABLE и ALTER TABLE; процедурные ограничения целостности, которые реализуются посредством триггеров.

В обычных ситуациях следует использовать декларативные ограничения для обеспечения целостности, поскольку они поддерживаются системой и не требуют реализации пользователем. Применение триггеров рекомендуется только в тех случаях, для которых декларативные ограничения для обеспечения целостности отсутствуют.

В примере ниже показано принудительное обеспечение ссылочной целостности посредством триггеров для таблиц Employee и Works_on:

```

USE SampleDb;

GO
CREATE TRIGGER trigger_WorksonIntegrity
    ON Works_on AFTER INSERT, UPDATE
    AS IF UPDATE(EmpId)
    BEGIN

```

```

IF (SELECT Employee.Id
    FROM Employee, inserted
    WHERE Employee.Id = inserted.EmpId) IS NULL
BEGIN
    ROLLBACK TRANSACTION
    PRINT 'Строка не была вставлена/модифицирована'
END
ELSE
    PRINT 'Строка была вставлена/модифицирована'
END

```

Триггер trigger_WorksonIntegrity в этом примере проверяет ссылочную целостность для таблиц Employee и Works_on. Это означает, что проверяется каждое изменение столбца Id в ссылочной таблице Works_on, и при любом нарушении этого ограничения выполнение этой операции не допускается. (То же самое относится и к вставке в столбец Id новых значений.) Инструкция ROLLBACK TRANSACTION во втором блоке BEGIN выполняет откат инструкции INSERT или UPDATE в случае нарушения ограничения для обеспечения ссылочной целостности.

В этом примере триггер выполняет проверку на проблемы ссылочной целостности первого и второго случая между таблицами Employee и Works_on. А в примере ниже показан триггер, который выполняет проверку на проблемы ссылочной целостности третьего и четвертого случая между этими же таблицами:

```
USE SampleDb;
```

```

GO
CREATE TRIGGER trigger_RefintWorkson2
ON Employee AFTER DELETE, UPDATE
AS IF UPDATE (Id)
BEGIN
    IF (SELECT COUNT(*)
        FROM Works_on, deleted
        WHERE Works_on.EmpId = deleted.Id) > 0
    BEGIN
        ROLLBACK TRANSACTION
        PRINT 'Строка не была вставлена/модифицирована'
    END
    ELSE
        PRINT 'Строка была вставлена/модифицирована'
    END

```

Триггеры INSTEAD OF

Триггер с предложением **INSTEAD OF** заменяет соответствующее действие, которое запустило его. Этот триггер выполняется после создания соответствующих таблиц inserted и deleted, но перед выполнением проверки ограничений целостности или каких-либо других действий.

Триггеры **INSTEAD OF** можно создавать как для таблиц, так и для представлений. Когда инструкция Transact-SQL ссылается на представление, для которого определен триггер **INSTEAD OF**, система баз данных выполняет этот триггер вместо выполнения любых действий с любой таблицей. Данный тип триггера всегда использует информацию в таблицах **inserted** и **deleted**, созданных для представления, чтобы создать любые инструкции, требуемые для создания запрошенного события.

Значения столбцов, предоставляемые триггером **INSTEAD OF**, должны удовлетворять определенным требованиям:

- значения не могут задаваться для вычисляемых столбцов;
- значения не могут задаваться для столбцов с типом данных **timestamp**;
- значения не могут задаваться для столбцов со свойством **IDENTITY**, если только параметру **IDENTITY_INSERT** не присвоено значение **ON**.

Эти требования действительны только для инструкций **INSERT** и **UPDATE**, которые ссылаются на базовые таблицы. Инструкция **INSERT**, которая ссылается на представления с триггером **INSTEAD OF**, должна предоставлять значения для всех столбцов этого представления, не допускающих пустые значения **NULL**. (То же самое относится и к инструкции **UPDATE**. Инструкция **UPDATE**, ссылающаяся на представление с триггером **INSTEAD OF**, должна предоставить значения для всех столбцов представления, которое не допускает пустых значений и на которое осуществляется ссылка в предложении **SET**.)

В примере ниже показана разница в поведении при вставке значений в вычисляемые столбцы, используя таблицу и ее соответствующее представление:

```
USE SampleDb;
```

```
CREATE TABLE Orders (  
    OrderId INT NOT NULL,  
    Price MONEY NOT NULL,  
    Quantity INT NOT NULL,  
    OrderDate DATETIME NOT NULL,  
    Total AS Price * Quantity,  
    ShippedDate AS DATEADD (DAY, 7, orderdate)  
);
```

```
GO  
CREATE VIEW view_AllOrders  
    AS SELECT *  
    FROM Orders;
```

```
GO  
CREATE TRIGGER trigger_orders  
    ON view_AllOrders INSTEAD OF INSERT  
    AS BEGIN
```



```

INSERT INTO Orders
SELECT OrderId, Price, Quantity, OrderDate
FROM inserted
END

```

В этом примере используется таблица Orders, содержащая два вычисляемых столбца. Представление view_AllOrders содержит все строки этой таблицы. Это представление используется для задания значения в его столбце, которое соотносится с вычисляемым столбцом в базовой таблице, на которой создано представление. Это позволяет использовать триггер INSTEAD OF, который в случае инструкции INSERT заменяется пакетом, который вставляет значения в базовую таблицу посредством представления view_AllOrders. (Инструкция INSERT, обращающаяся непосредственно к базовой таблице, не может задавать значение вычисляемому столбцу.)

Триггеры first и last

Компонент Database Engine позволяет создавать несколько триггеров для каждой таблицы или представления и для каждой операции (INSERT, UPDATE и DELETE) с ними. Кроме этого, можно указать порядок выполнения для нескольких триггеров, определенных для конкретной операции. С помощью системной процедуры **sp_settriggerorder** можно указать, что один из определенных для таблицы триггеров AFTER будет выполняться первым или последним для каждого обрабатываемого действия. Эта системная процедура имеет параметр @order, которому можно присвоить одно из трех значений:

- first - указывает, что триггер является первым триггером AFTER, выполняющимся для модифицирования действия;
- last - указывает, что данный триггер является последним триггером AFTER, выполняющимся для инициирования действия;
- none - указывает, что для триггера отсутствует какой-либо определенный порядок выполнения. (Это значение обычно используется для того, чтобы выполнить сброс ранее установленного порядка выполнения триггера как первого или последнего.)

Изменение структуры триггера посредством инструкции ALTER TRIGGER отменяет порядок выполнения триггера (первый или последний). Применение системной процедуры sp_settriggerorder показано в примере ниже:

```
USE SampleDb;
```

```
EXEC sp_settriggerorder @triggername = 'trigger_ModifyBudget',
    @order = 'first', @stmttype='update'
```

Для таблицы разрешается определить только один первый и только один последний триггер AFTER. Остальные триггеры AFTER выполняются в неопределенном порядке. Узнать порядок выполнения триггера можно с помощью системной процедуры **sp_helptrigger** или функции OBJECTPROPERTY.

Возвращаемый системной процедурой `sp_helptrigger` результирующий набор содержит столбец `order`, в котором указывается порядок выполнения указанного триггера. При вызове функции `objectproperty` в ее втором параметре указывается значение `ExecelsFirstTrigger` или `ExecelsLastTrigger`, а в первом параметре всегда указывается идентификационный номер объекта базы данных. Если указанное во втором параметре свойство имеет значение `true`, функция возвращает значение 1.

Поскольку триггер `INSTEAD OF` выполняется перед тем, как выполняются изменения в его таблице, для триггеров этого типа нельзя указать порядок выполнения "первым" или "последним".

Триггеры DDL и области их применения

Ранее мы рассмотрели триггеры DML, которые задают действие, предпринимаемое сервером при изменении таблицы инструкциями `INSERT`, `UPDATE` или `DELETE`. Компонент Database Engine также позволяет определять триггеры для инструкций DDL, таких как `CREATE DATABASE`, `DROP TABLE` и `ALTER TABLE`. Триггеры для инструкций DDL имеют следующий синтаксис:

```
CREATE TRIGGER [schema_name.]trigger_name  
ON {ALL SERVER | DATABASE }  
[WITH {ENCRYPTION | EXECUTE AS clause_name}  
{FOR | AFTER } { event_group | event_type | LOGON }  
AS {batch | EXTERNAL NAME method_name}
```

Как можно видеть по их синтаксису, триггеры DDL создаются таким же способом, как и триггеры DML. А для изменения и удаления этих триггеров используются те же инструкции `ALTER TRIGGER` и `DROP TRIGGER`, что и для триггеров DML. Поэтому в этом разделе рассматриваются только те параметры инструкции `CREATE TRIGGER`, которые новые для синтаксиса триггеров DDL.

Первым делом при определении триггера DDL нужно указать его область действия. *Предложение DATABASE* указывает в качестве области действия триггера DDL текущую базу данных, а *предложение ALL SERVER* - текущий сервер.

После указания области действия триггера DDL нужно в ответ на выполнение одной или нескольких инструкций DDL указать способ запуска триггера. В параметре `event_type` указывается инструкция DDL, выполнение которой запускает триггер, а в альтернативном параметре `event_group` указывается группа событий языка Transact-SQL. Триггер DDL запускается после выполнения любого события языка Transact-SQL, указанного в параметре `event_group`. Ключевое слово **LOGON** указывает триггер входа.

Кроме сходства триггеров DML и DDL, между ними также есть несколько различий. Основным различием между этими двумя видами триггеров является то, что для триггера DDL можно задать в качестве его области действия всю базу данных или даже весь сервер, а не всего лишь

отдельный объект. Кроме этого, триггеры DDL не поддерживают триггеров **INSTEAD OF**. Как вы, возможно, уже догадались, для триггеров DDL не требуются таблицы **inserted** и **deleted**, поскольку эти триггеры не изменяют содержимого таблиц.

В следующих подразделах подробно рассматриваются две формы триггеров DDL: триггеры уровня базы данных и триггеры уровня сервера.

Триггеры DDL уровня базы данных

В примере ниже показано, как можно реализовать триггер DDL, чья область действия распространяется на текущую базу данных:

```
USE SampleDb;
```

```
GO
```

```
CREATE TRIGGER trigger_PreventDrop  
ON DATABASE FOR DROP_TRIGGER
```

```
AS PRINT 'Перед тем, как удалить триггер, вы должны отключить  
"trigger_PreventDrop"'
```

```
ROLLBACK
```

Триггер в этом примере предотвращает удаление любого триггера для базы данных **SampleDb** любым пользователем. Предложение **DATABASE** указывает, что триггер **trigger_PreventDrop** является триггером уровня базы данных. Ключевое слово **DROP_TRIGGER** указывает predetermined тип события, запрещающий удаление любого триггера.

Триггеры DDL уровня сервера

Триггеры уровня сервера реагируют на серверные события. Триггер уровня сервера создается посредством использования предложения **ALL SERVER** в инструкции **CREATE TRIGGER**. В зависимости от выполняемого триггером действия, существует два разных типа триггеров уровня сервера: обычные триггеры DDL и триггеры входа. Запуск обычных триггеров DDL основан на событиях инструкций DDL, а запуск триггеров входа - на событиях входа.

В примере ниже демонстрируется создание триггера уровня сервера, который является триггером входа:

```
USE master;
```

```
GO
```

```
CREATE LOGIN loginTest WITH PASSWORD = '12345!'  
CHECK_EXPIRATION = ON;
```

```
GO
```

```
GRANT VIEW SERVER STATE TO loginTest;
```

```
GO
```

```
CREATE TRIGGER trigger_ConnectionLimit  
ON ALL SERVER WITH EXECUTE AS 'loginTest'  
FOR LOGON AS
```

```

BEGIN
    IF ORIGINAL_LOGIN()= 'loginTest' AND
        (SELECT COUNT(*) FROM sys.dm_exec_sessions
         WHERE is_user_process = 1 AND
              original_login_name = 'loginTest') > 1
        ROLLBACK;
END;

```

Здесь сначала создается имя входа SQL Server loginTest, которое потом используется в триггере уровня сервера. По этой причине, для этого имени входа требуется разрешение VIEW SERVER STATE, которое и предоставляется ему посредством инструкции GRANT. После этого создается триггер trigger_ConnectionLimit. Этот триггер является триггером входа, что указывается ключевым словом LOGON.

С помощью представления **sys.dm_exec_sessions** выполняется проверка, был ли уже установлен сеанс с использованием имени входа loginTest. Если сеанс уже был установлен, выполняется инструкция ROLLBACK. Таким образом имя входа loginTest может одновременно установить только один сеанс.

Лабораторная работа №9. Управление временем в реляционных базах данных

Цель работы: модификация одной из таблиц, добавляя атрибуты времени «с», «по».

При помощи добавления к полю ограничения DEFAULT мы можем задать значение по умолчанию, которое будет подставляться в случае, если при вставке новой записи данное поле не будет перечислено в списке полей команды INSERT. Данное ограничение можно задать непосредственно при создании таблицы.

Давайте добавим в таблице Employees новое поле «Дата приема» и назовем его HireDate и скажем, что значение по умолчанию у данного поля будет текущая дата:

```
ALTER TABLE Employees ADD HireDate date NOT NULL DEFAULT  
SYSDATETIME()
```

Или если столбец HireDate уже существует, то можно использовать следующий синтаксис:

```
ALTER TABLE Employees ADD DEFAULT SYSDATETIME() FOR  
HireDate
```

Или

```
ALTER TABLE Employees ADD CONSTRAINT  
DF_Employees_HireDate DEFAULT SYSDATETIME() FOR HireDate
```

Та как данного столбца раньше не было, то при его добавлении в каждую запись в поле HireDate будет вставлено текущее значение даты.

При добавлении новой записи, текущая дата так же будет вставлена автоматически, конечно если мы ее явно не зададим, т.е. не укажем в списке столбцов. Покажем это на примере, не указав поле HireDate в перечне добавляемых значений:

```
INSERT Employees(ID,Name,Email)VALUES(1004,N'Сергеев  
С.С.','s.sergeev@test.tt')
```

Посмотрим, что получилось:

```
SELECT * FROM Employees
```

Лабораторная работа №10. Пользовательские функции

Цели:

1. Изучить порядок создания пользовательских функций
2. Освоить применение пользовательских функций

Пользовательские функции очень похожи на хранимые процедуры. Так же в них можно передавать параметры и они выполняют некоторые действия, однако их главным отличием от хранимых процедур является то, что они выводят (возвращают) какой то результат. Более того, они вызываются только при помощи оператора SELECT, аналогично встроенным функциям. Все пользовательские функции делятся на 2 вида:

1. Скалярные функции - функции, которые возвращают число или текст, то есть одно или несколько значений;
2. Табличные функции - функции, которые выводят результат в виде таблицы.

В языках программирования обычно имеется два типа подпрограмм:

- хранимые процедуры;
- определяемые пользователем функции (UDF).

Как уже было рассмотрено, хранимые процедуры состоят из нескольких инструкций и имеют от нуля до нескольких входных параметров, но обычно не возвращают никаких параметров. В отличие от хранимых процедур, функции всегда возвращают одно значение. В этом разделе мы рассмотрим создание и использование *определяемых пользователем функций (User Defined Functions - UDF)*.

Создание и выполнение определяемых пользователем функций

Определяемые пользователем функции создаются посредством инструкции **CREATE FUNCTION**, которая имеет следующий синтаксис:

```
CREATE FUNCTION [schema_name.]function_name  
[( { @param } type [= default] ) {,...}  
    RETURNS {scalar_type | [ @variable] TABLE }  
    [WITH {ENCRYPTION | SCHEMABINDING }  
    [AS] {block | RETURN (select_statement)}
```

Параметр `schema_name` определяет имя схемы, которая назначается владельцем создаваемой UDF, а параметр `function_name` определяет имя этой функции. Параметр `@param` является входным параметром функции (формальным аргументом), чей тип данных определяется параметром `type`. Параметры функции - это значения, которые передаются вызывающим объектом определяемой пользователем функции для использования в ней. Параметр `default` определяет значение по умолчанию для соответствующего параметра функции. (Значением по умолчанию также может быть NULL.)

Предложение **RETURNS** определяет тип данных значения, возвращаемого UDF. Это может быть почти любой стандартный тип данных, поддерживаемый системой баз данных, включая тип данных **TABLE**.

Единственным типом данных, который нельзя указывать, является тип данных timestamp.

Определяемые пользователем функции могут быть либо скалярными, либо табличными. Скалярные функции возвращают атомарное (скалярное) значение. Это означает, что в предложении RETURNS скалярной функции указывается один из стандартных типов данных. Функция является табличной, если предложение RETURNS возвращает набор строк.

Параметр *WITH ENCRYPTION* в системном каталоге кодирует информацию, содержащую текст инструкции CREATE FUNCTION. Таким образом, предотвращается несанкционированный просмотр текста, который был использован для создания функции. Данная опция позволяет повысить безопасность системы баз данных.

Альтернативное предложение *WITH SCHEMABINDING* привязывает UDF к объектам базы данных, к которым эта функция обращается. После этого любая попытка модифицировать объект базы данных, к которому обращается функция, претерпевает неудачу. (Привязка функции к объектам базы данных, к которым она обращается, удаляется только при изменении функции, после чего параметр SCHEMABINDING больше не задан.)

Для того чтобы во время создания функции использовать предложение SCHEMABINDING, объекты базы данных, к которым обращается функция, должны удовлетворять следующим условиям:

- все представления и другие UDF, к которым обращается определяемая функция, должны быть привязаны к схеме;
- все объекты базы данных (таблицы, представления и UDF) должны быть в той же самой базе данных, что и определяемая функция.

Параметр block определяет блок BEGIN/END, содержащий реализацию функции. Последней инструкцией блока должна быть инструкция RETURN с аргументом. (Значением аргумента является возвращаемое функцией значение.) Внутри блока BEGIN/END разрешаются только следующие инструкции:

- инструкции присвоения, такие как SET;
- инструкции для управления ходом выполнения, такие как WHILE и IF;
- инструкции DECLARE, объявляющие локальные переменные;
- инструкции SELECT, содержащие списки столбцов выборки с выражениями, значения которых присваиваются переменным, являющимися локальными для данной функции;
- инструкции INSERT, UPDATE и DELETE, которые изменяют переменные с типом данных TABLE, являющиеся локальными для данной функции.

По умолчанию инструкцию CREATE FUNCTION могут использовать только члены предопределенной роли сервера sysadmin и предопределенной роли базы данных db_owner или db_ddladmin. Но члены этих ролей могут присвоить это право другим пользователям с помощью инструкции GRANT CREATE FUNCTION.

В примере ниже показано создание функции ComputeCosts:

```
USE SampleDb;
```

```
-- Эта функция вычисляет возникающие дополнительные общие  
затраты,
```

```
-- при увеличении бюджетов проектов
```

```
GO
```

```
CREATE FUNCTION ComputeCosts (@percent INT = 10)
```

```
RETURNS DECIMAL(16, 2)
```

```
BEGIN
```

```
DECLARE @addCosts DEC (14,2), @sumBudget DEC(16,2)
```

```
SELECT @sumBudget = SUM (Budget) FROM Project
```

```
SET @addCosts = @sumBudget * @percent/100
```

```
RETURN @addCosts
```

```
END;
```

Функция ComputeCosts вычисляет дополнительные расходы, возникающие при увеличении бюджетов проектов. Единственный входной параметр, @percent, определяет процентное значение увеличения бюджетов. В блоке BEGIN/END сначала объявляются две локальные переменные: @addCosts и @sumBudget, а затем с помощью инструкции SELECT переменной @sumBudget присваивается общая сумма всех бюджетов. После этого функция вычисляет общие дополнительные расходы и посредством инструкции RETURN возвращает это значение.

Вызов определяемой пользователем функции

Определенную пользователем функцию можно вызывать с помощью инструкций Transact-SQL, таких как SELECT, INSERT, UPDATE или DELETE. Вызов функции осуществляется, указывая ее имя с парой круглых скобок в конце, в которых можно задать один или несколько аргументов. Аргументы - это значения или выражения, которые передаются входным параметрам, определяемым сразу же после имени функции. При вызове функции, когда для ее параметров не определены значения по умолчанию, для всех этих параметров необходимо предоставить аргументы в том же самом порядке, в каком эти параметры определены в инструкции CREATE FUNCTION.

В примере ниже показан вызов функции ComputeCosts в инструкции SELECT:

```
USE SampleDb;
```

```
-- Вернет проект "p2 - Gemini"
```

```
SELECT Number, ProjectName
```

```
FROM Project
```

```
WHERE Budget < dbo.ComputeCosts(25);
```

Инструкция SELECT в примере отображает названия и номера всех проектов, бюджеты которых меньше, чем общие дополнительные расходы по всем проектам при заданном значении процентного увеличения.

В инструкциях Transact-SQL имена функций необходимо задавать, используя имена, состоящие из двух частей: schema name и function name, поэтому в примере мы использовали префикс схемы dbo.

Возвращающие табличное значение функции

Как уже упоминалось ранее, функция является возвращающей табличное значение, если ее предложение RETURNS возвращает набор строк. В зависимости от того, каким образом определено тело функции, возвращающие табличное значение функции классифицируются как *встраиваемые (inline)* и *многоинструкционные (multistatement)*. Если в предложении RETURNS ключевое слово TABLE указывается без сопровождающего списка столбцов, такая функция является встроенной. Инструкция SELECT встраиваемой функции возвращает результирующий набор в виде переменной с типом данных TABLE.

Многоинструкционная возвращающая табличное значение функция содержит имя, определяющее внутреннюю переменную с типом данных TABLE. Этот тип данных указывается **ключевым словом TABLE**, которое следует за именем переменной. В эту переменную вставляются выбранные строки, и она служит возвращаемым значением функции.

Создание возвращающей табличное значение функции показано в примере ниже:

```
USE SampleDb;
```

```
GO
```

```
CREATE FUNCTION EmployeesInProject (@projectNumber CHAR(4))
```

```
RETURNS TABLE
```

```
AS RETURN (SELECT FirstName, LastName
```

```
FROM Works_on, Employee
```

```
WHERE Employee.Id = Works_on.EmpId
```

```
AND ProjectNumber = @projectNumber)
```

Функция EmployeesInProject отображает имена всех сотрудников, работающих над определенным проектом, номер которого задается входным параметром @projectNumber. Тогда как функция в общем случае возвращает набор строк, предложение RETURNS в определении данной функции содержит ключевое слово TABLE, указывающее, что функция возвращает табличное значение. (Обратите внимание на то, что в примере блок BEGIN/END необходимо опустить, а предложение RETURN содержит инструкцию SELECT.)

Использование функции Employees_in_Project приведено в примере ниже:

```
USE SampleDb;
```

```
SELECT *
```

```
FROM EmployeesInProject('p3')
```

Результат выполнения:

Results		Messages
	FirstName	LastName
1	Анна	Иванова
2	Василий	Фролов
3	Елена	Лебедеенко

Возвращающие табличное значение функции и инструкция APPLY

Реляционная инструкция **APPLY** позволяет вызывать возвращающую табличное значение функцию для каждой строки табличного выражения. Эта инструкция задается в предложении FROM соответствующей инструкции SELECT таким же образом, как и инструкция JOIN. Инструкция APPLY может быть объединена с табличной функцией для получения результата, похожего на результирующий набор операции соединения двух таблиц. Существует две формы инструкции APPLY:

- CROSS APPLY
- OUTER APPLY

Инструкция *CROSS APPLY* возвращает те строки из внутреннего (левого) табличного выражения, которые совпадают с внешним (правым) табличным выражением. Таким образом, логически, инструкция CROSS APPLY функционирует так же, как и инструкция INNER JOIN.

Инструкция *OUTER APPLY* возвращает все строки из внутреннего (левого) табличного выражения. (Для тех строк, для которых нет совпадений во внешнем табличном выражении, он содержит значения NULL в столбцах внешнего табличного выражения.) Логически, инструкция OUTER APPLY эквивалентна инструкции LEFT OUTER JOIN.

Применение инструкции APPLY показано в примерах ниже:
USE SampleDb;

GO

-- Создать функцию

CREATE FUNCTION GetJob(@empid AS INT)

RETURNS TABLE AS

RETURN

SELECT Job

FROM Works_on

WHERE EmpId = @empid

AND Job IS NOT NULL

AND ProjectNumber = 'p1';

Функция GetJob() возвращает набор строк с таблицы Works_on. В примере ниже этот результирующий набор "соединяется" предложением APPLY с содержимым таблицы Employee:

USE SampleDb;

-- Используется CROSS APPLY

SELECT E.Id, FirstName, LastName, Job

```
FROM Employee as E
CROSS APPLY GetJob(E.Id) AS A
```

-- *Используется OUTER APPLY*

```
SELECT E.Id, FirstName, LastName, Job
FROM Employee as E
OUTER APPLY GetJob(E.Id) AS A
```

Результатом выполнения этих двух функций будут следующие две таблицы (отображаются после выполнения второй функции):

Results		Messages		
	Id	FirstNa...	LastName	Job
1	10102	Анна	Иванова	Аналитик
2	9031	Елена	Лебедеко	Менеджер
3	29346	Олег	Маменко	Консультант

	Id	FirstNa...	LastName	Job
1	2581	Василий	Фролов	NULL
2	9031	Елена	Лебедеко	Менеджер
3	10102	Анна	Иванова	Аналитик
4	18316	Игорь	Соловьев	NULL
5	25348	Дмитрий	Волков	NULL
6	28559	Наталья	Вершини...	NULL
7	29346	Олег	Маменко	Консульт...

В первом запросе примера результирующий набор табличной функции GetJob() "соединяется" с содержимым таблицы Employee посредством инструкции CROSS APPLY. Функция GetJob() играет роль правого ввода, а таблица Employee - левого. Выражение правого ввода вычисляется для каждой строки левого ввода, а полученные строки комбинируются, создавая конечный результат.

Второй запрос похожий на первый (но в нем используется инструкция OUTER APPLY), который логически соответствует операции внешнего соединения двух таблиц.

Возвращающие табличное значение параметры

Во всех версиях сервера, предшествующих SQL Server 2008, задача передачи подпрограмме множественных параметров была сопряжена со значительными сложностями. Для этого сначала нужно было создать временную таблицу, вставить в нее передаваемые значения, и только затем можно было вызывать подпрограмму. Начиная с версии SQL Server 2008, эта задача упрощена, благодаря возможности использования возвращающих табличное значение параметров, посредством которых результирующий набор может быть передан соответствующей подпрограмме.

Использование возвращающего табличное значение параметра показано в примере ниже:

```
USE SampleDb;
```

```
CREATE TYPE departmentType AS TABLE
```

```
(Number CHAR(4), DepartmentName CHAR(40), Location CHAR(40));  
GO
```

```
CREATE TABLE #moscowTable
```

```
(Number CHAR(4), DepartmentName CHAR(40), Location CHAR(40));  
GO
```

```
CREATE PROCEDURE InsertProc
```

```
@Moscow departmentType READONLY
```

```
AS SET NOCOUNT ON
```

```
INSERT INTO #moscowTable (Number, DepartmentName, Location)
```

```
SELECT * FROM @Moscow
```

```
GO
```

```
DECLARE @Moscow AS departmentType;
```

```
INSERT INTO @Moscow (Number, DepartmentName, Location)
```

```
SELECT * FROM department
```

```
WHERE location = 'Москва';
```

```
EXEC InsertProc @Moscow;
```

В этом примере сначала определяется табличный тип departmentType. Это означает, что данный тип является типом данных TABLE, вследствие чего он разрешает вставку строк. В процедуре InsertProc объявляется переменная @Moscow с типом данных departmentType. (Предложение READONLY указывает, что содержимое этой таблицы нельзя изменять.) В последующем пакете в эту табличную переменную вставляются данные, после чего процедура запускается на выполнение. В процессе исполнения процедура вставляет строки из табличной переменной во временную таблицу #moscowTable. Вставленное содержимое временной таблицы выглядит следующим образом:

Results		Messages	
	Number	DepartmentName	Location
1	d1	Исследования	Москва
2	d3	Маркетинг	Москва

Использование возвращающих табличное значение параметров предоставляет следующие преимущества:

- упрощается модель программирования подпрограмм;
- уменьшается количество обращений к серверу и получений соответствующих ответов;
- таблица результата может иметь произвольное количество строк.

Изменение структуры определяемых пользователями инструкций

Язык Transact-SQL также поддерживает инструкцию **ALTER FUNCTION**, которая модифицирует структуру определяемых

пользователями инструкций (UDF). Эта инструкция обычно используется для удаления привязки функции к схеме. Все параметры инструкции ALTER FUNCTION имеют такое же значение, как и одноименные параметры инструкции CREATE FUNCTION.

Для удаления UDF применяется **инструкция DROP FUNCTION**. Удалить функцию может только ее владелец или член предопределенной роли db_owner или sysadmin.

Задание для лабораторной работы №10.

Реализовать слой бизнес-логики на уровне хранимых процедур и функций СУБД (в соответствии с вариантом):

- не менее 1 скалярной функции;
- не менее 1 табличной функции;
- не менее, чем в 1 хранимой процедуре использовать разработанные функции.

Лабораторная работа №11 Транзакции

Как уже знаете, данные в базе данных обычно используются совместно многими прикладными пользовательскими программами (приложениями). Ситуация, когда несколько прикладных пользовательских программ одновременно выполняют операции чтения и записи одних и тех же данных, называется *одновременным конкурентным (параллельным) доступом (concurrency)*. Таким образом, каждая система управления базами данных должна обладать каким-либо типом механизма управления для решения проблем, возникающих вследствие одновременного конкурентного доступа.

В системе баз данных, которая может обслуживать большое число активных пользовательских приложений таким образом, чтобы эти приложения не мешали друг другу, возможен высокий уровень одновременного конкурентного доступа. И наоборот, система баз данных, в которой разные активные приложения мешают друг другу, поддерживает низкий уровень одновременного конкурентного доступа.

Здесь показано, как проблемы, связанные с одновременным конкурентным доступом, можно решить посредством транзакций. Здесь дается вводное представление о свойствах транзакций, называемых *свойствами ACID (Atomicity, Consistency, Isolation, Durability - атомарность, согласованность, изолированность, долговечность)*, обзор инструкций языка Transact-SQL, применяемых для работы с транзакциями, и введение в журналы транзакций.

Модели одновременного конкурентного доступа

Компонент Database Engine поддерживает две разные модели одновременного конкурентного доступа:

- пессимистический одновременный конкурентный доступ;
- оптимистический одновременный конкурентный доступ.

В модели *пессимистического одновременного конкурентного доступа* для предотвращения одновременного доступа к данным, которые используются другим процессом, применяются блокировки. Иными словами, система баз данных, использующая модель пессимистического одновременного конкурентного доступа, предполагает, что между двумя или большим количеством процессов в любое время может возникнуть конфликт и поэтому блокирует ресурсы (строку, страницу, таблицу), как только они потребуются в течение периода транзакции. Модель пессимистического одновременного конкурентного доступа устанавливает блокировку с обеспечением разделяемого доступа, иначе *немонопольную блокировку (shared lock)* на считываемые данные, чтобы никакой другой процесс не мог изменить эти данные. Кроме этого, механизм пессимистического одновременного конкурентного доступа устанавливает *монопольную блокировку (exclusive lock)* на изменяемые данные, чтобы никакой другой процесс не мог их считывать или модифицировать.

Работа *оптимистического одновременного конкурентного доступа* основана на предположении маловероятности изменения данных

одной транзакцией одновременно с другой. Компонент Database Engine применяет оптимистический одновременный конкурентный доступ, при котором сохраняются старые версии строк, и любой процесс при чтении данных использует ту версию строки, которая была активной, когда он начал чтение. Поэтому процесс может модифицировать данные без каких-либо ограничений, поскольку все другие процессы, которые считывают эти же данные, используют свою собственную сохраненную версию. Конфликтная ситуация возможна только при попытке двух операций записи использовать одни и те же данные. В таком случае система выдает ошибку, которая обрабатывается клиентским приложением.

Понятие оптимистического одновременного конкурентного доступа обычно определяется в более широком смысле. Работа управления оптимистического одновременного конкурентного доступа основана на предположении маловероятности конфликтов между несколькими пользователями, поэтому разрешается исполнение транзакций без установки блокировок. Только когда пользователь пытается изменить данные, выполняется проверка ресурсов, чтобы определить наличие конфликтов. Если таковые возникли, то приложение требуется перезапустить.

Использование транзакций

Транзакция задает последовательность инструкций языка Transact-SQL, применяемую программистами базы данных для объединения в один пакет операций чтения и записи для того, чтобы система базы данных могла обеспечить согласованность данных. Существует два типа транзакций:

- Неявная транзакция - задает любую отдельную инструкцию INSERT, UPDATE или DELETE как единицу транзакции.
- Явная транзакция - обычно это группа инструкций языка Transact-SQL, начало и конец которой обозначаются такими инструкциями, как BEGIN TRANSACTION, COMMIT и ROLLBACK.

Понятие транзакции лучше всего объяснить на примере. Допустим, в базе данных SampleDb сотруднику "Василий Фролов" требуется присвоить новый табельный номер. Этот номер нужно одновременно изменить в двух разных таблицах. В частности, требуется одновременно изменить строку в таблице Employee и соответствующие строки в таблице Works_on. Если обновить данные только в одной из этих таблиц, данные базы данных SampleDb будут несогласованны, поскольку значения первичного ключа в таблице Employee и соответствующие значения внешнего ключа в таблице Works_on не будут совпадать. Реализация этой транзакции посредством инструкций языка Transact-SQL показана в примере ниже:

USE SampleDb;

-- Начало транзакции

BEGIN TRANSACTION

UPDATE Employee

SET Id = 14568

WHERE Id = 10102

```
IF (@@error <> 0)
    -- Отменить транзакцию, если есть ошибки
    ROLLBACK
```

```
UPDATE Works_on
SET EmpId = 14568
WHERE EmpId = 10102
```

```
IF (@@error <> 0)
    ROLLBACK
-- Завершение транзакции
COMMIT
```

Согласованность данных, обрабатываемых в примере, можно обеспечить лишь в том случае, если выполнены обе инструкции UPDATE либо обе не выполнены. Успех выполнения каждой инструкции UPDATE проверяется посредством **глобальной переменной @@error**. В случае ошибки этой переменной присваивается отрицательное значение и выполняется откат всех выполненных на данный момент инструкций транзакции.

В следующем разделе мы познакомимся со свойствами транзакций ACID. Эти свойства обеспечивают согласованность данных, обрабатываемых прикладными программами.

Свойства транзакций

Транзакции обладают следующими свойствами, которые все вместе обозначаются сокращением ACID (Atomicity, Consistency, Isolation, Durability):

- атомарность (Atomicity);
- согласованность (Consistency);
- изолированность (Isolation);
- долговечность (Durability).

Свойство атомарности обеспечивает неделимость набора инструкций, который модифицирует данные в базе данных и является частью транзакции. Это означает, что или выполняются все изменения данных в транзакции, или в случае любой ошибки осуществляется откат всех выполненных изменений.

Свойство согласованности обеспечивает, что в результате выполнения транзакции база данных не будет содержать несогласованных данных. Иными словами, выполняемые транзакцией трансформации данных переводят базу данных из одного согласованного состояния в другое.

Свойство изолированности отделяет все параллельные транзакции друг от друга. Иными словами, активная транзакция не может видеть модификации данных в параллельной или незавершенной транзакции. Это означает, что для обеспечения изоляции для некоторых транзакций может потребоваться выполнить откат.

Свойство долговечности обеспечивает одно из наиболее важных требований баз данных: сохраняемость данных. Иными словами, эффект транзакции должен оставаться действенным даже в случае системной ошибки. По этой причине, если в процессе выполнения транзакции происходит системная ошибка, то осуществляется откат для всех выполненных инструкций этой транзакции.

Инструкции Transact-SQL и транзакции

Для работы с транзакциями язык Transact-SQL предоставляет некоторые инструкции. Инструкция **BEGIN TRANSACTION** запускает транзакцию. Синтаксис этой инструкции выглядит следующим образом:

```
BEGIN TRANSACTION [ {transaction_name | @trans_var }  
[WITH MARK ['description']]]
```

В параметре `transaction_name` указывается имя транзакции, которое можно использовать только в самой внешней паре вложенных инструкций **BEGIN TRANSACTION/COMMIT** или **BEGIN TRANSACTION/ROLLBACK**. В параметре `@trans_var` указывается имя определяемой пользователем переменной, содержащей действительное имя транзакции. *Параметр WITH MARK* указывает, что транзакция должна быть отмечена в журнале. Аргумент `description` - это строка, описывающая эту отметку. В случае использования параметра **WITH MARK** требуется указать имя транзакции.

Инструкция **BEGIN DISTRIBUTED TRANSACTION** запускает распределенную транзакцию, которая управляется Microsoft Distributed Transaction Coordinator (MS DTC - координатором распределенных транзакций Microsoft). Распределенная транзакция - это транзакция, которая используется на нескольких базах данных и на нескольких серверах. Поэтому для таких транзакций требуется координатор для согласования выполнения инструкций на всех вовлеченных серверах. Координатором распределенной транзакции является сервер, запустивший инструкцию **BEGIN DISTRIBUTED TRANSACTION**, и поэтому он и управляет выполнением распределенной транзакции.

Инструкция **COMMIT WORK** успешно завершает транзакцию, запущенную инструкцией **BEGIN TRANSACTION**. Это означает, что все выполненные транзакцией изменения фиксируются и сохраняются на диск. Инструкция **COMMIT WORK** является стандартной формой этой инструкции. Использовать предложение **WORK** не обязательно.

Язык Transact-SQL также поддерживает инструкцию **COMMIT TRANSACTION**, которая функционально равнозначна инструкции **COMMIT WORK**, с той разницей, что она принимает определяемое пользователем имя транзакции. Инструкция **COMMIT TRANSACTION** является расширением языка Transact-SQL, соответствующим стандарту SQL.

В противоположность инструкции **COMMIT WORK**, инструкция **ROLLBACK WORK** сообщает о неуспешном выполнении транзакции. Программисты используют эту инструкцию, когда они полагают, что база данных может оказаться в несогласованном состоянии. В таком

случае выполняется откат всех произведенных инструкциями транзакции изменений. Инструкция **ROLLBACK WORK** является стандартной формой этой инструкции. Использовать предложение **WORK** не обязательно. Язык **Transact-SQL** также поддерживает инструкцию **ROLLBACK TRANSACTION**, которая функционально равнозначна инструкции **ROLLBACK WORK**, с той разницей, что она принимает определяемое пользователем имя транзакции.

Инструкция **SAVE TRANSACTION** устанавливает точку сохранения внутри транзакции. Точка сохранения (savepoint) определяет заданную точку в транзакции, так что все последующие изменения данных могут быть отменены без отмены всей транзакции. (Для отмены всей транзакции применяется инструкция **ROLLBACK**.) Инструкция **SAVE TRANSACTION** в действительности не фиксирует никаких выполненных изменений данных. Она только создает метку для последующей инструкции **ROLLBACK**, имеющей такую же метку, как и данная инструкция **SAVE TRANSACTION**.

Использование инструкции **SAVE TRANSACTION** показано в примере ниже:

```
USE SampleDb;
```

```
BEGIN TRANSACTION;
```

```
    INSERT INTO Department (Number, DepartmentName)
```

```
        VALUES ('d4', 'Скидки');
```

```
    SAVE TRANSACTION a;
```

```
    INSERT INTO Department (Number, DepartmentName)
```

```
        VALUES ('d5', 'Исследование');
```

```
    SAVE TRANSACTION b;
```

```
    INSERT INTO Department (Number, DepartmentName)
```

```
        VALUES ('d6', 'Менеджмент');
```

```
    ROLLBACK TRANSACTION b;
```

```
    INSERT INTO Department (Number, DepartmentName)
```

```
        VALUES ('d7', 'Поддержка');
```

```
    ROLLBACK TRANSACTION a;
```

```
COMMIT TRANSACTION;
```

Единственной инструкцией, которая выполняется в этом примере, является первая инструкция **INSERT**. Для третьей инструкции **INSERT** выполняется откат с помощью инструкции **ROLLBACK TRANSACTION b**, а для двух других инструкций **INSERT** будет выполнен откат инструкцией **ROLLBACK TRANSACTION a**.

Инструкция **SAVE TRANSACTION** в сочетании с инструкцией **IF** или **WHILE** является полезной возможностью, позволяющей выполнять отдельные части всей транзакции. С другой стороны, использование этой инструкции противоречит принципу работы с базами данных, гласящему, что транзакция должна быть минимальной длины, поскольку длинные транзакции обычно уменьшают уровень доступности данных.

Как вы уже знаете, каждая инструкция Transact-SQL всегда явно или неявно принадлежит к транзакции. Для удовлетворения требований стандарта SQL компонент Database Engine предоставляет поддержку неявных транзакций. Когда сеанс работает в режиме неявных транзакций, выполняемые инструкции неявно выдают инструкции **BEGIN TRANSACTION**. Это означает, что для того чтобы начать неявную транзакцию, пользователю или разработчику не требуется ничего делать. Но каждую неявную транзакцию нужно или явно зафиксировать или явно отменить, используя инструкции **COMMIT** или **ROLLBACK** соответственно. Если транзакцию явно не зафиксировать, то все изменения, выполненные в ней, откатываются при отключении пользователя.

Для разрешения неявных транзакций параметру *implicit_transactions* оператора **SET** необходимо присвоить значение **ON**. Это установит режим неявных транзакций для текущего сеанса. Когда для соединения установлен режим неявных транзакций и соединение в данный момент не используется в транзакции, выполнение любой из следующих инструкций запускает транзакцию:

- **ALTER TABLE**;
- **FETCH**;
- **REVOKE**;
- **CREATE TABLE**;
- **GRANT**;
- **SELECT**;
- **DELETE**;
- **INSERT**;
- **TRUNCATE TABLE**;
- **DROPTABLE**;
- **OPEN**;
- **UPDATE**.

Иными словами, если имеется последовательность инструкций из предыдущего списка, то каждая из этих инструкций будет представлять транзакцию.

Начало явной транзакции помечается инструкцией **BEGIN TRANSACTION**, а окончание - инструкцией **COMMIT** или **ROLLBACK**. Явные транзакции можно вкладывать друг в друга. В таком случае, каждая пара инструкций **BEGIN TRANSACTION/COMMIT** или **BEGIN TRANSACTION/ROLLBACK** используется внутри каждой такой пары или большего количества вложенных транзакций. (Вложенные транзакции

обычно используются в хранимых процедурах, которые сами содержат транзакции и вызываются внутри другой транзакции.) Глобальная переменная @@trancount содержит число активных транзакций для текущего пользователя.

Инструкции BEGIN TRANSACTION, COMMIT и ROLLBACK могут использоваться с именем заданной транзакции. (Именованная инструкция ROLLBACK соответствует или именованной транзакции, или инструкции SAVE TRANSACTION с таким же именем.) Именованную транзакцию можно применять только в самой внешней паре вложенных инструкций BEGIN TRANSACTION/COMMIT или BEGIN TRANSACTION/ROLLBACK.

Журнал транзакций

Реляционные системы баз данных создают запись для каждого изменения, которые они выполняют в базе данных в процессе транзакции. Это требуется на случай ошибки при выполнении транзакции. В такой ситуации все выполненные инструкции транзакции необходимо отменить, осуществив для них откат. Как только система обнаруживает ошибку, она использует сохраненные записи, чтобы вернуть базу данных в согласованное состояние, в котором она была до начала выполнения транзакции.

Компонент Database Engine сохраняет все эти записи, в особенности значения до и после транзакции, в одном или более файлов, которые называются *журналами транзакций (transaction log)*. Для каждой базы данных ведется ее собственный журнал транзакций. Таким образом, если возникает необходимость отмены одной или нескольких операций изменения данных в таблицах текущей базы данных, компонент Database Engine использует записи в журнале транзакций, чтобы восстановить значения столбцов таблиц, которые существовали до начала транзакции.

Журнал транзакций применяется для отката или восстановления транзакции. Если в процессе выполнения транзакции еще до ее завершения возникает ошибка, то система использует все существующие в журнале транзакций исходные значения записей (которые называются *исходными образами записей (before image)*), чтобы выполнить откат всех изменений, выполненных после начала транзакции. Процесс, в котором исходные образы записей из журнала транзакций используются для отката всех изменений, называется *операцией отмены записей (undo activity)*.

В журналах транзакций также сохраняются преобразованные образы записей (*after image*). Преобразованные образы - это модифицированные значения, которые применяются для отмены отката всех изменений, выполненных после старта транзакции. Этот процесс называется *операцией повторного выполнения действий (redo activity)* и применяется при восстановлении базы данных.

Каждой записи в журнале транзакций присваивается однозначный идентификатор, называемый *порядковым номером журнала транзакции (log sequence number - LSN)*. Все записи журнала, являющиеся частью

определенной транзакции, связаны друг с другом, чтобы можно было найти все части этой транзакции для операции отмены или повтора.

Лабораторная работа №12. Блокировки

Одновременный конкурентный доступ может вызывать разные отрицательные эффекты, например чтение несуществующих данных или потерю модифицированных данных. Рассмотрим следующий практический пример, иллюстрирующий один из этих отрицательных эффектов, называемый грязным чтением. Пользователь U1 из отдела кадров получает извещение, что сотрудник "Василий Фролов" поменял место жительства. Он вносит соответствующее изменение в базу данных для данного сотрудника, но при просмотре другой информации об этом сотруднике он понимает, что изменил адрес не того человека. (В компании работают два сотрудника по имени Василий Фролов.) К счастью, приложение позволяет отменить это изменение одним нажатием кнопки. Он нажимает эту кнопку, уверенный в том, что данные после отмены операции изменения адреса уже не содержат никакой ошибки.

В то же самое время пользователь U2 в отделе проектирования обращается к данным второго сотрудника с именем Василий Фролов, чтобы отправить ему домой последнюю техническую документацию, поскольку этот служащий редко бывает в офисе. Однако пользователь U2 обратился к базе данных после того, как адрес этого второго сотрудника с именем Василий Фролов был ошибочно изменен, но до того, как он был исправлен. В результате письмо отправляется не тому адресату.

Чтобы предотвратить подобные проблемы в модели пессимистического одновременного конкурентного доступа, каждая система управления базами данных должна обладать механизмом для управления одновременным доступом к данным всеми пользователями. Для обеспечения согласованности данных в случае одновременного обращения к данным несколькими пользователями компонент Database Engine, подобно всем СУБД, применяет блокировки. Каждая прикладная программа блокирует требуемые ей данные, что гарантирует, что никакая другая программа не сможет модифицировать эти данные. Когда другая прикладная программа пытается получить доступ к заблокированным данным для их модификации, то система или завершает эту попытку ошибкой, или заставляет программу ожидать снятия блокировки.

Блокировка имеет несколько разных свойств: длительность блокировки, режим блокировки и гранулярность блокировки. *Длительность блокировки* - это период времени, в течение которого ресурс удерживает определенную блокировку. Длительность блокировки зависит, среди прочего, от режима блокировки и выбора уровня изоляции.

Режимы блокировки и уровень гранулярности блокировки рассматриваются в следующих двух разделах. Последующее обсуждение относится к модели пессимистического одновременного конкурентного доступа.

Режимы блокировки

Режимы блокировки определяют разные типы блокировок. Выбор определенного режима блокировки зависит от типа ресурса, который требуется заблокировать. Для блокировок ресурсов уровня строки и страницы применяются следующие три типа блокировок:

Разделяемая блокировка (shared lock)

Резервирует ресурс (страницу или строку) только для чтения. Другие процессы не могут изменять заблокированный таким образом ресурс, но, с другой стороны, несколько процессов могут одновременно накладывать разделяемую блокировку на один и тот же ресурс. Иными словами, чтение ресурса с разделяемой блокировкой могут одновременно выполнять несколько процессов.

Монопольная блокировка (exclusive lock)

Резервирует страницу или строку для монопольного использования одной транзакции. Блокировка этого типа применяется инструкциями DML (INSERT, UPDATE и DELETE), которые модифицируют ресурс. Монопольную блокировку нельзя установить, если на ресурс уже установлена разделяемая или монопольная блокировка другим процессом, т.е. на ресурс может быть установлена только одна монопольная блокировка. На ресурс (страницу или строку) с установленной монопольной блокировкой нельзя установить никакую другую блокировку.

Блокировка обновления (update lock)

Может быть установлена на ресурс только при отсутствии на нем другой блокировки обновления или монопольной блокировки. С другой стороны, этот тип блокировки можно устанавливать на объекты с установленной разделяемой блокировкой. В таком случае блокировка обновления накладывает на объект другую разделяемую блокировку. Если транзакция, которая модифицирует объект, подтверждается, и у объекта нет никаких других блокировок, блокировка обновления преобразовывается в монопольную блокировку. У объекта может быть только одна блокировка обновления.

Система баз данных автоматически выбирает соответствующий режим блокировки, в зависимости от типа операции (чтение или запись). Блокировка обновления применяется для предотвращения определенных распространенных типов взаимоблокировок.

Возможность совмещения разных типов блокировок приводится в таблице ниже:

Матрица совместимости разных типов блокировок

	Разделяемая	Обновления	Монопольная
Разделяемая	Да	Да	Нет

Обновления	Да	Нет	Нет
Монопольная	Нет	Нет	Нет

Эта таблица интерпретируется следующим образом: предположим транзакция T1 имеет блокировку, указанную в заголовке соответствующей строки таблицы, а транзакция T2 запрашивает блокировку, указанную в соответствующем заголовке столбца таблицы. Значение "Да" в ячейке на пересечении строки и столбца означает, что транзакция T2 может иметь запрашиваемый тип блокировки, а значение "Нет", что не может.

Компонент Database Engine также поддерживает и другие типы блокировок, такие как кратковременные блокировки (latch lock) и взаимоблокировки (spin lock).

На уровне таблицы существует пять разных типов блокировок:

- разделяемая (shared, S);
- монопольная (exclusive, X);
- разделяемая с намерением (intent shared, IS);
- монопольная с намерением (intent exclusive, IX);
- разделяемая с монопольным намерением (shared with intent exclusive, SIX).

Разделяемые и монопольные типы блокировок для таблицы соответствуют одноименным блокировкам для строк и страниц. Обычн**облокировка с намерением (intent lock)** означает, что транзакция намеревается заблокировать следующий нижележащий в иерархии объектов базы данных ресурс. Таким образом, блокировка с намерением помещается на уровне иерархии объектов, который выше того объекта, который этот процесс намеревается заблокировать. Это является действенным способом узнать, возможна ли подобная блокировка, а также устанавливается запрет другим процессам заблокировать более высокий уровень, прежде чем процесс может установить требуемую ему блокировку.

Возможность совмещения разных типов блокировок на уровне таблиц базы данных приведена в таблице ниже. Эта таблица интерпретируется точно таким же образом, как и предыдущая таблица.

Возможность совмещения разных типов блокировок

	S	X	IS	SIX	IX
S	Да	Нет	Да	Нет	Нет
X	Нет	Нет	Нет	Нет	Нет

IS	Да	Нет	Да	Да	Да
SIX	Нет	Нет	Да	Нет	Нет
IX	Нет	Нет	Да	Нет	Да

Гранулярность блокировки

Гранулярность блокировки определяет, какой ресурс блокируется в одной попытке блокировки. Компонент Database Engine может блокировать следующие ресурсы: строки, страницы, индексный ключ или диапазон индексных ключей, таблицы, экстенд, саму базу данных. Система выбирает требуемую гранулярность блокировки автоматически.

Строка является наименьшим ресурсом, который можно заблокировать. Блокировка уровня строки также включает как строки данных, так и элементы индексов. Блокировка на уровне строки означает, что блокируется только строка, к которой обращается приложение. Поэтому все другие строки данной таблицы остаются свободными и их могут использовать другие приложения. Компонент Database Engine также может заблокировать страницу, на которой находится подлежащая блокировке строка.

В кластеризованных таблицах страницы данных хранятся на уровне узлов (кластеризованной) индексной структуры, и поэтому для них вместо блокировки строк применяется блокировка с индексными ключами.

Блокироваться также могут единицы дискового пространства, которые называются экстендами и имеют размер 64 Кбайт. Экстенды блокируются автоматически, и когда растет таблица или индекс, то для них требуется выделять дополнительное дисковое пространство.

Гранулярность блокировки оказывает влияние на одновременный конкурентный доступ. В общем, чем выше уровень гранулярности, тем больше сокращается возможность совместного доступа к данным. Это означает, что блокировка уровня строк максимизирует одновременный конкурентный доступ, т.к. она блокирует всего лишь одну строку страницы, оставляя все другие строки доступными для других процессов. С другой стороны, низкий уровень блокировки увеличивает системные накладные расходы, поскольку для каждой отдельной строки требуется отдельная блокировка. Блокировка на уровне страниц и таблиц ограничивает уровень доступности данных, но также уменьшает системные накладные расходы.

Укрупнение блокировок

Если в процессе транзакции имеется большое количество блокировок одного уровня, то компонент Database Engine автоматически объединяет эти блокировки в одну уровня таблицы. Этот процесс преобразования большого числа блокировок уровня строки, страницы или индекса в одну блокировку уровня таблицы называется *укрупнением блокировок (lock escalation)*. Порогом укрупнения называется граница, на которой система баз данных

применяет укрупнение блокировок. Пороги укрупнения устанавливаются динамически системой и не требуют настройки. (В настоящее время пороговым значением укрупнения блокировок является 5000 блокировок.)

Основной проблемой, касающейся укрупнения блокировок, является то обстоятельство, что решение, когда осуществлять укрупнение, принимает сервер баз данных, и это решение может не быть оптимальным для приложений, имеющих различные требования. Механизм укрупнения блокировок можно модифицировать с помощью инструкции ALTER TABLE. Эта инструкция поддерживает параметр TABLE и имеет следующий синтаксис:

```
ALTER TABLE table_name  
SET ( LOCK_ESCALATION = { TABLE | AUTO | DISABLE } )
```

Параметр table является значением по умолчанию и задает укрупнение блокировок на уровне таблиц. Параметр auto позволяет компоненту Database Engine самому выбирать уровень гранулярности, который соответствует схеме таблицы. Наконец, параметр disable отключает укрупнение блокировок в большинстве случаев. (В некоторых случаях компоненту Database Engine требуется наложить блокировку на уровне таблиц, чтобы предохранить целостность данных.)

В примере ниже показана отмена укрупнения блокировок для таблицы Employee:

```
USE SampleDb;
```

```
ALTER TABLE Employee SET (LOCK_ESCALATION = DISABLE);
```

Настройка блокировок

Настройку блокировок можно осуществлять, используя *подсказки блокировок (locking hints)* или параметр LOCK_TIMEOUT инструкции SET. Эти возможности описываются в следующих разделах.

Подсказки блокировок (locking hints)

Подсказки блокировок задают тип блокировки, используемой компонентом Database Engine для блокировки табличных данных. Подсказки блокировки уровня таблиц применяются, когда требуется более точное управление типами блокировок, накладываемых на ресурс. (Подсказки блокировок перекрывают текущий уровень изоляции для сеанса.)

Все подсказки блокировок указываются в предложении FROM инструкции SELECT. Далее приводится список и краткое описание доступных подсказок блокировок:

UPDLOCK

Устанавливается блокировка обновления для каждой строки таблицы при операции чтения. Все блокировки обновления удерживаются до окончания транзакции.

TABLOCK

Устанавливается разделяемая (или монопольная) блокировка для таблицы. Все блокировки удерживаются до окончания транзакции.

ROWLOCK

Существующая разделяемая блокировка таблицы заменяется разделяемой блокировкой строк для каждой отвечающей требованиям строки таблицы.

PAGLOCK

Разделяемая блокировка таблицы заменяется разделяемой блокировкой страницы для каждой страницы, содержащей указанные строки.

NOLOCK

Синоним для READUNCOMMITTED, который мы рассмотрим при обсуждении уровней изоляции.

HOLDLOCK

Синоним для REPEATABLEREAD.

XLOCK

Устанавливается монопольная блокировка, удерживаемая до завершения транзакции. Если подсказка xlock указывается с подсказкой rowlock, paglock или tablock, монопольные блокировки устанавливаются на соответствующем уровне гранулярности.

READPAST

Указывает, что компонент Database Engine не должен считывать строки, заблокированные другими транзакциями.

Все эти параметры можно объединять вместе в любом имеющем смысл порядке. Например, комбинация подсказок TABLOCK с PAGLOCK не имеет смысла, поскольку каждая из них применяется для разных ресурсов.

Параметр LOCK_TIMEOUT

Чтобы процесс не ожидал освобождения блокируемого объекта до бесконечности, можно в инструкции SET использовать параметр LOCK_TIMEOUT. Этот параметр задает период в миллисекундах, в течение которого транзакция будет ожидать снятия блокировки с объекта. Например, если вы хотите чтобы период ожидания был равен восемь секунд, то это следует указать следующим образом:

SET LOCK_TIMEOUT 8000

Если данный ресурс не может быть предоставлен процессу в течение этого периода времени, инструкция завершается аварийно и выдается соответствующее сообщение об ошибке. Значение LOCK_TIMEOUT равно -1 (значение по умолчанию) указывает отсутствие периода ожидания, т.е. транзакция не будет ожидать освобождения ресурса совсем. (Подсказка блокировки READPAST предоставляет альтернативу параметру LOCK_TIMEOUT.)

Отображение информации о блокировках

Наиболее важным средством для отображения информации о блокировках является динамическое административное представление **sys.dm_tran_locks**. Это представление возвращает информацию о текущих активных ресурсах диспетчера блокировок. Каждая строка представления отображает активный в настоящий момент запрос на блокировку, которая была предоставлена или предоставление которой

ождается. Столбцы представления соответствуют двум группам: ресурсам и запросам. Группа ресурсов описывает ресурсы, на блокировку которых делается запрос, а группа запросов описывает запрос блокировки. Наиболее важными столбцами этого представления являются следующие:

- resource_type - указывает тип ресурса;
- resource_database_id - задает идентификатор базы данных, к которой принадлежит данный ресурс;
- request_mode - задает режим запроса;
- request_status - задает текущее состояние запроса.

В примере ниже показан запрос, использующий представление sys.dm_tran_locks для отображения блокировок в состоянии ожидания:

```
USE SampleDb;
```

```
SELECT resource_type, DB_NAME(resource_database_id) as db_name,  
       request_session_id, request_mode, request_status  
FROM sys.dm_tran_locks  
WHERE request_status = 'WAIT';
```

Взаимоблокировки

Взаимоблокировка (deadlock) - это особая проблема одновременного конкурентного доступа, в которой две транзакции блокируют друг друга. В частности, первая транзакция блокирует объект базы данных, доступ к которому хочет получить другая транзакция, и наоборот. (В общем, взаимоблокировка может быть вызвана несколькими транзакциями, которые создают цикл зависимостей.) В примере ниже показана взаимоблокировка двумя транзакциями. (При использовании базы данных небольшого размера, одновременный конкурентный доступ процессов нельзя получить естественным образом, вследствие очень быстрого выполнения каждой транзакции. Поэтому в примере ниже используется инструкция WAITFOR, чтобы приостановить обе транзакции на десять секунд, чтобы эмулировать взаимоблокировку.)

```
USE SampleDb;
```

```
BEGIN TRANSACTION  
UPDATE Works_on  
SET Job = 'Менеджер'  
WHERE EmpId = 25348  
AND ProjectNumber = 'p2'
```

```
WAITFOR DELAY '00:00:10'
```

```
UPDATE Employee  
SET LastName = 'Фролова'  
WHERE LastName = 'Вершинина'
```

```
= 'p2'
```

```
COMMIT
```

```
BEGIN TRANSACTION  
UPDATE Employee  
SET DepartamentNumber = 'd2'  
WHERE Id = 28559
```

```
WAITFOR DELAY '00:00:10'
```

```
DELETE FROM Works_on  
WHERE EmpId = 25348  
AND ProjectNumber
```

```
COMMIT
```

Если обе транзакции в примере выше будут выполняться в одно и то же время, то возникнет взаимоблокировка и система возвратит следующее сообщение об ошибке:

Сообщение 1205, уровень 13, состояние 45

Транзакция (процесс с идентификатором 56) находится во взаимной блокировке с другим процессом и выбрана в качестве потерпевшей взаимоблокировки. Повторите выполнение команды.)

Как можно видеть по результатам выполнения примера, система баз данных обрабатывает взаимоблокировку, выбирая одну из транзакций (на самом деле, транзакцию, которая замыкает цикл в запросах блокировки) в качестве "жертвы" и выполняя ее откат. После этого выполняется другая транзакция. На уровне прикладной программы взаимоблокировку можно обрабатывать посредством реализации условной инструкции, которая выполняет проверку на возврат номера ошибки (1205), а затем снова выполняет инструкцию, для которой был выполнен откат.

Вы можете повлиять на то, какая транзакция будет выбрана системой в качестве "жертвы" взаимоблокировки, присвоив в инструкции SET параметру *DEADLOCK_PRIORITY* один из 21 (от -10 до 10) разных уровней приоритета взаимоблокировки. Константа LOW соответствует значению -5, NORMAL (значение по умолчанию) - значению 0, а константа HIGH - значению 5. Сеанс "жертва" выбирается в соответствии с приоритетом взаимоблокировки сеанса.

Лабораторная работа №13. Администрирование баз данных

Аутентификация заключается в предоставлении ответа на следующий вопрос: "Имеет ли данный пользователь законное право на доступ к системе?" Таким образом, данная концепция безопасности определяет процесс проверки подлинности учетных данных пользователя, чтобы не допустить использование системы несанкционированными пользователями. Аутентификацию можно реализовать путем запроса, требуя, чтобы пользователь предоставил, например, следующее:

- нечто, что известно пользователю (обычно пароль);
- нечто, что принадлежит пользователю, например, идентификационную карту;
- физические характеристики пользователя, например, подпись или отпечатки пальцев.

Наиболее применяемый способ подтверждения аутентификации реализуется посредством использования имени пользователя и пароля. Система проверяет достоверность этой информации, чтобы решить, имеет ли данный пользователь законное право на доступ к системе или нет. Этот процесс может быть усилен применением шифрования.

Шифрование данных представляет собой процесс кодирования информации таким образом, что она становится непонятной, пока не будет расшифрована пользователем.

Аутентификация

Система безопасности компонента Database Engine состоит из двух разных подсистем безопасности:

- системы безопасности Windows;
- системы безопасности SQL Server.

Система безопасности Windows определяет безопасность на уровне операционной системы, т.е. метод, посредством которого пользователи входят в систему Windows, используя свои учетные записи Windows. (Аутентификация посредством этой подсистемы также называется аутентификацией Windows.)

Система безопасности SQL Server определяет дополнительную безопасность, требуемую на уровне системы баз данных, т.е. способ, посредством которого пользователи, уже вошедшие в операционную систему, могут подключаться к серверу базы данных. Система безопасности SQL Server определяет регистрационное имя входа (или просто называемое логином) в SQL Server, которое создается в системе и ассоциируется с определенным паролем. Некоторые регистрационные имена входа в SQL Server идентичны существующим учетным записям Windows. (Аутентификация посредством этой подсистемы также называется аутентификацией SQL Server.)

На основе этих двух подсистем безопасности компонент Database Engine может работать в одном из следующих режимов аутентификации: в режиме Windows и в смешанном режиме.

Режим Windows требует, чтобы пользователи входили в систему баз данных исключительно посредством своих учетных записей Windows. Система принимает данные учетной записи пользователя, полагая, что они уже были проверены и одобрены на уровне операционной системы. Такой способ подключения к системе баз данных называется доверительным соединением (trusted connection), т.к. система баз данных доверяет, что операционная система уже проверила подлинность учетной записи и соответствующего пароля.

Смешанный режим позволяет пользователям подключаться к компоненту Database Engine посредством аутентификации Windows или аутентификации SQL Server. Это означает, что некоторые учетные записи пользователей можно настроить для использования подсистемы безопасности Windows, а другие, вдобавок к этому, могут использовать также и подсистему безопасности SQL Server.

Аутентификация SQL Server предоставляется исключительно в целях обратной совместимости. Поэтому зачастую следует использовать аутентификацию Windows.

Реализация режима аутентификации

Выбор одного из доступных режимов аутентификации осуществляется посредством среды SQL Server Management Studio. Чтобы установить режим аутентификации Windows, щелкните правой кнопкой сервер баз данных в окне Object Explorer и в контекстном меню выберите пункт Properties. Откроется диалоговое окно Server Properties, в котором нужно выбрать страницу Security, а на ней Windows Authentication Mode. Для выбора смешанного режима в этом же диалоговом окне Server Properties вам нужно выбрать Server and Windows Authentication Mode.

После успешного подключения пользователя к компоненту Database Engine его доступ к объектам базы данных становится независимым от использованного способа аутентификации для входа в базу данных - аутентификации Window или SQL Server аутентификации.

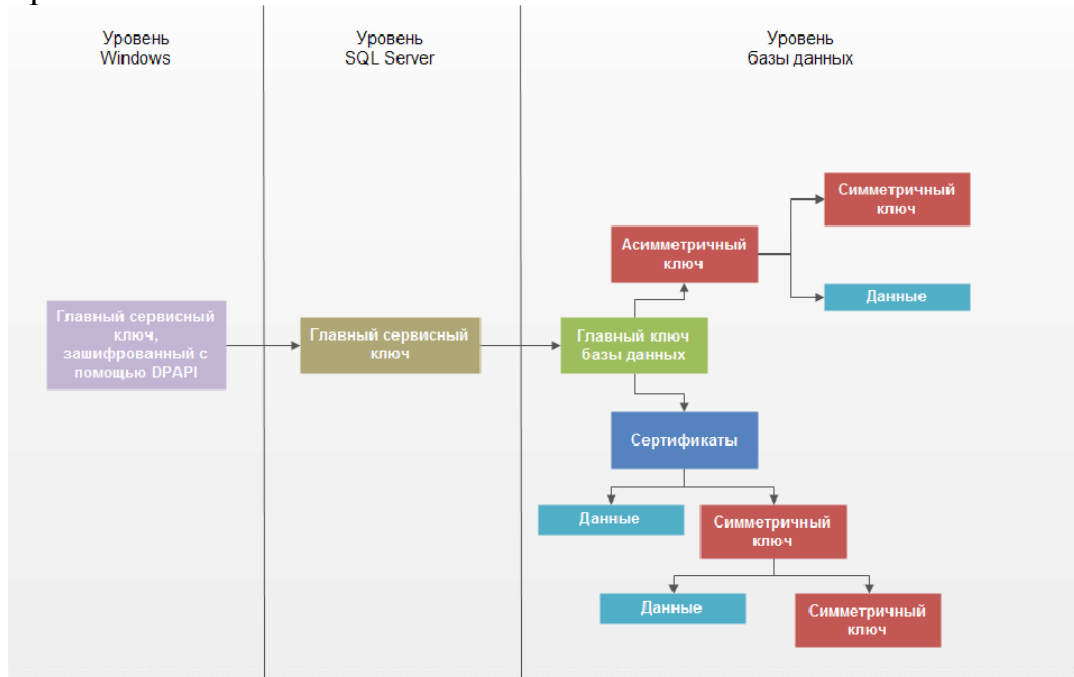
Прежде чем приступить к изучению настройки безопасности сервера базы данных, вам необходимо понять соглашения и механизмы шифрования, которые рассмотрены в следующих разделах.

Шифрование данных

Шифрование - это процесс приведения данных в запутанное непонятное состояние, вследствие чего повышается уровень их безопасности. Обычно конкретная процедура шифрования осуществляется с использованием определенного алгоритма. Наиболее важный алгоритм шифрования называется **RSA**, по первым буквам фамилий его создателей - Rivers, Shamir и Adelman.

Компонент Database Engine обеспечивает безопасность данных посредством иерархии уровней шифрования и инфраструктуры управления

ключами. Каждый уровень защищает следующий за ним уровень шифрования, используя комбинацию сертификатов, асимметричных и симметричных ключей:



На рисунке выше главный сервисный ключ задает ключ, который управляет всеми другими ключами и сертификатами. Главный сервисный ключ создается автоматически при установке компонента Database Engine. Этот ключ зашифрован с помощью *API-интерфейса защиты данных Windows (DPAPI - Data Protection API)*.

Важным свойством главного сервисного ключа является то, что он управляется системой. Хотя системный администратор может выполнять разные задачи по обслуживанию ключа, ему следует выполнять лишь одну из них - резервное копирование главного сервисного ключа, чтобы его можно было восстановить в случае повреждения.

Как можно видеть на рисунке, главный сервисный ключ базы данных является корневым объектом шифрования на уровне базы данных для всех ключей, сертификатов и данных. Каждая база данных имеет один главный ключ базы данных, который создается посредством инструкции **CREATE MASTER KEY**. Поскольку главный ключ базы данных защищен главным сервисным ключом системы, то система может автоматически расшифровывать главный ключ базы данных.

Существующий главный ключ базы данных можно использовать для создания пользовательских ключей. Существует три формы пользовательских ключей:

- симметричные ключи;
- асимметричные ключи;
- сертификаты.

Эти формы пользовательских ключей рассмотрены в следующих далее подразделах.

Симметричные ключи

В системе шифрования с использованием симметричного ключа оба участника обмена - отправитель и получатель сообщения - применяют один и тот же ключ. Иными словами, для шифрования информации и ее обратного расшифровывания используется этот единственный ключ.

Использование симметричных ключей имеет много преимуществ и один недостаток. Одним из преимуществ использования симметричных ключей является то обстоятельство, что с их помощью можно защитить значительно больший объем данных, чем с помощью двух других типов ключей. Кроме этого, использование ключей этого типа намного быстрее, чем использование несимметричных ключей.

Но с другой стороны, использование симметричного ключа в среде распределенной системы может сделать задачу обеспечения целостности шифрования почти невыполнимой, поскольку один и тот же ключ применяется на обоих концах обмена данными. Таким образом, основной рекомендацией является то, что симметричные ключи должны использоваться только с теми приложениями, в которых зашифрованные данные сохраняются в одном месте.

Язык Transact-SQL поддерживает несколько инструкций и системных функций применительно к симметричным ключам. В частности, для создания симметричного ключа применяется инструкция **CREATE SYMMETRIC KEY**, а для удаления существующего симметричного ключа - инструкция **DROP SYMMETRIC KEY**. Прежде чем симметричный ключ можно использовать для шифрования данных или для защиты другого ключа, его нужно открыть. Для этого используется инструкция **OPEN SYMMETRIC KEY**.

После того как вы откроете симметричный ключ, вам нужно для шифрования использовать системную функцию **EncryptByKey**. Эта функция имеет два входных параметра: идентификатор ключа и текст, который требуется зашифровать. Для расшифровки зашифрованной информации применяется системная функция **DecryptByKey**.

Асимметричные ключи

В случае если у вас имеется распределенная система или если использование симметричных ключей не обеспечивает достаточного уровня безопасности данных, то следует использовать асимметричные ключи. *Асимметричный ключ* состоит из двух частей: личного закрытого ключа (private key) и соответствующего общего открытого ключа (public key). Каждый из этих ключей может расшифровывать данные, зашифрованные другим ключом. Благодаря наличию личного закрытого ключа асимметричное шифрование обеспечивает более высокий уровень безопасности данных, чем симметричное шифрование.

Язык Transact-SQL поддерживает несколько инструкций и системных функций применительно к асимметричным ключам. В частности, для создания нового асимметричного ключа применяется инструкция **CREATE ASYMMETRIC KEY**, а для изменения свойств асимметричного ключа используется инструкция **ALTER ASYMMETRIC KEY**. Для удаления

асимметричного ключа применяется инструкция **DROP ASYMMETRIC KEY**.

После того как вы создали асимметричный ключ, для шифрования данных используйте системную функцию **EncryptByAsymKey**. Эта функция имеет два входных параметра: идентификатор ключа и текст, который требуется зашифровать. Для расшифровки информации, зашифрованной с использованием асимметричного ключа, применяется системная функция **DecryptByAsymKey**.

Сертификаты

Сертификат открытого ключа, или просто сертификат, представляет собой предложение с цифровой подписью, которое привязывает значение открытого ключа к определенному лицу, устройству или службе, которая владеет соответствующим открытым ключом. Сертификаты выдаются и подписываются *центром сертификации (Certification Authority - CA)*. Сущность, которая получает сертификат из центра сертификации (CA), является субъектом данного сертификата (certificate subject).

Между сертификатами и асимметричными ключами нет значительной функциональной разницы. Как первый, так и второй используют алгоритм RSA. Основная разница между ними заключается в том, что асимметричные ключи создаются вне сервера.

Сертификаты содержат следующую информацию:

- значение открытого ключа субъекта;
- информацию, идентифицирующую субъект;
- информацию, идентифицирующую издателя сертификата;
- цифровую подпись издателя сертификата.

Основным достоинством сертификатов является то, что они освобождают хосты от необходимости содержать набор паролей для отдельных субъектов. Когда хост, например, безопасный веб-сервер, обозначает издателя как надежный центр авторизации, этот хост неявно доверяет, что данный издатель выполнил проверку личности субъекта сертификата.

Сертификаты предоставляют самый высший уровень шифрования в модели безопасности компонента Database Engine. Алгоритмы шифрования с использованием сертификатов требуют большого объема процессорных ресурсов. По этой причине сертификаты следует использовать при реальной необходимости.

Наиболее важной инструкцией применительно к сертификатам является **инструкция CREATE CERTIFICATE**. Использование этой инструкции показано в примере ниже:

USE SampleDb;

CREATE MASTER KEY

ENCRYPTION BY PASSWORD = '12345!'

GO

CREATE CERTIFICATE cert01

WITH SUBJECT = 'Сертификат для схемы dbo';

Чтобы создать сертификат без параметра **ENCRYPTION BY**, сначала нужно создать главный ключ базы данных. (Все инструкции **CREATE CERTIFICATE**, которые не содержат этот параметр, защищаются главным ключом базы данных.) По этой причине первой инструкцией в примере выше является инструкция **CREATE MASTER KEY**. После этого инструкция **CREATE CERTIFICATE** используется для создания нового сертификата cert01, владельцем которого является объект dbo базы данных SampleDb, если этот объект является текущим пользователем.

Редактирование пользовательских ключей

Наиболее важными представлениями каталога применительно к шифрованию являются следующие:

- **sys.symmetric_keys**
- **sys.asymmetric_keys**
- **sys.certificates**
- **sys.database_principals**

Первые три представления каталога предоставляют информацию обо всех симметричных ключах, всех асимметричных ключах и всех сертификатах, установленных в текущей базе данных, соответственно. Представление каталога **sys.database_principals** предоставляет информацию обо всех принципах в текущей базе данных. ***Принципалы (principals)*** - это субъекты, которые имеют разрешение на доступ к определенной сущности. Типичными принципалами являются учетные записи Windows и SQL Server. Кроме этих принципалов, также существуют группы Windows и роли SQL Server. Группа Windows - это коллекция учетных записей и групп Windows. Присвоение учетной записи пользователя членство в группе дает этому пользователю все разрешения, предоставленные данной группе. Подобным образом роль является коллекцией учетных записей.

Представление каталога **sys.database_principals** можно соединить с любым из первых трех, чтобы получить информацию о владельце определенного ключа. В примере ниже показано получение информации о существующих сертификатах. Подобным образом можно получить информацию о симметричных и асимметричных ключах.

USE SampleDb;

SELECT p.name, c.name, certificate_id

FROM sys.certificates c, sys.database_principals p

WHERE p.principal_id = c.principal_id

Часть результата этого запроса:

Results		Messages	
	name	name	certificate...
1	public	cert01	256
2	dbo	cert01	256
3	guest	cert01	256
4	INFORMATION_SCHEMA	cert01	256
5	sys	cert01	256
6	db_owner	cert01	256
7	db_accessadmin	cert01	256
8	db_securityadmin	cert01	256
9	db_ddladmin	cert01	256
10	db_backupoperator	cert01	256
11	db_datareader	cert01	256
12	db_datawriter	cert01	256
13	db_denydatareader	cert01	256
14	db_denydatawriter	cert01	256

Расширенное управление ключами SQL Server

Следующим шагом к обеспечению более высокого уровня безопасности ключей является использование *расширенного управления ключами (Extensible Key Management - EKM)*. Основными целями расширенного управления ключами являются следующие:

- повышение безопасности ключей посредством выбора поставщика функций шифрования;
- общее управление ключами по всему приложению.

Расширенное управление ключами позволяет сторонним разработчикам регистрировать свои устройства в компоненте Database Engine. Когда такие устройства зарегистрированы, регистрационные имена (logins) в SQL Server могут использовать хранящиеся в них ключи шифрования, а также эффективно использовать продвинутые возможности шифрования, поддерживаемые этими модулями. Расширенное управление ключами позволяет защитить данные от доступа администраторов базы данных (за исключением членов группы sysadmin). Таким образом система защищается от пользователей с повышенными привилегиями. Данные можно зашифровывать и расшифровывать, используя инструкции шифрования языка Transact-SQL, а SQL Server может использовать внешнее устройство расширенного управления ключами для хранения ключей.

Способы шифрования данных

SQL Server поддерживает два способа шифрования данных:

- шифрование на уровне столбцов;
- прозрачное шифрование данных.

Шифрование на уровне столбцов позволяет шифровать конкретные столбцы данных. Для реализации этого способа шифрования используется несколько пар сопряженных функций. Далее мы не будем рассматривать этот метод шифрования, поскольку его реализация является сложным ручным процессом, требующим внесения изменений в приложение.

Прозрачное шифрование данных является новой возможностью базы данных, которая зашифровывает файлы базы данных автоматически, не

требуя внесения изменений в какие-либо приложения. Таким образом можно предотвратить доступ к информации базы данных несанкционированным лицам, даже если они смогут получить в свое распоряжение основные или резервные файлы базы данных.

Файлы базы данных зашифровываются на уровне страниц. Страницы зашифрованной базы данных шифруются перед тем, как они записываются на диски и расшифровываются при считывании с диска в память.

Как и большинство других методов шифрования, прозрачное шифрование данных основано на ключе шифрования. В нем используется симметричный ключ, посредством которого и защищается база данных.

Применения прозрачного шифрования для защиты определенной базы данных реализуется в четыре этапа:

1. Используя инструкцию **CREATE MASTER KEY**, создается главный ключ базы данных.
2. С помощью инструкции **CREATE CERTIFICATE** создается сертификат.
3. Используя инструкцию **CREATE DATABASE ENCRYPTION KEY**, создается ключ шифрования.
4. Выполняется конфигурирование базы данных для использования шифрования. (Этот шаг можно реализовать, присвоив параметру **ENCRYPTION** инструкции **ALTER DATABASE** значение **ON**.)

Настройка безопасности компонента Database Engine

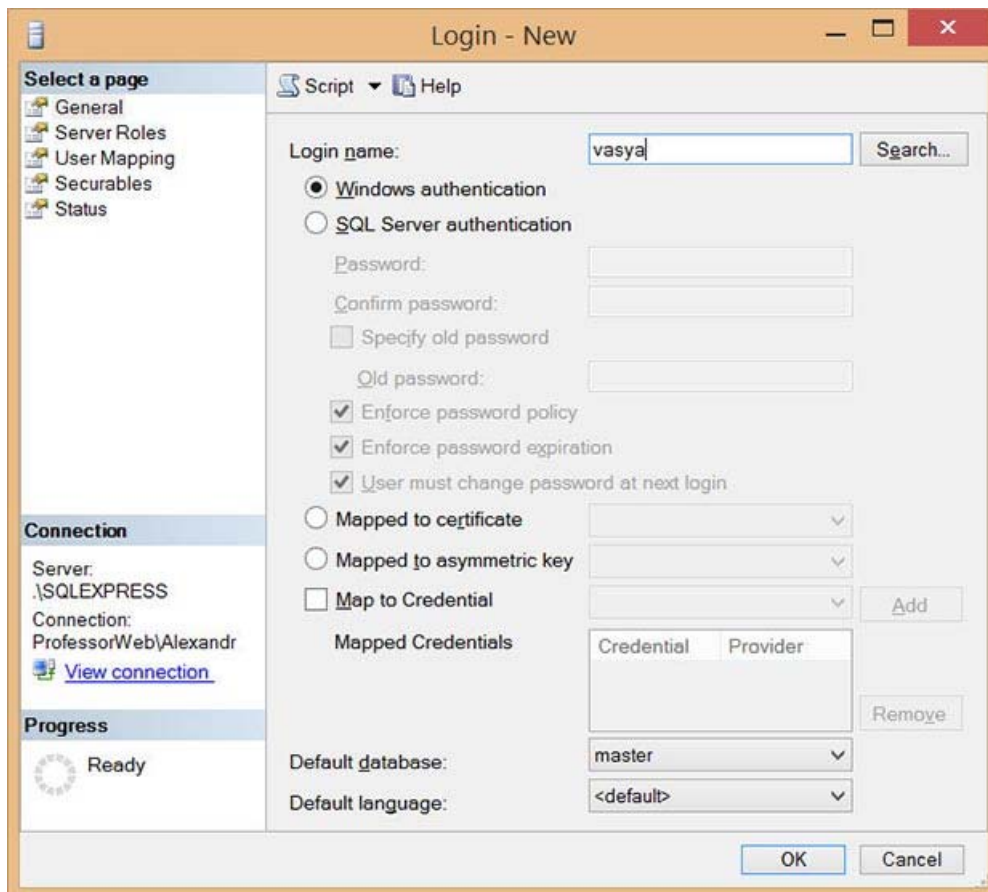
Настройку безопасности компонента Database Engine можно выполнить одним из следующих способов:

- с помощью среды управления Management Studio сервера SQL Server;
- используя инструкции языка Transact-SQL.

Эти два метода рассматриваются в последующих подразделах.

Управление безопасностью с помощью среды Management Studio

Чтобы с помощью среды Management Studio создать новое регистрационное имя, разверните в обозревателе объектов Object Explorer узел сервера, затем разверните папку "Security", в этой папке щелкните правой кнопкой папку "Logins" и в контекстном меню выберите опцию **New Login**. Откроется диалоговое окно **Login - New**:



Первым делом нужно решить, какой способ аутентификации применять: Windows или SQL Server. В случае выбора аутентификации Windows, в качестве регистрационного имени (Login name) необходимо указать действительное имя пользователя Windows в форме domain\user_name (домен\имя_пользователя). А если выбрана аутентификация SQL Server, необходимо ввести новое регистрационное имя (Login name) и соответствующий пароль (Password). Можно также указать базу данных и язык по умолчанию. База данных по умолчанию - это база данных, к которой пользователь автоматически подключается сразу же после входа в компонент Database Engine. Выполнив все эти действия, пользователь может входить в систему под этой новой учетной записью.

Управление безопасностью посредством инструкций Transact-SQL

Для управления безопасностью компонента Database Engine применяются три инструкции языка Transact-SQL: CREATE LOGIN, ALTER LOGIN и DROP LOGIN. Инструкция **CREATE LOGIN** создает новое регистрационное имя входа в SQL Server. Синтаксис этой инструкции следующий:

```
CREATE LOGIN login_name
{ WITH option_list1 |
FROM {WINDOWS [ WITH option_list2 [...]]
| CERTIFICATE certname | ASYMMETRIC KEY key_name }}
```

В параметре login_name указывается создаваемое регистрационное имя. Как можно видеть в синтаксисе этой инструкции, в предложении WITH

можно указать один или несколько параметров для регистрационного имени или указать в предложении FROM сертификат, асимметричный ключ или учетную запись пользователя Windows, связанную с соответствующим регистрационным именем.

В списке option_list1 указывается несколько параметров, наиболее важным из которых является параметр password, который задает пароль для данного регистрационного имени. (Другие возможные параметры - DEFAULT_DATABASE, DEFAULT_LANGUAGE и CHECK_EXPIRATION.)

Как видно из синтаксиса инструкции CREATE LOGIN, предложение FROM может содержать один из следующих параметров:

Параметр WINDOWS

Указывает, что данное регистрационное имя соотносится с существующей учетной записью пользователя Window. Этот параметр можно указать с другими подпараметрами, такими как default_database и default_language.

Параметр CERTIFICATE

Задаёт имя сертификата для привязки к данному регистрационному имени.

Параметр ASYMMETRIC KEY

Задаёт имя асимметричного ключа для привязки к данному регистрационному имени. (Сертификат и асимметричный ключ уже должны присутствовать в базе данных master.)

В примерах ниже показано создание разных форм регистрационного имени. В следующем примере создается регистрационное имя "Vasya" с паролем "12345!":

```
USE SampleDb;
```

```
CREATE LOGIN Vasya WITH PASSWORD = '12345!';
```

В примере ниже создается регистрационное имя "Vasya", которое сопоставляется с учетной записью пользователя Windows с таким же самым именем пользователя:

```
USE SampleDb;
```

```
CREATE LOGIN [ProfessorWeb\vasya] FROM WINDOWS;
```

Для конкретной системной среды нужно должным образом изменить имя компьютера и имя пользователя (в примере выше это ProfessorWeb и vasya соответственно).

Вторая инструкция языка Transact-SQL для обеспечения безопасности **ALTER LOGIN** - изменяет свойства определенного регистрационного имени. С помощью этой инструкции можно изменить текущий пароль и его конечную дату действия, параметры доступа, базу данных по умолчанию и язык по умолчанию. Также можно задействовать или отключить определенное регистрационное имя.

Наконец, инструкция **DROP LOGIN** применяется для удаления существующего регистрационного имени. Однако регистрационное имя, которое ссылается (владеет) на другие объекты, удалить нельзя.

Схема базы данных.

Схемы используются в модели безопасности компонента Database Engine для упрощения взаимоотношений между пользователями и объектами, и, следовательно, схемы имеют очень большое влияние на взаимодействие пользователя с компонентом Database Engine. В этом разделе рассматривается роль схем в безопасности компонента Database Engine. В первом подразделе описывается взаимодействие между схемами и пользователями, а во втором обсуждаются все три инструкции языка Transact-SQL, применяемые для создания и модификации схем.

Разделение пользователей и схем

Схема - это коллекция объектов базы данных, имеющая одного владельца и формирующая одно пространство имен. (Две таблицы в одной и той же схеме не могут иметь одно и то же имя.) Компонент Database Engine поддерживает именованные схемы с использованием понятия принципала (principal). Как уже упоминалось, принципалом может быть индивидуальный принципал и групповой принципал.

Индивидуальный принципал представляет одного пользователя, например, в виде регистрационного имени или учетной записи пользователя Windows. Групповым принципалом может быть группа пользователей, например, роль или группа Windows. Принципалы владеют схемами, но владение схемой может быть с легкостью передано другому принципалу без изменения имени схемы.

Отделение пользователей базы данных от схем дает значительные преимущества, такие как:

- один принципал может быть владельцем нескольких схем;
- несколько индивидуальных принципалов могут владеть одной схемой посредством членства в ролях или группах Windows;
- удаление пользователя базы данных не требует переименования объектов, содержащихся в схеме этого пользователя.

Каждая база данных имеет схему по умолчанию, которая используется для определения имен объектов, ссылки на которые делаются без указания их полных уточненных имен. В схеме по умолчанию указывается первая схема, в которой сервер базы данных будет выполнять поиск для разрешения имен объектов. Для настройки и изменения схемы по умолчанию применяется параметр **DEFAULT_SCHEMA** инструкции **CREATE USER** или **ALTER USER**. Если схема по умолчанию **DEFAULT_SCHEMA** не определена, в качестве схемы по умолчанию пользователю базы данных назначается **схема dbo**.

Инструкция CREATE SCHEMA

В примере ниже показано создание схемы и ее использование для управления безопасностью базы данных. Прежде чем выполнять этот пример, необходимо создать пользователей базы данных Alex и Vasya.


```
USE SampleDb;
```

```
GO
```

```
CREATE SCHEMA poco AUTHORIZATION Vasya
```

```
GO
```

```
CREATE TABLE Product (  
    Number CHAR(10) NOT NULL UNIQUE,  
    Name CHAR(20) NULL,  
    Price MONEY NULL);
```

```
GO
```

```
CREATE VIEW view_Product  
    AS SELECT Number, Name  
    FROM Product;
```

```
GO
```

```
GRANT SELECT TO Alex;  
DENY UPDATE TO Alex;
```

В этом примере создается схема poco, содержащая таблицу Product и представление view_Product. Пользователь базы данных Vasya является принципалом уровня базы данных, а также владельцем схемы. (Владелец схемы указывается посредством *параметра AUTHORIZATION*. Принципал может быть владельцем других схем и не может использовать текущую схему в качестве схемы по умолчанию.)

Две другие инструкции, применяемые для работы с разрешениями для объектов базы данных, GRANT и DENY, подробно рассматриваются позже. В этом примере инструкция GRANT предоставляет инструкции SELECT разрешения для всех создаваемых в схеме объектов, тогда как инструкция DENY запрещает инструкции UPDATE разрешения для всех объектов схемы.

С помощью инструкции **CREATE SCHEMA** можно создать схему, сформировать содержащиеся в этой схеме таблицы и представления, а также предоставить, запретить или удалить разрешения на защищаемый объект. Как упоминалось ранее, защищаемые объекты - это ресурсы, доступ к которым регулируется системой авторизации SQL Server. Существует три основные области защищаемых объектов: сервер, база данных и схема, которые содержат другие защищаемые объекты, такие как регистрационные имена, пользователи базы данных, таблицы и хранимые процедуры.

Инструкция **CREATE SCHEMA** является атомарной. Иными словами, если в процессе выполнения этой инструкции происходит ошибка, не выполняется ни одна из содержащихся в ней подынструкций.

Порядок указания создаваемых в инструкции **CREATE SCHEMA** объектов базы данных может быть произвольным, с одним исключением: представление, которое ссылается на другое представление, должно быть указано после представления, на которое оно ссылается.

Принципалом уровня базы данных может быть пользователь базы данных, роль или роль приложения. (Роли и роли приложения рассматриваются в одной из следующих статей.) Принципал, указанный в предложении AUTHORIZATION инструкции CREATE SCHEMA, является владельцем всех объектов, созданных в этой схеме. Владение содержащихся в схеме объектов можно передавать любому принципалу уровня базы данных посредством инструкции ALTER AUTHORIZATION.

Для исполнения инструкции CREATE SCHEMA пользователь должен обладать правами базы данных CREATE SCHEMA. Кроме этого, для создания объектов, указанных в инструкции CREATE SCHEMA, пользователь должен иметь соответствующие разрешения CREATE.

Инструкция ALTER SCHEMA

Инструкция ALTER SCHEMA перемещает объекты между разными схемами одной и той же базы данных. Инструкция ALTER SCHEMA имеет следующий синтаксис:

ALTER SCHEMA schema_name TRANSFER object_name

Использование инструкции ALTER SCHEMA показано в примере ниже:

USE AdventureWorks2012;

ALTER SCHEMA HumanResources TRANSFER Person.ContactType;

Здесь изменяется схема HumanResources базы данных AdventureWorks2012, перемещая в нее таблицу ContactType из схемы Person этой же базы данных. Инструкцию ALTER SCHEMA можно использовать для перемещения объектов между разными схемами только одной и той же базы данных. (Отдельные объекты в схеме можно изменить посредством инструкции ALTER TABLE или ALTER VIEW.)

Инструкция DROP SCHEMA

Для удаления схемы из базы данных применяется инструкция DROP SCHEMA. Схему можно удалить только при условии, что она не содержит никаких объектов. Если схема содержит объекты, попытка выполнить инструкцию DROP SCHEMA будет неуспешной.

Как указывалось ранее, владельца схемы можно изменить посредством инструкции ALTER AUTHORIZATION, которая изменяет владение сущностью. Язык Transact-SQL не поддерживает инструкции CREATE AUTHORIZATION и DROP AUTHORIZATION. Владелец схемы указывается с помощью инструкции CREATE SCHEMA.

Добавление пользователя БД.

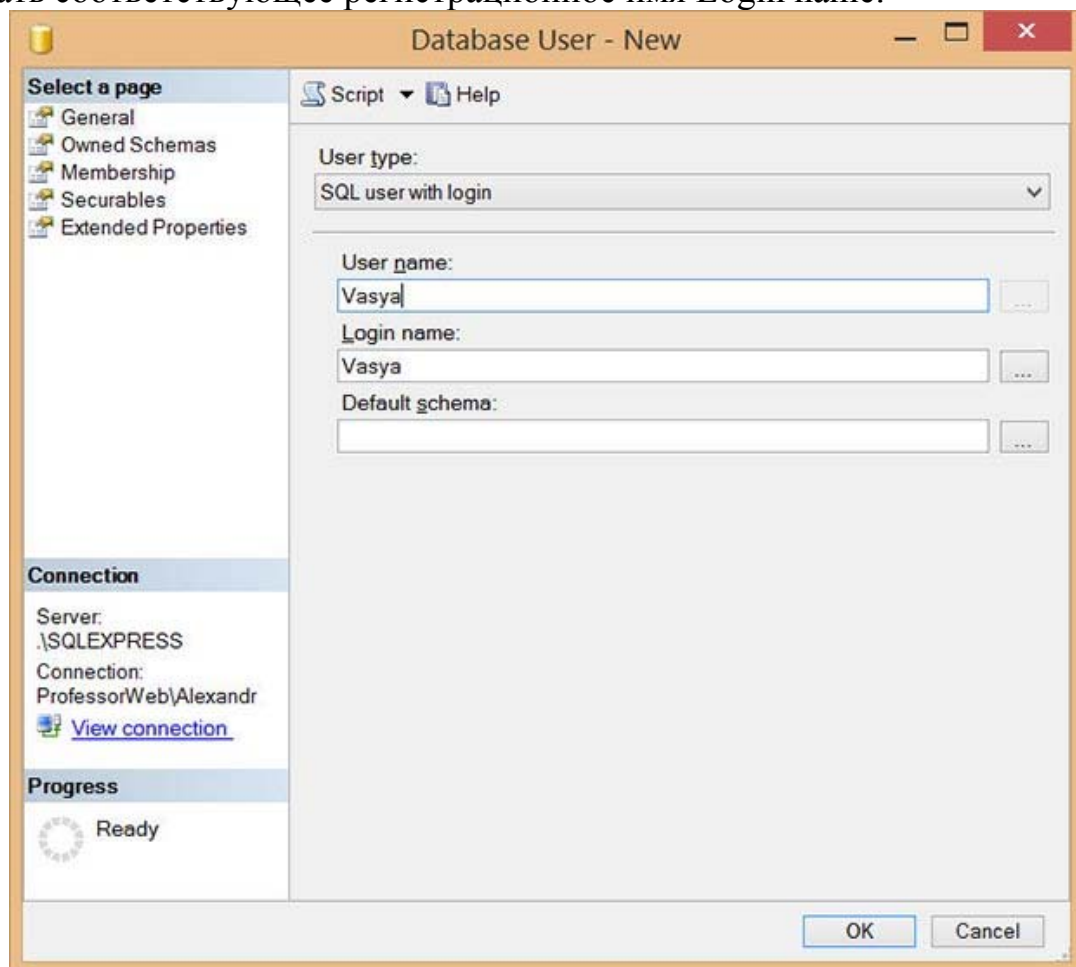
Пользователь может войти в систему баз данных, используя учетную запись пользователя Windows или регистрационное имя входа в SQL Server. Для последующего доступа и работы с определенной базой данных пользователь также должен иметь учетную запись пользователя базы данных. Для работы с каждой отдельной базой данных требуется иметь учетную запись пользователя именно для этой базы данных. Учетную запись пользователя базы данных можно сопоставить с существующей учетной

записью пользователя Windows, группой Windows (в которой пользователь имеет членство), регистрационным именем или ролью.

Управлять пользователями баз данных можно с помощью среды Management Studio или инструкций языка Transact-SQL. Оба эти способа рассматриваются в следующих подразделах.

Управление пользователями базы данных с помощью среды Management Studio

Чтобы добавить пользователя базы данных с помощью среды Management Studio, разверните узел сервера в окне Object Explorer и в нем папку "Databases", в этой папке разверните узел требуемой базы данных, а в ней папку "Security". Щелкните правой кнопкой мыши папку "Users" и в контекстном меню выберите пункт New User. Откроется диалоговое окно Database User - New, в котором следует ввести имя пользователя User name и выбрать соответствующее регистрационное имя Login name:



Здесь можно также выбрать схему по умолчанию для данного пользователя.

Управление безопасностью базы данных посредством инструкций языка Transact-SQL

Для добавления пользователя в текущую базу данных используется **инструкция CREATE USER**. Синтаксис этой инструкции выглядит таким образом:

CREATE USER user_name

```
[FOR {LOGIN login |CERTIFICATE cert_name | ASYMMETRIC KEY
key_name}]
[WITH DEFAULT_SCHEMA = schema_name]
```

Параметр `user_name` определяет имя, по которому пользователь идентифицируется в базе данных, а в параметре `login` указывается регистрационное имя, для которого создается данный пользователь. В параметрах `cert_name` и `key_name` указываются соответствующий сертификат и асимметричный ключ соответственно. Наконец, в параметре **WITH DEFAULT_SCHEMA** указывается первая схема, с которой сервер базы данных будет начинать поиск для разрешения имен объектов для данного пользователя базы данных.

Применение инструкции **CREATE USER** показано в примере ниже:
USE SampleDb;

```
CREATE USER Vasya FOR LOGIN Vasya;
CREATE USER Alex FOR LOGIN [ProfessorWeb\Alexandr]
WITH DEFAULT_SCHEMA = росо;
```

Для успешного выполнения на вашем компьютере второй инструкции примера требуется сначала создать учетную запись Windows для пользователя `Alexandr` и вместо домена (сервера) `ProfessorWeb` указать имя вашего сервера.

В этом примере первая инструкция **CREATE USER** создает пользователя базы данных `Vasya` для пользователя `Vasya` учетной записи Windows. Схемой по умолчанию для пользователя `Vasya` будет `dbo`, поскольку для параметра `DEFAULT_SCHEMA` значение не указано. Вторая инструкция **CREATE USER** создает нового пользователя базы данных `Alex`. Схемой по умолчанию для этого пользователя будет схема `росо`. (Параметру `DEFAULT_SCHEMA` можно присвоить в качестве значения схему, которая в данное время не существует в базе данных.)

Каждая база данных имеет своих конкретных пользователей. Поэтому инструкцию **CREATE USER** необходимо выполнить для каждой базы данных, для которой должна существовать учетная запись пользователя. Кроме этого, для определенной базы данных регистрационное имя входа в SQL Server может иметь только одного пользователя базы данных.

С помощью инструкции **ALTER USER** можно изменить имя пользователя базы данных, изменить схему пользователя по умолчанию или переопределить пользователя с другим регистрационным именем. Подобно инструкции **CREATE USER**, пользователю можно присвоить схему по умолчанию прежде, чем она создана.

Для удаления пользователя из текущей базы данных применяется инструкция **DROP USER**. Пользователя, который является владельцем защищаемых объектов (объектов базы данных), удалить нельзя.

Схемы базы данных по умолчанию

Каждая база данных в системе имеет следующие схемы по умолчанию:

- guest
- dbo
- information_schema
- sys

Компонент Database Engine позволяет пользователям, которые не имеют учетной записи пользователя, работать с базой данных, используя схему guest. (Каждая созданная база данных имеет эту схему.) Для схемы guest можно применять разрешения таким же образом, как и для любой другой схемы. Кроме этого, схему guest можно удалить из любой базы данных, кроме системных баз данных master и tempdb.

Каждый объект базы данных принадлежит одной, и только одной схеме, которая является схемой по умолчанию для данного объекта. Схема по умолчанию может быть определена явно или неявно. Если при создании объекта его схема по умолчанию не определена явно, этот объект принадлежит к схеме dbo. Кроме этого, при использовании принадлежащей ему базы данных регистрационное имя всегда имеет специальное имя пользователя dbo.

Вся информация о схемах содержится в *схеме information_schema*. Схема sys, как можно догадаться, содержит системные объекты, такие как представления каталога.

Добавление роли.

Когда нескольким пользователям требуется выполнять подобные действия в определенной базе данных и при этом они не являются членами соответствующей группы Windows, то можно воспользоваться ролью базы данных, задающей группу пользователей базы данных, которые могут иметь доступ к одним и тем же объектам базы данных.

Членами роли базы данных могут быть любые из следующих:

- группы и учетные записи Windows;
- регистрационные имена входа в SQL Server;
- другие роли.

Архитектура безопасности компонента Database Engine включает несколько "системных" ролей, которые имеют специальные явные разрешения. Кроме ролей, определяемых пользователями, существует два типа предопределенных ролей: фиксированные серверные роли и фиксированные роли базы данных.

Фиксированные серверные роли

Фиксированные серверные роли определяются на уровне сервера и поэтому находятся вне баз данных, принадлежащих серверу баз данных. В таблице ниже приводится список фиксированных серверных ролей и краткое описание действий, которые могут выполнять члены этих ролей:

Фиксированные серверные роли

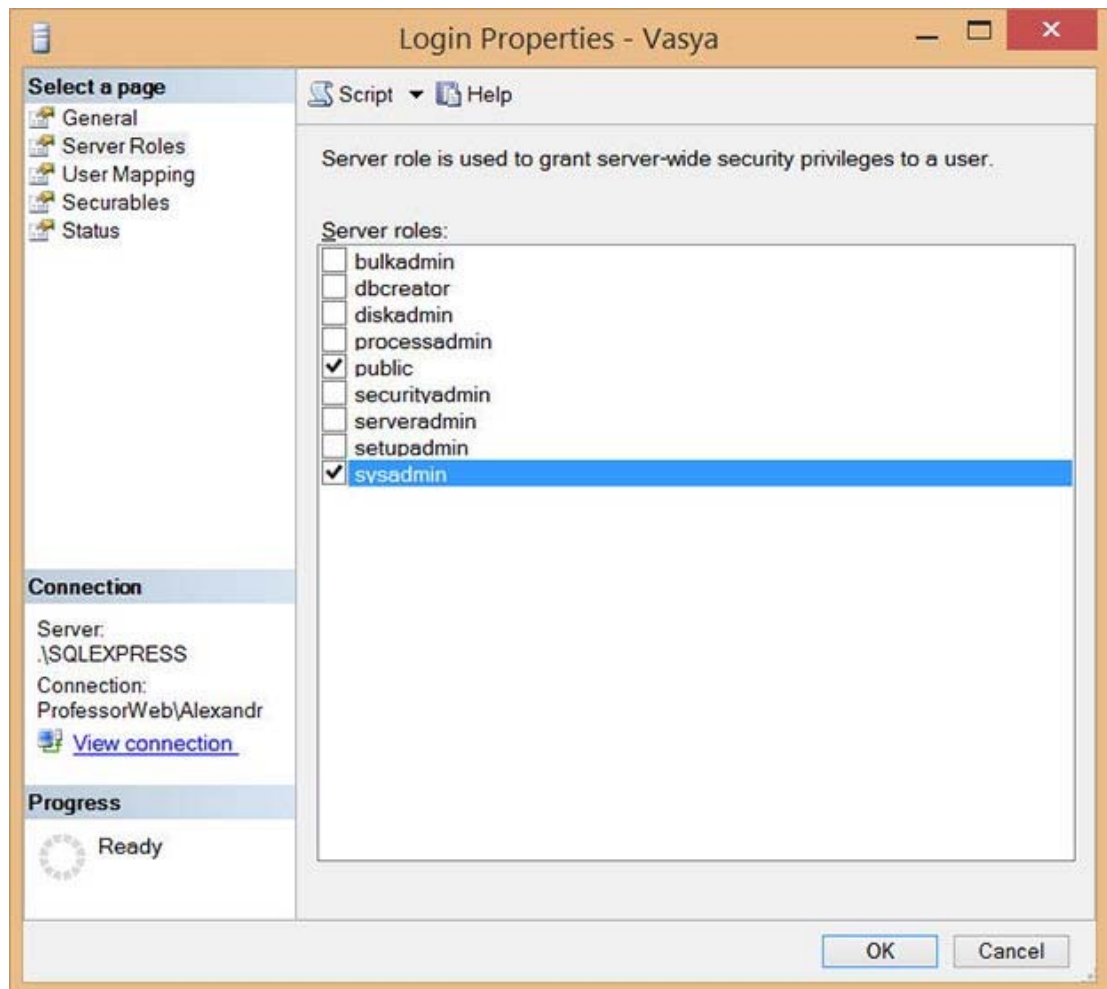
Фиксированная	Описание
---------------	----------

серверная роль	
<i>sysadmin</i>	Выполняет любые действия в системе баз данных
<i>serveradmin</i>	Конфигурирует параметры сервера
<i>setupadmin</i>	Устанавливает репликацию и управляет расширенными процедурами
<i>securityadmin</i>	Управляет регистрационными именами и разрешениями для инструкции CREATE DATABASE и чтением журналов логов
<i>processadmin</i>	Управляет системными процессами
<i>dbcreator</i>	Создает и модифицирует базы данных
<i>diskadmin</i>	Управляет файлами на диске

Членов фиксированной серверной роли можно добавлять и удалять двумя способами:

- используя среду Management Studio;
- используя инструкции языка Transact-SQL.

Чтобы добавить регистрационное имя в члены фиксированной серверной роли посредством среды Management Studio, разверните в обозревателе объектов узел сервера, в нем папку "Security", а в ней разверните папку "Logins". Выберите имя пользователя, для которого нужно изменить роль, щелкните правой кнопкой мыши и выберите в контекстном меню Properties. В открывшемся окне Login Properties перейдите на вкладку Server Role, где можно задавать или удалять пользователя в фиксированной роли:



Для добавления и удаления членов в фиксированные серверные роли используются инструкции языка Transact-SQL **CREATE SERVER ROLE** и **DROP SERVER ROLE** соответственно. А для изменения членства в серверной роли используется инструкция **ALTER SERVER ROLE**.

Фиксированные серверные роли нельзя добавлять, удалять или переименовывать. Кроме этого, только члены фиксированных серверных ролей могут выполнять системные процедуры для добавления или удаления регистрационного имени в роли.

Регистрационное имя sa

Регистрационное имя sa является регистрационным именем системного администратора. В версиях более ранних, чем SQL Server 2005, в которых роли отсутствовали, регистрационное имя sa предоставляло все возможные разрешения для задач системного администрирования. В более новых версиях SQL Server регистрационное имя sa включено единственно с целью обратной совместимости. Это регистрационное имя всегда является членом фиксированной серверной роли sysadmin и его нельзя удалить из этой роли.

Регистрационное имя sa следует использовать только в тех случаях, когда нет другого способа войти в систему базы данных.

Фиксированные роли базы данных

Фиксированные роли базы данных определяются на уровне базы данных и поэтому существуют в каждой базе данных, принадлежащей серверу баз данных. В таблице ниже приводится список фиксированных

ролей базы данных и краткое описание действий, которые могут выполнять члены этих ролей.

Фиксированные роли базы данных

Фиксированная роль базы данных	Описание
<i>db_owner</i>	Пользователи, которые могут выполнять почти все действия в базе данных
<i>db_accessadmin</i>	Пользователи, которые могут добавлять и удалять пользователей
<i>db_datareader</i>	Пользователи, которые могут просматривать данные во всех таблицах пользователей базы данных
<i>db_datawriter</i>	Пользователи, которые могут добавлять, изменять или удалять данные во всех пользовательских таблицах базы данных
<i>db_ddladmin</i>	Пользователи, которые могут выполнять инструкции DDL в базе данных
<i>db_securityadmin</i>	Пользователи, которые могут управлять всеми действиями в базе данных, связанными разрешениями безопасности
<i>db_backupoperator</i>	Пользователи, которые могут выполнять резервное копирование базы данных
<i>db_denydatareader</i>	Пользователи, которые не могут просматривать любые данные в базе данных
<i>db_denydatawriter</i>	Пользователи, которые не могут изменять никакие данные в базе данных

Члены фиксированных ролей баз данных могут выполнять разные действия. Подробную информацию о действиях, которые могут выполнять

члены каждой фиксированной роли базы данных смотрите в электронной документации.

Фиксированная роль базы данных public

Кроме перечисленных в таблице фиксированных ролей базы данных, существует специальная фиксированная роль базы данных public. Фиксированная роль базы данных public является специальной ролью, членом которой являются все законные пользователи базы данных. Она охватывает все разрешения по умолчанию для пользователей базы данных. Это позволяет предоставить всем пользователям, которые не имеют должных разрешений, набор разрешений (обычно ограниченный). Роль public предоставляет все разрешения по умолчанию для пользователей базы данных и не может быть удалена. Пользователям, группам или ролям нельзя присвоить членство в этой роли, поскольку они имеют его по умолчанию.

По умолчанию роль public разрешает пользователям выполнять следующие действия:

- просматривать системные таблицы и отображать информацию из системной базы данных master, используя определенные системные процедуры;
- выполнять инструкции, для которых не требуются разрешения, например, PRINT.

Присвоения пользователю членства в фиксированной роли базы данных

Чтобы присвоить пользователю базы данных членство в фиксированной роли базы данных с помощью среды Management Studio, разверните сервер и папку "Databases", а в ней базу данных, затем разверните папку "Security", "Roles" и папку "Databases Roles". Щелкните правой кнопкой мыши роль, в которую требуется добавить пользователя, и в контекстном меню выберите пункт Properties. В диалоговом окне свойств роли базы данных нажмите кнопку Add и выберите пользователей, которым нужно присвоить членство в этой роли. Теперь этот пользователь является членом данной роли базы данных и наследует все параметры доступа, предоставленные этой роли.

Роли приложений

Роли приложения позволяют принудительно обеспечивать безопасность для определенного приложения. Иными словами, роли приложения позволяют приложению взять на себя ответственность за аутентификацию пользователя, вместо того, чтобы это делала система баз данных. Например, если служащие компании могут изменять данные о сотрудниках только посредством какого-либо приложения (а не посредством инструкций языка Transact-SQL или какого-либо другого средства), для этого приложения можно создать роль приложения.

Роли приложений существенно отличаются от всех других типов ролей. Во-первых, роли приложений не имеют членов, поскольку они используют только приложения, и поэтому им нет необходимости

предоставлять разрешения непосредственно пользователям. Во-вторых, для активации роли приложения требуется пароль.

Когда приложение активирует для сеанса роль приложения, этот сеанс утрачивает все разрешения, применимые к именам ввода, учетным записям пользователей, группам пользователей или ролям во всех базах данных. Так как эти роли применимы только к базе данных, в которой они находятся, сеанс может получить доступ к другой базе данных только посредством разрешений, предоставленных пользователю guest базы данных, к которой требуется доступ. Поэтому, если база данных не имеет пользователя guest, сеанс не может получить доступ к этой базе данных.

В следующих двух подразделах описывается управление ролями приложений.

Управление ролями приложений посредством среды Management Studio

Чтобы создать роль приложения с помощью среды Management Studio, разверните узел сервера, папку "Databases", требуемую базу данных, папку "Security". Щелкните правой кнопкой папку "Roles", в появившемся контекстном меню выберите пункт New, а во вложенном меню выберите пункт New Application Role. В открывшемся диалоговом окне Application Role - New введите в соответствующие поля имя новой роли приложения, пароль и, необязательно, схему по умолчанию. Нажмите кнопку ОК, чтобы сохранить роль.

Управление ролями приложений посредством инструкций Transact-SQL

Для создания, изменения и удаления ролей приложений применяются инструкции языка Transact-SQL CREATE APPLICATION ROLE, ALTER APPLICATION ROLE и DROP APPLICATION ROLE соответственно.

Инструкция CREATE APPLICATION ROLE, создающая роль приложения для текущей базы данных, имеет два параметра. В первом параметре указывается пароль роли, а во втором - схема по умолчанию, т.е. первая схема, к которой будет обращаться сервер для разрешения имен объектов для этой роли.

В примере ниже показано создание роли приложения WeeklyReports в базе данных SampleDb:

```
USE SampleDb;
```

```
CREATE APPLICATION ROLE WeeklyReports  
WITH PASSWORD = '12345!'  
DEFAULT_SCHEMA = poc;
```

Инструкция ALTER APPLICATION ROLE применяется для изменения имени, пароля или схемы по умолчанию существующей роли приложения. Синтаксис этой инструкции очень сходен с синтаксисом инструкции CREATE APPLICATION ROLE. Для выполнения инструкции ALTER APPLICATION ROLE для этой роли необходимо иметь разрешение ALTER.

Для удаления роли приложения используется **инструкция DROP APPLICATION ROLE**. Роль приложения нельзя удалить, если она владеет какими-либо объектами (защищаемыми объектами).

Активация ролей приложений

При установлении соединения необходимо выполнить системную процедуру **sp_setuprole**, чтобы активировать разрешения, связанные с ролью приложения. Эта процедура имеет следующий синтаксис:

```
sp_setapprole [@rolename =] 'role' ,  
               [@password =] 'password'  
               [,[@encrypt =] 'encrypt_style']
```

В параметре **role** указывается имя роли приложения, определенного в текущей базе данных, в параметре **password** - пароль для этой роли, а в параметре **encrypt_style** - метод шифрования, указанный для пароля.

При активации роли приложения с помощью системной процедуры **sp_setuprole** необходимо иметь в виду следующее:

- активированную роль приложения нельзя деактивировать в текущей базе данных, пока сеанс не отсоединится от системы;
- роль приложения всегда привязана к базе данных, т.е. ее область видимости ограничивается текущей базой данных. Если в течение сеанса изменить текущую базу данных, то в ней можно будет выполнять действия, зависящие от разрешений в этой базе данных.

Конструкция ролей приложений в SQL Server 2012 не является оптимальной вследствие своей неоднородности. В частности, роли приложения создаются посредством инструкций языка Transact-SQL, после чего они активируются с помощью системной процедуры.

Определяемые пользователем роли сервера

Определяемые пользователем роли сервера впервые стали применяться в SQL Server 2012. Для создания и удаления этих ролей используются инструкции языка Transact-SQL **CREATE SERVER ROLE** и **DROP SERVER ROLE** соответственно. Для добавления или удаления членов роли сервера используется инструкция **ALTER SERVER ROLE**. Использование инструкций **CREATE SERVER ROLE** и **ALTER SERVER ROLE** показано в примере ниже:

```
USE SampleDb;
```

```
GO
```

```
CREATE SERVER ROLE program_admin;
```

```
ALTER SERVER ROLE program_admin ADD MEMBER Vasya;
```

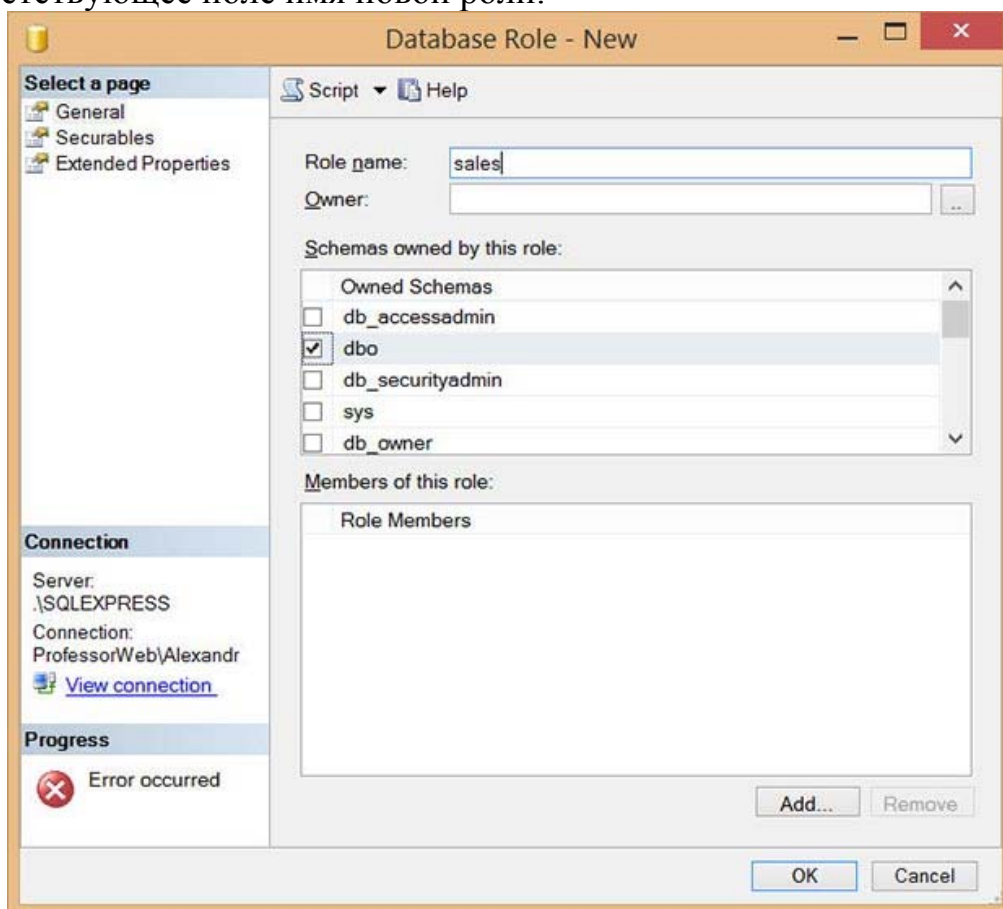
Определяемые пользователем роли баз данных

Обычно определяемые пользователем роли базы данных применяются, когда группе пользователей базы данных требуется выполнять общий набор действий в базе данных и отсутствует применимая группа пользователей Windows. Для создания, изменения и удаления этих ролей применяется или среда Management Studio, или инструкции языка Transact-SQL **CREATE**

ROLE, ALTER ROLE и DROP ROLE. Управление определяемыми пользователем ролями базы данных рассматривается в следующих двух подразделах.

Управление определяемыми пользователем ролями базы данных с помощью среды Management Studio

Чтобы создать определяемую пользователем роль базы данных с помощью среды Management Studio, разверните узел сервера, папку "Databases", требуемую базу данных, папку "Security". Щелкните правой кнопкой папку "Roles", в появившемся контекстном меню выберите пункт New, а во вложенном меню выберите пункт New Database Role. В открывшемся диалоговом окне Database Role - New введите в соответствующее поле имя новой роли:



Нажмите кнопку Add, чтобы добавить членов в новую роль. Выберите требуемых членов (пользователей и/или другие роли) новой роли базы данных и нажмите кнопку OK.

Управление определяемыми пользователем ролями базы данных с помощью инструкций Transact-SQL

Для создания новой определяемой пользователем роли базы данных в текущей базе данных применяется **инструкция CREATE ROLE**. Синтаксис этой инструкции выглядит таким образом:

CREATE ROLE role_name [AUTHORIZATION owner_name]

В параметре role_name инструкции указывается имя создаваемой определяемой пользователем роли, а в параметре owner_name - пользователь базы данных или роль, которая будет владельцем новой роли. (Если

пользователь не указан, владельцем роли будет пользователь, исполняющий инструкцию CREATE ROLE.)

Для изменения имени определяемой пользователем роли базы данных применяется **инструкция ALTER ROLE**, а для удаления роли из базы данных - **инструкция DROP ROLE**. Роль, которая является владельцем защищаемых объектов (т.е. объектов базы данных), удалить нельзя. Чтобы удалить такую роль, сначала нужно изменить владельца этих объектов.

В примере ниже показано создание определяемой пользователем роли базы данных и добавление в нее членов:

```
USE SampleDb;
```

```
CREATE ROLE marketing AUTHORIZATION Vasya;
```

```
GO
```

```
ALTER ROLE marketing ADD MEMBER 'Vasya';
```

```
ALTER ROLE marketing ADD MEMBER 'ProfessorWeb\Alexandr';
```

В этом примере сначала создается определяемая пользователем роль базы данных marketing, а затем, *предложением ADD MEMBER* инструкции ALTER ROLE, в нее добавляются два члена - Vasya и ProfessorWeb\Alexandr.