

Цветанка Георгиева

ПРАКТИКУМ ПО
БАЗИ ОТ ДАННИ

ЧАСТ II

Велико Търново

2009

Настоящият практикум по Базы от данни е продължение на „Практикум по Базы от данни” – I част. Разработен е в съответствие с учебните планове на специалностите „Математика и информатика”, „Информатика” и „Компютърни науки” във Великотърновския университет „Св. св. Кирил и Методий”. Той е предназначен за студентите от редовна форма на обучение, както и за обучаващите се в магистърските програми по Информатика. Практикумът може да бъде полезен и на студентите от други специалности и други висши учебни заведения, които изучават учебни дисциплини, свързани с бази от данни.

Примерните SQL конструкции са тествани в средата на Microsoft SQL Server 2005. Всички включени в темите примери могат да бъдат изтеглени от адрес <http://practicum.host22.com>.

Предлаганият материал е обсъден и утвърден за печат на заседание на катедра „Информационни технологии” на Великотърновския университет „Св. св. Кирил и Методий”.

Рецензент: ст.н.с. I ст. д.м.н Петър Любомиров Станчев

© Цветанка Любомирова Георгиева, автор, 2009

Съдържание

Предговор	5
Тема 1 Програмиране с Transact-SQL	6
Използване на променливи	6
Управление на реда на изпълнение	8
Извеждане на съобщения	12
Функциите @@ERROR и @@ROWCOUNT	16
Тема 2 Работа с XML данни	18
Съхраняване на XML данни	18
Извличане на XML данни	20
Модифициране на XML данни	27
Преобразуване на XML данни в релационни	28
Тема 3 Създаване на изгледи	32
Създаване, променяне и изтриване на изгледи	32
Променяне на данни посредством изгледи	34
Тема 4 Временни таблици	40
Частни временни таблици	40
Глобални временни таблици	40
Директно използване на <i>tempdb</i>	41
Тема 5 Транзакции	42
Свойства на транзакциите	42
Проверка за грешки в транзакциите	43
Нива на изолация на транзакциите	45
Вложени блокове транзакции	47
Точки на записване в транзакциите	49
Тема 6 Създаване на съхранени процедури	52
Роля, създаване и извикване на съхранени процедури	52
Създаване на съхранени процедури с параметри	54
Използване на резултатите, върнати от изходните параметри	56
Съхранени процедури и INSERT	58
Тема 7 Работа с йерархични данни в бази от данни на Microsoft SQL Server	61
Тема 8 Дефинирани от потребителя функции	67
Скаларни функции	67
Функции, връщащи таблица	70
Многоструктурни функции, връщащи таблица	70
Тема 9 Създаване на тригери	73
Създаване и използване на тригери	73
Каскадни и рекурсивни тригери	76
Използване на тригери за изпълняване на действия за запазване на целостта на данните	77
Тема 10 Използване на курсори	82
Деклариране на курсори	82
Отваряне на курсори и извличане на ред от курсор	83
Претърсване на съдържанието на курсора	84
Актуализиране на курсори	85
Затваряне и освобождаване на курсори	85
Курсорни променливи	86

Динамично създаване на кръстосани заявки чрез генериране на CASE изрази	88
Примери за предимството от използване на системните таблици.....	89
Тема 11 Импортиране и експортиране на данни	96
Използване на конструкцията BULK INSERT за импортиране на данни	96
Използване на помощната програма bcp за импортиране и експортиране на данни	99
Използване на SSIS (<i>SQL Server Integration Services</i>) за импортиране и експортиране на данни	105
Тема 12 Установяване на връзка между клиентското приложение и Microsoft SQL Server	110
ODBC (Open Database Connectivity – отворена система за свързване)	110
OLE DB (Object Linking and Embedding – свързване и вграждане на обекти) ...	113
Тема 13 Създаване на клиентски приложения в Microsoft Access за Microsoft SQL Server	116
Създаване на клиентски MDB и ADP приложения за Microsoft SQL Server ...	116
Създаване на форми в Microsoft Access	118
Контроли във формите на Microsoft Access	121
Работа с макроси	125
Тема 14 Системата за сигурност на Microsoft SQL Server	133
Режими за автентикация на Microsoft SQL Server	133
Microsoft SQL Server логици	134
Роли	136
Създаване на потребителски идентификатори за база от данни	138
Привилегии	141
Роли за бази от данни	143
Състояния на привилегиите	150
Конфигуриране на привилегии за обекти чрез Transact-SQL	151
Конфигуриране на привилегии за конструкции чрез Transact-SQL	155
Създаване на план на сигурността	155
Приложение 1	157
Литература	159

Предговор

Базите от данни са в центъра на нашето информационно общество. Практически всяка система, с която работим, съдържа база от данни. През следващото десетилетие ние ще бъдем изключително зависими от коректността и ефективността на тези системи. Системите за управление на бази от данни са част от съвременната технология за работа с бази от данни. Настоящият практикум разглежда различни аспекти на работата с релационни бази от данни и може да бъде използван от хора, имащи основни познания.

Материалът е представен по лесен за усвояване начин. Множество решени задачи са дадени. Изброени са и задачи за самостоятелна работа.

Практикумът по Базите от данни е разработен в съответствие с учебните планове на специалностите „Математика и информатика“, „Информатика“ и „Компютърни науки“ във Великотърновския университет „Св. св. Кирил и Методий“. Той е предназначен за студентите от редовна форма на обучение, както и за обучаващите се в магистърските програми по Информатика. Практикумът може да бъде полезен и на студентите от други специалности и други висши учебни заведения, които изучават учебни дисциплини, свързани с бази от данни. Примерните SQL конструкции са тествани в средата на Microsoft SQL Server 2005. Като предимство може да се посочи, че всички включени в темите примери могат да бъдат изтеглени от адрес <http://practicum.host22.com>.

Настоящият практикум по Базите от данни (втора част) включва четиринадесет теми: програмиране с Transact-SQL, работа с XML данни, създаване на изгледи, временни таблици, транзакции, създаване на съхранени процедури, работа с йерархични данни в бази от данни на Microsoft SQL Server, създаване на дефинирани от потребителя функции, създаване на тригери, използване на курсори, импортиране и експортиране на данни, установяване на връзка между клиентското приложение и Microsoft SQL, създаване на клиентски приложения в Microsoft Access за Microsoft SQL Server, системата за сигурност на Microsoft SQL Server, както и едно приложение. Дадени са основни знания за работа с XML данни; въпроси, свързани със SQL заявки като изгледи; тригери; управление на транзакции; йерархични структури; сигурност и упълномощаване; създаване на приложения.

Практикумът може да послужи и на хора, желаещи да създадат свои бази от данни, дава решение на голям брой въпроси за използването на тази технология.

ст.н.с. I ст. д.м.н Петър Л. Станчев
Ръководител
на секция „Информационни системи“
Институт по математика и информатика
Българска академия на науките

Dr. Peter L. Stanchev
Professor of Computer Science
Kettering University, Flint, Michigan
Computer Science Department
<http://www.kettering.edu/~pstanche>

Програмиране с Transact-SQL

Всяка система за управление на релационни бази от данни използва SQL (*Structured Query Language* – език за структурирани заявки). Наличието на разнообразни продукти за управление на бази от данни води до необходимостта от дефиниране на стандарт, на който да отговарят. Стандартът SQL се дефинира съвместно от ANSI (*American National Standards Institute* – Американски национален институт по стандартизация) и ISO (*International Organization for Standardization* – Международна организация по стандартизация). Първоначалният стандарт е разработен през 1986 година, след което е няколкократно преработван, като получените в резултат на това главни ANSI/ISO SQL стандарти са: SQL-86, SQL-89, SQL-92, SQL-99, SQL-2003.

Повечето SQL-базирани продукти за управление на бази от данни, включително Microsoft SQL Server, поддържат напълно SQL-92, като включват избрани характеристики от SQL-99 и SQL-2003. Езикът, който Microsoft SQL Server използва, се нарича *Transact-SQL* (*T-SQL*). Microsoft SQL Server 2005 предоставя среда за работа с Transact-SQL посредством инструмента SQL Server Management Studio. Този инструмент е включен и в безплатното издание SQL Server 2005 Express Edition, което може да бъде изтеглено от <http://www.microsoft.com/Sqlserver/2005/en/us/express-down.aspx>.

Програмирането на Transact-SQL позволява да се подобри производителността на приложението, тъй като се намалява предаването на съобщения между клиентския процес и сървъра. При архитектурата клиент/сървър е от съществено значение да се минимизира обменът на данни между клиента и сървъра. Чрез програмиране на Transact-SQL е възможно приложението да изпълнява дадена процедура, като я извика и ѝ предаде някои параметри, след което процедурата се изпълнява отдалечено, без да е необходима междинна обработка при клиента.

Друго предимство от използването на Transact-SQL е изолиране на приложението от промените. Процедурата при базата от данни може да бъде променена и приложението да остане напълно незасегнато. Докато входът и изходът остават неизменени, приложението е защитено от направените промени в базата от данни.

Transact-SQL не предлага потребителски интерфейс, нито вход/изход към файлове и устройства.

Използване на променливи

Всички променливи са локални, т.е. имат диапазон и видимост само в пакета или съхранената процедура, в която са декларирани. Името на променливата трябва да започва със символа @. Глобални променливи не съществуват, временните таблици предоставят възможност за разделяне на стойности между конекциите.

Декларирането на променливите се прави обикновено в началото на пакета или съхранената процедура чрез следния синтаксис:

```
DECLARE @var_name datatype
```

Присвояването на стойност на променлива се осъществява чрез SET или SELECT по следния начин:

```
SET @var_name = var_value
```

или

```
SELECT @var_name = var_value
```

Така нареченият *присвояващ* SELECT се използва, когато стойностите, които трябва да бъдат присвоени, са в колона от таблица. Позволява присвояване на стойност

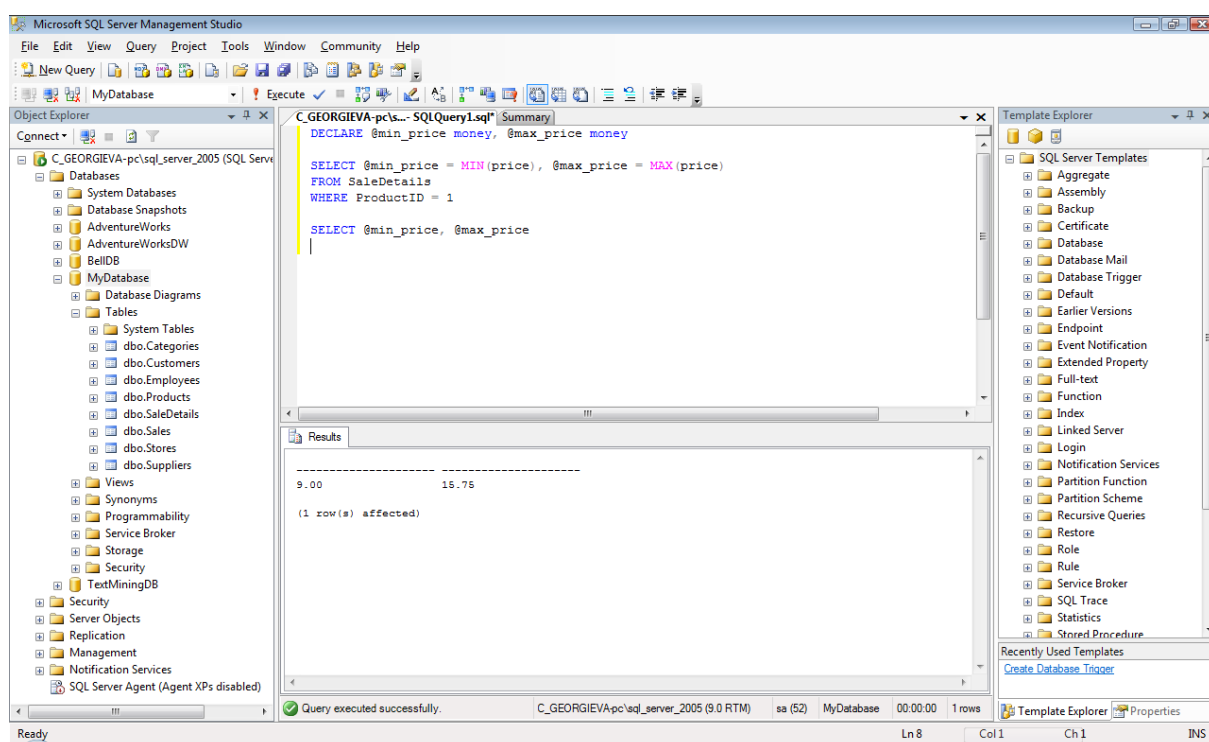
наведнъж на повече от една променлива. При използване на SET са необходими отделни конструкции за всяко присвояване. Например^{*}:

```
DECLARE @min_price money, @max_price money
```

```
SELECT @min_price = MIN(price),
       @max_price = MAX(price)
FROM SaleDetails
WHERE ProductID = 1
```

```
SELECT @min_price, @max_price
```

Второто използване на SELECT връща стойностите на клиента като резултатен набор (фиг. 1).



Фиг. 1 Изпълнение на примерния код от SQL Server Management Studio

При присвояване на стойност на променлива чрез избиране на стойност от базата от данни трябва да се гарантира, че конструкцията SELECT ще върне само един ред. В противен случай няма да бъде върнато съобщение за грешка и променливата ще има стойността на последния върнат ред. Обикновено се използват обобщаващи функции (MAX(), MIN(), SUM(), AVG(), COUNT(), StDev(), VAR()) или SELECT TOP 1 (с ORDER BY). Възможно е да се използва функцията @@ROWCOUNT непосредствено след присвояването и ако нейната стойност е по-голяма от единица, да се направи разклонение към подпрограма за грешка.

Преди да бъде зададена някаква стойност на една променлива чрез SET или SELECT, нейната стойност е NULL.

Ако SELECT не върне никаква стойност, нищо няма да бъде присвоено и променливата ще запази стойността, която е имала преди да бъде изпълнена присвояващата конструкция SELECT.

^{*} В примерите се използват таблиците, структурата на които е описана в Приложение 1.

Пример 1 Ако в таблицата `Products` има данни за продукт с идентификатор 1 и не съществува продукт с идентификатор 1898695, в резултат от изпълнението на следния код два пъти ще се изведе наименованието на продукта с `ProductID` единица:

```
DECLARE @ProductName varchar(40)

SELECT @ProductName = ProductName
FROM Products
WHERE ProductID = 1

SELECT @ProductName

SELECT @ProductName = ProductName
FROM Products
WHERE ProductID = 1898695

SELECT @ProductName -- връща като резултат предишната стойност
```

Пример 2 Когато се използва обобщаваща функция (без `GROUP BY`), винаги се връща точно една стойност. Ако е зададено условие и няма редове, които да го удовлетворяват, върнатата стойност е `NULL`.

```
DECLARE @total money

SELECT @total = SUM(sd.price*sd.quantity*(1-sd.discount))
FROM SaleDetails sd
INNER JOIN Sales s ON sd.SaleID = s.SaleID
WHERE DATEDIFF(day, s.SaleDate, GetDate()) = 0

SELECT @total

SELECT @total = SUM(sd.price*sd.quantity*(1-sd.discount))
FROM SaleDetails sd
INNER JOIN Sales s ON sd.SaleID = s.SaleID
WHERE DATEDIFF(day, s.SaleDate, DATEADD(day, 1, GetDate())) = 0
-- или DATEDIFF(day, s.SaleDate, GetDate()) = -1

SELECT @total -- връща като резултат стойност NULL,
               -- тъй като няма продажби за утрешна дата
```

Възможно е формулиране и изпълняване на даден низ динамично в **SQL Server** с командата `EXECUTE('string')` или `EXEC('string')`. Може да се подаде низ или променлива от тип `char` или `varchar`. **Например:**

```
DECLARE @table_name varchar(30)
SET @table_name = 'Products'
EXEC ('SELECT * FROM ' + @table_name)
```

Управление на реда на изпълнение

- условен оператор;

```
IF condition
BEGIN
```



```

        statements
    END
ELSE
    BEGIN
        statements
    END

```

Например:

```

IF EXISTS
(
    SELECT * FROM Customers
    WHERE City = 'Велико Търново'
)
SELECT COUNT(*) AS CountOfCustomers
FROM Customers
WHERE City = 'Велико Търново'
ELSE
    SELECT 'Няма клиенти от избрания град!'

```

- оператор за цикли;

```

WHILE condition
    BEGIN
        statements
    END

```

Изразите в тялото на цикъла се повтарят, докато условието е изпълнено.

Например:

```

DECLARE @counter int

SET @counter = 1
WHILE @counter <= 10
    BEGIN
        INSERT INTO Sales
            (CustomerID, EmployeeID, SaleDate)
        VALUES (1, 1, DATEADD(mi, @counter, GetDate()))

        SET @counter = @counter + 1
    END

```

- оператор за изход от най-вътрешния цикъл WHILE;

BREAK

Например:

```

DECLARE @counter int

SET @counter = 1
WHILE @counter <= 10
    BEGIN
        SELECT @counter
        SET @counter = @counter + 1
        IF @counter = 7 BREAK
    END

```

В този пример върнатите като резултатен набор стойности са 1, 2, 3, 4, 5 и 6.

- оператор за рестартиране на цикъла WHILE;

CONTINUE

Например:

```

DECLARE @counter int

SET @counter = 0
WHILE @counter < 10
BEGIN
    SET @counter = @counter + 1
    IF @counter = 7 CONTINUE
    SELECT @counter
END

```

В случая върнатите като резултатен набор стойности са 1, 2, 3, 4, 5, 6, 8, 9 и 10.

- оператор за безусловен преход;

```
GOTO label_name
```

Продължава обработката от конструкцията, следваща зададения етикет.

```

...
GOTO label_name

```

```

...
label_name:

```

- оператор за безусловен изход;

```
RETURN [n]
```

Обикновено се използва в съхранена процедура или тригер. Цялото число *n* е опционално и се установява като изходен статус, който да бъде присвоен на променлива при изпълняването на съхранената процедура. Ако се пропусне, RETURN връща 0.

- оператор за установяване на време за изпълнение на конструкцията;

```
WAITFOR
```

Може времето да бъде интервал (до 24 часа), тогава операторът има вида:

```
WAITFOR DELAY 'интервал за изчакване'
```

Може времето да бъде определен час от деня, тогава операторът има вида:

```
WAITFOR TIME 'час'
```

Времето може да бъде представено като константа или променлива. Например:

- WAITFOR DELAY '5:20'
- WAITFOR TIME '14:00'
- DECLARE @Morning datetime
SET @Morning = '12:06:30'
WAITFOR TIME @Morning
SELECT @Morning

В стойността на @Morning е от значение само часа, т.е. не може да се зададе определена дата и час, а само час от деня.

- оператор за обработка на грешки.

```

BEGIN TRY
    statements
END TRY
BEGIN CATCH
    statements
END CATCH

```

Ако възникне грешка в TRY блока, изпълнението преминава към групата от конструкции, заградени в CATCH блока. За да се получи информация за грешката, предизвикала изпълнението на CATCH блока, могат да се използват следните системни функции:

- `ERROR_NUMBER()` – номера на грешката;
- `ERROR_SEVERITY()` – степента на строгост на грешката;
- `ERROR_STATE()` – кода на състоянието, свързан с грешката;
- `ERROR_PROCEDURE()` – наименованието на съхранената процедура или тригера, в който е възникнала грешката;
- `ERROR_LINE()` – номера на реда, в който е възникнала грешката;
- `ERROR_MESSAGE()` – пълния текст на съобщението на грешката. Текстът включва стойностите, получени от всички параметри за заместване.

Ако тези функции се извикат извън `CATCH` блок, те връщат стойност `NULL`.

Пример 3 Нека в таблицата `SaleDetails` съществува ред със стойност 1 в `ProductID`.

```
BEGIN TRY
    -- Генерира грешка за нарушаване на ограничение външен ключ.
    DELETE FROM Products
    WHERE ProductID = 1
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber,
           ERROR_SEVERITY() AS ErrorSeverity,
           ERROR_STATE() AS ErrorState,
           ERROR_PROCEDURE() AS ErrorProcedure,
           ERROR_LINE() AS ErrorLine,
           ERROR_MESSAGE() AS ErrorMessage
END CATCH
```

Тогава резултатът от изпълнението на горния код е следния набор от данни:

ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine	ErrorMessage
547	16	0	NULL	3	The DELETE statement conflicted with the REFERENCE constraint "FK_ProductInformation_Product". The conflict occurred in database "MyDatabase", table "dbo.ProductInformation", column 'ProductID'.

Пример 4 `TRY ... CATCH` конструкцията не обработва грешки при компилиране (като например синтактичните грешки), освен ако те не са възникнали при изпълнението на динамична заявка или съхранена процедура. Затова резултатът от изпълнението на следния код:

```
BEGIN TRY
    SELECT * FROM NonExistentTable
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber,
           ERROR_MESSAGE() AS ErrorMessage
END CATCH
```

е системното съобщение за грешка, която възниква при използване на несъществуваща таблица:

Msg 208, Level 16, State 1, Line 2

Invalid object name 'NonExistentTable'.

От друга страна динамичната заявка в следващия пример предизвиква грешка, която се обработва от TRY ... CATCH конструкцията:

```
BEGIN TRY
    DECLARE @sql varchar(2000)
    SET @sql = 'SELECT * FROM NonExistentTable'
    EXEC (@sql)
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber,
           ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

Резултатът от изпълнението на последния код е:

```
ErrorNumber ErrorMessage
```

```
-----
208          Invalid object name 'NonExistentTable'.
```

Извеждане на съобщения

Като другите езици за програмиране и Transact-SQL предоставя възможности за извеждане на съобщения чрез конструкциите PRINT и RAISERROR.

PRINT визуализира символните низове до 8000 символа. Може да се изведе низ, ограден в апострофи или променлива от тип *char*, *varchar*, *nchar*, *nvarchar*, *datetime*, *smalldatetime*. Може да се конкатенират низове и да се използват функции, връщащи низове или дати.

Пример 5 PRINT 'низ-съобщение'

Пример 6 PRINT 'Today is ' + CONVERT(char(30), GetDate())

Разликата между PRINT и SELECT е, че PRINT извежда съобщение, а SELECT извежда резултатен набор (следва съобщението (1 row(s) affected)). За клиентското приложение има голяма разлика между съобщение и набор от данни. PRINT се използва предимно за проверка на грешки в съхранени процедури или пакети. Не се отпечатва нищо чрез PRINT от клиентско приложение с графичен интерфейс.

RAISERROR се използва за връщане на съобщения за грешки. Общият вид на конструкцията е:

```
RAISERROR( {msg_num OR msg_str}, severity_num, state
           [, argument1 [,... ]] )
[WITH option [,... ]]
```

- Трябва да се посочи номерът на грешката или низ за нейното описание. За стойностите на параметъра номер на грешка *msg_num* трябва да се има предвид, че числата в диапазона 0÷50000 се използват от SQL Server и за потребителя са тези над 50000. Максималната възможна стойност е $2^{31}-1$. Допустимите стойности на параметъра низ за описание на грешката *msg_str* са символните низове до 2047 символа.
- Необходимо е да се посочи и число 0÷25, което представлява степен на сериозност (строгост) на грешката. За стойностите на *severity_num* трябва да се има предвид, че числата 19÷25 се използват от системните администратори; 0÷18 се използват за по-малко сериозни грешки от потребителя; 10, 17, 18, 19, 20, 21, 22, 23, 24 са запазени за определени цели. Типът на съобщението в дневника на събития на Windows зависи от строгостта: съобщение със строгост 14 и по-ниска са информационни съобщения; със строгост 15 – предупреждения; с 16 и по-висока са грешки.

- *state* е произволно число от 1 до 127, обозначаващо състоянието на системата в момента на възникване на грешката. Тъй като това число е произволно, потребителят може сам да определи какво ще означава то. Например може да се предаде номера на реда, който показва къде се е получила грешката. Но за SQL Server *state* няма действително значение.
- Възможно е да се включат следните опции:
 - LOG – записва кода на грешката в *log* файл за грешки на SQL Server и *log* файл за приложения на Windows;
 - SETERROR – задава стойност на @@ERROR с посочения номер на грешката. Ако не е подаден номер на грешка, грешката се поражда с номер 50000 и това е стойността на функцията @@ERROR;
 - NOWAIT – съобщението се изпраща към клиента без забавяне.

Например:

```
RAISERROR ('Message', 1, 1)
RAISERROR ('Error', 16, 1)
RAISERROR (50001, 1, 1) WITH SETERROR
```

- За допълнителна информация в низа се използват параметри за заместване.

Например:

```
RAISERROR('Идентификаторът на клиента е %d.', 11, 1, 87)
```

Извежда се като резултат:

```
Server: Msg 50000, Level 11, State 1, Line 1
```

Идентификаторът на клиента е 87.

След основната част от конструкцията се записват във вид на списък със запетаи стойностите, с които трябва да бъдат заместени параметрите. Знакът процент % показва, че следващият го знак е позицията, която трябва да се заеме от заместващата стойност, както и типа данни на заместващата стойност: d или i – цяло число със знак; o – осмично число без знак; p – указател; s – символен низ; u – цяло число без знак; x или X – шестнадесетично число без знак.

Пример 7 Извеждане на съобщение за налично количество на даден по идентификатора си продукт:

```
DECLARE @s decimal(10, 3), @sn varchar(40),
        @st varchar(10)
```

```
SELECT @sn = ProductName, @s = Stock
FROM Products
WHERE ProductID = 1
```

```
SET @st = LTRIM(STR(@s, 10, 3))
```

```
IF @sn IS NOT NULL
```

```
    RAISERROR ('The %s's stock is %s.', 11, 1, @sn, @st)
ELSE RAISERROR('The product doesn't exist.', 16, 1)
```

Резултатът, който се извежда, ако продукт с идентификатор 1 съществува, има вида:

```
Server: Msg 50000, Level 11, State 1, Line 10
The product_name's stock is 318.000.
```

Пример 8 Извеждане на съобщение за идентификатора на даден по наименованието си клиент:

```
DECLARE @i int
```

```

SELECT @i = CustomerID
FROM Customers
WHERE CompanyName = 'name'

IF @i IS NOT NULL
    RAISERROR ('Идентификаторът на клиента е %d.',
              11, 1, @i)
ELSE RAISERROR('Не съществува такъв клиент.', 16, 1)

```

За всеки параметър на RAISERROR може да се използва променлива с подходящо зададени тип и стойност.

Ако дадено съобщение се извежда често, е по-ефективно, то да бъде добавено в таблицата *sysmessages* на системната база от данни *master* и да се прави обръщение към него, като се използва уникалният му идентификатор.

Добавяне на дефинирано от потребителя съобщение за грешка

За добавяне на дефинирано от потребителя съобщение за грешка може да се използва системната съхранена процедура *sp_addmessage*:

```

sp_addmessage [@msgnum =] msg_num,
              [@severity =] severity,
              [@msgtext =] 'msg_text'
              [, [@lang =] 'language']
              [, [@with_log =] 'with_log']
              [, [@replace =] 'replace']

```

Информацията за новото съобщение за грешка се добавя в таблицата *sysmessages* от системната база от данни *master*. Параметрите на *sp_addmessage* са:

- *[@msgnum =] msg_num* е идентификатор на съобщението. Комбинацията *msg_num* и *language* трябва да е уникална. Допустимите стойности на *msg_num* за потребителски дефинирани съобщения са цели числа по-големи или равни на 50001.
- *[@severity =] severity* е нивото на строгост.
- *[@msgtext =] 'msg_text'* е текста на съобщението за грешка. *msg_text* е от тип *nvarchar(255)*.
- *[@lang =] 'language'* е езика за това съобщение. Ако се пропусне, езикът е подразбиращия се за сесията.
- *[@with_log =] 'with_log'* определя дали съобщението да бъде записано в Microsoft Windows log за приложения, когато възникне грешката. Възможните стойности са FALSE (по подразбиране) или TRUE. Ако стойността на параметъра е FALSE, грешката се вписва в зависимост от начина, по който е възникнала; ако стойността на параметъра е TRUE – винаги. Тази опция се задава само от членове на *sysadmin*.
- *[@replace =] 'replace'* е низа *replace* или *NULL* (по подразбиране). Ако е зададена стойност *replace*, съществуващото съобщение за грешка се припокрива с новия текст на съобщението и степен на строгост. Тази стойност на параметъра трябва да бъде определена, ако вече съществува съобщение със същия идентификатор *msg_num*. Ако се замести *us_english* съобщение, степента на строгост се замества за всички съобщения на всички други езици, които имат същия идентификационен номер *msg_num*; ако се замества

съобщение на някакъв друг език, трябва да се използва старата степен на строгост и се замества само текста.

Пример 9 Добавяне на ново съобщение за грешка с идентификатор 50001, степен на строгост 16, зададен текст и подразбиращи се стойности за останалите параметри:

```
EXEC sp_addmessage 50001, 16,  
    'Текстът на съобщението за грешка'
```

Пример 10 Първо трябва да се създаде английската версия на съобщението, за да могат да се добавят версии на същото съобщение на други езици със същия идентификатор и същата степен на строгост:

```
EXEC sp_addmessage @msgnum = 60001,  
    @severity = 16,  
    @msgtext = N'The item named %s already exists in %s.',  
    @lang = 'us_english'  
/* Преди началото на всеки низ може да се постави N. По този начин се  
определя, че данните, които следват символа N, са Unicode данни. */
```

```
EXEC sp_addmessage @msgnum = 60001, @severity = 16,  
    @msgtext = N'L'élément nommé %1! existe déjà dans %2!.',  
    @lang = 'French'
```

```
EXEC sp_addmessage @msgnum = 60001, @severity = 16,  
    @msgtext = N'Елемент с име %1! вече съществува в %2!.',  
    @lang = 'bulgarian'
```

За определяне на мястото на параметрите за заместване се използват числа, съответстващи на параметрите в оригиналното съобщение по последователност. Добавя се удивителен знак (!) след всеки номер на параметър. Извеждането на добавеното съобщение може да се осъществи по следния начин:

```
RAISERROR(60001, 16, 1, 'MyTable', 'MyDatabase')
```

Коя версия на съобщението ще се изведе, зависи от настройката за текущата конекция на опцията SET LANGUAGE 'language'. **Например** изпълнението на:

```
SET LANGUAGE 'bulgarian'  
RAISERROR(60001, 16, 1, 'MyTable', 'MyDatabase')
```

ще изведе българската версия на съобщението:

```
Changed language setting to български.  
Server: Msg 60001, Level 16, State 1, Line 2  
Елемент с име MyTable вече съществува в MyDatabase.
```

Пример 11 Припокриване на вече съществуващо съобщение – на текста и/или степента на строгост за английската версия; само на текста (като се спази съответствие на параметрите за заместване по брой и тип) за версиите на останалите езици:

```
EXEC sp_addmessage @msgnum = 60001,  
    @severity = 11,  
    @msgtext = N'New message text',  
    @lang = 'us_english', @replace = 'replace'
```

```
EXEC sp_addmessage @msgnum = 60001,  
    @severity = 11,  
    @msgtext = N'Нов текст на съобщението',
```

```
@lang = 'bulgarian', @replace = 'replace'
```

Функциите @@ERROR и @@ROWCOUNT

Функцията @@ERROR представлява номера на грешката, генерирана от последната конструкция в текущата конекция. Стойността на функцията е 0, ако не е генерирана грешка. Например:

```
DECLARE @error int
```

```
/* Генерира се грешка, ако не съществува SaleID със стойност 999999 в Sales */
```

```
INSERT INTO SaleDetails
```

```
        (SaleID, ProductID, Price, Quantity, Discount)
```

```
VALUES (999999, 11, 10.00, 10, 0)
```

```
/* @@ERROR се установява на кода на грешката, получена от изпълнението на тази конструкция. */
```

```
-- Кодът на грешката се съхранява в променлива.
```

```
SELECT @error = @@ERROR
```

```
-- Стойността, запазена в променливата, е очакваната стойност.
```

```
PRINT 'The value of @error is ' +
```

```
        CONVERT(varchar(10), @error)
```

```
/* Стойността на @@ERROR е променена. След изпълнението на конструкцията PRINT тя е върната на 0. */
```

```
PRINT 'The value of @@ERROR is ' +
```

```
        CONVERT(varchar(10), @@ERROR)
```

Функцията @@ROWCOUNT представлява броя на избраните или засегнатите редове от последната конструкция в текущата конекция. Например:

```
SELECT * FROM Customers
```

```
SELECT @@ROWCOUNT
```

```
SELECT @@ROWCOUNT
```

Първата конструкция SELECT ще изведе списък с данните за клиентите; втората – броя редове в таблицата Customers, изведени с предишната конструкция; третата – 1, тъй като предишната команда извежда един ред, съдържащ броя на клиентите.

Задачи

Задача 1. Да се напише код за извеждане на съобщение за доставната цена на продукт с идентификатор 10.

Задача 2. Да се определи резултата от изпълнението на следния код:

2.1.

```
DECLARE @string varchar(20)
```

```
SET @string = 'SearchDatabase.com'
```

```
SELECT LEFT(@string, 6) AS FirstSix,
```

```
        RIGHT(@string, 4) AS LastFour
```

2.2.

```
DECLARE @string varchar(25)
```



```
SELECT @string = '      SearchDatabase.com      '
SELECT LTRIM(@string) AS NoLeadingSpaces,
       RTRIM(@string) AS NoTrailingSpaces
```

2.3.

```
DECLARE @string varchar(20)
SELECT @string = 'SearchDatabase.com'
SELECT SUBSTRING(@string, 7, 8) AS MiddlePortion
```

2.4.

```
DECLARE @string varchar(20)
SELECT @string = 'SearchDatabase.com'
SELECT SUBSTRING(@string, CHARINDEX('d', @string),
               LEN(@string) - (CHARINDEX('d', @string) - 1) -
               CHARINDEX('.', REVERSE(@string))) AS MiddlePortion
```

Задача 3. Да се напише код, чрез който да се определи броя на думите в символен низ, съдържащ думи, разделени със запетая, например *'sqlserver,html,xml,access,adp'*.

Задача 4. Да се напише код, чрез който да се отделят думите в символен низ, съдържащ разделител запетая, например *'sqlserver,html,xml,access,adp'*.

Задача 5. Да се напише код за извеждане на символен низ по такъв начин, че всички думи в него да започват с главни букви.

Работа с XML данни

SQL Server 2005 осигурява поддръжка на XML (*eXtensible Markup Language* – *разширяем маркиращ език*) данни в съответствие със спецификацията SQLXML (*SQL Standard XML-Related Specifications*), която определя как една релационна база от данни да управлява XML данни.

XML данните са организирани йерархично като дървовидна структура. Приложенията, използващи XML, изпълняват различни действия с данните като създаване на нов XML документ; филтриране на XML документ и извличане на върхове на дървовидната структура на XML документ, отговарящи на зададени условия; преобразуване на XML фрагмент в друга XML структура; актуализиране или модифициране на текущите данни в XML структура.

Съхраняване на XML данни

Съхраняването на XML данни предлага някои предимства:

- XML е самоописателен, поради което приложенията могат да използват XML данни, без да имат информация за тяхната схема или структура.
- Декларирането на схемата на XML данните осигурява информация за валидиране на типа и структурата на данните. Езикът XML Schema е стандартен език, който се използва за дефиниране на валидна структура за определен XML документ или фрагмент. Освен това XML схемите предоставят информация за типа на данните в XML структурата. Чрез XML Schema е възможно да се декларират опционални части в схемата или общи типове, които допускат всякакви XML фрагменти. По този начин могат да се представят не само структурирани данни, но и полуструктурирани, и неструктурирани данни.
- Възможно е да се извършва търсене на XML данни. Тъй като структурата на XML данните е йерархична, могат да бъдат прилагани различни алгоритми за търсене в дървовидната структура. XQuery и XPath са езици за заявки, предназначени за търсене в XML данни.
- XML данните са разширяеми. Възможно е манипулиране на XML данни чрез добавяне, модифициране, изтриване на върхове в дървото.

SQL Server 2005 осигурява два начина за съхраняване на XML данните – в текстова колона или в колона от тип XML на база от данни.

Съхраняване на XML в текстова колона

XML данни могат се съхраняват в текстова колона чрез използване на типовете данни *char(n)*, *nchar(n)*, *varchar(n)*, *nvarchar(n)*. Типовете данни *char(max)*, *nchar(max)*, *varchar(max)*, *nvarchar(max)*, въведени в SQL Server 2005, допускат максимален размер за съхраняваните данни 2 GB.

Пример 1 Следният код съхранява XML данни в колона от тип *varchar(max)*:

```
DECLARE @myXML AS varchar(max)
CREATE TABLE log_table
( log_tableID int NOT NULL IDENTITY,
  log_information varchar(max) NOT NULL )

SET @myXML =
  '<log>
    <application>Sales</application>
    <description>Can not connect to SQL_Server_2005.
    </description>
```

```
</log>'
```

```
INSERT INTO log_table (log_information)
VALUES (@myXML)
```

```
SELECT * FROM log_table
```

Недостатъците от съхраняването на XML данните в текстова колона са: търсенето на XML данни изисква четене на целия XML документ, тъй като XML се интерпретира като текст от сървъра на базата от данни; необходим е допълнителен програмен код, реализиращ валидирането на XML документа, както и операциите за манипулиране на XML данните.

Съхраняване на XML в колона от тип XML

В SQL Server 2005 XML данните могат се съхраняват в колона от тип XML.

Пример 2 Следният код съхранява XML данни в колона от тип XML:

```
DECLARE @myXML AS xml
CREATE TABLE log_table
( log_tableID int NOT NULL IDENTITY,
  log_information xml NOT NULL )

SET @myXML =
  '<log>
    <application>Sales</application>
    <description>Can not connect to SQL_Server_2005.
    </description>
  </log>'
```

```
INSERT INTO log_table (log_information)
VALUES (@myXML)
```

```
SELECT * FROM log_table
```

Основното предимство от съхраняването на XML данните чрез използване на колона от тип XML се състои в това, че в SQL Server 2005 се поддържат възможности за извличане, добавяне, модифициране и изтриване на върхове в XML документа. Съществуват и някои ограничения при този начин на съхраняване на XML данни: празните интервали, XML декларацията в началото на документа, коментарите в XML, наредбата на атрибутите не се запазват; максималната дълбочина на върховете е 128 нива; максималният размер на съхраняваните данни е 2 GB.

В SQL Server 2005 е възможно дефиниране на XML схема.

Пример 3 Създаване на XML схема:

```
CREATE XML SCHEMA COLLECTION LogRecordSchema
AS
  '<schema xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="log">
      <complexType>
        <sequence>
          <element name="application" type="string"/>
          <element name="description" type="string"/>
        </sequence>
      </complexType>
```

```
</element>
</schema>'
```

В следващия примерен код SQL Server валидира съдържанието на променливата @myXML според правилата, зададени в XML схемата LogRecordSchema:

```
DECLARE @myXML AS xml(LogRecordSchema)
SET @myXML = '<log>Can not connect to MyHome_SQL_Server.</log>'
```

XML данните не отговарят на XML структурата, декларирана в схемата, поради което горният код генерира следното съобщение за грешка:

```
Msg 6909, Level 16, State 1, Line 2
```

```
XML Validation: Text node is not allowed at this location, the
type was defined with element only content or with simple
content. Location: /*:log[1]
```

Пример 4 XML схемата може да бъде заредена от файл с разширение *.xsd*. За целта се използва командата OPENROWSET в SQL Server 2005:

```
DECLARE @schema xml
DECLARE @t table (c xml)

INSERT INTO @t (c)
SELECT *
FROM OPENROWSET (BULK 'MyXMLSchema.xsd', SINGLE_BLOB) AS c

SELECT @schema = c FROM @t
```

```
CREATE XML SCHEMA COLLECTION MySchema1 AS @schema
```

Командата BULK зарежда съдържанието на файла. Опцията SINGLE_BLOB в OPENROWSET гарантира, че XML парсерът в SQL Server импортира данните според кодировката на схемата, зададена в XML декларацията.

Извличане на XML данни

SQL Server 2005 предлага множество възможности за извличане на XML данни при различните начини за представяне на данните – в релационен модел, в текстова колона, или в колона от тип XML.

Преобразуване на релационни данни в XML данни

SQL Server 2005 позволява преобразуване на релационни данни в XML данни чрез използване на опцията FOR XML в конструкцията SELECT. Опцията FOR XML преобразува наборът от данни, получен от изпълнението на дадена заявка в XML структура и осигурява различни начини за форматиране: FOR XML RAW, FOR XML AUTO, FOR XML PATH, FOR XML EXPLICIT.

Общият вид на опцията **FOR XML RAW** е:

```
FOR XML RAW [ ( 'ElementName' ) ]
[ [, XMLSCHEMA] [, ELEMENTS] ]
[ , ROOT [ ( 'RootName' ) ] ]
```

По подразбиране опцията FOR XML RAW създава нов XML елемент <row> за всеки ред в набора от данни, получен като резултат от изпълнението на конструкцията SELECT. Освен това се добавя XML атрибут на елемента <row> за всяка колона, избрана в конструкцията SELECT, като се използва името на колоната за име на атрибута.

За да се преименува елементът <row>, може да се зададе ново наименование на тага *ElementName* след ключовата дума RAW. За да се създаде по един елемент за всяка колона вместо атрибут, се използва ключовата дума ELEMENTS след FOR XML RAW. Коренен елемент в XML структурата се добавя, като се използва ключовата дума ROOT, като в скоби може да се зададе неговото наименование. При използване на опцията FOR XML RAW всички колони се представят по един и същи начин (като елементи или атрибути) и се създава йерархия от едно ниво.

Пример 5 Заявка за извличане на данни за служителите, подредени по дата на назначаване в магазин:

```
SELECT s.StoreID, e.EmployeeID, e.HireDate, s.StoreName,
       DATEDIFF(year, e.HireDate, GetDate()) AS YearsToDate
FROM Stores s
INNER JOIN Employees e ON s.StoreID = e.StoreID
WHERE TerminationDate IS NULL
ORDER BY s.StoreID ASC, e.HireDate ASC
FOR XML RAW('OldestEmployeeByStore'), ELEMENTS
```

Примерен резултат от изпълнението на последната заявка е:

```
<OldestEmployeeByStore>
  <StoreID>1</StoreID>
  <EmployeeID>3</EmployeeID>
  <HireDate>1999-01-05T00:00:00</HireDate>
  <StoreName>StoreName1</StoreName>
  <YearsToDate>10</YearsToDate>
</OldestEmployeeByStore>
<OldestEmployeeByStore>
  <StoreID>1</StoreID>
  <EmployeeID>5</EmployeeID>
  <HireDate>1999-01-05T00:00:00</HireDate>
  <StoreName>StoreName1</StoreName>
  <YearsToDate>10</YearsToDate>
</OldestEmployeeByStore>
```

Чрез опцията **FOR XML AUTO** се създава вложена XML структура. За всяка таблица, избрана в SELECT заявката, се генерира ново ниво в XML структурата. Редът на влагане на XML данните се определя от реда, по който колоните са изброени в SELECT конструкцията. Подобно на XML RAW, по подразбиране се създават само атрибути за колоните в заявката. За да се добавят елементи за колоните, се прилага ключовата дума ELEMENTS след XML AUTO. Коренен елемент в XML структурата също се добавя с помощта на ключовата дума ROOT. При използване на опцията FOR XML AUTO всички колони се представят по един и същи начин (като елементи или атрибути) и се създава йерархия от толкова нива, колкото са таблиците, избрани в SELECT заявката. Първото ниво на йерархията съответства на таблицата, в която се намира първата колона, изброена в SELECT конструкцията и т.н. Например:

```
SELECT store.StoreID, employee.EmployeeID,
       employee.HireDate, store.StoreName,
       DATEDIFF(year, employee.HireDate, GetDate()) AS YearsToDate
FROM Stores store
INNER JOIN Employees employee
       ON store.StoreID = employee.StoreID
WHERE TerminationDate IS NULL
ORDER BY store.StoreID ASC, employee.HireDate ASC
```

FOR XML AUTO, ELEMENTS

Примерен резултат от изпълнението на последната заявка е:

```
<store>
  <StoreID>1</StoreID>
  <StoreName>StoreName1</StoreName>
  <employee>
    <EmployeeID>3</EmployeeID>
    <HireDate>1999-01-05T00:00:00</HireDate>
    <YearsToDate>10</YearsToDate>
  </employee>
  <employee>
    <EmployeeID>5</EmployeeID>
    <HireDate>1999-01-05T00:00:00</HireDate>
    <YearsToDate>10</YearsToDate>
  </employee>
</store>
```

Чрез **FOR XML PATH** е възможно изцяло да се контролира XML структурата, която се генерира. За всяка колона се задава псевдоним (*alias*), който определя местоположението на съответния връх в XML йерархията. Ако дадена колона няма дефиниран псевдоним, се използва подразбиращият се връх <row>. Декларирането на псевдоними се осъществява чрез XPath изрази. Някои опции за конфигуриране на колони във **FOR XML PATH** са:

- '*elementName*' – създаване на XML елемент <elementName> със съдържанието на колоната в текущия връх;
- '@*attributeName*' – създаване на XML атрибут attributeName със съдържанието на колоната в текущия връх;
- '*elementName/nestedElement*' – създаване на XML елемент <nestedElement>, вложен в елемента <elementName> със съдържанието на колоната;
- '*elementName/@attributeName*' – създаване на XML атрибут attributeName на елемента <elementName> със съдържанието на колоната.

Например:

```
SELECT s.StoreID AS 'Store/@StoreID',
       s.StoreName AS 'Store',
       e.EmployeeID AS 'Store/Employee/@EmployeeID',
       e.HireDate AS 'Store/Employee/HireDate',
       DATEDIFF(year, e.HireDate, GetDate()) AS
           'Store/Employee/YearsToDate'

FROM Stores s
INNER JOIN Employees e ON s.StoreID = e.StoreID
WHERE TerminationDate IS NULL
ORDER BY s.StoreID ASC, e.HireDate ASC
FOR XML PATH('OldestEmployeeByStore'), ROOT ('QueryResult')
```

Примерен резултат от изпълнението на последната заявка е:

```
<QueryResult>
  <OldestEmployeeByStore>
    <Store StoreID="1">StoreName1
      <Employee EmployeeID="3">
        <HireDate>1999-01-05T00:00:00</HireDate>
        <YearsToDate>10</YearsToDate>
      </Employee>
    
```

```

    </Store>
</OldestEmployeeByStore>
<OldestEmployeeByStore>
  <Store StoreID="1">StoreName1
    <Employee EmployeeID="5">
      <HireDate>1999-01-05T00:00:00</HireDate>
      <YearsToDate>10</YearsToDate>
    </Employee>
  </Store>
</OldestEmployeeByStore>
</QueryResult>

```

Чрез **FOR XML EXPLICIT** може да се контролира структурата на генерираните XML данни, като се следва шаблон, наречен универсална таблица (*universal table*). Тази универсална таблица изисква дефиниране на колони и техните псевдоними, форматирувани по определен шаблон:

- колона *Tag* – първата колона в набора от данни, задаваща дълбочината на XML структурата;
- колона *Parent* – втората колона в набора от данни, определяща родителския връх в XML структурата;
- шаблон на име на колона *ElementName!TagName!AttributeName!Directive* – колоните трябва да имат зададен псевдоним, съответстващ на този шаблон: *ElementName* е името на съответния елемент; *TagName* е нивото, на което върхът да бъде разположен; *AttributeName* е име на атрибут; *Directive* предоставя допълнителна информация за форматирането. Възможните стойности на *Directive* са: *hide* (колоната не се включва в получената XML структура); *element* (създава се XML елемент, като NULL стойностите се игнорират); *elementxsinil* (създава се XML елемент, като NULL стойностите не се игнорират); *cdata* (стойността на колоната се добавя като коментар).

Например:

```

SELECT 1 AS Tag,
      NULL AS Parent,
      s.StoreID AS [Store!1!StoreID],
      s.StoreName AS [Store!1!StoreName!ELEMENT],
      NULL AS [Employee!2!EmployeeID],
      NULL AS [Employee!2!HireDate!ELEMENT],
      NULL AS [Employee!2!YearsToDate!ELEMENT]
FROM Stores s
UNION ALL
SELECT 2 AS Tag,
      1 AS Parent,
      s.StoreID,
      s.StoreName,
      e.EmployeeID,
      e.HireDate,
      DATEDIFF(year, e.HireDate, GetDate())
FROM Stores s
INNER JOIN Employees e ON s.StoreID = e.StoreID
WHERE TerminationDate IS NULL
ORDER BY [Store!1!StoreID] ASC,
        [Employee!2!HireDate!ELEMENT] ASC
FOR XML EXPLICIT, ROOT ('QueryResult')

```

Примерен резултат от изпълнението на последната заявка е:

```
<QueryResult>
  <Store StoreID="1">
    <StoreName>StoreName1</StoreName>
    <Employee EmployeeID="3">
      <HireDate>1999-01-05T00:00:00</HireDate>
      <YearsToDate>10</YearsToDate>
    </Employee>
    <Employee EmployeeID="5">
      <HireDate>1999-01-05T00:00:00</HireDate>
      <YearsToDate>10</YearsToDate>
    </Employee>
  </Store>
</QueryResult>
```

Извличане на XML данни от колони от тип XML

Съществуват два езика за заявки XQuery и XPath, разработени от W3C. Комбинираното им използване осигурява възможности за манипулиране на XML структури.

XPath изразите се използват за избиране на върхове в XML документ. Най-често използваните изрази са:

- `nodename` – всички наследници на върха `nodename`;
- `/` – коренния връх;
- `//` – всички върхове в документа от текущия;
- `.` – текущия връх;
- `..` – родителския връх на текущия връх;
- `@` – атрибут;
- `*` – произволен елемент;
- `@*` – произволен атрибут;
- `node()` – произволен връх (елемент или атрибут).
- `|` – няколко набора от върхове;

В XPath изразите могат да се включват функции, най-често използваните от които са:

- `name()` – името на даден връх;
- `text()` – съдържанието на даден връх;
- `position()` – позицията на даден връх в набор от върхове;
- `last()` – позицията на последния връх в набор от върхове;
- `count(arg)` – броя на върховете в даден набор;
- `sum(arg)` – сумата от числовите стойности в набор от върхове;
- `number(arg)` – преобразува текст в числова стойност;
- `string(arg)` – преобразува числова стойност в низ;
- `string-length(string)` – дължината на даден низ;
- `contains(string1, string2)` – връща като резултат стойност *true*, ако *string2* се съдържа в *string1* и *false* в противен случай;
- `substring(string, start, len)` – подниз на *string* от позиция *start* с дължина *len*.

XQuery е език за заявки, проектиран за извличане на XML данни. Най-често използвани в XQuery са FLWOR изразите (For, Let, Where, Order by, Return):

- `for` – асоциира една или повече променливи с изрази;

- `let` – асоциира променлива с резултата от израз. Не се поддържа от SQL Server 2005;
- `where` – задава критерий за филтриране;
- `order by` – задава критерий за сортиране;
- `return` – определя какъв резултат да бъде върнат от FLWOR изрази.

В XQuery заявките се използват XPath изрази за избиране на върхове в XML структура.

Пример 6 Нека е даден XML документ с данни за книги. Следващият пример съдържа FLWOR израз, който връща като резултат заглавията на книгите, издадени през 2000 година.

```
DECLARE @booksXML xml
SET @booksXML =
'<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last><first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the UNIX Environment
    </title>
    <author>
      <last>Stevens</last><first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author>
      <last>Abiteboul</last><first>Serge</first>
    </author>
    <author>
      <last>Buneman</last><first>Peter</first>
    </author>
    <author>
      <last>Suciu</last><first>Dan</first>
    </author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>65.95</price>
  </book>
  <book year="1999">
    <title>
      The Economics of Technology and Content for Digital TV
    </title>
    <editor><last>Gerbarg</last><first>Darcy</first>
    <affiliation>CITI</affiliation></editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>'
```

```
SELECT @booksXML.query('for $b in /bib/book
                        where $b/@year = "2000"
                        return $b/title')
```

```
FOR XML PATH(''), ROOT
```

В следващите примери от тази тема е използван същия XML документ, но присвояването на стойността на променливата @booksXML е пропуснато.

Пример 7 Извеждане на заглавията на книгите с цени над 100:

```
SELECT @booksXML.query('for $b in /bib/book
                        where $b/price > 100
                        return $b/title')
```

```
FOR XML PATH(''), ROOT
```

Пример 8 Сортиране на заглавията по азбучен ред:

```
SELECT @booksXML.query('for $b in /bib/book/title
                        order by $b
                        return $b')
```

```
FOR XML PATH(''), ROOT
```

Пример 9 Сортиране на заглавията по година на издаване в низходящ ред:

```
SELECT @booksXML.query('for $b in /bib/book
                        order by $b/@year descending
                        return $b/title')
```

```
FOR XML PATH(''), ROOT
```

Пример 10 Пресмятане на броя на авторите за всяко заглавие:

```
SELECT @booksXML.query(
    'for $b in /bib/book
    return <book>{$b/title,
                <count>{count($b/author)}</count>}
    </book>')
```

```
FOR XML PATH(''), ROOT
```

Пример 11 Извличане на заглавията на книгите по издателства:

```
SELECT @booksXML.query(
    'for $p in distinct-values(//publisher)
    order by $p
    return <result> { $p }
    {
        for $b in /bib/book
        where $b/publisher = $p
        return $b/title
    }
    </result>')
```

```
FOR XML PATH(''), ROOT
```

Функцията `distinct-values` извлича стойностите на набор от върхове, като отстранява дублиращите се.

Пример 12 XQuery заявките позволяват използване на If-Then-Else изрази. В следващия пример се проверява дали заглавието на книгата съдържа като подниз "Web" и създава елемент <web>, в противен случай създава елемент <other>:

```
SELECT @booksXML.query(
    'for $x in /bib/book/title
    return if (contains($x, "Web"))
        then <web>{data($x)}</web>
        else <other>{data($x)}</other>')
FOR XML PATH(''), ROOT
```

Пример 13 Увеличаване на цените на книгите с 10%:

```
DECLARE @new xml
SELECT @new = @booksXML.query('
    for $i in /bib/book
    return <r> {
        for $b in $i/price
        return <book>{ $i/@year, $i/title, $i/author,
            $i/publisher, $i/editor,
            <price>{$b*1.1}</price>}
        </book>
    } </r> ')
SELECT @new.query ('<bib>{
    for $b in /r/book
    return $b
} </bib> ')
```

Модифициране на XML данни

SQL Server 2005 предоставя възможност XML данните, съхранявани в колона или променлива от тип XML, да бъдат модифицирани с помощта на метода `modify()`.

Езикът XQuery, дефиниран от W3C, не предлага команди за модифициране на данни. Microsoft SQL Server 2005 поддържа разширение на XQuery, в което са включени команди за добавяне, актуализиране, изтриване на XML данни (XML DML – *XML Data Modification Language*). Методът `modify()`, приложим за данните от тип XML, приема един входен параметър, който трябва да бъде валиден XML DML израз. Езикът XML DML, предоставен от SQL Server 2005, поддържа следните ключови думи:

- `insert` – добавяне на един или повече върхове на същото ниво на даден връх или като негови наследници. Конструкцията `insert` се състои от два изрази и един оператор. Първият израз трябва да връща като резултат един връх или множество от върхове; вторият израз трябва да връща един връх. Двата изрази могат да представляват константни стойности или XQuery изрази. Операторът свързва двата изрази и може да бъде един от следните:
 - `into` – добавяне на върховете като наследници на върха, определен с втория израз. Ако върхът вече има наследници, XML DML изразът може да уточни дали новите върхове да бъдат добавени като първи (чрез `first into`) или като последни (чрез `last into`, което е по подразбиране).
 - `after` – добавяне на върховете след върха, определен от втория израз, така че да са на едно и също ниво.
 - `before` – добавяне на върховете преди върха, определен от втория израз, така че да са на едно и също ниво.

- `replace value of` – променяне на стойността на даден връх. Конструкцията `replace` съдържа два израза. Първият израз трябва да представлява един връх; вторият израз може да бъде съставен, като се използва константна стойност или XQuery израз, който да връща като резултат стойност от прост тип.
- `delete` – изтриване на един или повече върхове от XML структурата. Конструкцията `delete` се състои от един XQuery израз, връщащ като резултат множество от върхове, които трябва да бъдат изтрети от XML структурата.

Пример 14 Добавяне на нов елемент преди първия елемент `book`:

```
SET @booksXML.modify('
  insert <book year="2006">
    <title>Databases</title>
  </book>
  before (/bib/book)[1]')
```

Пример 15 Добавяне на нов автор на книга със заглавие "Data on the Web", т.е. добавяне на нов елемент `author` като наследник на първия елемент `book`, отговарящ на условието за стойността на елемента `title`:

```
SET @booksXML.modify('
  insert <author>
    <last>Ivanov</last><first>Ivan</first>
  </author>
  into (/bib/book[title="Data on the Web"])[1]')
```

Пример 16 Променяне на цената на книга със заглавие "Data on the Web", т.е. на стойността на елемента `price`, отговарящ на зададеното условие:

```
SET @booksXML.modify('
  replace value of
    (/bib/book[title="Data on the Web"]/price/text())[1]
  with "55"')
```

Пример 17 Увеличаване на цената на книга със заглавие "Data on the Web" с 10%:

```
SET @booksXML.modify('
  replace value of
    (/bib/book[title="Data on the Web"]/price/text())[1]
  with
    (/bib/book[title="Data on the Web"]/price/text())[1]*1.1
  ')
```

Пример 18 Изтриване на книга със заглавие "Data on the Web", т.е. на елемент `book`, отговарящ на зададеното условие:

```
SET @booksXML.modify('
  delete /bib/book[title="Data on the Web"]')
```

Преобразуване на XML данни в релационни

В SQL Server 2005 преобразуването на XML данни в релационни се извършва чрез използване на конструкцията `OPENXML` и метода `nodes()` на данните от тип XML.

Използване на OPENXML

Конструкцията **OPENXML** се прилага заедно със съхранените процедури **sp_xml_preparedocument** и **sp_xml_removedocument**, които служат съответно за зареждане и освобождаване на документа в паметта. Общият вид на конструкцията **OPENXML** е:

```
OPENXML ( idoc, row_pattern, [ flags ] )
[ WITH ( SchemaDeclaration ) ]
```

където

- параметърът *idoc* е указател към вътрешното представяне на XML документа, което се създава при изпълнението на съхранената процедура **sp_xml_preparedocument**;
- параметърът *row_pattern* е XPath израз, предназначен за определяне на върховете, които трябва да бъдат преобразувани в редове;
- параметърът *flags* задава съответствието между XML данните и релационната структура. Възможните стойности са:
 - 1 – всеки XML атрибут се преобразува в колона;
 - 2 – всеки XML елемент се преобразува в колона;
- параметърът *SchemaDeclaration* служи за деклариране на релационната схема.

Пример 19 Неявно преобразуване на XML данните в релационни:

```
DECLARE @h int
EXEC sp_xml_preparedocument @h OUTPUT, @booksXML
```

```
SELECT * FROM OPENXML( @h , '/bib/book', 1)
WITH (
    year char(4),
    title varchar(30),
    price money
)
EXEC sp_xml_removedocument @h
```

При неявното преобразуване, за да се определят колоните в получената релационна структура, се използват наименованията на елементите и атрибутите в XML структурата, като се прави разлика между малки и главни букви.

Пример 20 Явно преобразуване на XML данните в релационни:

```
DECLARE @h int
EXEC sp_xml_preparedocument @h OUTPUT, @booksXML
```

```
SELECT * FROM OPENXML( @h , '/bib/book')
WITH (
    BookTitle varchar(30) 'title',
    YearOfPublishing int '@year',
    Publisher varchar(30) 'publisher',
    BookPrice money 'price'
)

EXEC sp_xml_removedocument @h
```

Използване на метода `nodes()` на данните от тип XML

Методът **nodes()** връща като резултат табличен набор от данни, който съдържа една колона от тип XML. Нов ред се генерира за всеки XML връх, който

съответства на зададения XPath израз. Например, в резултат от изпълнението на следната заявка ще се получи таблица с една колона `result`, а всеки ред ще съдържа по един елемент `book`:

```
SELECT c.query('.') AS result
FROM @booksXML.nodes('/bib/book') AS T(c)
```

Задачи

Задача 1. Да се напише:

1.1. SELECT заявка, която връща като резултат следната XML структура:

```
<ProductList>
  <product>
    <ProductName>PrName</ProductName>
    <CategoryName>CategoryName</CategoryName>
    <price>10.50</price>
    <stock>10.50</stock>
  </product>
  ...
</ProductList>
```

1.2. XQuery заявка, която от XML документа, получен в 1.1, да извлече наименованията на продуктите и цените им, като избере само продуктите с цени под 10;

1.3. XQuery заявка, която от XML документа, получен в 1.1, да извлече наименованията на продуктите по категории.

Задача 2. Да се напише SELECT заявка, която връща като резултат следната XML структура:

```
<EmployeeList>
  <Employee employeeID="3">
    <FirstName>fName</FirstName>
    <LastName>lName</LastName>
    <Store storeID="1">
      <StoreName>company name</StoreName>
    </Store>
  </Employee>
  ...
</EmployeeList>
```

Задача 3. Да се напише XQuery заявка, която преобразува XML документа, получен от SELECT заявката от задача 2, в следната XML структура:

```
<StoreList>
  <Store>
    <StoreName>company name</StoreName>
    <Employees>
      <Employee>
        <FirstName>fName</FirstName>
        <LastName>lName</LastName>
      </Employee>
      ...
    </Employees>
  </Store>
```

```
...  
</StoreList>
```

Задача 4. Да се напише XQuery заявка, която преобразува XML документа, получен от XQuery заявката от задача 3, в следната XML структура:

```
<StoreList>  
  <Store>  
    <StoreName>company name</StoreName>  
    <NumberOfEmployees>10</NumberOfEmployees>  
  </Store>  
  ...  
</StoreList>
```

Създаване на изгледи

За разлика от *постоянните базови таблици*, които съдържат постоянно данните, съхранявани в базата от данни, изгледите са друг вид таблици, поддържани от SQL Server. *Изгледите (views)* са таблици, чието съдържание се взема или извлича от други таблици. Известни са под името виртуални таблици или изгледни таблици. Може да се направи обръщение към изглед от SELECT, INSERT, UPDATE или DELETE конструкция по същия начин, както се прави обръщение към базовата таблица с тази разлика, че изгледите не съдържат собствени данни. Изгледът представлява заявка, която се извиква за изпълнение винаги, когато в някоя конструкция се извършва препращане към този изглед. Изходните данни от заявката, дефинираща изгледа, формират неговото съдържание.

Създаване, променяне и изтриване на изгледи

Дефинирането на изглед в Microsoft SQL Server 2005 се извършва посредством конструкцията CREATE VIEW, която има следния общ вид:

```
CREATE VIEW view_name [(column_list)]
AS
    SELECT statement
[WITH CHECK OPTION]
```

Пример 1

```
CREATE VIEW All_Employees_And_Customers
AS
    SELECT FirstName+' '+LastName AS Name,
           CompanyName
    FROM Employees e
    LEFT JOIN Sales s
        ON e.EmployeeID = s.EmployeeID
    FULL JOIN Customers c
        ON s.CustomerID = c.CustomerID
```

Тази конструкция не генерира изходни данни, а потвърждение за създаване на нов обект в базата от данни. Създаденият изглед може да се използва като всяка друга таблица. Може да се отправят заявки към него, да се актуализират, вмъкват данни, да се изтриват данни и да бъде съединяван с други таблици и изгледи. Например:

```
SELECT * FROM All_Employees_And_Customers
WHERE Name LIKE 'Ab%'
ORDER BY Name
```

Първо се изпълнява заявката, която се съдържа в дефиницията на изгледа All_Employees_And_Customers, изходните данни от тази заявка стават съдържание на изгледа. След това SQL изпълнява командата SELECT на всички колони от този изглед.

Изглед може да се създаде на базата на която и да е таблица, дори на базата на друг изглед.

Изгледите увеличават значително възможностите за контролиране на данните. Позволяват начин за предоставяне на достъп до ограничена част, вместо до цялата информация в дадена таблица.

Изгледите могат да съдържат GROUP BY или да се базират на други изгледи, които са дефинирани с помощта на GROUP BY.

Пример 2

```
CREATE VIEW TotalForDay
```



```

AS
SELECT CONVERT(datetime, CONVERT(char(10), SaleDate, 112))
    AS DateOfSale,
    COUNT(DISTINCT CustomerID) AS CustomersCount,
    COUNT(DISTINCT EmployeeID) AS EmployeesCount,
    COUNT(SaleID) AS SalesCount,
    AVG(TotalForSale) AS AverageTotal,
    SUM(TotalForSale) AS SumTotal
FROM Sales
GROUP BY CAST(CONVERT(char(10), SaleDate, 112) AS datetime)

```

След създаване на изгледа могат да се изпълняват заявки към изгледа.

Например:

- SELECT * FROM TotalForDay
- SELECT * FROM TotalForDay
WHERE SumTotal=(SELECT MAX(SumTotal) FROM TotalForDay)
- SELECT * FROM TotalForDay
WHERE SalesCount>=2 AND SumTotal>=1000

Възможно е изгледите да се дефинират чрез подзаявки, включително взаимосвързани подзаявки.

Пример 3

```

CREATE VIEW MaxTotalForCustomer
AS
SELECT c.CustomerID, c.CompanyName,
    s.TotalForSale, s.SaleDate
FROM Customers c
INNER JOIN Sales s
    ON c.CustomerID=s.CustomerID
WHERE TotalForSale =
    ( SELECT MAX(TotalForSale)
      FROM Sales s1
      WHERE s.CustomerID = s1.CustomerID )

```

Заявката, дефинираща изгледа, извлича данните за най-голямата покупка на клиентите измежду всичките им собствени покупки. Пример за заявка към изгледа е следната:

```

SELECT * FROM MaxTotalForCustomer
WHERE CompanyName LIKE 'L%'

```

Изгледите могат да се базират на няколко заявки, комбинирани с оператора UNION.

Пример 4

```

CREATE VIEW CityPeople
AS
SELECT CustomerID AS ID, CompanyName, City,
    'customer' AS Name
FROM Customers
UNION
SELECT SupplierID, CompanyName, City, 'supplier'
FROM Suppliers

```

Включват се низове, за да се постави етикет на всеки ред, който показва коя заявка го е генерирала. Следната заявка към изгледа извежда тези редове, за които името на града е в зададения азбучен обхват:

```

SELECT * FROM CityPeople

```

```
WHERE City BETWEEN 'A' AND 'L'
ORDER BY City
```

Не е допустимо използването на ORDER BY в дефиницията на изглед. Според стандарта изходните данни от заявката формират съдържанието на изгледа, което подобно на базовата таблица по дефиниция е неподредено. Заявките към изгледа могат да използват ORDER BY.

Исключение прави изглед, включващ в дефиницията си TOP *n*. Например:

```
CREATE VIEW TopProductPrice
AS
SELECT TOP 3 WITH TIES ProductID, ProductName, Price
FROM Products
ORDER BY Price DESC
GO
SELECT * FROM TopProductPrice
```

Променяне на изгледи

SQL Server позволява да се променя дефиницията на изгледа:

```
ALTER VIEW view_name [(column_list)]
AS
SELECT statement
[WITH CHECK OPTION]
```

Изгледът трябва да съществува, а указаната дефиниция замества тази, която е имал изгледът преди изпълнението на ALTER VIEW.

Изтриване на изгледи

```
DROP VIEW view_name
```

Премахването на изглед не оказва влияние върху базовата таблица, от която той се извлича.

Променяне на данни посредством изгледи

Възможно е модифициране на редове в таблицата, на която е базиран изгледът, като се използва самият изглед. Съществуват някои ограничения при променяне на данни чрез изгледа:

- Модификациите са ограничени до една таблица, т.е. конструкциите INSERT и UPDATE са позволени за изгледи на множество таблици, ако промяната засяга само една таблица. Конструкциите DELETE не са разрешени за изгледи на множество таблици.
- Някои изгледи са само за четене. Не могат да се добавят (INSERT), изтриват (DELETE) редовете или променят (UPDATE) колони от изглед, включващ обобщаващи функции, UNION, INTERSECT, EXCEPT, ключовите думи GROUP BY или DISTINCT, както и колоните в изгледа, които са изчислими колони или резултат от вградени функции.
- По подразбиране конструкциите за добавяне (INSERT) и променяне (UPDATE) на данни чрез изгледи не се проверяват за определяне на това, дали засегнатите редове отговарят на критериите на изгледа. Например конструкцията INSERT за даден изглед може да добавя ред в базовата таблица, но да не добавя ред към самия изглед, тъй като стойностите на колоните не отговарят на критериите на изгледа. В този случай добавеният ред съдържа стойности, валидни за таблицата и могат да бъдат добавени, но не се избират от заявката, дефинираща изгледа.

Ако трябва всички промени да бъдат проверявани, то е необходимо при създаване на изгледа да се използва опцията WITH CHECK OPTION. Включването на опцията WITH CHECK OPTION за изгледи, които са само за четене е безсмислено.

Пример 5 Изглед за данните на клиентите от даден град:

```
CREATE VIEW Customers_VT
AS
    SELECT * FROM Customers
    WHERE City = 'Велико Търново'
```

WITH CHECK OPTION

Поради наличието на опцията WITH CHECK OPTION в дефиницията на изгледа Customers_VT, всички редове, които се добавят или променят чрез изгледа, трябва да отговарят на критерия на изгледа (т.е. на условието City = 'Велико Търново'). Изпълнението на конструкцията от вида:

```
UPDATE Customers_VT
    SET City = 'София'
WHERE CustomerID = 2
```

ще предизвика извеждане на следното съобщение за грешка:

Msg 550, Level 16, State 1, Line 1

The attempted insert or update failed because the target view either specifies WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION and one or more rows resulting from the operation did not qualify under the CHECK OPTION constraint.

Освен това ако в списъка на полетата се пропусне колоната City, изгледът ще може да се актуализира, да се изтриват редове, но в него не може да се вмъкват редове, освен ако не е определена стойност по подразбиране за колоната City да е 'Велико Търново'. Следователно е добре да се включи колоната, а ако информацията в нея не е нужна, тя може да се пропусне в заявката към изгледа:

```
SELECT CustomerID, CompanyName, Address, PhoneNumber
FROM Customers_VT
```

Пример 6 Опцията WITH CHECK OPTION се прилага по отношение на всички изгледи, които съдържат изглед с тази опция.

```
CREATE VIEW ListCustomers_VT
```

```
AS
    SELECT CustomerID, CompanyName, City
    FROM Customers_VT
```

Пример 7 Освен това опцията WITH CHECK OPTION се прилага не само по отношение на условието на изгледа, който директно я съдържа, но и по отношение на други изгледи, които този изглед съдържа.

```
CREATE VIEW Product_List
AS
    SELECT ProductID, ProductName, Price,
           Stock, ReorderLevel, Discontinued
    FROM Products
    WHERE Discontinued = 0
```

Да предположим, че трябва да се създаде друг изглед, базиран на предишния:

```
CREATE VIEW ProductsForReorder
```

```

AS
    SELECT *
    FROM Product_List
    WHERE Stock<=ReorderLevel
WITH CHECK OPTION

```

Няма да може след това да се извърши следната актуализация:

```

UPDATE ProductsForReorder
    SET Discontinued = 1
WHERE ProductID = 87

```

Тя не нарушава непременно условието в изгледа ProductsForReorder, но нарушава условието в изгледа Product_List, на който се базира. Изтриване на ред чрез изглед води до изтриването на ред от съответната таблица:

```

DELETE FROM ProductsForReorder
WHERE ProductID = 12

```

Пример 8

```

CREATE VIEW Suppliers_vw
AS
    SELECT SupplierID, CompanyName, PhoneNumber
    FROM Suppliers
    WHERE PhoneNumber LIKE '(503)%'
WITH CHECK OPTION

```

Добавяне на ред чрез този изглед може да се извърши чрез следната команда:

```

INSERT INTO Suppliers_vw (CompanyName, PhoneNumber)
VALUES ('name', '(503)12345')

```

Пример 9

```

CREATE VIEW CustomerPhoneList_vw
AS
    SELECT CompanyName, ContactName, PhoneNumber
    FROM Customers

```

Пример 10

```

CREATE VIEW CurrentEmployees_vw
AS
    SELECT EmployeeID,
           FirstName, Surname, LastName,
           Title, HireDate, TerminationDate,
           ManagerEmpID, StoreName
    FROM Employees e
    INNER JOIN Stores s ON E.StoreID = S.StoreID
    WHERE TerminationDate IS NULL

```

Пример 11

```

CREATE VIEW Employees_vw
AS
    SELECT EmployeeID,
           FirstName, Surname, LastName,
           Title, HireDate, TerminationDate,
           ManagerEmpID, StoreName
    FROM Employees e

```

```
INNER JOIN Stores s
  ON e.StoreID = s.StoreID
```

Пример 12

```
CREATE VIEW CustomerSales_vw
AS
  SELECT cu.CompanyName, s.SaleDate,
         sd.ProductID, p.ProductName,
         sd.Quantity, sd.Price,
         sd.Quantity * sd.Price * (1 - sd.Discount)
         AS ExtendedPrice
  FROM Customers AS cu
  INNER JOIN Sales AS s
    ON cu.CustomerID = s.CustomerID
  INNER JOIN SaleDetails AS sd
    ON s.SaleID = sd.SaleID
  INNER JOIN Products p
    ON sd.ProductID = p.ProductID
```

Пример 13

```
CREATE VIEW YesterdaySales_vw
AS
  SELECT cu.CompanyName, s.SaleID,
         s.SaleDate, sd.ProductID,
         p.ProductName, sd.Quantity,
         sd.Price,
         sd.Quantity * sd.Price * (1-sd.Discount)
         AS ExtendedPrice
  FROM Customers AS cu
  INNER JOIN Sales AS s
    ON cu.CustomerID = s.CustomerID
  INNER JOIN SaleDetails AS sd
    ON s.SaleID = sd.SaleID
  INNER JOIN Products p
    ON sd.ProductID = p.ProductID
  WHERE CONVERT(varchar(12),s.SaleDate,101) =
         CONVERT(varchar(12),DATEADD(day,-10,GETDATE()),101)
  /* или DATEDIFF(day, SaleDate,
                  DATEADD(day, -10, Getdate())) = 0 */
```

Пример 14

```
CREATE VIEW CustomerSales_vw1
  WITH ENCRYPTION
AS
  SELECT cu.CompanyName, s.SaleDate,
         sd.ProductID, p.ProductName,
         sd.Quantity, sd.Price,
         sd.Quantity * sd.Price * (1 - sd.Discount)
         AS ExtendedPrice
  FROM Customers AS cu
  INNER JOIN Sales AS s
```

```

        ON cu.CustomerID = s.CustomerID
    INNER JOIN SaleDetails AS sd
        ON s.SaleID = sd.SaleID
    INNER JOIN Products p
        ON sd.ProductID = p.ProductID

```

Криптира се записва в колоната *text* на системната таблица *syscomments*, съдържащ текста на дефиницията на обекта, т.е. изгледа.

Пример 15

```

CREATE VIEW Product_Rollup
AS
    SELECT CASE
        WHEN GROUPING(CategoryName)=1 THEN 'ALL'
        ELSE ISNULL(CategoryName, 'UNKNOWN')
    END AS CategoryName,
    CASE
        WHEN GROUPING(ProductName)=1 THEN 'ALL'
        ELSE ISNULL(ProductName, 'UNKNOWN')
    END AS ProductName,
    AVG(sd.Price*sd.Quantity*(1-sd.Discount))
    AS Average
FROM Categories c
INNER JOIN Products p
    ON c.CategoryID=p.CategoryID
INNER JOIN SaleDetails sd
    ON p.ProductID=sd.ProductID
GROUP BY CategoryName, ProductName WITH ROLLUP

```

Пример 16 Изглед, показващ датата на продажба, продажбата на най-голяма стойност и сумата от продажбите за всички понеделници или вторници, или т.н. недели от текущия месец в зависимост от това кой ден от седмицата е текущият ден:

```

CREATE VIEW Sales_summary_by_weekday
AS
    SELECT
        CONVERT(datetime, CONVERT(char(10), SaleDate, 112))
        AS DateOfSale,
        SUM(TotalforSale) AS TotalSales,
        MAX(TotalforSale) AS BestSale
    FROM Sales
    WHERE DATEDIFF(month, SaleDate, GETDATE())=0 AND
        DATEPART(weekday, SaleDate) =
        DATEPART(dw, GETDATE())
    GROUP BY
        CONVERT(datetime, CONVERT(char(10), SaleDate, 112))

```

При използване на информацията за деня от седмицата на дати може да е необходимо да се промени подразбиращата се стойност на опцията:

```
SET DATEFIRST {number | @number_var}
```

Чрез тази опция се установява първия ден от седмицата като цяло число между 1 и 7. Подразбиращата се стойност е 7, т.е. неделя. Функцията @@DATEFIRST връща като резултат текущата настройка на опцията SET DATEFIRST. За да се зададе първият ден

от седмицата да е понеделник, трябва да се промени стойността на опцията за текущата конекция по следния начин:

```
SET DATEFIRST 1  
GO
```

Задачи

Задача 1. Да се създаде изглед, който да показва:

- 1.1.** имената на всички продукти и броя на техните продажби за текущия ден;
- 1.2.** данните за продуктите, чиято продажба не е преустановена и наличното количество (`Stock`) е критично (т.е. по-малко или равно на `ReorderLevel`); изгледът да не допуска промяна на данните, нарушаваща условието му;
- 1.3.** данните за продуктите, чиято доставна цена е по-голяма от доставните цени на всички продукти от дадена категория (`CategoryID = 1`). Да се напише заявка към изгледа, показваща 3^{те} най-ниски цени;
- 1.4.** датите, на които не е продаван даден продукт (`ProductID = 1`). Да се напише заявка към изгледа, показваща само датите от изминалите 6 месеца, подредени в намаляващ ред.

Временни таблици

Временните таблици могат да се използват за съхраняване на междинни резултати или за споделяне на данни, необходими за обработка, извършвана от други едновременни конекции. Създаването на временни таблици е възможно от всяка база от данни, но те винаги съществуват само в системната база от данни *tempdb*. Тази база от данни бива създавана при рестартиране на сървъра, а не възстановявана. При работа с SQL Server могат да се използват временни таблици по три начина: частно, глобално и директно.

Частни временни таблици (#)

От всяка база от данни може да бъде създадена частна (локална) временна таблица чрез поставяне на знака # пред името на таблицата (например `CREATE TABLE #my_table (...)`). Само конекцията, създава частната временна таблица, има достъп до нея. Такава временна таблица съществува само по време на съществуването на конекцията; тази конекция може да изтрие таблицата посредством `DROP TABLE`. Допустимо е да има временна таблица с име, използвано в друга конекция.

Пример 1

```
CREATE TABLE #MyTempTable
(c1 int NULL)
```

Може да се използва системната таблица *sysobjects* в базата от данни *tempdb* за претърсване на имената на всички потребителски таблици:

```
SELECT name
FROM tempdb..sysobjects
WHERE type = 'U'
```

Пълното име на създадената локална временна таблица се състои от 128 символа и съдържа в началото името, дадено при нейното създаване, 12-цифров идентификатор в края и запълващи символи по средата. Потребителят, който я е създал, може да прави обръщение към тази таблица (например чрез `SELECT * FROM #MyTempTable`), но ако друг потребител опита да го изпълни, той ще получи съобщение за грешка (*Invalid object name '#MyTempTable'.*).

Не е възможно предоставяне или отнемане на привилегии върху локални временни таблици.

Локалните временни таблици са много удобни за временно съхраняване на подмножество на данните в постоянните таблици. За целта е подходящо да се използва конструкцията `SELECT INTO`. Например:

```
SELECT *
INTO #MySales
FROM Sales
WHERE DATEDIFF(month, Saledate, GETDATE()) BETWEEN 1 AND 6
```

Конструкцията `SELECT INTO` може да бъде използвана за създаване на постоянни таблици.

Глобални временни таблици (##)

От всяка база от данни и от всяка конекция може да бъде създадена глобална временна таблица чрез ## пред името на таблицата (например `CREATE TABLE ##my_table (...)`). След това всяка конекция може да осъществява достъп до таблицата за извличане или променяне на данните, следователно не може да има друга глобална временна таблица със същото име. Такава временна таблица съществува до

прекръпяване на създамата я конекция и до приключване на текущото използване на таблицата. След като създамата я конекция бъде прекратена обаче, само на конекциите, които вече имат достъп до нея ще бъде разрешено да приключат, като не се допуска следващо използване на таблицата.

Директно използване на *tempdb*

Тъй като *tempdb* се пресъздава отново при всяко стартиране на SQL Server, тя може да се използва за създаване на таблица. За целта трябва в *tempdb* да се предостави на потребителя привилегия CREATE TABLE. Таблицы, създадени директно в *tempdb*, могат да съществуват след прекръпяване на създамата ги конекция и създателят може специално да предостави или отнеме привилегии за достъп на определени потребители.

Пример 2

```
CREATE TABLE tempdb..MyTemp
(coll int NOT NULL)
```

Всички ограничения могат да бъдат прилагани върху временни таблици, които са явно създадени в *tempdb* (без използване на префикси # или ##). За частни и глобални таблици (т.е. създадени с префикси # или ##) са приложими всички ограничения с изключение на FOREIGN KEY. Референция FOREIGN KEY към частна или глобална временна таблица не е допустимо да се прилага, тъй като би попречила на изтриването на таблицата в момента на прекръпяване на конекцията (за #) и при излизане на таблицата от диапазона (за ##), ако таблицата, извършваща референцията не бъде изтрита предварително.

Задачи

Задача 1. Да се напише код, чрез който да се създаде частна временна таблица, състояща се от колоните SaleID и TotalForSale, като в колоната TotalForSale да се запишат изчислените общи суми на продажбите; да се промени стойността на съществуващата в таблицата Sales колона TotalForSale със съответната във временната таблица; да се изтрие временната таблица.

Задача 2. Да се напише код, чрез който да се създаде частна временна таблица, състояща се от колоните CustomerID и CurrentBalance, като в колоната CurrentBalance да се запишат изчислените общи суми на покупките на клиентите; да се промени стойността на съществуващата в таблицата Customers колона CurrentBalance със съответната във временната таблица; да се изтрие временната таблица.

Транзакции

Транзакцията (transaction) е последователност от SQL конструкции, които биват потвърдени или отхвърлени като едно цяло. Една декларирана от потребителя транзакция се състои от няколко SQL команди, които четат и обновяват базата от данни.

Свойства на транзакциите (ACID)

- Атомарност (*atomicity*). Изпълняват се или всички, или нито една от модификациите, включени в транзакцията.
- Съгласуваност (*consistency*). След като приключи, транзакцията трябва да остави всички данни в състояние на съгласуваност, т.е. данните трябва да останат логически верни. В една релационна база от данни към модификациите на транзакцията трябва да се приложат всички ограничения, с цел да се поддържа целостта на данните.
- Изолиране (*isolation*). Модификациите, които се извършват от едновременни транзакции, трябва да бъдат изолирани една от друга. Изолирането между транзакциите се осъществява автоматично от SQL Server. Той заключва данните, за да могат множество паралелни потребители да работят с тях, така като че ли подават своите заявки последователно. Тази възможност представлява едно от нивата на изолиране, което SQL Server поддържа. Съществуват и други нива на изолиране, които позволяват да се направи подходящ компромис между паралелност и съгласуваност. Заключването намалява паралелността, тъй като заключените данни не са достъпни за други потребители, но осигурява предимство от по-висока съгласуваност.
- Дълготрайност (*durability*). След като една транзакция е приключила, резултатите от нейното изпълнение остават дори в случай на срив на системата. Ако се получи срив на системата, докато транзакцията се изпълнява, транзакцията трябва да бъде напълно отменена, без да оставя никакви частични следи върху данните. Например, ако преди да бъде завършена една транзакция, се получи прекъсване на хранването, когато системата бъде отново рестартирана, цялата транзакция трябва да бъде отменена автоматично. Ако хранването спре непосредствено след потвърждаването на транзакцията, тогава модификациите, извършени в транзакцията, трябва да бъдат отразени в базата от данни. За да се осигури тази дълготрайност, се извършва предварително записване в дневника на транзакциите и автоматично отменяне и възстановяване на транзакциите при стартиране на SQL Server.

Транзакция, състояща се от множество конструкции, не прави никакви промени в базата от данни, докато не бъде изпълнена командата `COMMIT TRANSACTION`. Освен това транзакцията може да отмени промените чрез `ROLLBACK TRANSACTION`.

`ROLLBACK TRANSACTION` не променя хода на управлението, той продължава със следващата конструкция. `ROLLBACK TRANSACTION` засяга само данни, съхранявани в базата от данни, не влияе върху локалните променливи или опциите, установявани чрез конструкциите `SET`, следователно промените в стойностите на локалните променливи и `SET`-опциите по време на транзакция не се връщат до стойностите, които са имали преди нейното започване.

Може да се конфигурира SQL Server да започва транзакция неявно, като се използва `SET IMPLICIT_TRANSACTIONS ON`. Ако са разрешени неявните

транзакции, всички конструкции се считат за част от транзакция и нищо не се потвърждава, докато изрично не се изпълни COMMIT TRANSACTION. Ако опцията не е включена, всяка **транзакция, състояща се от множество конструкции**, трябва да започва с BEGIN TRANSACTION.

Пример 1

```
DECLARE @sum money, @st varchar(20)
```

BEGIN TRANSACTION

```
UPDATE SaleDetails
    SET Quantity = 200
WHERE SaleID = 1 AND ProductID = 5

SELECT @sum = SUM(price * quantity * (1 - discount))
FROM SaleDetails
WHERE SaleID = 1

IF @sum > 100
    BEGIN
        ROLLBACK TRANSACTION
        SET @st = LTRIM(STR(@sum - 100, 20, 2))
        RAISERROR('Общата сума на продажбата
                    надхвърля 100 с %s.', 11, 1, @st)
    END
ELSE
    BEGIN
        COMMIT TRANSACTION

        SET @st = LTRIM(STR(@sum, 20, 2))

        RAISERROR('Общата сума на продажбата е %s.',
                    11, 1, @st)
    END
GO
```

Проверка за грешки в транзакциите

Ако възникне грешка и не се предприеме никакво действие спрямо нея, обработката преминава към следващата конструкция. Синтактичните грешки винаги предизвикват отхвърляне на целия пакет още преди изпълнението да е започнало, а обръщенията към несъществуващи обекти води до спиране на изпълнението на пакета до съответното място.

Ако целта е да се превърти назад транзакцията при възникване на каквато и да е грешка, трябва да се проверява стойността на функцията @@ERROR.

Пример 2

BEGIN TRANSACTION

```
INSERT INTO Sales
    (CustomerID, EmployeeID, SaleDate)
VALUES (1, 1, GETDATE())
```

```

IF @@ERROR <> 0 GOTO error

INSERT INTO SaleDetails
    (SaleID, ProductID, Price, Quantity)
VALUES (@@IDENTITY, 18, 6, 10)

IF @@ERROR <> 0 GOTO error

UPDATE Products
    SET Stock = Stock - 10
WHERE ProductID = 18

IF @@ERROR <> 0 GOTO error

COMMIT TRANSACTION
GOTO Finish

error:
ROLLBACK TRANSACTION

Finish:
GO

SELECT * FROM Products WHERE ProductID = 18
GO

```

След пораждање на грешката трябва да се установи дали транзакцията да продължи или да приключи, или да се превърти назад. В пример 2 генерирането на грешка от някоя конструкция превърта назад транзакцията; изпълнението на всички конструкции без грешки, води до потвърждаване на всички промени.

Последователност на проверките за цялост

Модификацията на даден ред се отменя, ако бъде нарушено някое ограничение или ако някой тригер прекрати операцията. Нарушаването на ограничение води до прекратяване на модификацията, следващите проверки за този ред не се извършват и за него не се активират никакви тригери, следователно последователността на проверките е съществена:

1. Присвояват се подразбиращите се стойности.
2. Проверяват се нарушения на NOT NULL.
3. Изчисляват се ограниченията CHECK.
4. Извършват се проверки на FOREIGN KEY в таблиците, които имат референция.
5. Извършват се проверки на FOREIGN KEY в таблиците, към които има референция.
6. UNIQUE (PRIMARY KEY) се проверява за коректност.
7. Активират се тригерите.

Контрол на едновременната работа (*concurrency control*)

Механизмите, които SQL Server използва, за да контролира едновременното извършване на модификации на данните от различни потребители, се наричат *заклучвания (locks)*. Заклучванията ограничават изпълнението на определени операции

върху базата от данни, докато други операции или транзакции са активни. Съществуват две общи категории заключвания:

- *Песимистичните заключвания* предотвратяват модифициране на данните от страна на потребителите, което може да се отрази върху модификациите, извършени от други потребители. Реализира се чрез заключване на всички или на част от данните, така че други потребители да не могат да изпълняват действия, които влизат в конфликт със заключването, докато неговият собственик не го преустанови. Песимистичният контрол на едновременната работа се използва най-вече в среда, в която има голяма конкуренция на данните.
- *Оптимистичните заключвания* осигуряват едновременна работа на потребителите, при която не се предотвратява никоя операция, но се анулират тези, които са създали конфликти. Когато се извършва актуализация, системата проверява дали някой друг потребител не е променил данните, след като те са били прочетени. Ако някой друг потребител е актуализирал данните, възниква грешка и транзакцията се анулира. Тази ситуация се реализира в среда, където има слаба конкуренция за данните.

Потребителят определя какъв да е видът на контрола на едновременната работа, като установява нива на изолация на транзакциите.

Нива на изолация на транзакциите

Нивото на изолация, на което се изпълнява дадена транзакция, определя чувствителността на приложението към промените, направени от други. То определя и колко дълго транзакцията трябва да държи заключванията на данните за защита от евентуални промени, извършвани от други едновременни транзакции. SQL Server предлага четири поведения за нива на изолация:

- READ UNCOMMITTED (*dirty read*) – прочитане на непотвърдени данни;
- READ COMMITTED – прочитане на потвърдени данни;
- REPEATABLE READ – повторяемо четене;
- SERIALIZABLE – последователно.

Когато се използва опцията

SET TRANSACTION ISOLATION LEVEL **READ UNCOMMITTED**

се допуска, ако една заявка бъде изпълнена няколко пъти, тя всеки път да връща различни резултати, независимо от това дали едновременните транзакции са били потвърдени. При това ниво на изолация е възможно прочитане на данни, които логически никога не са съществували (т. нар. мръсно четене). Съгласуваността на съвместното изпълнение е висока, но не се поддържа съгласуваност на данните.

Например:

Транзакция 1	Стойност на Stock	Транзакция 2
SELECT Stock FROM Products WHERE ProductID = 87	100	SELECT Stock FROM Products WHERE ProductID = 87
UPDATE Products SET Stock = Stock - 10 WHERE ProductID = 87	90	
ROLLBACK TRANSACTION	100	

Заявката в транзакция 2 извлича стойност, която междувременно бива отхвърлена. Тъй като транзакция 1 се отменя, всичко изглежда така, сякаш стойността на наличното количество `Stock` никога не е била променяна на 90, въпреки че точно това е стойността, която се появява при изпълнението на транзакция 2.

Когато се използва опцията

`SET TRANSACTION ISOLATION LEVEL READ COMMITTED`

се допуска, ако една заявка бъде изпълнена няколко пъти, тя всеки път да връща различни резултати, но само когато едновременните транзакции са били потвърдени. При това ниво на изолация конекцията, четяща данните, трябва да изчака, докато транзакцията по обновяване на данните приключи. Следователно се постига по-висока съгласуваност на данните, тъй като непотвърдените данни не се виждат от други потребители, но съгласуваността на съвместното изпълнение е понижена. Например:

Транзакция 1	Стойност на Stock	Транзакция 2
	100	SELECT Stock FROM Products WHERE ProductID = 87
UPDATE Products SET Stock = Stock - 10 WHERE ProductID = 87 COMMIT TRANSACTION	90	SELECT Stock FROM Products WHERE ProductID = 87
	90	

Заявката в транзакция 2 връща два различни резултата в рамките на една и съща транзакция. И двата резултата са коректни, тъй като действително данните са се променили междувременно.

Committed Read е подразбиращото се ниво на изолация за SQL Server. Въпреки че не е възможно четене на данни, чиято обработка не е приключила, при повторно извличане на същите данни те вече може да са променени или може внезапно да се появят нови редове, които удовлетворяват условието на заявката, т.е. така наречените *фантоми* (*phantom insert*).

Когато се използва опцията

`SET TRANSACTION ISOLATION LEVEL REPEATABLE READ`

се допуска *phantom insert*, но във всички останали случаи се гарантира, че ако дадена заявка бъде изпълнена повторно, данните няма да са се променили. Ако една и съща заявка се изпълни повече от един път в рамките на една транзакция, на резултата от тази заявка няма да се отразят промените в стойностите на данните, направени от други транзакции. Например:

Транзакция 1	Транзакция 2
	SELECT SUM(quantity) FROM SaleDetails WHERE ProductID = 87
INSERT INTO SaleDetails (SaleID, ProductID, Quantity, Price) VALUES (23, 87, 15.500, 5.90) COMMIT TRANSACTION	

	<pre>SELECT SUM(quantity) FROM SaleDetails WHERE ProductID = 87</pre>
--	---

Добавянето на ред е възможно след обработване на конструкцията `SELECT`. Тъй като вмъкването на нов ред в транзакция 1 се извършва по средата на транзакция 2, то в резултат от двете изпълнения на една и съща заявка се получават две различни стойности. При това ниво на изолация съществуващите редове не могат да бъдат променяни или изтрити. Обаче могат да бъдат добавяни нови редове към областта от данни, указана чрез `WHERE` на конструкцията `SELECT`. Всички прочетени данни са заключени, така че не могат да се правят промени. Но тъй като конструкцията `INSERT` добавя нови данни, заключванията върху съществуващите данни не пречат да бъде изпълнена.

Когато се използва опцията

`SET TRANSACTION ISOLATION LEVEL SERIALIZABLE`

всяка транзакция не оказва влияние върху нито една друга транзакция, която се извършва едновременно с нея. При това ниво на изолация се гарантира, че ако дадена заявка бъде изпълнена отново, междувременно няма да са добавени редове, т.е. няма да се появят фантоми. В този случай изпълняването на транзакции по едно и също време е еквивалентно на изпълняването им последователно без значение от реда им. При това ниво на изолация съгласуваността на съвместното изпълнение е значително намалена, тъй като втората конекция трябва да изчака, докато първата приключи транзакцията напълно, но съгласуваността на данните е висока.

Вложени блокове транзакции

Възможно е блокове, започващи с `BEGIN TRANSACTION` и завършващи с `COMMIT TRANSACTION` или `ROLLBACK TRANSACTION`, да бъдат вложени един в друг. Резултатът е следния:

- `ROLLBACK TRANSACTION` превърта назад всички нива на транзакцията, а не само нейния най-вътрешен блок;
- `COMMIT TRANSACTION` потвърждава всички нива на транзакцията, ако конструкцията съответства на най-външния блок, в противен случай не прави нищо за потвърждаване на транзакцията.

Системната функция `@@TRANCOUNT` връща дълбочината на вложените `BEGIN TRANSACTION` блокове. Ако няма нито една активна транзакция, стойността на `@@TRANCOUNT` е 0. Изпълняването на:

- `BEGIN TRANSACTION` увеличава с единица стойността на `@@TRANCOUNT`;
- `COMMIT TRANSACTION` намалява с единица стойността на `@@TRANCOUNT`;
- `ROLLBACK TRANSACTION` превърта назад цялата транзакция и установява `@@TRANCOUNT` на 0.

Възможно е именуване на `BEGIN TRANSACTION`, `COMMIT TRANSACTION` и `ROLLBACK TRANSACTION` конструкциите. `ROLLBACK TRANSACTION` не изисква име на транзакция, но може да се включи, за да се подчертае, че превърта назад цялата транзакция. `COMMIT TRANSACTION` също може да бъде именувана и няма да се генерира грешка, ако името не съответства на `BEGIN TRANSACTION` от най-горно ниво, но в този случай тя не потвърждава транзакцията, а само намалява с единица стойността на `@@TRANCOUNT`.

Пример 3

`SELECT @@TRANCOUNT -- Резултатът е 0.`

```

BEGIN TRANSACTION A
    SELECT @@TRANCOUNT -- Резултатът е 1.
    -- изпълняват се конструкции за модифициране на данни
BEGIN TRANSACTION B
    SELECT @@TRANCOUNT -- Резултатът е 2.
    -- изпълняват се конструкции за модифициране на данни
ROLLBACK TRANSACTION B
/* Неуспешна конструкция с грешка 6401:
Cannot roll back B. No transaction or savepoint of that name was found. */
SELECT @@TRANCOUNT -- Резултатът е 2.
-- изпълняват се конструкции за модифициране на данни
ROLLBACK TRANSACTION A -- тази конструкция е успешна
SELECT @@TRANCOUNT -- Резултатът е 0.

```

Пример 4

```

SELECT @@TRANCOUNT -- Резултатът е 0.
BEGIN TRANSACTION A
    SELECT @@TRANCOUNT -- Резултатът е 1.
    -- изпълняват се конструкции за модифициране на данни
BEGIN TRANSACTION B
    SELECT @@TRANCOUNT -- Резултатът е 2.
    -- изпълняват се конструкции за модифициране на данни
ROLLBACK TRANSACTION -- прекратява транзакцията
SELECT @@TRANCOUNT -- Резултатът е 0.
-- изпълняват се конструкции за модифициране на данни
ROLLBACK TRANSACTION
/* Неуспешна конструкция с грешка 3903:
The ROLLBACK TRANSACTION request has no corresponding BEGIN
TRANSACTION. */
SELECT @@TRANCOUNT -- Резултатът е 0.

```

Пример 5

```

SELECT @@TRANCOUNT -- Резултатът е 0.
BEGIN TRANSACTION A
    SELECT @@TRANCOUNT -- Резултатът е 1.
    -- изпълняват се конструкции за модифициране на данни
BEGIN TRANSACTION B
    SELECT @@TRANCOUNT -- Резултатът е 2.
    -- изпълняват се конструкции за модифициране на данни
COMMIT TRANSACTION B
/* Не потвърждава транзакцията, но намалява стойността на
@@TRANCOUNT. */
SELECT @@TRANCOUNT -- Резултатът е 1.
-- изпълняват се конструкции за модифициране на данни
COMMIT TRANSACTION A
-- Потвърждава промените и затваря транзакциите.
SELECT @@TRANCOUNT -- Резултатът е 0.

```


Точки на записване в транзакциите

Точката на записване (*savepoint*) е точка, до която модификациите в транзакцията могат да бъдат отменени. Не е възможно точката на записване да бъде използвана за потвърждаване на някакви промени в базата от данни.

Добавянето на точки на записване в SQL Server се извършва с конструкцията `SAVE TRANSACTION savepoint_name`. Тя не се отразява върху стойността на @@TRANCOUNT. Превъртането назад до точка на записване, а не на цялата транзакция, също не засяга стойността, връщана от @@TRANCOUNT. При превъртането назад трябва явно да се укаже името на точката на записване, тъй като използването на `ROLLBACK TRANSACTION` без специфично име винаги връща назад цялата транзакция.

Пример 6

```
SELECT @@TRANCOUNT -- Резултатът е 0.
BEGIN TRANSACTION A
    SELECT @@TRANCOUNT -- Резултатът е 1.
    -- изпълняват се конструкции за модифициране на данни
    SAVE TRANSACTION B
        SELECT @@TRANCOUNT /* Резултатът е 1, тъй като функцията не
        се влияе от точката на записване. */
        -- изпълняват се конструкции за модифициране на данни
    ROLLBACK TRANSACTION B
    SELECT @@TRANCOUNT -- Резултатът е 1.
    /* Последният ROLLBACK засяга само точката на записване, а не
    транзакцията. */
    -- изпълняват се конструкции за модифициране на данни
ROLLBACK TRANSACTION A
/* успешно превъртане на транзакцията, отхвърляне на промените и
затваряне на транзакциите */
SELECT @@TRANCOUNT -- Резултатът е 0.
```

Пример 7

```
BEGIN TRANSACTION TranStart
-- начало на транзакцията
INSERT INTO Sales (CustomerID, EmployeeID, SaleDate)
VALUES (2, 2, DEFAULT)
-- Добавяне на ред, който се запазва в базата от данни.
SAVE TRANSACTION FirstPoint
/* Създаване на точка на записване, до която може да се извърши връщане
при необходимост. */
INSERT INTO Sales (CustomerID, EmployeeID, SaleDate)
VALUES (1, 1, DEFAULT)
-- Добавяне на друг ред, който не се запазва в базата от данни.
ROLLBACK TRANSACTION FirstPoint
/* Отменят се конструкциите до първата точка на записване FirstPoint.
Конструкциите над тази точка са още част от транзакцията. */
INSERT INTO Sales (CustomerID, EmployeeID, SaleDate)
VALUES (2, 2, DEFAULT)
/* Добавяне на трети ред. Това е вторият запис, който се запазва в базата
от данни. */
```

SAVE TRANSACTION SecondPoint

/ Създаване на друга точка на записване, до която да се извършват отменения. */*

```
INSERT INTO Sales (CustomerID, EmployeeID, SaleDate)
VALUES (1, 1, DEFAULT)
```

-- Добавяне на още един ред, който не се запазва.

ROLLBACK TRANSACTION SecondPoint

-- отменяне до втората точка на записване

```
INSERT INTO Sales (CustomerID, EmployeeID, SaleDate)
VALUES (2, 2, DEFAULT)
```

/ Добавяне на още един ред. Той е третия, който се запазва в базата от данни. */*

```
COMMIT TRANSACTION TranStart
```

-- потвърждаване на транзакцията

```
GO
```

-- извличане на потвърдените редове

```
SELECT TOP 3 *
FROM Sales
ORDER BY SaleID DESC
```

Ако в пример 7 COMMIT TRANSACTION TranStart се замени с ROLLBACK TRANSACTION TranStart, всички добавяния на редове се отменят, т.е. няма въведени редове.

Името след COMMIT TRANSACTION е без значение, изпълнението на тази конструкция винаги намалява с единица стойността, връщана от @@TRANCOUNT.

Възможно е влягане на SAVE TRANSACTION *savepoint_name*, но е необходимо първата конструкция ROLLBACK TRANSACTION *savepoint_name* да съответства на последната точка на записване и т.н.

Задачи

Задача 1. Да се напише кода в пример 2 за проверка за грешки в транзакциите, но с използване на конструкцията TRY...CATCH вместо оператора GOTO.

Задача 2. Да се определи резултата от изпълнението на следния код, при условие че съществува продукт с идентификатор 1, но не съществува продукт с идентификатор 2000:

```
DECLARE @Ident int
```

```
BEGIN TRANSACTION
```

```
INSERT INTO Sales
      (CustomerID, EmployeeID, SaleDate)
VALUES (1, 1, DATEADD(day, -1, GETDATE()))
```

```
SET @Ident = @@IDENTITY
```

```
SAVE TRANSACTION ProductA
```

```
INSERT INTO SaleDetails
      (SaleID, ProductID, Price, Quantity)
```

```
VALUES (@Ident, 2000, 50, 25)

IF @@ERROR <> 0 ROLLBACK TRANSACTION ProductA

SAVE TRANSACTION ProductB

INSERT INTO SaleDetails
        (SaleID, ProductID, Price, Quantity)
VALUES (@Ident, 1, 15.50, 2)

IF @@ERROR <> 0 ROLLBACK TRANSACTION ProductB

COMMIT TRANSACTION

SELECT SUM(Price*Quantity) AS Total
FROM SaleDetails
WHERE SaleID = @Ident
GO
```

Създаване на съхранени процедури

Всяка система за управление на бази от данни (СУБД) предлага възможности за създаване и съхраняване в схемата на база от данни на процедури и/или функции, които могат да се стартират от SQL заявки или други SQL изрази. Стандартът SQL/PSM (*Persistent Stored Modules – постоянно съхранявани модули*) е част от SQL-99 и обуславя основните направления на развитието на технологията за работа със съхранен код. В съответствие с този стандарт са разработени конкретните реализации (като например в СУБД Microsoft SQL Server 2005).

Роля, създаване и извикване на съхранени процедури

Съхранените процедури представляват последователност от SQL конструкции, съхранени в база от данни на SQL Server. Възможно е да се напише кода на сложни заявки и транзакции под формата на съхранени процедури, след което да бъдат извикани директно от приложението с клиентския интерфейс. Обща тенденция при програмирането на бази от данни е клиентската част (която съдържа обекти за връзка, свързващи се към базата от данни и командни обекти, които извикват съхранени процедури) да се разтовари колкото се може повече, като се разчита сървърът да поеме голяма част от обработката.

Синтаксисът на съхранените процедури се проверява и компилирането им се извършва първия път, когато бъдат изпълнени. SQL сървърът съхранява компилираната версия в своя кеш, след което за обработване на следващите заявки използва кешираната и компилирана версия, което води до по-бързо изпълнение. Съхранените процедури могат да приемат и параметри, следователно една процедура може да бъде изпълнявана от множество приложения, като използва различни входни данни.

Съхранените процедури, освен че предоставят по-добра производителност, те осигуряват изолация от промени в критериите и логиката на системата. Когато се налагат изменения, съхранените процедури се променят само веднъж върху сървъра и не е необходимо клиентските приложения да бъдат променяни.

Създаване на съхранени процедури

За създаване на съхранена процедура в Microsoft SQL Server 2005 се използва следната конструкция:

```
CREATE PROCEDURE stored_procedure_name
[ { @parameter_name_IN datatype [=default_value]
  | @parameter_name_OUT datatype OUTPUT } [, ...] ]
AS
    sql_queries
```

Създадената процедура може впоследствие да бъде изпълнена чрез следния синтаксис:

```
{EXECUTE | EXEC}
[@return_value =] stored_procedure_name
[ { [@parameter_name_IN =] parameter_value
  | [@parameter_name_OUT =]
    @received_var_name OUTPUT } [, ...] ]
```

Пример 1

```
CREATE PROCEDURE SalesByEmployee
AS
    SELECT FirstName, LastName,
           ( SELECT COUNT(*) FROM Sales s
```

```

        WHERE s.EmployeeID = e.EmployeeID )
        AS TotalSales
    FROM Employees e
    ORDER BY FirstName, LastName
GO

```

-- изпълняване на създадената процедура:

```
EXECUTE SalesByEmployee
```

-- или

```
EXEC SalesByEmployee
```

Конструкцията EXEC изисква от сървъра да изпълни процедурата и да върне резултата към програмата, от която е извикана процедурата.

Променяне на съхранена процедура

За променяне на съхранена процедура се използва следната конструкция:

```

ALTER PROCEDURE stored_procedure_name
[ { @parameter_name_IN datatype [=default_value]
  | @parameter_name_OUT datatype OUTPUT } [, ...] ]
AS
    sql_queries

```

Изтриване на съхранена процедура

За изтриване на съхранена процедура се използва следната конструкция:

```
DROP PROCEDURE stored_procedure_name
```

Използване на върнатите стойности

Върната стойност е стойността, която съхранената процедура изпраща обратно към извикателя я процедура. По този начин може да се връща само една целочислена стойност. За изпращане на върната стойност обратно към извикващата процедура се използва оператора:

```
RETURN [expression]
```

Ако се пропусне израз (*expression*), се връща стойност 0, в противен случай – съответната стойност на израза.

Пример 2

```

CREATE PROCEDURE CurrentEmployees
AS
    SELECT EmployeeID,
           FirstName, Surname, LastName,
           Title, HireDate, TerminationDate,
           ManagerEmpID, StoreName
    FROM Employees e
    INNER JOIN Stores s
        ON e.StoreID=s.StoreID
    WHERE TerminationDate IS NULL

```

```
RETURN @@ROWCOUNT
```

За получаване на стойността, която връща дадена съхранена процедура, се използва EXEC конструкция от вида:

```
EXEC @some_var_name = stored_procedure_name
```

Например:

```
DECLARE @count int
```

```
EXEC @count = CurrentEmployees
PRINT @count -- неявно се преобразува в низ
```

Когато в процеса на изпълнение SQL Server се натъкне на RETURN, той спира изпълнението, т.е. извършава се безусловен изход от съхранената процедура или тригера. Каквато и конструкция да има след RETURN, тя няма да се изпълни.

Създаване на съхранени процедури с параметри

Върнатата стойност работи само в рамките на SQL, но не може да бъде получена обратно в програмата, която крайният потребител използва за въвеждане на данни.

Възможно е предаване на информация към съхранените процедури и изпращане на информация от SQL обратно към програмата, която я е извикала. Прехвърлянето на информация в двете посоки се осъществява чрез използването на *параметър*. Параметрите са два вида – входни и изходни. За да се създаде съхранена процедура с параметър, трябва да се въведе списък с променливи непосредствено след името на процедурата и преди AS по следния начин:

```
CREATE PROCEDURE stored_procedure_name
    @parameter_name_IN datatype,          -- входен параметър
    @parameter_name_OUT datatype OUTPUT   -- изходен параметър
AS
    sql_queries
```

При дефинирането на изходни параметри се използва задължително ключовата дума OUTPUT. Максималният брой параметри в една процедура е 2100.

Пример 3 Съхранена процедура с един входен параметър:

```
CREATE PROCEDURE SalesForEmployee
    @EmployeeID int
AS
    SELECT LastName, FirstName,
        (SELECT SUM(TotalForSale*(1-Discount))
         FROM Sales s
         WHERE e.EmployeeID = s.EmployeeID )
        AS TotalSales
    FROM Employees e
    WHERE e.EmployeeID = @EmployeeID
GO
```

Стойностите се прехвърлят по време на изпълнение на съхранената процедура. Ако параметрите са повече от един, се разделят със запетай:

```
EXEC [@some_var_name =] stored_procedure_name
    [@parameter_name_IN1 =] parameter_value1 [,...]
```

В пример 3 съхранената процедура се извиква по следния начин:

```
EXEC SalesForEmployee 1
или
EXEC SalesForEmployee @EmployeeID = 1
```

В съхранените процедури могат да се използват и подразбиращи се стойности на входните параметри. За целта се прилага следния синтаксис:

```
CREATE PROCEDURE stored_procedure_name
    @parameter_name_IN datatype = default_value [,...]
AS
    sql_queries
```

Ако не се прехвърлят никакви стойности към входните параметри с дефинирани подразбиращи се стойности, съхранената процедура използва подразбиращите се стойности. Например:

```
ALTER PROCEDURE SalesForEmployee
    @EmployeeID int = 1
AS
    SELECT LastName, FirstName,
           (SELECT SUM(TotalForSale*(1-Discount))
            FROM Sales s
            WHERE e.EmployeeID = s.EmployeeID )
           AS TotalSales
    FROM Employees e
    WHERE e.EmployeeID = @EmployeeID
GO
/* извикване на съхранената процедура, използващо дефинираната стойност
по подразбиране за входния параметър: */
EXEC SalesForEmployee
-- или
EXEC SalesForEmployee DEFAULT
```

Пример 4 Съхранена процедура, предназначена да изведе доставчиците, за които колоната CompanyName започва с даден символен низ, предаден като параметър или да изведе всички доставчици, ако не се предаде стойност:

```
CREATE PROCEDURE GetSuppliersRaw
    @CompanyName varchar(50) = NULL
AS
    IF @companyName IS NOT NULL
        SELECT * FROM Suppliers
        WHERE CompanyName LIKE @CompanyName + '%'
    ELSE
        SELECT * FROM Suppliers

    RETURN @@ROWCOUNT
GO
EXEC GetSuppliersRaw 'C'
-- или
EXEC GetSuppliersRaw NULL
-- заради стойността по подразбиране на параметъра, е еквивалентно на:
EXEC GetSuppliersRaw
/* В резултат от последното изпълнение на процедурата ще се изведат
всички редове от таблицата Suppliers, т.е. когато не се въведе нищо при
запитване за стойността на параметъра, се извеждат всички записи от
таблицата за доставчици, в противен случай тези, които удовлетворяват
условието – имената им да започват със съответния символен низ. */
```

Върнатата от процедурата стойност е броя на редовете, които се извличат. Следният код извиква процедурата и извежда съобщение за броя на извлечените редове:

```
DECLARE @count int
EXEC @count = GetSuppliersRaw 'Abc'
RAISERROR('%d реда са избрани.', 11, 1, @count)
```

GO

Пример 5 Съхранена процедура, извеждаща данните за дадена по идентификатора си продажба:

```
CREATE PROCEDURE get_sale
    @SaleID int
AS
    SELECT SaleID,
           CONVERT(char(10), SaleDate, 104) AS Sale_date,
           EmployeeID, CustomerID
    FROM Sales
    WHERE SaleID = @SaleID
GO
EXEC get_sale 1000
```

Пример 6 Съхранена процедура, извеждаща данните за продуктите, продадени с дадена по идентификатора си продажба:

```
CREATE PROCEDURE get_sale_details
    @saleID int
AS
    SELECT SaleID, sd.ProductID, ProductName,
           Quantity, sd.Price,
           sd.Quantity*sd.Price*(1-sd.Discount) AS Cost
    FROM SaleDetails sd
    INNER JOIN Products p
        ON sd.ProductID = p.ProductID
    WHERE SaleID = @SaleID
GO
EXEC get_sale_details 1000
```

Използване на резултатите, върнати от изходните параметри

За да може да се използват стойностите на изходните параметри, се прилага следния синтаксис:

```
EXEC [@return_value =] stored_procedure_name
    [@parameter_name_OUT =] @received_var_name OUTPUT
```

Пример 7 Съхранена процедура, която чрез изходен параметър връща доставната цена на продукт с идентификатор, предаден като входен параметър:

```
CREATE PROCEDURE Get_Product_price
    @ProductID int, @price money OUTPUT
AS
    SELECT @price = price
    FROM Products
    WHERE ProductID = @ProductID
GO
```

За използване на стойността, върната от изходния параметър, се изпълнява следния код:

```
DECLARE @price money
EXEC Get_Product_price 1, @price OUTPUT
SELECT @price AS Price
GO
```


Пример 8 Съхранена процедура, въвеждаща нов ред в таблицата Sales, с входни параметри за клиент, служител и дата, изходен параметър за идентификатор на продажбата; върнатата от процедурата стойност е номера на грешката, генерирана след въвеждането на продажбата:

```
CREATE PROCEDURE spInsertSale
    @CustomerID int,
    @EmployeeID int,
    @SaleDate datetime = NULL,
    @SaleID int OUTPUT
AS
    DECLARE @e int

    /* Въвежда се нов ред: */
    INSERT INTO Sales
        (CustomerID, EmployeeID, SaleDate)
    VALUES (@CustomerID, @EmployeeID, @SaleDate)

    /* Стойността на колоната със свойство IDENTITY за новия ред се
    записва в изходната променлива. */
    SELECT @e = @@ERROR, @SaleID = @@IDENTITY
    RETURN @e
GO
```

Кодът, показан по-долу, извиква процедурата от пример 8, така че ако е получена грешка, извежда съобщение; в противен случай добавя съответен ред в таблицата SaleDetails за продажба на даден продукт на цена, получена като 10%-но увеличение на доставната му цена. Ако добавянето на ред в SaleDetails генерира грешка, се отменя въвеждането на новия ред в таблицата Sales.

```
DECLARE @e int, @Ident int, @price money

BEGIN TRANSACTION

EXEC @e = spInsertSale @CustomerID = 1,
    @EmployeeID = 2,
    @SaleDate = '2008/06/18',
    @SaleID = @Ident OUTPUT

IF @e <> 0
    BEGIN
        ROLLBACK TRANSACTION
        RAISERROR('Грешката е %d.', 16, 1, @e)
    END
ELSE
    BEGIN
        EXEC Get_Product_price 87, @price OUTPUT

        INSERT INTO SaleDetails
            (SaleID, ProductID, Price, Quantity)
        VALUES (@Ident, 87, @price * 1.10, 20)

        SET @e = @@ERROR
```

```

IF @e <> 0
BEGIN
    ROLLBACK TRANSACTION
    RAISERROR('Грешката е %d.', 16, 1, @e)
END
ELSE COMMIT TRANSACTION

-- извеждане на добавените редове
SELECT * FROM Sales
WHERE SaleID = @Ident

SELECT * FROM SaleDetails
WHERE SaleID = @Ident
END
GO

```

Не е препоръчително създадените от потребителя съхранени процедури да се именуват с префикс `sp_`, тъй като системните съхранени процедури започват с него и SQL Server първо проверява за наличието на системна съхранена процедура със съответното име и това забавя нейното изпълнение.

Съхранените процедури могат да се извикват *рекурсивно*, при което трябва да се контролира приключване, но безкраен цикъл не се получава, тъй като се допускат до 32 извиквания, след което се отхвърля пакета.

Съхранени процедури и INSERT

Конструкцията `INSERT EXECUTE` от T-SQL позволява използването на съхранена процедура като източник на записи за конструкцията `INSERT`. Например добавянето на редове в следната частна временна таблица:

```

CREATE TABLE #my_Stores
( StoreName varchar(50) NOT NULL,
  City varchar(25) NULL )

```

може да се осъществи чрез `INSERT SELECT`:

```

INSERT INTO #my_Stores (StoreName, City)
SELECT StoreName, City
FROM Stores
WHERE City LIKE 'A%'

```

Освен това е възможно да се използва `INSERT EXECUTE` след създаване на следната съхранена процедура:

```

CREATE PROCEDURE Stores_select
    @p varchar(30) = ''
AS
    SELECT StoreName, City
    FROM Stores
    WHERE City LIKE @p + '%'
GO
INSERT INTO #my_Stores (StoreName, City)
EXECUTE Stores_select 'A'
GO

```

Другият начин за прилагане на `INSERT EXECUTE` е директно без създаване на съхранена процедура:

```

INSERT INTO #my_Stores (StoreName, City)

```

```
EXECUTE (' SELECT StoreName, City
          FROM Stores
          WHERE City LIKE 'A%' ' ')
```

Задачи

Задача 1. Да се създаде съхранена процедура, извеждаща:

- 1.1. клиентите, за които колоната CustomerID започва с дадена последователност от цифри, предадена като параметър или да изведе всички клиенти, ако не се предаде стойност;
- 1.2. продажбите, за които колоната CustomerID започва с дадена последователност от цифри, предадена като параметър или да изведе всички продажби, ако не се предаде стойност;
- 1.3. доставната цена на продукт с идентификатор, предаден като параметър със стойност по подразбиране NULL или да изведе средната аритметична стойност на доставните цени на всички продукти, ако не се предаде стойност.

Задача 2. Да се създаде съхранена процедура, чрез която се извежда различна информация от таблицата за продажбите в зависимост от деня от седмицата, в който е поискана. От вторник до петък показва продажбата на най-голяма стойност и сумата от продажбите за предишния ден. От събота до понеделник показва продажбата на най-голяма стойност и сумата от продажбите за изминалата седмица.

Задача 3. Да се създаде съхранена процедура, изтриваща всички доставчици, за които колоната CompanyName започва с даден символен низ, предаден като параметър. Ако изтриването на редове генерира грешка, върнатата от процедурата стойност да е -1; ако се предаде празен низ или NULL, върнатата от процедурата стойност да е -2; в противен случай – броя на изтритите редове. Да се напише код, който извиква процедурата и извежда съответното съобщение за липсващ префикс; за наличие на грешка; за броя на изтритите редове.

Задача 4. Да се създаде съхранена процедура, изчисляваща отстъпката на дадена по идентификатора си продажба, т.е. изчисляваща и актуализираща стойността на колоната Discount в таблицата за продажбите Sales в зависимост от стойностите на TotalForSale и CurrentBalance: за обща стойност на продажбата над 100 и текущ баланс на клиента над 1000 да се зададе отстъпка 15%; за обща стойност на продажбата над 100 и текущ баланс на клиента под 1000 – 10%; за обща стойност на продажбата под 100 и текущ баланс на клиента над 1000 – 12%; за обща стойност на продажбата под 100 и текущ баланс на клиента под 1000 – 0%.

Задача 5. Да се създаде съхранена процедура, извеждаща служителите, назначени след дадена дата, предадена като параметър или да изведе всички служители, ако не се предаде стойност. Върнатата от процедурата стойност да е броя на редовете, които се извличат. Да се напише код, който извиква процедурата и извежда съобщение за броя на извлечените редове.

Задача 6. Да се създаде съхранена процедура, която увеличава наличното количество на продукт с идентификатор, предаден като параметър или да изведе съобщение, ако не се предаде стойност. Ако продукт със зададения идентификатор не съществува или ако актуализирането генерира грешка, да се изведе съответното

съобщение; ако се изпълни успешно, да се изведе съобщение, което потвърждава извършената промяна и да се извлече новата стойност на колоната `Stock` за съответния продукт.

Задача 7. Да се създаде съхранена процедура, въвеждаща нов ред в таблицата `Sales`, с входни параметри за клиент, служител и дата, изходен параметър за идентификатор на продажбата. Ако датата на продажба за реда, който трябва да бъде добавен, е преди повече от една седмица, конструкцията за добавяне на нов ред не се изпълнява, извежда се дефинирано от потребителя съобщение за грешка, а върнатата от процедурата стойност е номера на грешката. В противен случай конструкцията за добавяне на нов ред се изпълнява, върнатата от процедурата стойност е номера на грешката, генерирана след въвеждането на продажбата. Да се напише код, който извиква процедурата, така че ако е получена грешка, извежда съобщение с номера на съответната грешка; в противен случай извежда добавения ред в таблицата `Sales`.

Задача 8. Да се създаде съхранена процедура, която чрез изходен параметър връща наличното количество на продукт с идентификатор, предаден като входен параметър. Да се напише код, който изпълнява съхранената процедура и извежда съобщение, ако не съществува продукт със зададения идентификатор; в противен случай извежда получената от изходния параметър стойност.

Работа с йерархични данни в бази от данни на Microsoft SQL Server

Често се изисква данните да бъдат представени във вид на йерархия или дърво. Например извеждане във формат, показващ зависимостите на схема, която съдържа всички служители в дадена организация и всеки служител е свързан със своя ръководител чрез неговия идентификатор. Тъй като ръководителите са също служители, техните подробни данни също се съхраняват в таблицата за служителите. Други примери за ситуации, при които възниква необходимост от работа с йерархични данни, са реализиране на йерархия от позволения; реализиране на каталог на продукти за онлайн магазин.

При всички подобни случаи данните трябва да бъдат съхранявани в йерархична или дървовидна форма, но SQL Server е система за управление на релационни бази от данни, а не йерархични. Затова е необходимо данните да се съхраняват в нормализирани, релационни таблици, но да се прилагат програмни техники за обработка на тези данни по йерархичен начин.

Рекурсията е често използвана програмна техника за обработка на дървовидни структури. Тя може да бъде използвана в Transact-SQL чрез *рекурсивно извикване на съхранени процедури*.

Например при създаването на таблицата, съдържаща данните за служителите в една организация, всеки служител е свързан с неговия ръководител чрез идентификатора му:

```
CREATE TABLE Employees
( EmployeeID int IDENTITY NOT NULL
  CONSTRAINT PK_employees PRIMARY KEY,
  FirstName varchar(25) NOT NULL,
  Surname varchar(25) NULL,
  LastName varchar(25) NOT NULL,
  ...,
  ManagerEmpID int NULL
  CONSTRAINT FK_Employee_ManagerEmpID
  FOREIGN KEY
  REFERENCES Employees (EmployeeID),
  ... )
```

Тъй като EmployeeID е деклариран като първичен ключ, по подразбиране се създава клъстериран индекс. За подобряване на производителността на заявките е полезно да се създаде неклъстериран индекс върху ManagerEmpID:

```
CREATE NONCLUSTERED INDEX Emp_MgrID
ON Employees (ManagerEmpID)
```

Стойността на колоната ManagerEmpID за един служител е NULL, ако той е главния ръководител на организацията. Останалите служители са свързани с техните съответни ръководители, като се използва колоната ManagerEmpID.

Може да се създаде съхранена процедура, която да обхожда йерархията рекурсивно и представя служителите във форма, очертаваща дървовидната структура. Съхранената процедура ShowHierarchy, показана по-долу, има един входен параметър, който взема идентификатора на служител (еквивалентен на идентификатора на връх в дърво) и визуализира всички служители, ръководени от него и техните подчинени. В нея се използва системната функция @@NESTLEVEL, която връща нивото на влягане на текущото изпълнение на съхранената процедура. Ако не е стартирана съхранена процедура, стойността на функцията е 0.

```
CREATE PROCEDURE ShowHierarchy
```

```

    @Root int
AS
    DECLARE @EmpID int, @EmpTitle varchar(30)

    SELECT @EmpTitle = Title FROM Employees
    WHERE EmployeeID = @Root
    PRINT REPLICATE('-', @@NESTLEVEL * 4) + @EmpTitle

    SELECT @EmpID = MIN(EmployeeID)
    FROM Employees
    WHERE ManagerEmpID = @Root

    WHILE @EmpID IS NOT NULL
    BEGIN
        EXEC ShowHierarchy @EmpID

        SELECT @EmpID = MIN(EmployeeID)
        FROM Employees
        WHERE ManagerEmpID = @Root
        AND EmployeeID > @EmpID
    END
END
GO

```

Една съхранена процедура може да извиква себе си най-много до 32 нива на влагане. Ако бъде превишена тази граница, се получава следната грешка:

Server: Msg 217, Level 16, State 1, Procedure ShowHierarchy, Line 15

Maximum stored procedure, function, trigger, or view nesting level exceeded (limit 32).

Стартиране на процедурата за получаване на пълната йерархия:

```
EXEC ShowHierarchy 1
```

```
GO
```

Извежда се резултат от следния вид:

```

----Президент
-----Заместник президент
-----Управител на магазин 1
-----Заместник управител 1
-----Продавач-консултант 1
-----Касиер 1
-----Заместник управител 2
-----Снабдител 1
-----Шофьор 1
-----Управител на магазин 2
-----Заместник управител 3
-----Продавач-консултант 2
-----Касиер 2
-----Заместник управител 4
-----Снабдител 2
-----Снабдител 3
-----Шофьор 2

```

За извеждане на йерархията от Управител на магазин 1 нататък:

```
EXEC ShowHierarchy 3
```

```

GO
----Управител на магазин 1
-----Заместник управител 1
-----Продавач-консултант 1
-----Касиер 1
-----Заместник управител 2
-----Снабдител 1
-----Шофьор 1

```

Въпреки че Transact-SQL поддържа рекурсия, много по-ефективно е да се използва *временна таблица* като *стек* за съхраняване на информация за всички елементи, за които обработката е започната, но не е завършена. Когато обработката за отделен елемент е приключила, той се изтрива от стека. При идентифициране на нови елементи, те се добавят в стека. Съхранената процедура `expand`, показана по-долу, извежда закодирана йерархия на произволна дълбочина (за разлика от рекурсията, която ограничава до 32 вложения).

```

CREATE PROCEDURE expand
    @current int
AS
    SET NOCOUNT ON
    DECLARE @lvl int, @line varchar(25)

    CREATE TABLE #stack
    ( item int NOT NULL,
      EmpName varchar(25) NOT NULL,
      lvl int NOT NULL )
    INSERT INTO #stack (item, EmpName, lvl)
    SELECT @current, LastName, 1
    FROM Employees
    WHERE EmployeeID = @current

    SET @lvl = 1
    WHILE @lvl > 0
    BEGIN
        IF EXISTS (SELECT * FROM #stack
                   WHERE lvl = @lvl)
        BEGIN
            SELECT TOP 1 @current = item,
                          @line = EmpName
            FROM #stack
            WHERE lvl = @lvl

            PRINT SPACE(@lvl*4) + @line

            DELETE FROM #stack
            WHERE lvl = @lvl AND item = @current

            INSERT INTO #stack(item, EmpName, lvl)
            SELECT EmployeeID, LastName, @lvl + 1
            FROM Employees
            WHERE ManagerEmpID = @current
        END
    END

```

```

        IF @@ROWCOUNT > 0
            SET @lvl = @lvl + 1
        END
    ELSE SET @lvl = @lvl - 1
END -- WHILE
GO

```

Входният параметър @current определя началната точка в йерархията. Използват се две локални променливи:

- @lvl, която съхранява информация за текущото ниво на йерархията;
- @line, на която се присвоява съответния ред за извеждане.

Задаването на опцията SET NOCOUNT ON предотвратява извеждането на съобщението за броя на обработените редове от всяка изпълнена конструкция. Временната таблица #stack се създава и в нея се добавя ред за елемента, определен като начална точка в йерархията. Задава се съответна стойност на @lvl. Когато @lvl има стойност по-голяма от 0, процедурата изпълнява следните стъпки:

1. Ако има някакви елементи в стека на текущо ниво, процедурата избира един и извлича неговата стойност чрез @current.
2. Вмъква @lvl*4 на брой интервала пред името на елемента и извежда елемента.
3. Изтрива елемента от стека, за да не бъде обработван отново и добавя всички негови подчинени елементи в стека на следващо ниво (@lvl+1). С Transact-SQL е възможно намирането и добавянето на всички подчинени елементи с една конструкция, като се избягват други вложени търсения (за разлика от стандартните езици за програмиране).
4. Ако има подчинени елементи (@@ROWCOUNT > 0), се преминава на по-високо ниво, за да бъде обработено (SET @lvl = @lvl + 1); в противен случай се продължава обработката на текущо ниво.
5. Ако няма елементи в стека, чакащи обработка на текущо ниво, се извършва връщане назад с едно ниво, за да се провери дали има чакащи обработка елементи на предходно ниво (SET @lvl = @lvl - 1). Когато няма предходно ниво, извеждането приключва.

За извеждане на резултата във вид на множество набори от данни може да се замени PRINT със SELECT. Тъй като връщането на група резултатни набори към клиентското приложение е неефективно, то трябва да бъде избягвано. Затова е добре да се използва временна таблица, чрез която накрая да се изведе йерархията като един резултатен набор. Това може да се реализира по следния начин:

```

ALTER PROCEDURE expand
    @current int
AS
    SET NOCOUNT ON
    DECLARE @lvl int

    CREATE TABLE #stack
    ( item int NOT NULL,
      lvl int NOT NULL )
    INSERT INTO #stack (item, lvl)
    VALUES (@current, 1)

```



```

-- Временна таблица за извеждане на един резултатен набор
CREATE TABLE #result
( item int NOT NULL,
  lvl int NOT NULL )

SET @lvl = 1
WHILE @lvl > 0
BEGIN
  IF EXISTS (SELECT * FROM #stack
             WHERE lvl = @lvl)
  BEGIN
    SELECT TOP 1 @current = item
    FROM #stack
    WHERE lvl = @lvl

    INSERT INTO #result (item, lvl)
    VALUES (@current, @lvl)

    DELETE FROM #stack
    WHERE lvl = @lvl AND item = @current

    INSERT INTO #stack(item, lvl)
    SELECT EmployeeID, @lvl + 1
    FROM Employees
    WHERE ManagerEmpID = @current

    IF @@ROWCOUNT > 0
      SET @lvl = @lvl + 1
  END
  ELSE SET @lvl = @lvl - 1
END -- WHILE

/* За да се изведе името и длъжността на служителя, временната таблица
#result се съединява с постоянната таблица Employees. */
SELECT SPACE(r.lvl*4) + e.LastName + ', ' + e.Title
      AS [Expanding hierarchy]
FROM #result r
INNER JOIN Employees e ON r.item = e.EmployeeID
GO

```

Задачи

Задача 1. Дадени са таблиците Courses и PrerequisiteCourses, съхраняващи информация за предлаганите курсове и необходимите за тях предварително изкарани курсове.

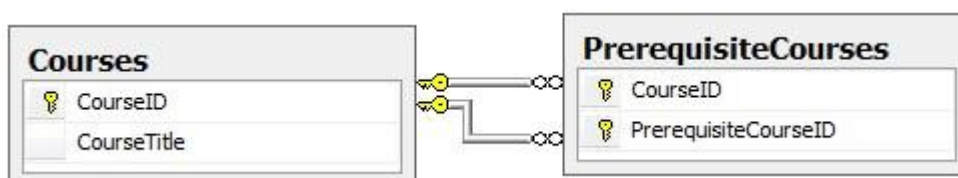
```

CREATE TABLE Courses
(
  CourseID int NOT NULL IDENTITY PRIMARY KEY,
  CourseTitle varchar(50) NOT NULL
)

```

```
CREATE TABLE PrerequisiteCourses
(
    CourseID int NOT NULL
        REFERENCES Courses(CourseID),
    PrerequisiteCourseID int NOT NULL
        REFERENCES Courses(CourseID),
    PRIMARY KEY (CourseID, PrerequisiteCourseID)
)
```

Получената рекурсивна релационна връзка е показана на фигура 1.



Фиг. 1 Рекурсивна релационна връзка „много към много”

Да се напише процедура, която по даден идентификатор на курс визуализира всички курсове, предварително изисквани за него, както и необходимите за тях курсове.

Дефинирани от потребителя функции

Освен вградените функции, които предлага, Microsoft SQL Server разполага с възможност за създаване на *дефинирани от потребителя функции* (*user-defined functions* – *UDF*). Дефинираните от потребителя функции представляват последователност от една или повече Transact-SQL конструкции. Те съхраняват код, който може да бъде изпълняван многократно; връщат стойност и могат да приемат параметри (от 0 до 1024). Типът на параметрите може да бъде произволен скаларен тип данни, включително *bigint* и *sql_variant*, но не може да бъде *timestamp* или потребителски дефиниран тип данни. Дефинираните от потребителя функции не могат да връщат стойност от тип *text*, *ntext*, *image*, *cursor* или *timestamp*. Поддържат се три типа дефинирани от потребителя функции в зависимост от типа на върнатата от тях стойност – скаларни функции; функции, връщащи таблица (*inline table-valued functions*) и многоструктурни функции, връщащи таблица (*multistatement table-valued functions*).

Скаларни функции

Връщат една стойност от скаларен тип данни (например *integer*, *varchar*, *char*, *money*, *datetime*, *bit* и т.н.) или от дефиниран от потребителя тип данни, ако е базиран на скаларен тип данни. Общият вид на конструкцията за създаване на дефинирана от потребителя функция от този тип е:

```
CREATE FUNCTION [owner_name.]function_name
( [ { @parameter_name [AS] scalar_parameter_data_type
    [ = default ] } [ , ... n ] ] )
RETURNS scalar_return_data_type
[ WITH [ENCRYPTION] [,] SCHEMABINDING ]
[AS]
BEGIN
    function_body
    RETURN scalar_expression
END
```

Опцията **ENCRYPTION** позволява да се зададе криптиране на колоната в системната таблица, съдържаща текста на конструкциите, включени в дефиницията на функцията. Чрез опцията **SCHEMABINDING** се задава свързване с обектите на базата от данни, към които функцията прави обръщение, т.е. те не могат да бъдат променяни или изтривани.

Скаларните функции могат да бъдат извиквани, като се включат поне две части от името на функцията:

```
[database_name.]owner_name.function_name([argument_expr][,...])
```

Извикването се извършва при съставяне на скаларни изрази, включително за дефиниране на изчислими колони и **CHECK** ограничения.

Пример 1 Функция за установяване кои от продуктите имат нужда от нова доставка:

```
CREATE FUNCTION fnNeedToReorder
( @nReorderLevel int, @nUnitsInStock int,
  @bDiscontinued bit )
RETURNS varchar(3)
AS
BEGIN
    DECLARE @sReturnValue varchar(3)
```

```

IF @bDiscontinued = 1
    SET @sReturnValue = 'No'
ELSE
    IF @nUnitsInStock <= @nReorderLevel
        SET @sReturnValue = 'Yes'
    ELSE
        SET @sReturnValue = 'No'

```

```

    RETURN @sReturnValue

```

```

END

```

```

GO

```

Извикването на функцията може да се осъществи по следния начин:

```

SELECT ProductID, ProductName, Stock, ReorderLevel,
       Discontinued,
       dbo.fnNeedToReorder(ReorderLevel, Stock, Discontinued)
       AS sNeedToReorder
FROM Products

```

Възможно е тази функция да се напише, така че да изисква само идентификатора на продукта като параметър:

```

ALTER FUNCTION fnNeedToReorder (@nProductID int)
RETURNS varchar(3)
AS
BEGIN
    DECLARE @sReturnValue varchar(3)

    SELECT @sReturnValue =
        CASE Discontinued
            WHEN 1 THEN 'No'
            ELSE
                CASE
                    WHEN Stock <= ReorderLevel THEN 'Yes'
                    ELSE 'No'
                END
        END

    FROM Products
    WHERE ProductID = @nProductID

    RETURN @sReturnValue

```

```

END

```

```

GO

```

Извикването на функцията вече ще има вида:

```

SELECT ProductID, ProductName, Stock,
       ReorderLevel, Discontinued,
       dbo.fnNeedToReorder(ProductID)
       AS sNeedToReorder
FROM Products

```

Другият начин за използване на тази функция е създаването на изчислима колона в таблицата Products по следния начин:

```

ALTER TABLE Products
ADD NeedToReorder AS dbo.fnNeedToReorder(ProductID)

```

Стойностите в новата колона се изчисляват автоматично, като се използва резултатът, върнат от потребителски дефинираната функция.

Пример 2 Потребителски дефинираните функции могат да бъдат използвани за задаване на определено форматиране, например на телефонните номера:

```
CREATE FUNCTION fnFormatTelephoneNumber
    (@sPhone char(10))
    RETURNS varchar(15)
AS
BEGIN
    DECLARE @sPhoneFormat varchar(15)

    IF LEN(@sPhone) < 10
        SET @sPhoneFormat = @sPhone
    ELSE SET @sPhoneFormat =
        '(' + LEFT(@sPhone, 3) + ')' + ' ' +
        SUBSTRING(@sPhone, 4, 3) +
        '-' + RIGHT(@sPhone, 4)
    RETURN @sPhoneFormat
END
GO
Резултатът от следното извикване на функцията:
SELECT dbo.fnFormatTelephoneNumber ('1234448888')
        AS FormatTelephoneNumber
```

има вида:

```
FormatTelephoneNumber
-----
(123) 444-8888
```

Пример 3 Функция за форматиране на датата във вида DD/MM/YYYY с водещи нули:

```
CREATE FUNCTION fn2Digits (@sValue varchar(2))
    RETURNS varchar(2)
AS
BEGIN
    IF (LEN(@sValue) < 2)
        SET @sValue = '0' + @sValue

    RETURN @sValue
END
GO
```

Тази функция добавя водеща нула и се извиква от следващата, която осъществява окончателното форматиране на датата.

```
CREATE FUNCTION fnStandardDate (@dtDate datetime)
    RETURNS varchar(10)
AS
BEGIN
    RETURN
        dbo.fn2Digits (CAST(DAY(@dtDate)
                            AS varchar(2))) + '/' +
        dbo.fn2Digits (CAST(MONTH(@dtDate)
```

```

AS varchar(2))) + '/' +
CAST(YEAR(@dtDate)
AS varchar(4))

END
GO
SELECT dbo.fnStandardDate ('3/17/2009')

```

Функции, връщащи таблица

Една функция от този тип връща набор от данни от типа данни *table* в SQL Server. Дефинира се с една SELECT конструкция, определяща таблицата, която функцията трябва да върне като резултат и не може да съдържа други T-SQL конструкции. SELECT конструкцията не може да съдържа ORDER BY, освен когато включва TOP *n*. Общият вид на конструкцията за създаване на дефинирана от потребителя функция от този тип е:

```

CREATE FUNCTION [owner_name.]function_name
    ( [ { @parameter_name [AS] scalar_parameter_data_type
      [ = default ] } [ ,... n ] ] )
RETURNS TABLE
[ WITH [ENCRYPTION] [[,] SCHEMABINDING] ]
[AS]
RETURN [ ( ) select_statement [ ) ]

```

Функции, връщащи таблица, могат да бъдат извиквани, като се включи само името на функцията:

```

[database_name.][owner_name.]function_name
    ([argument_expr][,...])

```

Пример 4 Функция, която връща редовете, съдържащи данните за служителите от даден град, зададен като параметър:

```

CREATE FUNCTION fnGetEmployeesByCity
(@sCity varchar(30))
    RETURNS table
AS
RETURN
(
    SELECT EmployeeID, FirstName, LastName, Address
    FROM Employees
    WHERE City = @sCity
)
GO

```

Функцията се извиква по следния начин:

```
SELECT * FROM fnGetEmployeesByCity('CityA')
```

Резултатът от функцията от този вид може да бъде съединяван с други таблици.

Например:

```

SELECT * FROM fnGetEmployeesByCity('CityA') e
INNER JOIN Sales s
    ON e.EmployeeID = s.EmployeeID

```

Многоструктурни функции, връщащи таблица

При този вид функции изрично се дефинира структурата на таблицата, която се връща като резултат. Задават се имената на колоните и типа на данните им в RETURNS.

Могат да се включат множество и разнообразни T-SQL конструкции. Общият вид на конструкцията за създаване на дефинирана от потребителя функция от този тип е:

```
CREATE FUNCTION [owner_name.]function_name
    ( [ { @parameter_name [AS] scalar_parameter_data_type
      [ = default ] } [,... n ] ] )
RETURNS @return_variable TABLE
    ( { column_definition | table_constraint } [,... n ] )
    [ WITH [ENCRYPTION] [[,] SCHEMABINDING] ]
[AS]
BEGIN
    function_body
RETURN
END
```

Пример 5 Функцията от пример 4 може да се напише по следния начин:

```
CREATE FUNCTION fnGetEmployeesByCity2
(@sCity varchar(30))
RETURNS @tblMyEmployees table
(
    FirstName varchar(20),
    LastName varchar(40),
    Address varchar(120)
)
AS
BEGIN
    INSERT INTO @tblMyEmployees
        (FirstName, LastName, Address)
    SELECT FirstName, LastName, Address
    FROM Employees
    WHERE City = @sCity
    ORDER BY LastName -- възможно е задаване на сортиране

RETURN
END
GO
```

Функцията може да бъде извикана по следния начин:

```
SELECT * FROM fnGetEmployeesByCity2('CityA')
```

Тъй като при този вид функции е възможно въвеждане на допълнителен код, примерът може да се модифицира така, че да се въвежда пояснителен текст в колоната Address при подаване на град, в който няма служители:

```
ALTER FUNCTION fnGetEmployeesByCity2
(@sCity varchar(30))
RETURNS @tblMyEmployees table
(
    FirstName varchar(20),
    LastName varchar(40),
    Address varchar(120)
)
AS
BEGIN
    INSERT INTO @tblMyEmployees
        (FirstName, LastName, Address)
```

```

SELECT FirstName, LastName, Address
FROM Employees
WHERE City = @sCity
ORDER BY LastName

IF NOT EXISTS (SELECT * FROM @tblMyEmployees)
    INSERT INTO @tblMyEmployees (Address)
    VALUES ('Не са намерени служители в
            избрания град.')

RETURN
END
GO
SELECT * FROM fnGetEmployeesByCity2('OtherCityA')

```

Потребителски дефинираните функции поддържат *рекурсия*. Допускат се най-много 32 нива извиквания, след което се генерира грешка.

Задачи

Задача 1. Да се напише потребителски дефинирана функция, която да връща:

1.1. частта на датата от стойност от тип *datetime*;

1.2. частта на часа от стойност от тип *datetime*.

Задача 2. Да се напише потребителски дефинирана функция, която да връща броя на работните дни между две дати (т.е. без съботи и недели), изключвайки тези дати.

Създаване на тригери

Тригерът е специален вид съхранена процедура, която се стартира автоматично от СУБД при опит за модифициране на данните, с които тригерът е свързан (за разлика от съхранените процедури, които се изпълняват само когато бъдат изрично извикани). Тригерът може да бъде настроен да се стартира, когато данните бъдат променени по някакъв начин, т.е. чрез конструкциите INSERT, UPDATE и/или DELETE.

Тригерите могат да се използват за:

- осигуряване на начин за ограничаване на стойностите на данните;
Тригерите са сходни с ограниченията CHECK, тъй като системата автоматично активира тригери и ограничения CHECK, когато се прави опит за изменение на данните. Ограниченията CHECK извършват доста прости типове проверки на данните, докато тригерите могат да извършат значително сложни ограничения върху данните.
- реализиране на актуализации, включващи промени в няколко таблици;
- проследяване на промените в данните в таблиците на базата от данни.

Създаване и използване на тригери

Общият вид на конструкцията за създаване на тригер в Microsoft SQL Server 2005 е:

```
CREATE TRIGGER trigger_name
ON {table_name | view_name}
{FOR | AFTER | INSTEAD OF}
[DELETE] [,] [INSERT] [,] [UPDATE]
AS
    sql_queries
```

Тригерът се изпълнява по веднъж за всяка конструкция INSERT, UPDATE или DELETE независимо от броя на редовете, които засяга тя (0, 1 или повече). За да се избегне изпълнение на код, който е безсмислен, често първата конструкция в тригера проверява колко реда са засегнати. В SQL Server за тази цел се използва функцията @@ROWCOUNT.

Тригерите са два типа: AFTER и INSTEAD OF. Тригерите INSTEAD OF се изпълняват вместо модификациите, указани в тяхната дефиниция. За всяко действие може да се дефинира най-много един тригер INSTEAD OF за дадена таблица или изглед. Тригерите AFTER (или FOR) се задействат, след като данните бъдат модифицирани, т.е. добавени, изтрити или променени. За всяко действие могат да се дефинират повече от един тригери AFTER за дадена таблица. Този вид тригери не могат да се създават за изгледи.

Тригерът AFTER се стартира, след като конструкцията за модифициране на данни приключи работата си, но преди тази модификация да бъде потвърдена в базата данни. Както конструкцията, така и всякакви модификации, извършени в тригера, се третират неявно като транзакция. Поради това тригерът може да превърти назад модификацията на данните, предизвикали стартирането му чрез ROLLBACK TRANSACTION.

Тригерът има достъп до първоначалния образ и крайния образ на данните чрез специалните псевдотаблици *inserted* и *deleted*. Тези две таблици имат същия набор колони като основната таблица, върху която се извършват промените и толкова редове, колкото са засегнати от съответната конструкция за вмъкване, изриване или променяне на данни. Таблицата *inserted* съдържа редовете, повлияни от модификацията, т.е.

вмъкнати или променени и то техния краен образ. Таблицата *deleted* съдържа редовете, засегнати от съответната конструкция за променяне или изтриване и то техния първоначален образ. Може да се проверяват началните и крайните стойности на определени колони и да се предприемат действия в зависимост от тях. Псевдотаблиците *inserted* и *deleted* не могат да се модифицират директно, защото те не съществуват в действителност. Данните от тези таблици могат само да бъдат извлечени чрез заявка.

SQL Server позволява да се дефинира кой тригер AFTER да се стартира първи и кой последен, като се използва системната съхранена процедура `sp_settriggerorder`, която има следните параметри:

```
sp_settriggerorder [@triggername = ] 'trigger_name',
                  [@order = ] 'value',
                  [@stmttype = ] 'statement_type'
```

Параметърът `@triggername` определя името на тригера AFTER.

На `@order` може да се зададе една от следните стойности: *First*, *Last*, *None*. Тази процедура може да укаже кой тригер AFTER да се стартира първи и кой последен от всички тригери от този вид за дадена таблица. Останалите тригери AFTER в таблицата се изпълняват в произволен ред.

Параметърът `@stmttype` определя коя SQL конструкция задейства тригера.

Възможните стойности са DELETE, INSERT или UPDATE.

Например:

```
EXEC sp_settriggerorder
    @triggername = 'MyTrigger',
    @order = 'First',
    @stmttype = 'UPDATE'
```

Забраняване на тригери се извършва чрез следната конструкция:

```
ALTER TABLE table_name
DISABLE TRIGGER {ALL | trigger_name [,...]}
```

Разрешаване на тригери става с аналогична конструкция с опцията ENABLE.

Изтриване на тригер се осъществява чрез следната команда:

```
DROP TRIGGER trigger_name
```

Променяне на тригер се извършва чрез следната конструкция:

```
ALTER TRIGGER trigger_name
ON {table_name | view_name}
{FOR | AFTER | INSTEAD OF}
[DELETE] [,] [INSERT] [,] [UPDATE]
AS
    sql_queries
```

Чрез **IF UPDATE(*column_name*)** може да се зададе да се изпълнят определени действия в тригера, ако промените са отразили точно определени колони.

Пример 1 Тригер, който забранява изтриването на редове в таблицата, съдържаща данните за клиентите:

```
CREATE TRIGGER No_DeleteCustomers
ON Customers
INSTEAD OF DELETE
AS
    IF @@ROWCOUNT = 0 RETURN
```

```

        RAISERROR('Не е разрешено изтриване на
                    редове в таблицата.', 11, 1)
GO
-- стартиране на тригера:
DELETE FROM Customers
WHERE CustomerID = 1

```

Пример 2 Тригер, който изчислява отстъпката на дадена по идентификатора си продажба, т.е. изчислява и актуализира стойността на колоната Discount в таблицата за продажбите Sales в зависимост от стойностите на TotalForSale и CurrentBalance: за обща стойност на продажбата над 100 и текущ баланс на клиента над 1000 да се зададе отстъпка 15%; за обща стойност на продажбата над 100 и текущ баланс на клиента под 1000 – 10%; за обща стойност на продажбата под 100 и текущ баланс на клиента над 1000 – 12%; за обща стойност на продажбата под 100 и текущ баланс на клиента под 1000 – 0%:

```

CREATE TRIGGER SalesAffectsDiscount
ON Sales
AFTER INSERT, UPDATE
AS
    IF @@ROWCOUNT = 0 RETURN

    IF UPDATE(TotalForSale)
    BEGIN
        UPDATE Sales
        SET Discount =
            CASE
                WHEN i.TotalForSale >= 100
                     AND c.CurrentBalance >= 1000
                THEN 0.15
                WHEN i.TotalForSale >= 100
                THEN 0.10
                WHEN c.CurrentBalance >= 1000
                THEN 0.12
                ELSE 0
                -- ако се пропусне, се въвежда стойност NULL
            END
        FROM Sales s
        INNER JOIN inserted i
            ON s.SaleID = i.SaleID
        INNER JOIN Customers c
            ON s.CustomerID = c.CustomerID
    END
GO

```

Тъй като отстъпката, която се пресмята чрез този тригер, зависи от изчислимите колони TotalForSale и CurrentBalance, поддържани актуални също със съответни тригери, трябва да се осигури стартирането на тригера SalesAffectsDiscount да се осъществява след тях. За целта може да се зададе тригерът да се изпълнява последен при въвеждане и променяне на редове в Sales:

```

EXEC sp_settriggerorder
    @triggername = 'SalesAffectsDiscount',
    @order = 'Last',
    @stmttype = 'UPDATE'
EXEC sp_settriggerorder
    @triggername = 'SalesAffectsDiscount',
    @order = 'Last',
    @stmttype = 'INSERT'

```

Пример 3 Тригер, който забранява продажбата на продукти, определени в таблицата Products като продукти с преустановена продажба:

```
CREATE TRIGGER SaleDetailNotDiscontinued
ON SaleDetails
AFTER INSERT, UPDATE
AS
IF @@ROWCOUNT = 0 RETURN

IF UPDATE ( ProductID )
BEGIN
    IF EXISTS
    ( SELECT *
      FROM Inserted i
      INNER JOIN Products p
        ON i.ProductID = p.ProductID
      WHERE p.Discontinued = 1 )
    BEGIN
        RAISERROR('Продуктът е с преустановена
                  продажба. Транзакцията е отменена.',
                  16, 1)
        ROLLBACK TRANSACTION
    END
END
```

Каскадни и рекурсивни тригери

Ако тригерът модифицира данни в някоя друга таблица и тя има тригер, това дали този тригер ще бъде стартиран зависи от текущата стойност на `sp_configure` за опцията *nested triggers*. Ако тази опция е зададена на 1 (`EXEC sp_configure 'nested triggers', 1`), която е и стойността по подразбиране, тригерите могат да се извикват *каскадно* до максимална последователност от 32. Ако дадена операция предизвика стартиране на повече от 32 тригера последователно, пакетът ще бъде отхвърлен и всякакви транзакции ще бъдат превъртени назад. По този начин се предотвратява получаване на безкраен цикъл.

Ако тригерът модифицира данни в същата таблица, в която той съществува, използването на същата операция (`INSERT`, `UPDATE` или `DELETE`) по подразбиране няма да стартира тригера отново. Това поведение може да се промени, позволявайки тригерите да бъдат *рекурсивни*. Контролира се за конкретната база от данни чрез задаване на опцията *recursive triggers* на стойност `TRUE` (`EXEC sp_dboption 'database_name', 'recursive triggers', TRUE`). По подразбиране стойността на опцията е `FALSE`. Необходимо е контролиране на рекурсията и нейното правилно приключване от разработчика. Няма да се стигне до състояние на безкраен цикъл, тъй като както и при съхранените процедури тригерите могат да бъдат влягани един в друг само до достигане на максимално ниво 32. Дори и да не е включена опцията за рекурсивни тригери, ако съществуват отделни тригери за операциите `INSERT`, `UPDATE` и `DELETE`, един от тригерите за таблица може да предизвика стартиране на други тригери за същата таблица, но само ако е зададено `EXEC sp_configure 'nested triggers', 1`, т.е. ако са разрешени каскадните тригери.

Използване на тригери за изпълняване на действия за запазване на целостта на данните

В SQL Server 2000 са въведени декларативни ограничения за поддържане на каскадна цялост на връзките, които чрез CREATE TABLE или ALTER TABLE могат да се зададат в дефиницията на една таблица. Например чрез конструкцията ALTER TABLE се добавят по следния начин:

```
ALTER TABLE table_name
ADD [CONSTRAINT constraint_name]
    [FOREIGN KEY (col_name1 [, ...])]
    REFERENCES ref_table_name (ref_col_name1 [, ...])
    [ON UPDATE {CASCADE | NO ACTION}]
    [ON DELETE {CASCADE | NO ACTION}]
```

В предишните версии за реализиране на каскадните изтривания и каскадните обновявания на външни ключове могат да се използват само тригери. Все пак при необходимост от изпълняване на допълнителни действия към стандартните действия, поддържащи цялост на данните, ще трябва да се създаде тригер. Например след изтриване на редовете от свързаните таблици да се извърши добавяне на изтритите редове в друга таблица за по-късен преглед. За целта може да се използва тригер INSTEAD OF **ИЛИ** AFTER, чийто код програмно да реализира каскадното изтриване, както и другите специфични за конкретните условия действия.

Пример 4

```
CREATE TRIGGER DelCascadeSale
ON Sales
AFTER DELETE
AS

DECLARE @i int, @j int

SET @i = @@ROWCOUNT

IF @i = 0 RETURN

SET NOCOUNT ON

INSERT INTO SalesHistory
    (SaleID, CustomerID,
     EmployeeID, SaleDate,
     TotalForSale, DiscountFromTotal,
     ProductID, Price,
     Quantity, DiscountFromPrice)
SELECT d.SaleID, d.CustomerID,
       d.EmployeeID, d.SaleDate,
       d.TotalForSale, d.Discount,
       sd.ProductID, sd.Price,
       sd.Quantity, sd.Discount
FROM SaleDetails sd
RIGHT JOIN deleted d
    ON sd.SaleID = d.SaleID

DELETE FROM SaleDetails
FROM SaleDetails sd
INNER JOIN deleted d
    ON sd.SaleID = d.SaleID

SET @j = @@ROWCOUNT

IF @i > 0 OR @j > 0
    RAISERROR('%d реда от SaleDetails са
    изтрити като резултат от
```

```

        изтриването на %d реда в
        таблицата Sales и са добавени в
        SalesHistory.', 11, 1, @j, @i)
GO
DELETE FROM Sales
WHERE SaleID = 2

```

Проверките за нарушаване на ограниченията се извършват преди активирането на тригерите. Ако бъде открито нарушаване на ограничение, конструкцията се прекратява и изпълнението никога не достига до тригера, т.е. той не се активира. За по-добра четимост може все пак да се декларира ограничение FOREIGN KEY в скриптовете с CREATE TABLE, а след това да се забрани ограничението външен ключ със следната конструкция:

```

ALTER TABLE table_name
NOCHECK CONSTRAINT {ALL | constraint_name [,...]}

```

По този начин се гарантира, че ограничението ще се вижда в изхода на `sp_help table_name`, в диаграми и други такива процедури.

Пример 5 Тригерът UpdateProductName се стартира при актуализиране на колона с ограничение за уникалност. Необходимо е при конструкции, обновяващи множество редове, тригерът да игнорира тези стойности, които нарушават ограничението за уникалност, всички останали да се приемат (за действието INSERT този резултат може да се постигне чрез дефиниране на уникален индекс с опцията IGNORE_DUP_KEY).

```

CREATE TRIGGER UpdateProductName
ON Products
INSTEAD OF UPDATE
AS
    IF @@ROWCOUNT = 0 RETURN

    IF UPDATE(ProductName)
    BEGIN
        UPDATE Products
        SET ProductName = i.ProductName
        FROM Products p
        INNER JOIN inserted i
            ON p.ProductID = i.ProductID
        WHERE i.ProductName NOT IN
            (SELECT ProductName FROM Products)
            AND i.ProductName NOT IN
            (SELECT ProductName FROM inserted i1
             WHERE i1.ProductID < i.ProductID)
    END
    ELSE -- ако модификацията засяга друга колона
    UPDATE Products
    SET CategoryID = i.CategoryID,
        SupplierID = i.SupplierID,
        Price = i.Price,
        Stock = i.Stock,
        ReorderLevel = i.ReorderLevel,
        Discontinued = i.Discontinued
    FROM Products p
    INNER JOIN inserted i
        ON p.ProductID = i.ProductID
GO

```

За проследяване на действието на тригера, нека е създадена временна таблица #MyTemp със следните примерни данни:

```

SELECT t.ProductID AS MyID,
       t.ProductName AS MyName,
       p.ProductID, p.ProductName

```



```

UPDATE Products
  SET ProductName = i.ProductName
FROM Products p
INNER JOIN inserted i
  ON p.ProductID = i.ProductID
WHERE i.ProductName NOT IN
      (SELECT ProductName
       FROM Products)
      AND i.ProductName NOT IN
      (SELECT ProductName
       FROM inserted i1
       WHERE i1.ProductID < i.ProductID)
END
END
ELSE -- ако модификацията засяга друга колона
UPDATE Products
  SET CategoryID = i.CategoryID,
      SupplierID = i.SupplierID,
      Price = i.Price,
      Stock = i.Stock,
      ReorderLevel = i.ReorderLevel,
      Discontinued = i.Discontinued
FROM Products p
INNER JOIN inserted i
  ON p.ProductID = i.ProductID
GO

```

Задачи

Задача 1. Да се създаде тригер, забраняващ променянето на продажна цена, която да е по-ниска от съответната доставна цена на продукт.

Задача 2. Да се създаде тригер, забраняващ въвеждането на нов ред в *SaleDetails*, в който продажната цена да е по-ниска от съответната доставна цена на продукт. Ако не се зададе стойност или стойността е по-малка от доставната, да се въвежда стойността на доставната за новия ред.

Задача 3. Да се създаде тригер, който актуализира отстъпката от продажната цена за покупка на голямо количество от даден продукт.

Задача 4. Да се създаде тригер, който актуализира общата сума на продажба.

Задача 5. Да се създаде тригер, който актуализира текущия баланс на клиент.

Задача 6. Да се създаде тригер, който забранява продажбата на продукт в количество, по-голямо от наличното. Тригерът да актуализира наличното количество, като го намалява с продаденото.

Задача 7. Да се създаде тригер, който използва таблицата *ProductInformation*, съхраняваща допълнителна информация за продуктите. Нека в таблицата *Products* е добавена колона *InformationFlag* от тип *bit*. Стойност единица в тази колона показва, че съществува поне един съответен ред за този продукт в таблицата *ProductInformation*; стойност нула означава, че не съществува нито един съответен ред с допълнителна информация за продукта. Тригерът да актуализира стойността на колоната *InformationFlag* при добавяне или изтриване на редове в

ProductInformation. Тази колона да се поддържа актуална и при променяне на идентификатора на продукта, за който се отнася допълнителната информация.

Използване на курсори

Релационните бази от данни като Microsoft SQL Server са ориентирани към работа с набори от данни. Това означава, че всяка конструкция обработва набор от данни, състоящ се от нула, един или повече редове с данни. От друга страна повечето езици за програмиране са базирани на записи, т.е. обработката се извършва запис по запис.

SQL Server предоставя възможност за работа с курсори като връзка между двата подхода – между подхода, ориентиран към набори и подхода, базиран на записи. Курсорът позволява да се избере набор от записи, представляващи неговото съдържание. След създаването на курсор с дадено име могат да се извлекат редове от него и да се работи с всеки ред отделно от другите редове. Това е полезно, когато определени редове в таблицата трябва да бъдат обработени по начин, който не може да се осъществи лесно с конструкциите, ориентирани към набори.

Деклариране на курсори

За да се използва даден курсор, първо той трябва да се декларира. При деклариране на курсора могат да се зададат опции, които определят неговото поведение. Общият вид на конструкцията за деклариране на курсор изглежда по следния начин:

```
DECLARE cursor_name CURSOR FOR select_statement
```

Например:

```
DECLARE cur_Customer CURSOR FOR SELECT * FROM Customers
```

Декларираният курсор осигурява възможност за работа с отделните редове на набора от данни, получени от конструкцията SELECT. По подразбиране се позволява актуализиране и изтриване на редовете, но може чрез определени опции точно да се зададат тези и други характеристики на курсора. Transact-SQL курсорите могат да използват или синтаксиса, зададен от ANSI с опциите `INSENSITIVE` и `SCROLL`, или пълния диапазон от типове курсори, който се нарича Transact-SQL Extended Syntax.

Когато се използва *ANSI спецификацията* за деклариране на курсор, характеристиките за поведението се задават преди ключовата дума `CURSOR`:

```
DECLARE cursor_name [INSENSITIVE] [SCROLL]
CURSOR FOR select_statement
[FOR {READ ONLY | UPDATE [OF column_list]}]
```

- Ключовата дума `INSENSITIVE` води до създаване на копие от върнатите данни в отделна таблица в базата от данни *tempdb*. Всички действия с данните се извършват върху това изолирано копие. Курсор, деклариран с ключовата дума `INSENSITIVE`, не може да бъде използван за актуализиране на данни.

Например:

```
DECLARE cur_Customer INSENSITIVE CURSOR FOR
SELECT * FROM Customers
```

- Ключовата дума `SCROLL` дефинира *превъртащ* курсор, т.е. може да се осъществява обхождане на данните напред и назад. Ако не е зададена тази опция, ще може да се преминава само към следващ запис – такъв курсор се нарича еднопосочен (*forward-only*). Например:

```
DECLARE cur_Customer SCROLL CURSOR FOR
SELECT * FROM Customers
```

Двете опции могат да се използват едновременно:

```
DECLARE cur_Customer INSENSITIVE SCROLL CURSOR FOR
SELECT * FROM Customers
```

- Използването на READ ONLY с FOR след SELECT конструкцията установява курсора в режим “само за четене”, в резултат на което данните могат да се четат, но не и да се променят. Ако е необходимо да се актуализират записи в даден курсор, той трябва да се декларира с ключовите думи FOR UPDATE, след които е допустимо да се уточни кои колони, изброени със запетайки помежду им, могат да се актуализират в курсора.

Пример 1

```
DECLARE cur_Customer CURSOR FOR
        SELECT * FROM Customers
FOR READ ONLY
```

Пример 2

```
DECLARE cur_Customer CURSOR FOR
        SELECT * FROM Customers
FOR UPDATE OF PhoneNumber
```

Ако не се изброят колони след UPDATE, се допуска променяне на всички, в противен случай се ограничава възможността за актуализиране само до изброените колони.

Отваряне на курсори и извличане на ред от курсор

След като е деклариран даден курсор, той може да се отвори за използване с командата OPEN, която има следния синтаксис:

```
OPEN cursor_name
```

Изпълнението на конструкцията OPEN води до попълване на курсора с данните, които извлича заявката, дефинираща набора от записи на курсора. След това може да се използва курсорът за операции с данните. Отварянето на курсора създава инстанция на курсора и връща неговия набор от записи и ако е необходимо генерира и попълва временни таблици.

Чрез конструкцията FETCH се извлича ред от курсора в зададената посока:

```
FETCH [row_selector] FROM cursor_name
[INTO @v1, @v2, ...]
```

Ако курсорът е еднопосочен, *row_selector* може да бъде само NEXT. Ако курсорът е превъртащ, *row_selector* може да бъде NEXT, PRIOR, FIRST, LAST, ABSOLUTE *n* или RELATIVE *n*. Ако не е зададен *row_selector*, се подразбира NEXT. ABSOLUTE *n* връща $n^{тия}$ ред в набора от данни, като отрицателната стойност показва номера на реда при обратно броене от края на набора резултати. Извличанията, които използват NEXT, PRIOR или RELATIVE *n*, се извършват по отношение на текущата позиция на курсора. След отваряне на курсора текущата му позиция е преди първия ред, т.е. курсорът се позиционира преди първия ред. RELATIVE *n* връща $n^{тия}$ ред по отношение на последно извлечения ред и отрицателната стойност показва, че номерът на реда се брои в обратна посока, като се започне от последно извлечения ред към началото на набора от данни. RELATIVE 0 означава повторно извличане на текущия ред. Например:

```
DECLARE cur_Customer SCROLL CURSOR FOR
        SELECT CustomerID, CompanyName,
               PhoneNumber, Address, City
        FROM Customers
FOR UPDATE OF PhoneNumber
```

```

OPEN cur_Customer

FETCH ABSOLUTE 8 FROM cur_Customer
...
FETCH RELATIVE 7 FROM cur_Customer
...

```

Възможно е използване на променлива от целочислен тип за придвижване в посока към последния запис (при положителни стойности) или за придвижване към първия запис (при отрицателни стойности).

С помощта на следната конструкция може да се прехвърлят стойностите от избраните колони към конкретни променливи, предварително декларирани:

```

DECLARE @CustID int, @CompanyName varchar(50),
        @PhoneNumber char(10),
        @Address varchar(50),
        @City varchar(25)

```

```

FETCH NEXT FROM cur_Customer
INTO @CustID, @CompanyName,
     @PhoneNumber, @Address, @City

```

В този случай стойностите на колоните от следващия ред в набора от записи се присвояват на съответните променливи.

Претърсване на съдържанието на курсора

В повечето случаи конструкцията `FETCH` се изпълнява поне веднъж за всеки ред в резултатния набор. SQL Server предоставя системната функция `@@FETCH_STATUS`, чиято стойност може да се проверява след всяко извличане:

- стойност 0 на тази функция указва, че последното извличане е било успешно;
- стойност 1 указва, че няма повече редове, т.е. извличането ще придвижи курсора извън резултатния набор – след последния ред или преди първия ред;
- стойност 2 указва, че редът вече не съществува в курсора, т.е. той е бил изтрят от базата от данни след отварянето на курсора или е бил актуализиран по начин, който вече не съответства на критериите на конструкцията `SELECT`, генерирала резултатния набор.

Тази функция може да се използва за претърсване на съдържанието на курсора, за да се установи къде са редовете, които ще се актуализират или изтриват. Например:

```

DECLARE @EmpID int, @FirstName varchar(25),
        @LastName varchar(25)
DECLARE cur_Employee CURSOR FOR
        SELECT EmployeeID, FirstName, LastName
        FROM Employees

```

```

OPEN cur_Employee

```

```

-- извличане на първия ред
FETCH NEXT FROM cur_Employee
INTO @EmpID, @FirstName, @LastName

```

/ Проверка за това дали има още редове за извличане – ако стойността на функцията `@@FETCH_STATUS` е различна от 0, следва, че или има проблем с*

курсора, или няма повече редове – и в двата случая това показва, че трябва да се спре търсенето. */

```
WHILE @@FETCH_STATUS = 0
BEGIN
    -- конструкция с условие за търсене
    IF @LastName = 'name' PRINT 'message'

    -- извличане на следващия ред
    FETCH NEXT FROM cur_Employee
    INTO @EmpID, @FirstName, @LastName
END
```

Актуализиране на курсори

В рамките на курсора може да се променят или изтриват редове чрез конструкциите UPDATE и DELETE. Конструкцията UPDATE обновява реда от таблицата, съответстващ на текущия ред в курсора:

```
UPDATE table_name
SET assignment_list
WHERE CURRENT OF cursor_name
```

Нарича се *позиционирано обновяване*, чрез което се променя само последния извлечен ред. Променяне е възможно винаги, когато курсорът не е зададен като FOR READ ONLY. Ако в дефиницията на курсора е уточнено с FOR UPDATE OF кои колони могат да се променят, конструкцията UPDATE може да се отнася само за посочените колони.

Например:

```
UPDATE Employees
SET LastName = 'new_name'
WHERE CURRENT OF cur_Employee
```

Посочва се името на таблицата от курсора, която ще се променя, изразът SET е същия като във всяка конструкция UPDATE. Ключовите думи CURRENT OF показват, че актуализирането ще се извърши в моментното положение на курсора, чието име е посочено веднага след тези ключови думи.

Конструкцията DELETE изтрива от зададената таблица реда, съответстващ на текущата позиция на курсора:

```
DELETE FROM table_name
WHERE CURRENT OF cursor_name
```

Това е т.нар. *позиционирано изтриване*. Изтриването се допуска винаги, когато курсорът не е READ ONLY. Например:

```
DELETE FROM Employees
WHERE CURRENT OF cur_Employee
```

Ключовите думи CURRENT OF показват, че изтриването ще се извърши в моментното положение на посочения курсор.

При използване на Transact-SQL курсорите не е допустимо да се извършва позиционирано вмъкване. Тъй като курсорът може да се разглежда като наименован набор от данни, не може да се прави вмъкване в курсора, а в таблицата, от която той избира редове.

Затваряне и освобождаване на курсори

Когато се приключи работата с даден курсор, той трябва да се затвори. Това се извършва по следния начин:

```
CLOSE cursor_name
```

Затварянето на курсора прекратява активното му действие, т.е. от него вече не могат да се извличат или обновяват, или изтриват редове. Курсорът все още е деклариран и може да бъде отворен отново, без да е необходимо пак да бъде деклариран.

Според ANSI спецификацията курсорите се затварят при издаване на конструкцията COMMIT TRANSACTION, но това не е подразбиращото се поведение на SQL Server с цел по-висока ефективност. Включването на опцията

```
SET CURSOR_CLOSE_ON_COMMIT {ON | OFF}
```

дава възможност за затваряне на всички отворени курсори при изпълнение на COMMIT TRANSACTION или ROLLBACK TRANSACTION.

Освобождаването на курсор има за цел да се отстрани референция към него и се извършва чрез конструкцията:

```
DEALLOCATE cursor_name
```

Тази команда не е част от ANSI спецификацията. След освобождаване на последната референция към курсора, SQL Server освобождава структурите от данни, които съставят курсора. Тъй като вътрешните структури от данни заемат памет в процедурния кеш, би трябвало да се почисти, след като е приключила работата с курсора. След освобождаване на последната референция към курсора, той повече не може да бъде отворен, докато не се декларира отново чрез конструкцията DECLARE.

Курсорни променливи

SQL Server дава възможност да се декларират променливи от тип курсор. Не е допустимо колона от дадена таблица да бъде от тип курсор. За присвояване на стойност на курсорна променлива се използва конструкция SET (не е възможно със SELECT). Курсорните променливи могат да бъдат използвани във всички конструкции за работа с курсори: OPEN, FETCH, CLOSE, DEALLOCATE. Входните параметри на съхранените процедури не могат да бъдат курсорни променливи, но могат да се използват за изходни параметри. В следния пример е показано деклариране на курсор, след което е указано на курсорна променлива да реферира същия резултатен набор:

```
DECLARE cur_Customer CURSOR FOR
    SELECT CompanyName
    FROM Customers
DECLARE @cursor_var cursor
```

```
SET @cursor_var = cur_Customer
```

Може да се зададе стойност на курсорна променлива и като се използва дефиницията на курсора, а не името му. Например:

```
DECLARE @cursor_var cursor
```

```
SET @cursor_var = CURSOR FOR
    SELECT CompanyName FROM Customers
```

Декларираният курсор или курсорната променлива могат да се използват за отваряне на курсора. Декларираното име и курсорната променлива са две референции към една и съща вътрешна структура. След изпълнение на

```
OPEN @cursor_var
```

курсорът е отворен. Ако след това се изпълни

```
OPEN cur_Customer
```

ще се получи грешка, тъй като курсорът вече е отворен. Ако се изпълни CLOSE с някоя от референциите, курсорът се затваря и повече не могат да се извличат редове. Ако с

декларирания курсор се изпълни FETCH, следващ FETCH от курсорната променлива извлича следващия ред. Единственият случай, когато двете референции се разглеждат по различен начин, е в конструкцията DEALLOCATE. Освобождаването на една от референциите означава, че чрез нея вече няма да може да се осъществи достъп до курсора. Вътрешните структури от данни не се освобождават, докато не бъде освободена и последната референция. Ако курсорът все още не е затворен, освобождаването на последната референция го затваря.

Важно приложение на курсорните променливи е за изходни параметри в съхранените процедури. Курсорът се попълва с данни вътре в съхранената процедура с помощта на конструкцията SET и бива върнат във вид на изходен параметър. Когато съхранената процедура приключи изпълнението си, курсорната променлива запазва референция към набора от данни. Това е полезно в случаите, когато трябва като изход от дадена съхранена процедура да се получи набор от редове. Параметрите от тип курсори в съхранените процедури трябва да се декларират с ключовите думи VARYING и OUTPUT в този ред.

Пример 3 Съхранена процедура, която извлича реда с данните на първия назначен с дадена фамилия служител:

```
CREATE PROCEDURE My_sp_cursor
    @PassedCursor cursor VARYING OUTPUT,
    @EmployeeLastName varchar(25)
AS
    DECLARE @LastName varchar(25),
            @HireDate datetime

    SET @PassedCursor = CURSOR FOR
        SELECT LastName, HireDate
        FROM Employees
        ORDER BY HireDate ASC
    OPEN @PassedCursor

    FETCH NEXT FROM @PassedCursor
        INTO @LastName, @HireDate

    WHILE @@FETCH_STATUS = 0
    BEGIN
        IF @LastName = @EmployeeLastName
            -- намира първия назначен с дадено име
            RETURN 1
        FETCH NEXT FROM @PassedCursor
            INTO @LastName, @HireDate
    END
    RETURN 2
GO
/* Спартиране на съхранената процедура, след което се извежда служителя,
назначен непосредствено след дадения по фамилията си служител: */
DECLARE @PassedCursor cursor, @r int,
        @LastName varchar(25),
        @HireDate datetime
EXEC @r = My_sp_cursor
    @PassedCursor = @PassedCursor OUTPUT,
```

```

        @EmployeeLastName = 'Иванов'
IF @r = 2
    RAISERROR('Не е намерен служител
              с тази фамилия.',11,1)
ELSE
    BEGIN
        FETCH NEXT FROM @PassedCursor
            INTO @LastName, @HireDate
        /* Извлича се служителят, който е назначен след този с дадено име,
        ако има такъв: */
        IF @@FETCH_STATUS = -1
            RAISERROR('Последно назначен.', 11, 1)
        ELSE SELECT @LastName, @HireDate
    END
CLOSE @PassedCursor
DEALLOCATE @PassedCursor
GO

```

Динамично създаване на кръстосани заявки чрез генериране на CASE изрази

Една кръстосана заявка може да бъде генерирана динамично с помощта на курсор.

Пример 4 Динамично генериране на заявка, която определя общата сума от продажбите за всеки магазин за всяка категория продукти.

Нека категориите в таблицата `Categories` имат наименования *a*, *b*, *c*. Тогава за създаване на разглежданата кръстосана заявка са необходими следните CASE изрази:

```

SUM(CASE CategoryName
      WHEN 'a'
      THEN sd.quantity*sd.price*(1-sd.discount)
      ELSE 0
    END) AS 'a',
SUM(CASE CategoryName
      WHEN 'b'
      THEN sd.quantity*sd.price*(1-sd.discount)
      ELSE 0
    END) AS 'b',
SUM(CASE CategoryName
      WHEN 'c'
      THEN sd.quantity*sd.price*(1-sd.discount)
      ELSE 0
    END) AS 'c',

```

Те могат да се генерират със SELECT заявката:

```

SELECT 'SUM(CASE CategoryName
          WHEN '' + CategoryName + ''
          THEN sd.quantity*sd.price*(1-sd.discount)
          ELSE 0
        END) AS [' + CategoryName + '],'
FROM Categories

```

Чрез нея се дефинира курсор, чието съдържание се претърсва, за да се извлекат изразите, необходими за получаване на кръстосаната заявка.


```

DECLARE @st varchar(8000), @cn varchar(200)

DECLARE categories CURSOR FOR
    SELECT 'SUM(CASE CategoryName
        WHEN '' + CategoryName + ''
        THEN sd.quantity*sd.price*(1-sd.discount)
        ELSE 0
        END) AS [' + CategoryName + '], '
    FROM Categories

SET @st = ''

OPEN categories

FETCH NEXT FROM categories INTO @cn

WHILE @@FETCH_STATUS = 0
    BEGIN
        SET @st = @st + @cn
        FETCH NEXT FROM categories INTO @cn
    END

CLOSE categories
DEALLOCATE categories

SET @st =
    'SELECT e.StoreID, st.StoreName,
        SUM(sd.quantity*sd.price*(1-sd.discount))
        AS Total, ' +
    STUFF(@st, LEN(@st), 1, ' ') +
    -- за отстраняване на последната запетая
    'FROM Categories c
    INNER JOIN Products p
        ON c.CategoryID = p.CategoryID
    INNER JOIN SaleDetails sd
        ON p.ProductID = sd.ProductID
    INNER JOIN Sales s
        ON s.SaleID = sd.SaleID
    INNER JOIN Employees e
        ON e.EmployeeID = s.EmployeeID
    INNER JOIN Stores st
        ON st.StoreID = e.StoreID
    GROUP BY e.StoreID, st.StoreName'

EXECUTE (@st)
GO

```

Примери за предимството от използване на системните таблици

В някои случаи е изключително полезно познаването на архитектурата на SQL Server системните таблици и начина, по който към тях ефективно да бъдат отпращани

заявки. Най-често използваните системни таблици за осъществяване на достъп до тях и извличане на информация са syscolumns, sysobjects, sysindexes.

Пример 5 Извеждане на имената на всички колони и на таблиците, в които се намират, съдържащи даден символен низ (например 'name').

```
SELECT a.name AS column_name, b.name AS table_name
FROM syscolumns a
INNER JOIN sysobjects b ON a.ID = b.ID
WHERE b.type = 'u' AND a.name LIKE '%name%'
```

Таблицата sysobjects съдържа един ред за всеки обект в базата от данни – системни и потребителски таблици, ограничения, изгледи, съхранени процедури и т.н. Таблицата syscolumns, от друга страна съдържа ред за всяка колона в таблица, колона в изглед и ред за всеки параметър на съхранена процедура.

Задаването на условие b.type = 'u' ограничава търсенето само върху потребителските таблици. Останалите възможни стойности на тази колона са:

- C – CHECK ограничение
- D – Default или DEFAULT ограничение
- F – FOREIGN KEY ограничение
- K – PRIMARY KEY или UNIQUE ограничение
- L – Log
- P – Съхранена процедура
- R – Правило
- RF – Съхранена процедура, реализираща филтър за репликация
- S – Системна таблица
- TR – Тригер
- U – Потребителска таблица
- V – Изглед
- X – Разширена съхранена процедура
- FN – Скаларна функция
- IF – Функция, връщаща таблица
- TF – Многоструктурна функция, връщаща таблица

Пример 6 Извеждане на кода на създадените тригери.

Кодът на тригерите, както и на изгледите, съхранените процедури (системни или създадени от потребителя) се съхранява в колоната text на системната таблица syscomments:

```
SELECT a.text
FROM syscomments a
INNER JOIN sysobjects b ON a.ID = b.ID
WHERE b.type = 'TR'
```

За показване на кода на даден по името си обект (изглед, тригер, съхранена процедура, дефинирана от потребителя функция) може да се използва системната съхранена процедура sp_helptext [@objname =] 'name'. Например:

```
EXEC sp_helptext 'MyObject_name'
```

Друг начин за получаване на същия резултат е следната заявка:

```
SELECT a.text
FROM syscomments a
INNER JOIN sysobjects b ON a.ID = b.ID
WHERE b.name = 'MyObject_name'
```

Пример 7 Извеждане на броя на редовете на всяка таблица в текущата база от данни.

Необходимата информация за имената на всички съществуващи потребителски таблици в базата от данни се извлича от системната таблица `sysobjects`. Генерира се динамичен SQL израз за намиране на броя на редовете на всяка потребителска таблица.

```
SET NOCOUNT ON

DECLARE @table nvarchar(128), @sql nvarchar(200)

DECLARE table_cursor CURSOR FOR
    SELECT name
    FROM sysobjects WHERE type = 'u'
    ORDER BY 1

OPEN table_cursor

FETCH NEXT FROM table_cursor INTO @table

WHILE @@FETCH_STATUS = 0
    BEGIN
        SELECT 'Броят на редовете в ' + @table

        SELECT @sql = 'SELECT COUNT(*) FROM ' + @table

        EXEC (@sql)

        FETCH NEXT FROM table_cursor INTO @table
    END
CLOSE table_cursor
DEALLOCATE table_cursor
```

Пример 8 Извеждане на информация за всички индекси, създадени във всички таблици на текущата база от данни.

Системната таблица `sysindexes` съдържа един ред за всеки индекс и таблица в базата от данни. Ако колоната `indID` на таблицата `sysindexes` има стойност 0, то този ред представя потребителска или системна таблица без клъстериран индекс. Стойност на `indID`, равна на 1, означава клъстериран индекс; стойност, по-голяма от 1 – неклъстериран индекс. Примерът извежда информация на потребителя дали индексът е клъстериран или не.

Системната функция `INDEX_COL('table', index_id, key_id)` връща името на индексиранията колона, където първият аргумент е името на таблицата, в която е създаден индексът, `index_id` е идентификатора на индекса, `key_id` е идентификатора на индексиранията колона – цяло число между 1 и 16.

-- Деклариране на използваните в примера променливи:

```
DECLARE @id int, @msg varchar(2000),
        @indID smallint, @indname nvarchar(128),
        @indkey tinyint, @table nvarchar(128)
```

-- Изключване на извеждането на броя на върнатите редове:

```
SET NOCOUNT ON
```

```

DECLARE table_cursor CURSOR FOR
    SELECT id, name
    FROM sysobjects WHERE type = 'u'
    ORDER BY 2

OPEN table_cursor

FETCH NEXT FROM table_cursor INTO @id, @table

-- външен цикъл:
WHILE @@FETCH_STATUS = 0
    BEGIN
        SELECT 'Индекси за таблицата' + @table + ':'

        /* Осъществява се търсене на всички индекси във всички таблици.
        Ако indID = 0, редът съдържа информация за самата таблица, а не
        за индекс. Затова тази стойност не се взема предвид в другия курсор.
        */

        DECLARE index_cursor CURSOR FOR
            SELECT indID, name
            FROM sysindexes
            WHERE id = @id AND indID > 0 AND indID < 255

        OPEN index_cursor

        FETCH NEXT FROM index_cursor
            INTO @indID, @indname

        WHILE @@FETCH_STATUS = 0
            BEGIN

                /* Извежда се името и номера на индекса. Ако indID = 1,
                индексът е клъстериран. Тази информация се извежда в
                съобщението. */

                IF @indID = 1
                    SET @msg = 'Индекс номер 1 е ' +
                        @indname +
                        '. Това е клъстериран индекс върху: '
                ELSE SET @msg = 'Индекс номер ' +
                    CONVERT(varchar(3), @indID) +
                    ' е ' + @indname +
                    '. Това е неклъстериран индекс върху: '

                SET @indkey = 1

                WHILE @indkey <= 16 AND
                    INDEX_COL(@table, @indID, @indkey)
                    IS NOT NULL

```

```

BEGIN
    SET @msg = @msg +
        INDEX_COL(@table, @indID, @indkey) +
        ','

    SET @indkey = @indkey + 1
END

PRINT @msg

FETCH NEXT FROM index_cursor
    INTO @indID, @indname
END
CLOSE index_cursor
DEALLOCATE index_cursor
FETCH NEXT FROM table_cursor
    INTO @id, @table
END -- на външния цикъл
CLOSE table_cursor
DEALLOCATE table_cursor

```

В изведената информация има индекси с имена, започващи със символния низ '_WA_Sys_'. Те са псевдо-индекси, автоматично създадени и поддържани от SQL Server. Биват дефинирани върху колоните, до които най-често се осъществява достъп, за да се оптимизира производителността. Възможно е да се изключи извеждането на информация за тези псевдо-индекси, като се промени заявката в декларацията на втория курсор по следния начин:

```

DECLARE index_cursor CURSOR FOR
    SELECT indID, name
    FROM sysindexes
    WHERE id = @id AND indID > 0 AND indID < 255
    AND name NOT LIKE '_WA_Sys_%'

```

Пример 9 Съхранена процедура, която позволява да се изтрият всички създадени от потребителя индекси на дадена по името си таблица.

Използва се системната функция OBJECT_ID('object'), която връща идентификационния номер на обекта в текущата база от данни.

```

CREATE PROCEDURE Drop_Indexes
    @table nvarchar(128)    -- името на таблицата
AS
DECLARE @indname varchar(128), -- име на индекса
        @objID int           -- идентификатор на таблицата

SET @objID = OBJECT_ID (@table)
-- Връща идентификатора на обекта в базата от данни.
IF @objID IS NULL
BEGIN
    RAISERROR (15001, -1, -1, @table)
    -- TableName does not exist.
    RETURN 1
END

```

```

DECLARE index_cursor CURSOR FOR
    SELECT name FROM sysindexes
    WHERE id = @objID AND indID > 0 AND indID < 255

OPEN index_cursor
FETCH NEXT FROM index_cursor INTO @indname
IF @@FETCH_STATUS = -1
    BEGIN
        RAISERROR(15472,-1,-1)
        --'Object does not have any indexes.'
        CLOSE index_cursor
        DEALLOCATE index_cursor
        RETURN 2
    END

WHILE @@FETCH_STATUS=0
    BEGIN
        EXEC ('DROP INDEX ' + @table + '.' + @indname)

        FETCH NEXT FROM index_cursor
            INTO @indname
    END
CLOSE index_cursor
DEALLOCATE index_cursor
GO

/* Създаване на таблица, върху която да се приложи съхранената процедура:
*/
SELECT * INTO new_table
FROM Employees WHERE 1 = 2

CREATE INDEX LastName
ON new_table (LastName)

CREATE UNIQUE INDEX UniqueEGN
ON new_table (EGN)
    WITH IGNORE_DUP_KEY

CREATE INDEX StoreID
ON new_table (StoreID)
GO
EXEC Drop_Indexes 'new_table'

```

Задачи

Задача 1. Да се напише код с използване на курсор за извеждане на общите суми от продажбите за последните три месеца по магазини във вида:

Последните три месеца	Магазин А	Магазин В	Магазин С	Общо
януари	6000	5000	3000	14000
февруари	5500	6500	3210	15210
март	6800	3214	5432	15446

Задача 2. Нека е дадена таблица Totals с две колони – първата DateTotal съдържа датите на обобщаване на продажбите, а втората колона DayTotal съдържа сумите на продажбите за съответните дати, но са извлечени само данните за дните с общи суми на дневните продажби, надвишаващи средната стойност на общите суми на дневните продажби за всички дни.

2.1. Трябва да се намери промяната в колоната DayTotal между всяка дата и следващата в таблицата с високи стойности на продажбите във вида:

Current Date	Previous date	Current value	Previous value	Difference
2008-01-04	NULL	5421.43	NULL	NULL
2008-03-02	2008-01-04	4991.93	5421.43	-429.50
2008-03-05	2008-03-02	10269.01	4991.93	5277.09
2008-05-10	2008-03-05	4606.62	10269.01	-5662.40
2008-06-03	2008-05-10	13226.83	4606.62	8620.21

2.2. Да се намери промяната в колоната DayTotal, както в 2.1, но когато разликата между две съседни дати е по-малка от 7 дена, трябва да се отхвърли тази дата и да се използва вместо нея разликата до следващата дата.

Импортиране и експортиране на данни

Импортьт на данни е процес на получаване на информация от външни за SQL Server източници и нейното вмъкване в таблици в база от данни на SQL Server. *Експортьт* на данните е процес на извличане на информация от таблица на база от данни на SQL Server и нейното копиране в някой определен от потребителя формат (например ASCII текстов файл, Microsoft Access база от данни и други). За копиране на данни в и от база от данни на SQL Server може да се използва помощната програма *bcp*, която се изпълнява от командния ред. Конструкцията `BULK INSERT` от Transact-SQL е предназначена за импорт на информация от файл с данни в база от данни на SQL Server. С помощта на SSIS (*SQL Server Integration Services*) е възможно извличане, трансформиране и обединяване на данни от различни източници.

Използване на конструкцията `BULK INSERT` за импортиране на данни

Общият синтаксис на командата от Transact-SQL за импортиране на данни в база от данни на SQL Server е:

```
BULK INSERT [[database_name.][owner.]{table_name | view_name}
FROM 'data_file'
[WITH
(
[ BATCHSIZE [= batch_size]]
[[,] CHECK_CONSTRAINTS]
[[,] CODEPAGE [= 'ACP' | 'OEM' | 'RAW' | 'code_page']]
[[,] DATAFILETYPE [=
{'char' | 'native' | 'widechar' | 'widenative'}]]
[[,] FIELDTERMINATOR [= 'field_terminator']]
[[,] FIRSTROW [= first_row]]
[[,] FORMATFILE [= 'format_file_path']]
[[,] KEEPIDENTITY]
[[,] KEEPNULLS]
[[,] KILOBYTES_PER_BATCH [= kilobytes_per_batch]]
[[,] LASTROW [= last_row]]
[[,] MAXERRORS [= max_errors]]
[[,] ORDER ({column_name [ASC | DESC]} [,... n])]
[[,] ROWS_PER_BATCH [= rows_per_batch]]
[[,] ROWTERMINATOR [= 'row_terminator']]
[[,] TABLOCK]
)
]
```

За да се използва командата `BULK INSERT`, таблицата трябва да съществува предварително и данните се добавят към съществуващото съдържание на таблицата. Броят на колоните и техният тип трябва да съответства на броя на колоните и типа на данните във файла. Параметрите, използвани в командата, са:

- Първият параметър преди ключовата дума `FROM` определя името на базата от данни и нейния собственик, както и името на таблицата (или изгледа), в която се въвеждат данните.
- След ключовата дума `FROM` се определя файла-източник, съдържащ данните, които трябва да се заредят в таблицата на базата от данни. Това е текстов файл, който може да бъде локален или да се намира навсякъде по мрежата, достъпен

чрез UNC (*universal naming convention name* – във формата \\Server_name\Share_name\Path\File).

- Параметърът BATCHSIZE определя размера на пакета, който се копира на сървъра като една транзакция. Това се отразява на броя на редовете, които се вмъкват при всяко изпълнение на конструкцията BULK INSERT. Ако целият пакет бъде зареден успешно, данните се записват на диска. Ако вмъкването се преустанови по средата на операцията, всички редове от този пакет ще бъдат загубени, но редовете до края на предишния пакет все още ще бъдат налични. По подразбиране всички данни в един файл с данни са един пакет.
- Чрез параметъра CHECK_CONSTRAINTS се установява прилагане на всички ограничения върху таблицата по време на копирането. По подразбиране ограниченията не се прилагат.
- CODEPAGE определя кодовата таблица на данните във файла с данни. Този аргумент има значение само ако данните съдържат колони от тип *char*, *varchar* или *text* със символи, ASCII кодовете на които са по-големи от 127 или по-малки от 32. Стойност ACP се използва за данни с кодова таблица ANSI ISO 1252, стойност RAW – когато не трябва да се извършва конвертиране на данни, стойност OEM – за да се използва подразбиращата се кодова таблица на клиента, или се въвежда специфична стойност за кодовата таблица (например 850).
- DATAFILETYPE задава подразбиращ се формат на данни. Подразбиращата се стойност *char* е предназначена за копиране от файл, съдържащ символни данни. При стойност *widechar* се извършва копиране от файл с данни, съдържащ Unicode символи. При стойност *native* се копират данни, като се използват собствени (за бази от данни) типове данни. В този случай файлът с данни, който трябва да бъде зареден, се създава от SQL Server с помощта на *bcp*. При *widenative* операцията по копирането се осъществява както при *native*, с изключение на колоните от тип *char*, *varchar* и *text*, които се съхраняват като Unicode във файла с данни.
- FIELDTERMINATOR определя разделител на полетата, използван за *char* и *widechar* файлове с данни. Подразбиращата се стойност е \t (табулация).
- FIRSTROW определя номера на първия ред за копиране. Подразбиращата се стойност е 1.
- FORMATFILE определя името на файла с формата и пътя до него. Този файл се генерира с *bcp*. Използва се, в случай че файлът с данни съдържа повече или по-малко колони от таблицата или изгледа, колоните са в различен ред, разграничителите на колоните са различни. Чрез помощната програма *bcp* може да се зададе допълнителна информация в този файл.
- KEEPIDENTITY определя, че стойностите на колоната със свойството IDENTITY се намират във файла, който се импортира. Ако този параметър не е зададен, стойностите за колоната IDENTITY във файла с данни се игнорират и се генерират автоматично нови стойности на базата на началната стойност и стъпката на нарастване, дефинирани при създаването на таблицата. Ако файлът с данни не съдържа стойности за колоната IDENTITY в таблицата или изгледа, трябва да се използва файл с формат, за да се определи, че колоната ще бъде пропусната при вмъкването на данните (в този случай стойностите се генерират автоматично).
- При зададен аргумент KEEPNULLS празните колони получават стойност NULL при копирането вместо стойностите си по подразбиране.
- KILOBYTES_PER_BATCH определя количеството на данните в един пакет в KB.

- LASTROW определя последния ред за копиране. По подразбиране има стойност 0, което означава последния ред във файла с данни.
 - MAXERRORS определя максималния брой на грешките, които могат да възникнат преди копирането да бъде отхвърлено. Подразбиращата се стойност е 10. Всеки ред, който не може да бъде вмъкнат, поради нарушаване на уникално ограничение, ограничение първичен ключ или друго ограничение, се игнорира и се брои като една грешка.
 - При използване на ORDER файлът с данни вече е сортиран по определени колони. Производителността на операцията по копиране на данните се подобрява, ако сортирането на данните, които се зареждат, е съгласно клъстерирания индекс в таблицата. Ако файлът с данни е сортиран в различен ред или в таблицата не съществува клъстериран индекс, тази опция се игнорира. Имената на определените колони в {column_name [ASC | DESC]} [, ... n] трябва да са на съществуващи колони в таблицата-приемник. По подразбиране се приема, че файлът с данни не е сортиран.
 - ROWS_PER_BATCH определя броя на редовете с данни в един пакет. Използва се, когато параметърът BATCHSIZE не е зададен, в резултат на което целият файл бива изпратен към сървъра като една транзакция. Сървърът оптимизира зареждането на данните според ROWS_PER_BATCH. По подразбиране стойността на този аргумент е неопределена.
 - ROWTERMINATOR определя разделител на редовете, използван за char и widechar файлове с данни. Подразбиращата се стойност е \n (символ за нов ред).
 - Чрез използване на TABLOCK се оказва, че по време на копирането трябва да има заключване на ниво таблица. Това подобрява производителността, поради предотвратяване на съревнованието за заключване. Този специален тип заключване на BULK INSERT е съвместим само с други заключвания BULK INSERT, така че в таблицата могат да се заредят данни бързо от много клиенти, изпълняващи паралелно BULK INSERT.
- Само членове на фиксираните роли на сървъра *sysadmin* или *bulkadmin* могат да изпълняват BULK INSERT.

Пример 1 Съхранена процедура за четене на текстов файл:

```
CREATE PROCEDURE EnterTextFile_Sp
    @filename nvarchar(128)
AS
    SET NOCOUNT ON

    CREATE TABLE #tempf (line varchar(8000))

    EXEC ('BULK INSERT #tempf FROM ''' + @filename +
        ''' WITH ( CODEPAGE = ''RAW'' ) ')

    SELECT * FROM #tempf

    DROP TABLE #tempf
GO
-- Примерно извикване на процедурата:
EXEC EnterTextFile_Sp
    @filename = '\\Server\Sh_name\Dir\file_name.txt'
```

/ Ако файлът е локален, т.е. намира се на сървъра и командата се изпълнява от сървъра, може да се използва низ от следния вид:
'c:\dir\subdir\file_name.txt'.*/*

Пример 2 Четене на част от текстов файл:

```
CREATE TABLE #MyTemp
( col1 varchar(30),
  col2 varchar(30),
  col3 varchar(30) )

BULK INSERT #MyTemp
FROM '\\Server\Sh_name\Dir\file_name2.txt'
WITH
(
  FIRSTROW = 2,
  LASTROW = 12,
  FIELDTERMINATOR = ';',
  ROWTERMINATOR = ';\n',
  MAXERRORS = 100,
  CHECK_CONSTRAINTS
)
GO
SELECT * FROM #MyTemp

DROP TABLE #MyTemp
GO
```

Използване на помощната програма *bcp* за импортиране и експортиране на данни

Помощната програма *bcp* (*bulk copy program* – програма за масово копиране) копира данните в и от файл с данни. Прилага се най-често за трансфер на големи количества информация в таблица на SQL Server база от данни от друга програма, в повечето случаи друга система за управление на бази от данни, или обратно. Когато се използва програмата *bcp*, данните първо се експортират от програмата-източник към файл с данни и след това се импортират от файла в таблицата на SQL Server база от данни (или обратно). Данни, които се копират във файл с данни от таблица в SQL Server база от данни, се записват върху предишното съдържание на файла или файлът се създава, ако все още не съществува.

Общият вид на помощната програма *bcp* за импортиране и експортиране на данни е:

```
bcp [[database_name.] [owner].] {table_name|view_name}|"query"
{in | out | queryout | format} data_file
[-m max_errors] [-f format_file] [-e err_file]
[-F first_row] [-L last_row] [-b batch_size]
[-n] [-c] [-w] [-N] [-q] [-C code_page]
[-t field_term] [-r row_term]
[-i input_file] [-o output_file] [-a packet_size]
[-S server_name] [-U login_id] [-P password]
[-T] [-v] [-R] [-k] [-E] [-h "hint [,... n]" ]
```

Параметрите, използвани в помощната програма, са:

- Първият параметър определя името на базата от данни и нейния собственик, както и името на таблица или изглед. Възможно е да се запише заявка, която извлича резултатен набор, като в този случай е задължително използването на двойни кавички и параметъра *queryout*.
 - *in* определя, че процесът е импортиране, *out* – експортиране.
 - *format* задава създаването на форматен файл. Необходимо е да се определи име на форматния файл с опцията *-f* и да се зададе формата за този файл чрез *-n*, *-c*, *-w* или *-N*.
 - *queryout* се използва, когато се експортира резултатът от SQL заявка.
 - *data_file* определя името на файла за импортиране или името на файла, който трябва да се създаде при експортиране.
- Опциите, използвани в помощната програма, различават малки и големи букви:
- *-m max_errors* определя максималния брой грешки, които могат да възникнат преди *bcp* да бъде спряна. Подразбиращата се стойност е 10.
 - *-f format_file* определя името и пътя за достъп до форматния файл на *bcp*. Подразбиращото име на файл е *bcp.fmt*.
 - *-e err_file* определя име и път за достъп до файл, в който се запазват съобщенията за грешки. Ако тази опция не е използвана, не се създава файл за грешки.
 - *-F first_row* определя номера на първия ред, който ще се импортира или експортира. Подразбиращата се стойност е 1, което съответства на първия ред във файла с данни.
 - *-L last_row* определя номера на последния ред, който ще се импортира или експортира. По подразбиране има стойност 0, което означава последния ред във файла с данни.
 - *-b batch_size* задава броя на редовете, които се прехвърлят с един пакет. Всеки пакет се копира на сървъра като една транзакция. По подразбиране всички редове се копират в един пакет. Не бива да се използва с опцията *-h*.
 - *-n* указва собствен режим за данни, който е специфичен за SQL Server. Съхраняването на данни в собствен формат е полезно, когато трябва да се копират данни от една инстанция на SQL Server в друга. Използването на собствен формат спестява време, тъй като не се извършва преобразуване на типове данни в и от символен формат, но файл с данни в естествен формат не може да бъде четен от други програми, а само от *bcp*.
 - *-c* установява използване на символни формати на данните за всички колони, като се поставят табулации между полетата и символ за нов ред в края на всеки ред. Съхраняването на информация в символен формат е полезно, когато данните се използват с друга програма, притежаваща функционалността да експортира и импортира данни в обикновен текстов формат.
 - *-w* определя Unicode режим. Използват се Unicode символи за всички колони и осигурява като подразбиращи се разделители табулации между полетата и символ за нов ред в края на всеки ред. Този формат позволява данните да бъдат копирани от един сървър (използващ кодова таблица, различна от тази на клиента, който изпълнява *bcp*) на друг сървър, който използва същата или друга кодова таблица.
 - *-N* задава използване на собствен тип данни за всички несимволни данни и формат на Unicode символи за всички символни данни (*char*, *varchar*, *nchar*, *nvarchar*, *text* и *ntext*).
 - *-q* определя използване на оградени в кавички идентификатори.

- `-C code_page` задава кодовата таблица на данните във файла с данни. Възможните стойности са аналогични на тези за параметъра CODEPAGE в командата BULK INSERT.
- `-t field_term` определя разделител за поле. Подразбиращата се стойност е \t (табулация).
- `-r row_term` определя разделител за ред. Подразбиращата се стойност е \n (символ за нов ред).
- `-i input_file` определя името на файл с отговорите на въпросите от командния промпт за всяко поле при копиране на голямо количество данни в интерактивен режим, т.е. когато не е зададена някоя от опциите `-n`, `-c`, `-w` или `-N`.
- `-o output_file` определя файл, който да получава резултата от работата на *bcp*, пренасочен от командния промпт.
- `-a packet_size` определя броя на байтовете в един мрежов пакет, изпращан към и от сървъра. Подразбиращата се стойност е 4096. Може да се зададе стойност между 4096 и 65535 байта.
- `-S server_name` определя име на SQL Server, с който се осъществява връзка. Тази опция се изисква, когато се изпълнява *bcp* от отдалечен компютър в мрежата.
- `-U login_id` определя логин за свързване със SQL Server.
- `-P password` определя парола за логин.
- `-T` задава използване на доверена конекция, използвана от мрежов потребител. Логин и парола не се изискват.
- `-v` показва използваната версия на *bcp*.
- `-R` определя данните тип валута, дата и час да бъдат копирани в SQL Server, като се използва регионалният формат, дефиниран за локалните настройки на клиентския компютър. По подразбиране регионалните настройки се игнорират.
- `-k` определя, че празните колони получават стойност NULL при копирането вместо стойностите си по подразбиране.
- `-E` е аналогична на параметъра KEEPIDENTITY в конструкцията BULK INSERT.
- `-h "hint [,... n]"` задава използване на подсказки при зареждане на данни в таблица: ORDER ({column_name [ASC | DESC]} [,... n]); ROWS_PER_BATCH = bb; KILOBYTES_PER_BATCH = cc; TABLOCK; CHECK_CONSTRAINTS.

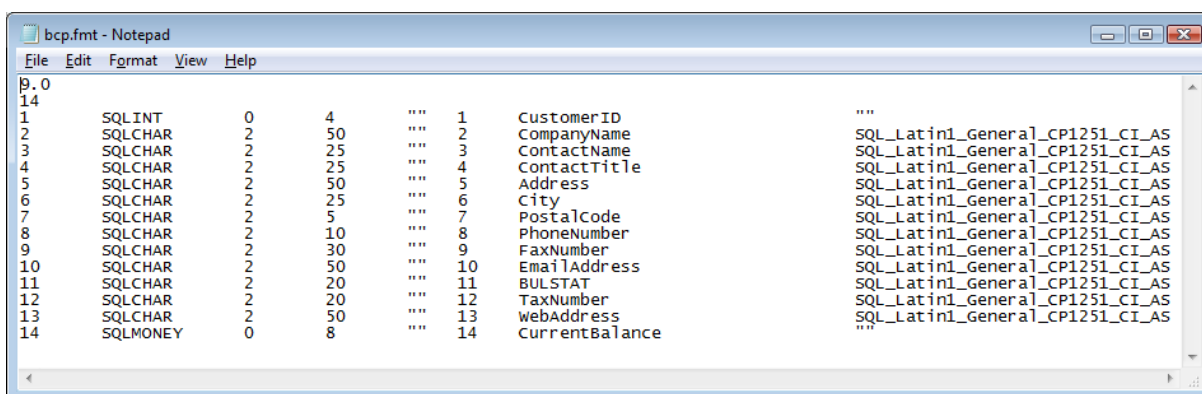
Въпреки че всеки потребител може да стартира *bcp*, само потребители с подходящи привилегии могат да осъществяват достъп до SQL Server и до определени обекти от базата от данни. Необходимата информация за свързване със SQL Server се задава чрез опциите `-U` и `-P`. За да импортира данни в една таблица, на потребителя трябва да е предоставена привилегия INSERT за таблицата-получател; за да експортира данни от една таблица, на потребителя трябва да му е предоставено разрешение SELECT за таблицата-източник.

Пример 3 Стартиране на интерактивна сесия от командния ред или от .bat файл:

```
bcp MyDatabase..Customers
out c:\dir\subdir\Customers.dat
-SMyComputer\SQL_Server_2005
-Ustudent -Pstudent
```

По време на интерактивната сесия за всяка колона от експортираната таблица се появяват промптове за типа за съхранение на данните във файла, дължината на префикса на полето, дължината на полето, разделител за полето. Тези промптове очакват въвеждане на съответната информация, като стойността по подразбиране е посочена в квадратни скоби и се приема, ако се натисне Enter, без да се въведе нищо за съответното запитване. Последният въпрос от интерактивната сесия е дали дадените отговори да бъдат съхранени във форматен файл и се дава възможност за въвеждане на името му (например c:\dir\Customers.fmt). Този форматен файл може да бъде използван и при други изпълнения на *bcp*, като се зададе стойност на опцията *-f*. Форматният файл (фиг. 1) има строг синтаксис, който трябва да се спазва – всеки ред от този файл съдържа полета с информация, които определят как ще се импортират данните. Написан е като ASCII текст и може да бъде разглеждан от произволен текстов редактор. Първият ред определя използваната версия на *bcp*, вторият – броя на колоните в таблицата, която се импортира или експортира, следващите редове определя форматирането за всяка колона от таблицата:

- номера на колоната във файла с данни;
- типа за съхранение на данните във файла;
- дължината на префикса при компресирани данни;
- дължината на полето в брой байтове, необходими за запомняне на този тип данни;
- разделител на полето (като например \t табулация, \r\n символ за връщане в началото на реда и символ за нов ред за последното име, др.);
- номера на колоната в таблицата от базата от данни;
- името на колоната в таблицата.

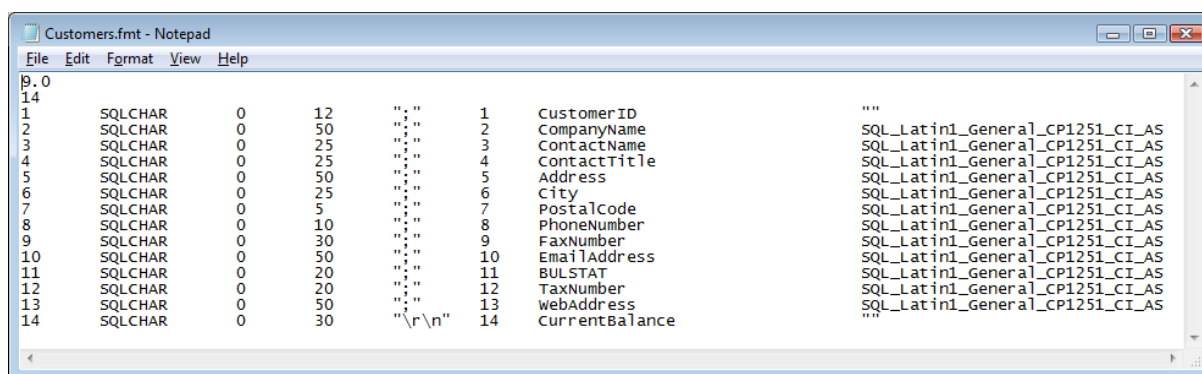


Фиг. 1 Примерен форматен файл

Пример 4 Създаване на форматен файл със зададено име и път за достъп, както и формат чрез опцията *-c*:

```
bcp MyDatabase..Customers
format C:\dir\subdir\Customers.dat
-c -t; -f c:\dir\subdir\Customers.fmt
-SMyComputer\SQL_Server_2005 -Ustudent -Pstudent
```

Съдържанието на създадения по този начин форматен файл е показано на фигура 2.



Фиг. 2 Форматен файл със зададено име, път за достъп и формат чрез опцията `-c`

Пример 5 Експортиране на таблицата за клиентите в зададен файл чрез използване на вече създаден форматен файл:

```
bcp MyDatabase..Customers
out C:\dir\subdir\Customers.dat
-f c:\dir\subdir\Customers.fmt -C1251
-SMyComputer\SQL_Server_2005
-Ustudent -Pstudent
```

В резултат от изпълнението на програмата *bcp* при зададени по този начин параметри се създава или припокрива файл с данни *Customers.dat*, който може да бъде преглеждан с текстов редактор, тъй като е определен символен формат.

Пример 6 Импортиране на данни в таблицата *NewCustomers* от зададен файл чрез използване на вече създаден форматен файл:

```
SELECT * INTO NewCustomers
FROM Customers WHERE 0 = 1
/* Първо се създава празна таблица NewCustomers със структурата на
Customers */
```

```
bcp MyDatabase..NewCustomers
in C:\dir\subdir\Customers.dat
-f c:\dir\subdir\Customers.fmt -C1251
-SMyComputer\SQL_Server2005
-Ustudent -Pstudent
```

```
SELECT * FROM NewCustomers
```

Тъй като *bcp* е помощна програма, която се стартира от командния ред, за разлика от конструкцията `BULK INSERT` (която обаче само импортира данни) тя не може директно да бъде изпълнявана от Transact-SQL код. За целта може да се използва разширената съхранена процедура:

```
xp_cmdshell {'command_string'} [, no_output]
```

Тя изпълнява даден команден низ подобно на среда за изпълняване на команди на операционната система и връща всякакъв резултат като редове от текст. Необходимо е на потребителя да се предостави привилегия `EXECUTE` за изпълняване на разширената съхранена процедура (изрично или чрез предоставяне на административни пълномощия). Опционалният параметър `no_output` предизвиква изпълнение на командния низ, без да бъде върнат някакъв резултат към клиента.

Например:

```
EXEC master..xp_cmdshell 'dir *.doc'
```

Редовете биват върнати като колони от тип *nvarchar(255)*.

```
EXEC master..xp_cmdshell 'cd c:\dir\subdir', no_output
```

Изпълнява се командния низ, но върнатият резултат е потвърждение, че командата е изпълнена успешно.

Пример 7 За успешно изпълнение на помощната програма *bcp* от T-SQL код е необходимо да се зададат опциите *-n*, *-c*, *-w* или *-N*, или да се посочи форматен файл чрез опцията *-f*, както и парола на логин (за нея също се появява запитване, ако не е посочена при изпълнение от командния ред):

```
EXEC master..xp_cmdshell
'bcp MyDatabase..NewCustomers
out c:\dir\subdir\Customers.dat -c
-SMyComputer\SQL_Server_2005
-Ustudent -Pstudent'
```

Пример 8 Съхранена процедура, която може да бъде създадена във всяка база от данни, за импортиране и експортиране на таблица в дадена база от данни от/към даден чрез име и път за достъп файл:

```
CREATE PROCEDURE Export_Or_Import_Table_sp
    @dbName    varchar(30),    @tbName    varchar(30),
    @filePath  varchar(80),    @cmode    char(6),
    @sep       char(1),        @serverName varchar(30),
    @usr       varchar(30),    @pwd      varchar(30)
AS
DECLARE @cmd varchar(200)

IF @cmode = 'EXPORT'
BEGIN
    SET @cmd = 'bcp.exe ' +
        @dbName + '..' + @tbName + ' out ' +
        @filePath + ' -c -q -C1251 -S' +
        @serverName + ' -U' + @usr + ' -P' +
        @pwd + ' -t' + @sep
    PRINT @cmd + '...'
    EXEC master..xp_cmdshell @cmd
END
IF @cmode = 'IMPORT'
BEGIN
    SET @cmd = 'bcp.exe ' +
        @dbName + '..' + @tbName + ' in ' +
        @filePath + ' -c -q -C1251 -S' +
        @serverName + ' -U' + @usr + ' -P' +
        @pwd + ' -t' + @sep
    PRINT @cmd + '...'
    EXEC master..xp_cmdshell @cmd
END
GO
```

За изпълнение на процедурата може да се използва следния примерен код:

- за експортиране на таблица от дадена база от данни във файл:
EXEC Export_Or_Import_Table_sp 'MyDatabase',


```
'Sales',
'C:\dir\subdir\my.dat',
'EXPORT',
'@',
'MyComputer\SQL_Server_2005',
'student',
'student'
```

- за импортиране в таблица на дадена база от данни от файл:

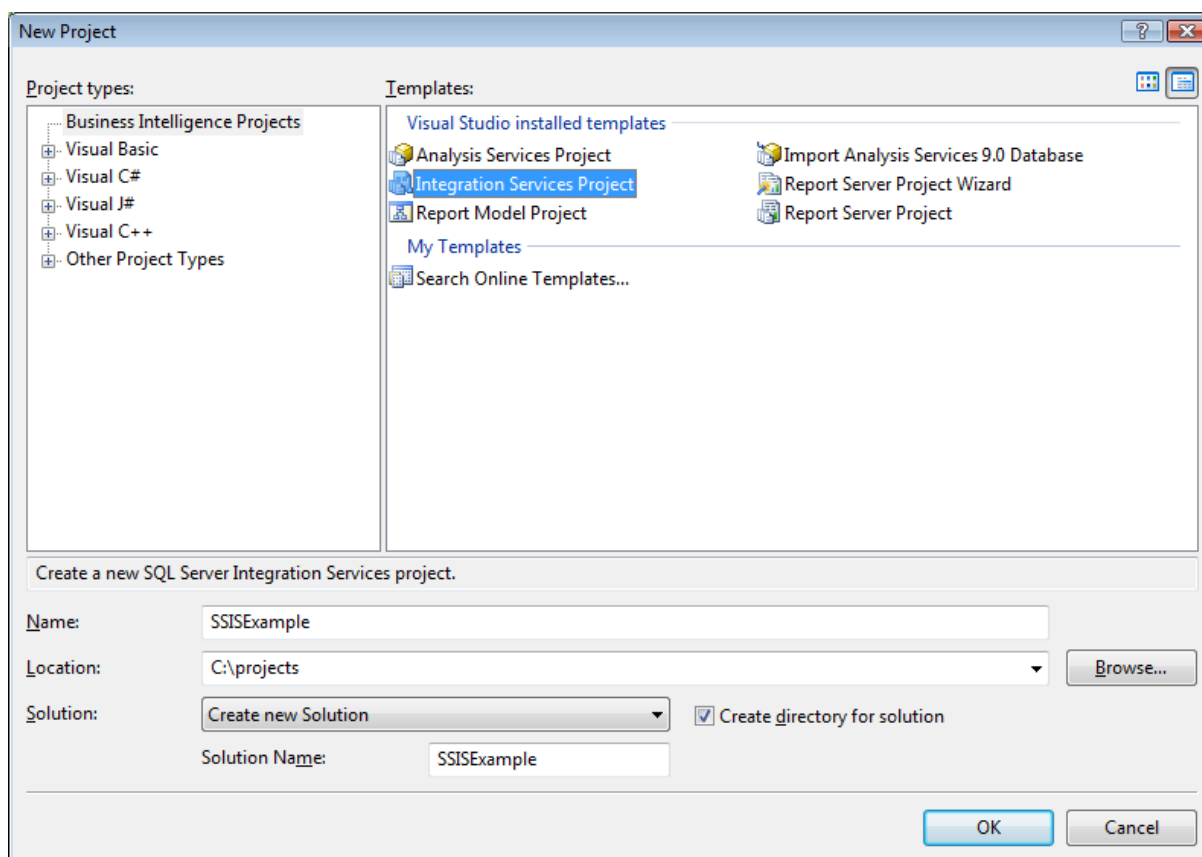
```
SELECT * INTO NewSales
FROM MyDatabase..Sales WHERE 0 = 1
GO
EXEC Export_Or_Import_Table_sp 'MyDatabase',
    'NewSales',
    'C:\dir\subdir\my.dat',
    'IMPORT',
    '@',
    'MyComputer\SQL_Server_2005',
    'student',
    'student'
SELECT * FROM MyDatabase..NewSales
```

Използване на SSIS (*SQL Server Integration Services*) за импортиране и експортиране на данни

SSIS (*SQL Server Integration Services*) е набор от графични инструменти, които дават възможност за извличане, трансформация, обединяване на данни от различни източници в един или множество приемници. SSIS може да бъде използван за преместване на данни към или от който и да е OLE DB източник на данни и да се прилагат всички трансформации на данните. Като се използват SSIS инструментите, може да се създадат SSIS пакети, в които да се дефинират и съхранят последователност от задачи за импортиране, преобразуване и експортиране на данни.

SQL Server Import and Export Wizard може да бъде стартиран от:

- SQL Server Management Studio чрез кликане с десен бутон на мишката върху името на базата от данни и избиране на *Task | Import Data* или *Task | Export Data* от контекстното меню;
- SQL Server Business Intelligence Development Studio чрез избиране на *Project | SSIS Import and Export Wizard* от менюто. Тази команда от менюто е достъпна при отворен проект, предназначен за разработка на пакети – *Integration Services Project*. Създаването на такъв проект се осъществява чрез *File | New | Business Intelligence Projects | Integration Services Project*, както е показано на фигура 3. След създаване на *Integration Services Project* е достъпен *Package Designer* – инструмент за разработване и поддържане на пакети. Той предоставя пълен контрол върху всяка стъпка от процеса на преобразуване. В повечето случаи е удобно един пакет да бъде създаден със SQL Server Import and Export Wizard и после да се използва Package Designer, за да се модифицира пакета и да се създадат допълнителни задачи.

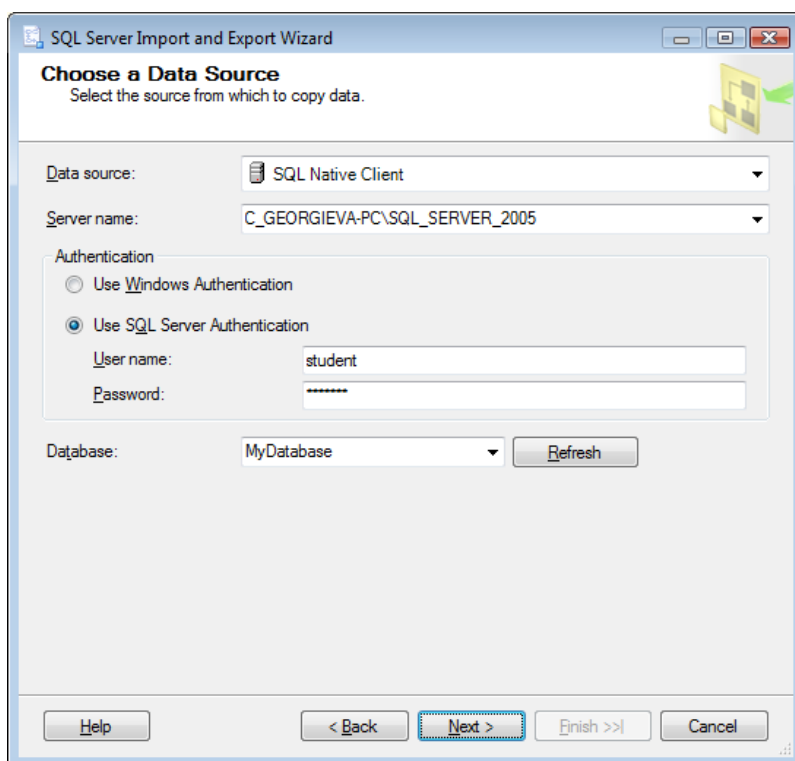


Фиг. 3 Създаване на нов *Integration Services Project*

- командния ред посредством помощната програма *DTSWizard* (DTSWizard.exe).

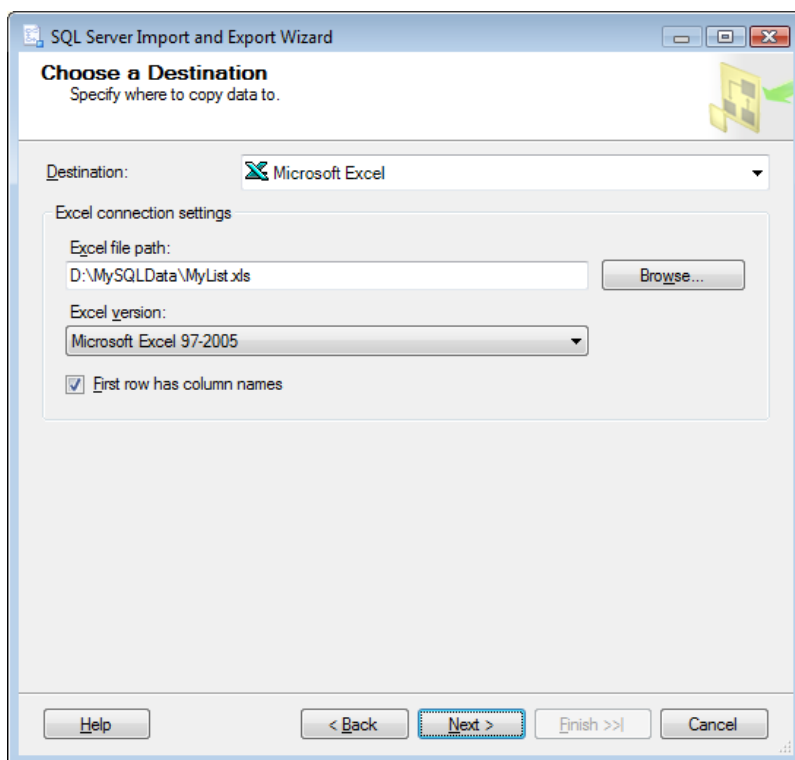
Пример 9 Използване на SQL Server Import and Export Wizard за експортиране на данните, които извлича изгледа *CustomerPhoneList_vw*, в работен лист на Microsoft Excel:

- Определяне на базата от данни-източник (фиг. 4);



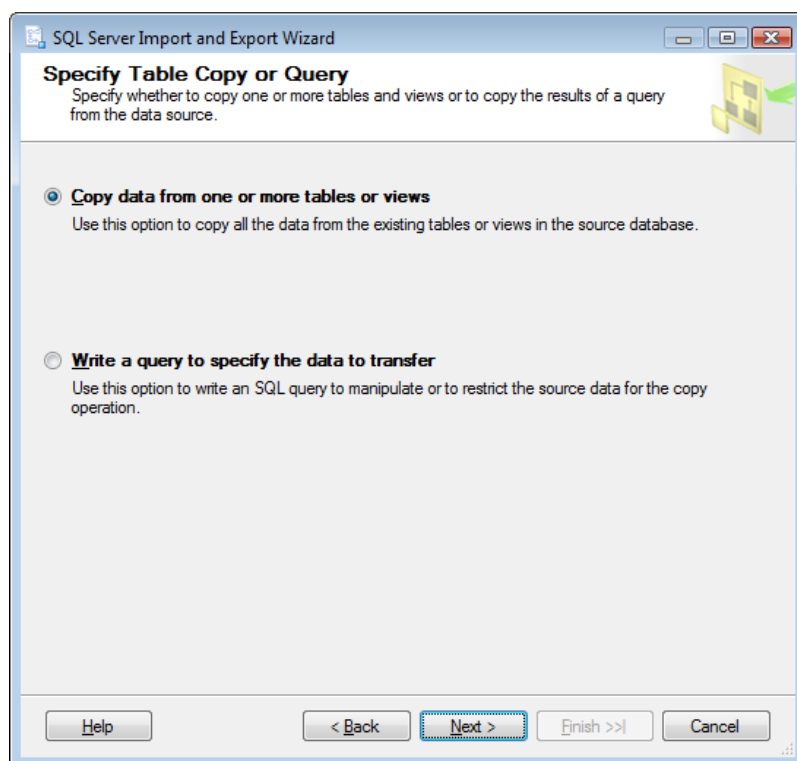
Фиг. 4 Определяне на източника

- Определяне на файла-приемник (фиг. 5);



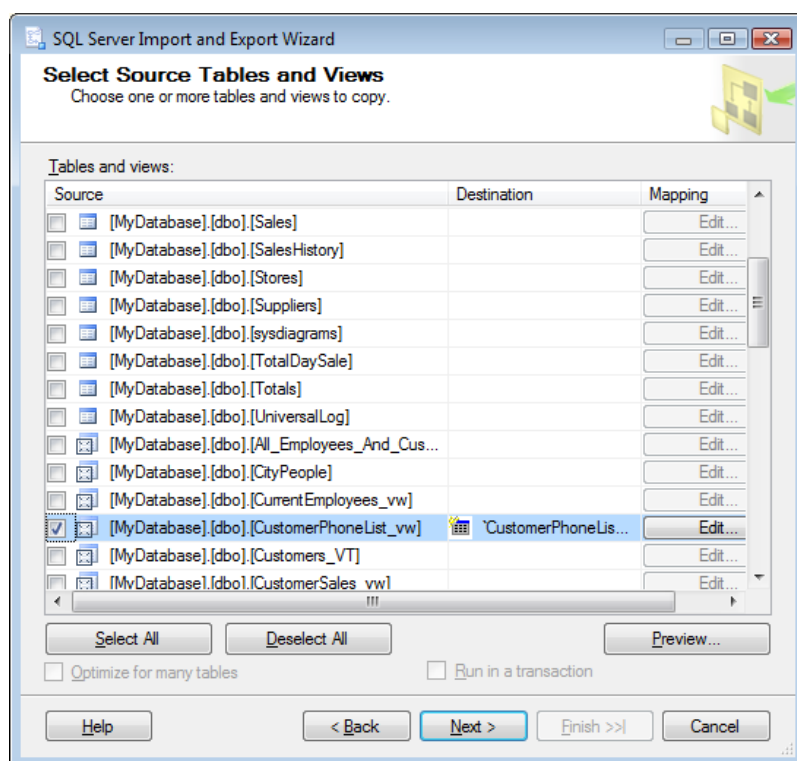
Фиг. 5 Определяне на приемника

- Определяне на обект за прехвърляне от базата от данни-източник. Тъй като трябва да се използва изглед, се избира първата опция (фиг. 6);



Фиг. 6 Определяне на обект за прехвърляне

- Избиране на обекта за прехвърляне, т.е. изгледа CustomerPhoneList_vw (фиг. 7);



Фиг. 7 Избиране на изгледа

Задачи

Задача 1. Да се експортират таблиците от базата от данни, описана в Приложение 1, в текстови файлове и да се импортират в нова празна база от данни.

Задача 2. Да се експортират таблиците от базата от данни, описана в Приложение 1, в база от данни на Microsoft Access, като се използва SQL Server Import and Export Wizard.

Установяване на връзка между клиентско приложение и Microsoft SQL Server

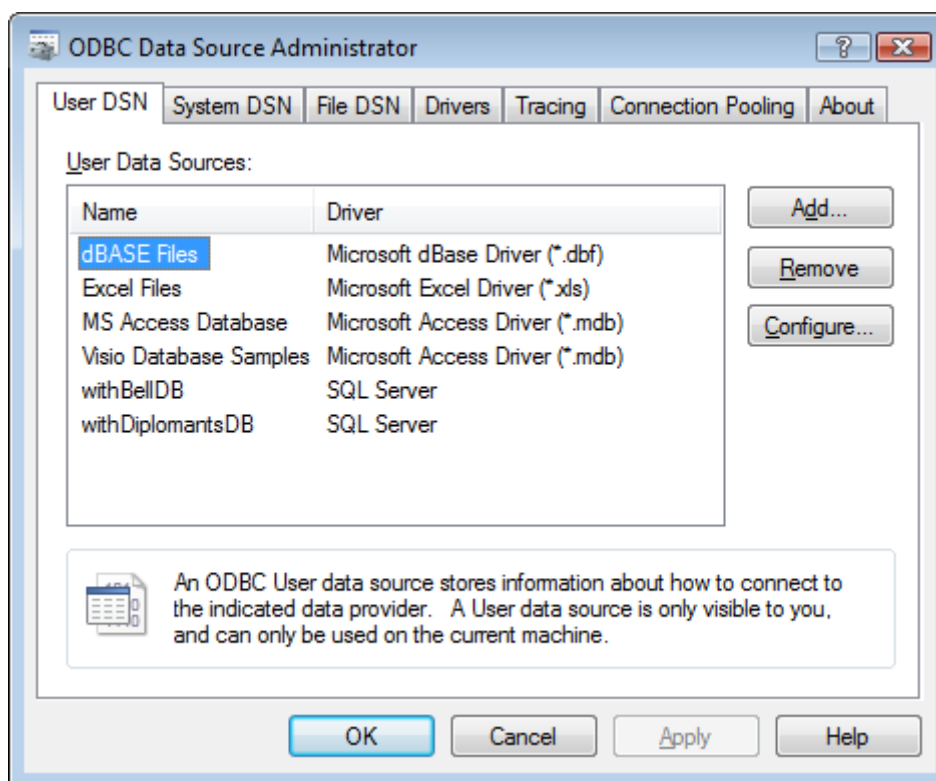
SQL Server предоставя няколко интерфейса, поддържащи разработването на клиентските приложения.

ODBC (*Open Database Connectivity – отворена система за свързване*)

ODBC е един от най-популярните интерфейси за бази от данни, използван от приложенията днес, получил признание като официален стандарт за интерфейс от ANSI и ISO. SQL Server осигурява високопроизводителен собствен ODBC интерфейс за всички среди за програмиране, базирани на Windows.

Определя се DSN (*Data Source Name – име на източника на данни*), който задава типа на базата от данни, нейното физическо местоположение, физическо име и други детайли на конфигурацията. Приложението след това отваря DSN по име и не се нуждае от специфичните за базата от данни детайли. Следователно ODBC е промеждутъчен софтуерен слой, който от страна на потребителя осигурява стандартизиран начин за комуникиране с базите от данни, а от страна на операционната система позволява на всяка фирма-производител да преобразува подаваната от потребителя информация във вида, изискван от нейната програма за работа с бази от данни.

Иконата на ODBC се намира в Control Panel на Windows, т.е. *Start / Settings / Control Panel / Administrative Tools* (фиг. 1).



Фиг. 1 Създаване на DSN

Има три вида имена на ODBC източници на данни:

- Потребителският DSN (*User DSN*) е обвързан с името на потребителя, който получава достъп. Този тип съхранява своята информация в регистратурата на системата за всеки потребител. Подходящ е за случаите, в които на един

компютър може да работят различни потребители, които обаче трябва да получат достъп до различни източници на данни. Потребител, който се е логнал, няма достъп до DSN, създадени от друг потребител.

- Системният DSN (*System DSN*) също съхранява информацията в регистратурата, но е достъпен независимо кой е регистриран в момента или дори никой да не е регистриран. Този тип източник на данни ще бъде достъпен винаги, когато компютърът работи. Системният DSN се прилага тогава, когато трябва всеки потребител на даден компютър да може да се свързва към базата от данни.
- Файловият DSN (*File DSN*) не се съхранява в системната регистратура, а в обикновен текстов файл. Подобно на системния DSN едно име на файлов DSN е достъпно независимо кой е регистриран. Може да бъде създаден с текстов редактор Notepad, като се зададат в съответния формат параметрите за вида на файла на базата от данни, с която ще се осъществява връзката. Чрез просто копиране на файла тази връзка може да се пренася от един компютър на друг. Папката, в която се създава файла, се задава с бутона *Set Directory*. Файлов DSN се използва в случаите, когато е необходимо името на източника на данни да може да се използва върху няколко компютъра. Файловете DSN могат да се намират навсякъде в мрежата.

И в трите случая процедурата по изграждане на DSN е аналогична. Например за създаване на системен DSN за връзка с база от данни на SQL Server са необходими следните стъпки:

1. Избира се драйвера, който се използва при свързването (например SQL Server).
2. Въвежда се името на свързването, чрез което ще се осъществява връзката с този източник на данни.
3. Въвежда се кратко описание в *Description*. То се вижда, когато се отваря връзката от Control Panel.
4. Избира се сървър, към който ще се осъществява връзката. Въвежда се името му или се избира от списък.
5. Избира се ниво на сигурност, което ще се използва: *With Windows NT authentication using the network login ID* – ако SQL сървърът е настроен да използва същите регистрационни идентификатори и пароли, които са използвани за регистриране в мрежа на Windows или *With SQL Server authentication using a login ID and password entered by the user* – ако SQL Server използва вградената си защита, т.е. използва се потребителско име и парола, които се получават от администратора на базата от данни.
6. Задават се някои по-особени параметри. В повечето случаи е най-добре да се приемат подразбиращите се стойности.
 - *Client Configuration...* – определя типа мрежова комуникация.
 - *Change the default database to* – промяна на подразбиращата се база от данни.
 - *Attach database filename* – определя главния файл с данни за присъединяване база от данни. Тази база от данни се присъединява и се използва като подразбираща се база от данни за източника на данни. Определя се пълният път и файлово име на главния файл с данни. Името на базата от данни, въведено в предишната опция, се използва като име за присъединената база от данни.
 - *Create temporary stored procedures for prepared SQL statements and drop the stored procedures* – за по стари версии на SQL Server за поддържане на SQL Prepare ODBC функция.
 - *Use ANSI quoted identifiers* – определя, че опцията QUOTED_IDENTIFIER се установява на ON, когато SQL Server ODBC драйверите се свържат, при

което двойните кавички се използват за заграждане на идентификатори (като имена на таблици и колони). За символните низове се използват апострофи.

- *Use ANSI nulls, paddings and warnings* – опциите ANSI_NULLS, ANSI_WARNINGS, ANSI_PADDING се установяват като ON, когато SQL Server ODBC драйверите се свържат. Когато ANSI_NULLS е ON, трябва да се използват изразите IS NULL и IS NOT NULL за сравняване на колони за стойност NULL. Не се поддържа T-SQL синтаксиса, който допуска израз = NULL. Когато ANSI_WARNINGS е ON, SQL Server използва предупредителни съобщения за условия, които нарушават ANSI правилата, но не нарушават правилата на T-SQL. Примери за такива грешки са прекъсване на изпълнението на INSERT или UPDATE израз и сблъскване със стойност NULL при изчисляване на агрегатни функции. Когато ANSI_PADDING е ON, интервалите в края на *varchar* и нулите в края на *varbinary* стойности не се отрязват автоматично.
- *Use the failover SQL Server if the primary SQL Server is not available* – съхранява се информация за свързване към резервен сървър. Ако приложението загуби връзката си с главния SQL Server, то изчиства текущите си транзакции и се опитва отново да се свърже с главния SQL Server. Ако драйверите определят, че главният сървър не е наличен, те автоматично се свързват с резервния сървър. Опцията не е активна, ако сървърът не поддържа операции с резервен сървър.
- *Change the language of SQL Server system messages to* – променя езика, на който се изписват системните съобщения.
- *Use strong encryption for data* – пренебрегват се подразбиращите се настройки за криптиране.
- *Perform translation for character data* – когато тази опция бъде избрана, SQL Server ODBC драйверите преобразуват ANSI символните низове, изпратени между клиентския компютър и SQL Server, като се използва Unicode. Това преобразуване изисква кодовата таблица, използвана от SQL Server да е една от кодовите таблици, налични на клиентския компютър. Когато опцията не е избрана, никакво преобразуване на разширени символи в ANSI символи не се прави, когато те се изпращат между клиентското приложение и сървъра. Ако клиентският компютър използва ANSI кодова таблица, различна от SQL Server кодовата таблица, може да не бъдат преведени правилно разширените символи в ANSI символи на низовете. В противен случай при една и съща кодова таблица превеждането е коректно.
- *Use regional settings when outputting currency, numbers, dates and times* – определя, че драйверите ще използват регионалните настройки на клиентския компютър за форматиране на валутата, числата, датите, времето в извеждащ символен низ. Драйверите използват регионалните настройки на регистрацията на потребител в Windows, свързан чрез източника на данни. Тази опция е добре да бъде избрана за приложения, които само извеждат данни, но не и за приложения, които ги обработват.
- *Save long running queries to the log file* – определя, че драйверът записва в *.log* файл всички заявки, които протичат по-дълго от *Long query time (milliseconds)* стойността.
- *Log ODBC driver statistic to the log file* – определя, че статистиката ще бъде записвана в *.log* файл. Файлът е *tab*-ограничителен файл, който може да бъде анализиран в MS Excel и други приложения, които поддържат *tab*-ограничителен файл.

Всяко свързване през ODBC става чрез един общ основен драйвер, който извиква един допълнителен и по-специален драйвер, предназначен за базите от данни на съответния производител. Тази двойка взаимодействащи си драйвери след това се нуждае от известно количество конфигурираща информация (тази информация може да се зададе, като се следват стъпките от разгледания вече ODBC Connection Wizard). Основният ODBC драйвер извлича от драйвера на производителя общите насоки затова как да се осъществи връзката. Тази информация може да бъде обединена в един непрекъснат низ, който се нарича *низ на свързване (connection string)*.

OLE DB (*Object Linking and Embedding* – свързване и вграждане на обекти)

OLE DB представлява обектен интерфейс за бази от данни. Чрез него може да се получи достъп до данни от източници на данни извън тези, до които ODBC може да получи достъп. Осигурява интерфейс към произволен източник на таблични данни, т.е. данни, които могат да бъдат представени в редове и колони (включително и данни в електронни таблици и дори текстови файлове).

ODBC осигурява универсален достъп до базите от данни. Ако е на разположение ODBC драйвер за съответната операционна система, ще е възможно да се осъществи връзка със съответната база от данни. Изграждането на връзка става с помощта на ресурсите на самата операционна система, следователно трябва да се настройва връзка отделно за всеки компютър (с изключение на файлов DSN).

С цел да се избегнат усложненията, свързани с настройването на връзката компютър по компютър, е създаден универсален достъп до бази от данни (*Universal Database Access – UDA*). В основата на UDA стои инсталируем драйвер, наречен *провайдер*. Като производител на операционни системи Microsoft осигурява провайдери за продуктите на основните производители на програми за управление на бази от данни, следователно на всеки компютър с инсталиран Windows може да се осъществи връзка със съответната база от данни. Използва се низ за свързване, който се изпраща към провайдъра, установяващ връзката. Низът за свързване съдържа цялата информация, необходима на провайдъра за инициране на връзката към базата от данни. За да може да се получи достъп до данните посредством OLE DB, трябва да се използва среда за програмиране, която да поддържа OLE DB.

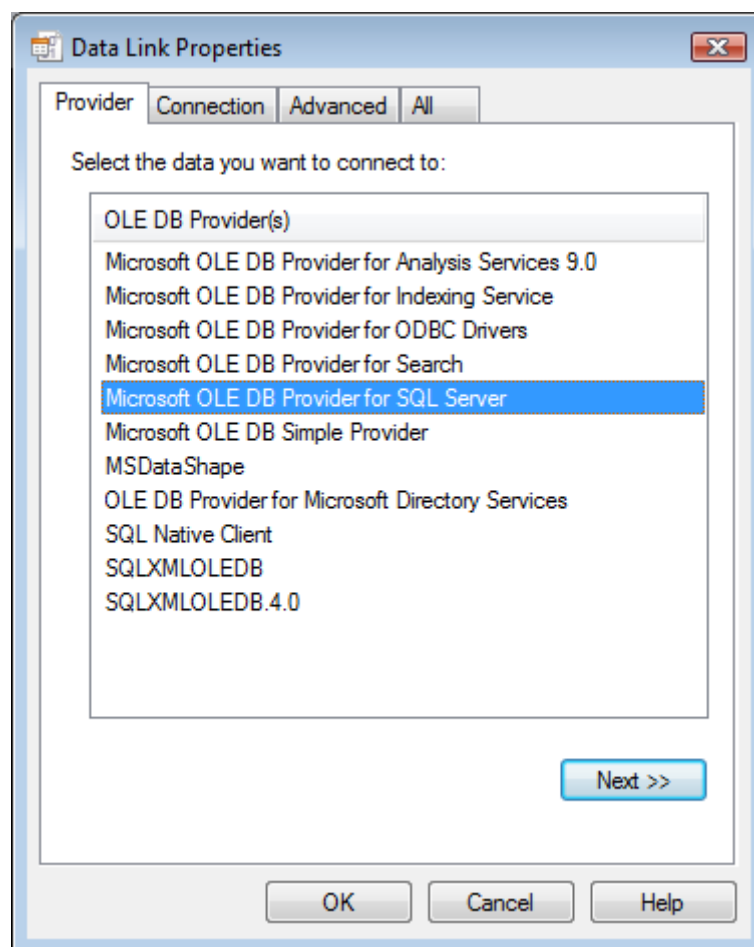
ADO (*ActiveX Data Objects*) представлява обектен интерфейс от по-високо ниво върху OLE DB, който осигурява голяма част от същата функционалност и производителност. ADO е обектен модел, който може да се използва за получаване на OLE DB данни. За да се осъществи връзка към дадена база от данни чрез ADO, трябва да се добавят към проекта референции към ADO. След това е необходимо да се създаде обект за връзка и да се зададат свойствата му, да се използват методите му за отваряне на връзката.

Използване на Microsoft Data Link за свързване към база от данни

Универсално свързване на данни (*Universal Data Links – UDL*) осигурява стандартизиран начин за представяне, зареждане и управление на OLE DB информацията за свързването. UDL компонентите осигуряват възможност за съхраняване на информацията за свързването в един *.udl* файл и след това позволяват да се отвори обектът Connection в ADO на базата на информацията, съхранена в *.udl* файла. Microsoft Universal Data Links се състои от графичен потребителски интерфейс за създаване на OLE DB свързване.

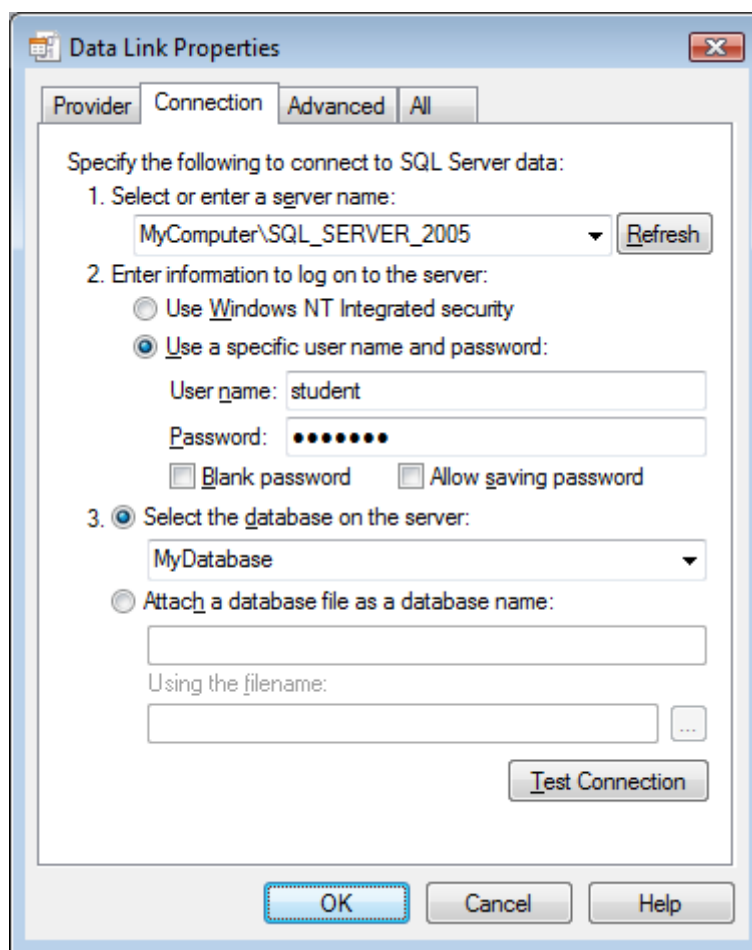
Създаването на връзка UDL се осъществява от Windows Explorer и *File / New / Microsoft Data Link*. Двойното щракване върху *.udl* файл позволява редактирането му:

- избор от списък с всички OLE DB доставчици, инсталирани на дадената система (фиг. 2);



Фиг. 2 Избор на OLE DB доставчик

- задаване на име на сървъра; потребителско име; парола; име на базата от данни (фиг. 3);



Фиг. 3 Задаване на име на сървър, потребителско име, парола, име на базата от данни

- настройка на специфични за доставчика свойства (*Advanced*) – мрежови настройки и други.

Задачи

Задача 1. Да се създаде ODBC връзка към база от данни на SQL Server.

Задача 2. Да се създаде *.udl* файл за връзка с база от данни на SQL Server.

Създаване на клиентски приложения в Microsoft Access за Microsoft SQL Server

Съществуват два вида приложения, които могат да бъдат създавани в Microsoft Access като клиентски за Microsoft SQL Server – MDB (ACCDB) и ADP (*Access Data Projects*).

Създаване на клиентски MDB (ACCDB) и ADP приложения за Microsoft SQL Server

Клиентските MDB (ACCDB) приложения на Access за SQL Server използват ODBC за комуникация с базата от данни на сървъра. Базите от данни в Access, които използват ODBC, зареждат двигателя на базата от данни Jet в качеството на посредник при всички комуникации между сървъра и съответния Access клиент.

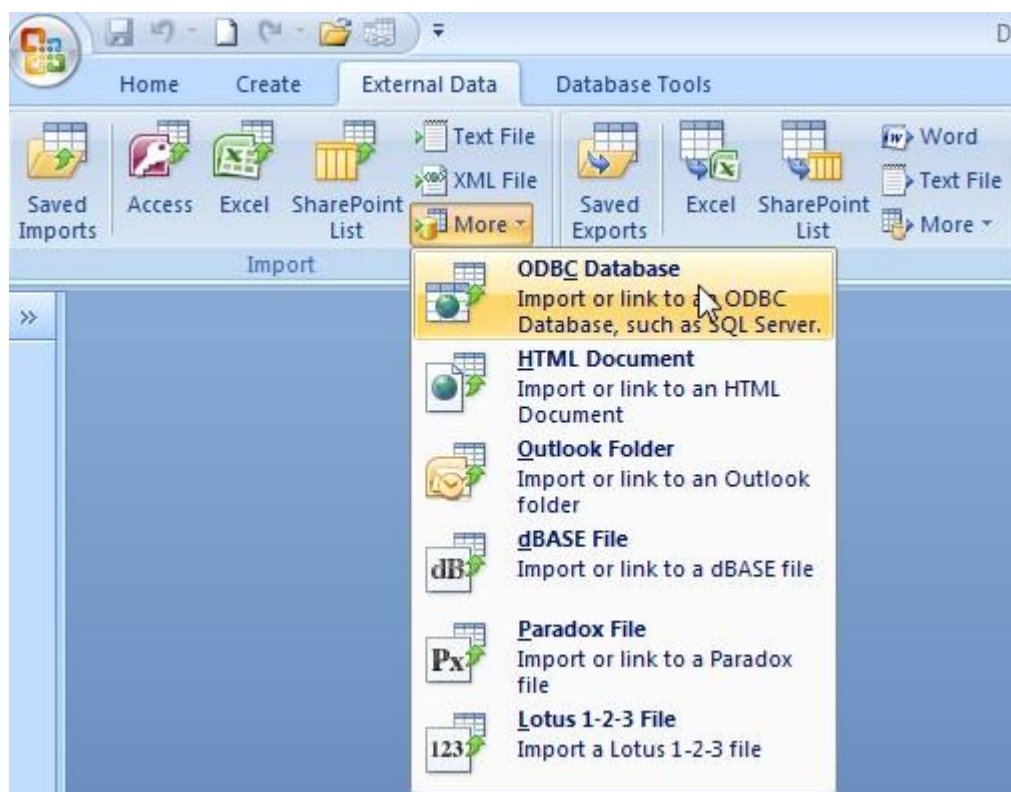
Клиентските ADP приложения към SQL Server използват технологията OLE DB за основна комуникация със сървъра без изобщо да зареждат Jet.

Установяване на връзка между клиентското MDB (ACCDB) приложение и SQL Server

За да се започне работа със сървърската SQL Server база от данни, трябва да се изгради комуникационна връзка с базата от данни. Инсталацията на Access зарежда ODBC драйвер за SQL Server на компютъра. След това трябва да се зададе източник на данни.

За свързване към таблиците на една ODBC база от данни, трябва да се следват следните стъпки:

1. Установява се ODBC източник на данни (DSN).
2. От менюто се избира *External Data | ODBC Database*, както е показано на фигура 1.



Фиг. 1 Свързване с ODBC база от данни в Access 2007

За предишни версии на Access (по-стари от 2007) от главното меню се избира *File / Get External Data / Link Tables*. От списъчното поле *File of Type* на диалоговата рамка *Link* се избира ODBC Databases, за да се изведе списък на всички ODBC източници на данни. Избира се DSN, с който ще се работи.

3. Позволява се промяна на въведената вече при създаването на DSN информация като потребителско име, парола, име на база от данни.
4. Извежда се списък с всички таблици, достъпни за свързване. Като се изберат таблиците, може да се зададе съхранение на паролата на базата от данни с таблиците, така че на потребителите няма да се извежда прозореца за свързване на ODBC, когато за пръв път зареждат приложението.
5. Access свързва таблиците към *.mdb* (*.accdb*) файла. Поставя префикс *dbo* (*Database Owner*) в началото на името на всяка таблица.
6. Ако SQL Server таблицата не притежава уникален ключ, Access няма да може да я свърже за четене и запис, затова се дава възможност за създаване на локален уникален индекс.

След като таблицата е свързана, може да бъде използвана както всяка друга свързана таблица в Access. Не се допуска промяна на структурата на таблицата от Access, а само модификация на данните в таблицата. Може да се създават заявки, форми, отчети, които да се обръщат към тези свързани таблици. Използват се два начина за достъп: свързани таблици и препращащи (*pass-through*) заявки.

Свързването на таблици от ODBC източник на данни към Access може да бъде много неефекасно. Ако двигателят Jet не може да използва уникален индекс на сървър, сървърът може да прехвърли на работната станция всички записи за обработка от Jet.

Клиентски ADP приложения

Microsoft интегрира SQL Server и Access посредством OLE DB. За тази цел е необходимо да се създаде *Access Data Project* (ADP). Едно ADP приложение представлява приложение на Microsoft Access с разширение *.adp*. То използва SQL Server като двигател за базата от данни. Преди да се използва SQL Server, трябва или да бъде инсталиран SQL Server на същата машина, или да е осигурен мрежов достъп до наличен SQL Server.

Някои предимства на ADP проектите са:

- ADP проектите позволяват да се използва Access като инструмент за разработка, тъй като е възможно да се извършва променяне на структурата на таблиците в средата на Access.
- С помощта на ADP проектите има една връзка за всички обекти от базата от данни.
- ADP проектите са лесни за използване.

Някои недостатъци на ADP проектите са:

- ADP проектите ограничават да се използва само SQL Server. Не могат да се използват например с Oracle, Sybase или DB2.
- Не може да се съхраняват никакви локални таблици или заявки.
- Не могат да се използват свързани таблици.

Създаване на ADP проекти

В Microsoft Access 2007 ADP проектите се създават, като се избере командата *New* от менюто на *Office Button*, след това се кликне върху бутона *Browse for a location to put your database* (фиг. 2) и в полето *Save as type* на диалоговата рамка *File New Database* се маркира *Microsoft Office Access 2007 Projects (*.adp)*.



Фиг. 2 Създаване на ADP в Microsoft Access 2007

При по-стари версии на Access се избира командата от главното меню *File / New* и от отворилата се диалогова рамка се избира *Project (Existing Database* – при свързване със съществуваща база от данни) или *Project (New Database* – за създаване на нова база от данни на сървър), т.е. дали новият ADP проект ще комуникира с вече съществуваща база от данни на SQL Server или с нова SQL Server база от данни.

Определя се местоположението на *.adp* файла; въвежда се информация за SQL Server базата от данни чрез прозореца *Data Link Properties* – име на сървър, име на потребител, парола, име на база от данни, която ще се използва.

Работата с проектите ADP и съществуваща SQL Server база от данни е аналогична на работата с обектите на базата от данни в Management Studio на SQL Server (таблицы, диаграми, изгледи, съхранени процедури, функции, тригери). За повторно свързване към SQL Server се използва командата *Office Button | Server | Connection* (или *File | Connection* за версии преди 2007), която отваря диалоговия прозорец *Data Link Properties* за определяне на информацията за свързващия OLE DB низ чрез Microsoft Universal Data Link.

При ADP проектите не само таблиците, но и заявките (т.е. изгледи, съхранени процедури, функции) директно се създават в SQL Server. Главното предимство е, че сървърът извършва много повече работа и приложението изисква само една връзка с него. Могат да се използват съхранени процедури за заявки с определен ред на сортиране за източник на списък в **Combo** и **List Box** на форми в Access.

Създаване на форми в Microsoft Access

Формите са основния интерфейс между потребителите и приложението на Microsoft Access. Използват се за изобразяване и редактиране на данни; контролиране на действието на приложението; въвеждане на нови данни; извеждане на съобщения. От гледна точка на всекидневната употреба формите са най-важните обекти, които са създадени в едно приложение на Access. Формите са това, което потребителите виждат и с което работят всеки път, когато стартират приложението. За да се създадат форми, които да са едновременно прости и удобни, е необходимо познаване на настройките, свойствата на формите, както и събитията, които протичат, когато потребителят работи с една форма на Access.

Възможно е автоматично създаване на форма на базата на даден източник на данни – таблица или заявка, като се маркира съответната таблица или заявка и от менюто се избере *Create | Form* (или *Insert | AutoForm* за версии преди 2007). За създаване на форма може да се използва *Form Wizard*, след което в проектен режим да се настроят съответните характеристики, така че формата да отговаря на специфични изисквания. Изграждането на нова форма с инструментите за проектиране се осъществява, като се избере *Create | Form Design* (или в прозореца *Database* се маркира *Forms* и се избира *New / Design View* за версии преди 2007).

Свойства на форми

Свойствата на формите се установяват в прозореца *Property Sheet*. Те се отнасят до данните, които стоят зад формите или до начина, по който формите изглеждат и по който се държат.

Свойства за данни (в страницата *Data* на прозореца *Property Sheet*) – описват начина, по който формата трябва да представят данните и да работи с тях:

- **Record Source** (източник на записи) е свойство, с което формите могат да се свързват към таблици, изгледи, съхранени процедури или да не се свързват с данни;
- **Allow Edits** (разрешаване на редакции). Когато е установено на *Yes* (по подразбиране), потребителят може да променя данните във формата; когато е *No* – редакциите са забранени. Това е удобен начин за бързо установяване на една форма само за четене, която обаче допуска въвеждане на нови редове и изтриване на съществуващи.
- **Allow Deletions** (разрешаване на изтривания). Когато е установено на *Yes* (по подразбиране), потребителят може да изтрива редове от формата. Това свойство позволява да се контролира кои данни могат да бъдат изтривани.
- **Allow Additions** (разрешаване на допълнения). Когато е установено на *No*, потребителят не може да добавя нови редове.
- **Data Entry** (въвеждане на данни). Установяването на това свойство на *Yes* забранява на потребителите да разглеждат съществуващите данни, но им позволява да добавят нови редове.
- **Allow Filters** (разрешаване на филтри), **Filter** (филтър), **Order By** (сортиране). Тези свойства позволяват ограничаване на набора от редове на готовата форма чрез задаване на условие в свойството **Filter** или ред за сортиране чрез свойството **Order By**. Свойството **Allow Filters** указва на Access дали да взема предвид тези стойности, когато се отваря формата. Вместо тези свойства може да се използва изглед, включващ условие чрез **WHERE** или съхранена процедура в **Record Source**, която да съдържа критерий и ред за сортиране.

Някои допълнителни характеристики на данни във форми на .adp проект

Източник на данни за форми в един .adp проект може да е таблица, изглед или съхранена процедура.

- При източник на данни *изглед*, за да е възможно актуализирането на данните от изглед, извличащ колони от повече от една таблица, трябва да се настрои характеристиката **Unique Table** на формата, като се установява на името на таблицата, която формата трябва да актуализира (само една таблица може да се актуализира). Ако не се зададе за форма, базирана за изглед или съхранена процедура, или SQL израз, съдържащ съединение (*join*), формата е само за четене.
- Свойството **Recordset Type** определя дали има група от записи, които могат да се актуализират в тази форма. Възможните стойности са *Snapshot* – някои таблици и контроли, свързани с техните колони не могат да се редактират; *Updatable Snapshot* – таблицата и контролите, свързани с нейните колони могат да се редактират (таблицата, определена с **Unique Table** при източник на данни изглед). Подразбиращата се стойност на свойството е *Updatable Snapshot*.
- Свойството **Max Records** определя максималния брой записи, върнати от източника на записи.

- Свойството **Resync Command** определя SQL израз или съхранена процедура, които ще се използват в *updatable snapshot* на тази таблица. Представлява SQL израз или съхранена процедура, които са параметризирани по ключовите колони от таблицата, определена в **Unique Table**, използвайки ? като параметризиращ знак. Параметрите трябва да съответстват по брой и наредба на множеството от ключови колони за таблицата, определена в свойството **Unique Table**. Целта на свойството **Resync Command** е да извлече фиксирани стойности на ред в набора от данни след актуализиране, включвайки актуализиране на свързаната колона. За да получи информация SQL Server какви ключови колони трябва да бъдат повторно синхронизирани, трябва да се добави условие **WHERE**, включващо ключовите колони на **Unique Table**. Access прочита свързаната информация от съответната таблица, която е “единичната” страна, когато се въведе нова стойност в колоните с външния ключ в таблицата от “множествената” страна.

- Свойството **Input Parameters** се използва за определяне на стойности на параметри, когато източникът на записи е съхранена процедура или SQL израз. Задаването на параметри на източника на записи, когато той е SQL израз, се осъществява по следния начин:

```
SELECT column_list FROM table_name
WHERE column_name = ?
```

Допустимо е използване и по-сложен израз, например:

```
column_name LIKE '%' + ? + '%'
```

Може да се зададат параметри за повече от една колони. Задаване на стойностите на параметрите се осъществява в свойството **Input Parameters** по следния начин:

```
[column_name [column_type] =] value_expression [,...]
```

Общият вид на **Input Parameters** при задаване на стойности на параметри на съхранена процедура е:

```
[@param_name [param_type] =] value_expression [,...]
```

За стойност на параметър може да се използва контрол на форма, като за целта се прилага следния синтаксис:

```
Forms![Form Name]![Control Name]
```

или по-кратко, ако се използва в субформа на главната форма, а не в същата или друга форма:

```
[Control Name]
```

За извличане на стойност на контрол в подформа на дадена форма се използва следния синтаксис:

```
Forms![Form Name]![Control SubformName].Form![Control in SubformName]
```




или по-кратко, ако се използва в главната форма, съдържаща субформата, а не в друга форма:

```
[Control SubformName].Form![Control in SubformName]
```

Форматиращи свойства (в страницата *Format* на прозореца *Property Sheet*) – описват начина, по който формата изглежда, т.е. използва се, за да се контролира външния вид на формата:

- Default View** (изглед по подразбиране) указва на Access как да отвори формата. Възможностите са: единична форма (*Single Form*) извежда един запис във всеки момент; продължителна (*Continuous Form*) извежда множество редове, всеки със свое копие на формата в секцията за детайли на формата; табличен

изглед (*Datasheet*) извежда данните в редове и колони; *PivotTable* и *PivotChart* позволяват анализиране на данните в съответно табличен и графичен вид.

- **Scroll Bars** (ленти за превъртане) разрешава или забранява вертикалните и хоризонтални линии за превъртане на формата.
- **Record Selectors** (селектори на записи) включва или изключва селекторите на записи в лявата страна на формата .
- **Navigation Buttons** (навигационни бутони) забранява или позволява появяването на лентата за избиране в долния край на формата .
- **Border Style** (стил на очертаване на рамка):
 - *оразмеряема* (*Sizable*) позволява потребителите да преоразмеряват формата по време на работа с нея;
 - *диалогова* (*Dialog*) фиксира размера на формата по време на нейното изпълнение. Бутоните *Minimize* и *Maximize* не са достъпни. Тази настройка е подходяща за диалогови рамки.
 - *тънка* (*Thin*) фиксира размера по време на нейното изпълнение. Бутоните *Minimize* и *Maximize* са достъпни.
 - *без рамка* (*None*) няма очертания на рамката. Подходяща настройка за форма, която съдържа лого на приложението и др.
- **Control Box** (контролна кутия) е свойство, което определя дали да се изведат или не бутоните *Minimize*, *Maximize*, *Close* и контролното меню на прозореца на формата (в горния ъгъл). При стойност *Yes* всички изброени се виждат; при стойност *No* – никое от тях.
- **Min Max Buttons** (бутони *Min* и *Max*) определя дали потребителят може да минимизира или максимизира формата на екрана. Възможните стойности са *Both Enabled*, *Min Enabled*, *Max Enabled*, *None*.
- **Close Button** (бутон за затваряне) – когато се установи на *Yes*, потребителят може да използва контролното меню за затваряне на формата и бутона ; при стойност *No* – трябва да се предвиди друг начин за затваряне на формата.

Допълнителни свойства (в страницата *Other* на прозореца *Property Sheet*):

- **Modal** (модална форма) – когато е установено на *Yes*, отварянето на формата забранява на друга форма да приема фокуса (моментът, когато формата става активна и към нея се насочва целия вход/изход), докато модалната форма не се затвори. Полезно е, когато потребителят трябва да извърши точно определени действия по точно определен начин преди да продължи работата си.
- **Pop Up** – когато това свойство е установено на *Yes*, формата ще може да стои най-отгоре на всички други форми и ще може да се разположи навсякъде по работния екран на Access (например формите на лентите с инструменти *Toolbar*).

За да се създаде **потребителска диалогова рамка**, трябва да се установи свойството **Pop Up** на *Yes*; **Modal** на *Yes*; **Border Style** на *Dialog*; да се премахнат навигационните бутони, селектори на записи, разделителните линии между записите (т.е. **Dividing Lines** да има стойност *No*); да се поставят несвързани контроли и бутони.








Контроли във формите на Microsoft Access

Информацията в една форма се съдържа в *контроли*. Прозорецът с инструменти *ToolBox* се визуализира от *Form Design Tools* в проектен изглед на форма (*View /*

ToolBars / ToolBox или от бутона *ToolBox*  за версии преди 2007). Този прозорец съдържа бутони за всички контроли, които могат да се използват във формата (фиг. 3).



Фиг. 3 Прозорец с инструменти *ToolBox* на форма






- **Label**  (етикет) се добавя за визуализиране на описателен текст като заглавия, кратки инструкции. Етикетите не могат да представят стойности от колони на източника на записи или изрази, те винаги са несвързани и не се променят при обхождане на редовете.
- **Text Box**  (текстово поле) се използва за визуализиране на данните от източника на записи. Текстовите полета могат да не бъдат свързани – например за показване на резултат от изчисление. Данните от несвързаните текстови полета не се съхраняват. За да бъде свързано текстовото поле с колона от източника на данни, се задава свойството му *Control Source* в страницата *Data* на прозореца *Property Sheet*.
- **Option Group**  (опционна група) се включва за визуализиране на ограничено множество от алтернативни стойности. Позволява изборът на стойност да се осъществи лесно чрез щракване с мишката на съответната стойност. Само една опция в групата може да бъде избрана в даден момент. Опционните групи могат да съдържат множество от **check box**  (контролни полета); **option button**  (опционни бутони); **toggle button**  (бутон с две състояния – натиснат или не). Ако опционната група е свързана с колона, само опционната рамка се свързва с колоната, не контролите вътре в рамката. Установява се свойството *Control Source* за опционната рамка. Когато се избере опция в опционната група, Microsoft Access установява стойността на колоната, с която опционната група е свързана, на стойността на свойството *Option Value* на избраната опция, която може да приема само целочислени стойности.
- **Toggle Button**, **Option Button**, **Check Box** са предназначени за визуализиране на колони от тип *bit* (*Yes/No*) от източника на записи. Най-полезни са, когато се използват в опционна група с други контроли от същия тип.
- **Combo Box**  (комбинирана кутия) се използва за създаване и визуализиране на списъци от предварително дефинирани стойности, от които потребителят да избира, за да се опрости процеса на въвеждане на данни. Възможно е комбинираната кутия да не е свързана с колона, т.е. съхранява стойност, която да се използва от друг контрол, например да се ограничат стойностите в друга комбинирана (или списъчна) кутия или субформа; в потребителска диалогова рамка, за да се намери запис, базиран на стойността, която е избрана в несвързания **Combo Box**.

Свойства за данни на *Combo Box*





- **Control Source** – име на колона или израз, който се използва за източник на контрола. Ако това свойство не е зададено, контролът е несвързан (*unbound*).
- **Row Source Type** определя типа на източника за данните от списъка за контрола: *Table/View/StoredProc* – таблица, изглед, SQL израз или съхранена процедура; *Value List* – списък от стойности, разделени с точка и запетая; *Field List* – имената на колоните от таблицата, изгледа, SQL израза или съхранената процедура.

- **Row Source** определя източника на данни за списъка на контрола. Установяването му зависи от типа, зададен в предишното свойство – конкретен SQL израз или име на таблица, изглед или съхранена процедура (при *Table/View/StoredProc* и *Field List*); списък от стойности (при *Value List*). Например за колоната длъжност на служител *Title* може да се използва контрол **Combo Box** със зададено свойство **Row Source** по следния начин: “управител”; “счетоводител”;... Възможно е списъкът да се състои от две колони, например първата да съдържа числови стойности, втората – символни низове, т.е. по следния начин: 1; “управител”; 2; “счетоводител”;... Това позволява да се съхранява числото, а да се избира от списък с низове.
- **Bound Column** определя номера на колоната от списъка, съдържаща стойностите, които се използват за стойност на контрола, т.е. стойността, която контролът въвежда, ако е свързан с колона от източника на записи.
- **Limit To List** – при *Yes* за стойност на контрола се приема само някоя от списъка.
- **Auto Expand** – при *Yes* автоматично се запълва текстовата част на **Combo Box** със стойност от списъка, която съответства на символите, въведени от клавиатурата.


Форматиращи свойства на **Combo Box**

- **Column Count** – брой колони в списъка на **Combo Box**.
 - **Column Heads** определя дали да се виждат заглавията на колоните, когато списъкът се извлича от таблица, изглед, SQL израз или съхранена процедура. При *Value List* за тип на източника заглавия на колоните са първите стойности от **Row Source**.
 - **Column Widths** – широчините на колоните, разделени със запетаи.
 - **List Rows** е максималния брой редове, които се виждат в списъка на контрола; ако има и други елементи на списъка, те се избират с помощта на *Scroll Bar*.
 - **List Width** определя широчината на списъка. Ако общата широчина на всички колони (**Column Widths**) е по-голяма от тази, се появяват хоризонтални ленти *Scroll Bar*.
- **List Box**  (списъчна кутия) се различава от **Combo Box** само по това, че винаги се вижда част от списъка, а не само избраният елемент. Свойствата му са аналогични на **Combo Box**.
 - **Command Button**  (команден бутон) се използва, за да се стартира действие или множество от действия, например отваряне или затваряне на форма или отчет, стартиране на съхранена процедура и други. За да може командният бутон да изпълни някакво действие във формата, трябва да се създаде макрос и да се асоциира (присъедини) към събитието *On Click* на бутона (в страницата *Event* на прозореца *Property Sheet*). За някои действия може да се използва *Command Button Wizard* – при поставяне на бутона във формата, трябва да е натиснат бутонът *Use Control Wizards* .
 - **Image**  е предназначен за вмъкване на статична картина във формата. Картината не може да бъде редактирана във формата (става част от формата), но Access я съхранява във формат, който е много удобен за обема на приложението и скоростта му на работа.
 - **Unbound Object Frame**  се използва за добавяне на обект, създаден с друга програма, поддържаща OLE (*Object Linking and Embedding* – свързване и вграждане на обекти). Обектът става част от формата, но не и част от данните в


някоя таблица. Могат да се добавят картини, диаграми, звуци, кадри, др. Позволява редактиране на обекта от формата с двойно щракване, както и установяване на връзка с оригиналния обект чрез включване на опцията *Link* в диалоговата рамка при избрана опция *Create from File*.

- **Bound Object Frame**  се добавя, за да се осигури достъп до обект, създаден с друга програма, поддържаща OLE, от данните, с които е свързана формата. Access изобразява по-голямата част от картините и диаграмите директно във формата. За другите обекти Access изобразява иконата на програмата, в която е създаден обектът (например звуков файл). Позволява редактиране от формата с двойно щракване, както и установяване на връзка с оригиналния обект. От контекстното меню на контрола, когато формата е във *Form View*, се избира *Insert Object*, за да се добавят нови стойности (т.е. обекти) в колоната от тип OLE (за *.mdb* (*.accdb*) файл) и *image* (за *.adp* файл).
- **Page Break**  се използва за добавяне на разделител на страница между страниците на формата с множество страници. Преминването от една страница на друга се извършва с клавишите *Pg Up*, *Pg Down* или като се използва вертикалната лента за превъртане на формата (*Scroll Bar*).
- **Tab Control**  е предназначен за създаване на поредица от страници във формата. Всяка страница може да съдържа определен брой други контроли, за да изобразява информацията.
- **Subform/Subreport**  се използва за вмъкване на друга форма в текущата форма. Субформите са добър начин за преглед на данни при релационни връзки от тип “едно към много”.


Свойства за данни на субформа

- **Source Object** определя формата, таблицата, изгледа или съхранената процедура, която е източник на субформата във формата;
- **Link Master Fields** и **Link Child Fields** определят колоните от главната и от подчинената форма (субформата), които са свързани – едно или повече, разделени с точка и запетая. Ако тези свойства са установени, Microsoft Access автоматично актуализира свързаните редове в субформата, когато се променя реда в главната форма. Възможно е вмъкване на повече от една субформи в една и съща главна форма.
- **Line, Rectangle** се използват за изчертаване на линии или правоъгълници във форма, за да се подобри нейният външен вид.
- **ActiveX Controls**  този бутон отваря диалогов прозорец, показващ всички ActiveX контроли, инсталирани на системата. Не всички ActiveX контроли работят с Microsoft Access.

Хипервръзки позволяват да се отворят други форми, отчети в същата или друга база от данни, текстови документи, електронни таблици, дори Web страници чрез щракване върху текста на хипервръзката. Една хипервръзка е етикет (контрол **Label**) със свойството **Hyperlink Address** или **Hyperlink SubAddress**, установено на стойност, различна от NULL. **Hyperlink Address** определя пътя до обекта, документа, Web страницата или друг приемник на хипервръзката. **Hyperlink SubAddress** определя мястото в обекта, зададен с предишното свойство, например обект в Microsoft Access база от данни, именуван област в Microsoft Excel работен лист, място в HTML документ и други.

Може да се използва диалоговата рамка *Insert Hyperlink*, която се отваря с кликане върху бутона , за да се зададат тези свойства. За определяне на обект от друга база от данни, създадена с Microsoft Access, се избира *Existing File or Web Page*. Посочените свойства могат освен за етикет, да се зададат и за команден бутон и

контрол **Image**. Възможно е хипервръзката да е свързана с колона от таблица, съхраняваща различен адрес за всеки ред от таблицата – тогава трябва да се използва **Text Box** със свойство **Control Source**, установено на колона от тип *Hyperlink* (за *.mdb* (*.accdb*) файл) и *varchar(n)* (за *.adp* файл). Свойството *Is Hyperlink* се установява в първия случай автоматично на *Yes*, докато във втория е необходимо да се зададе изрично.


Списъкът с колоните на източника на записи за формата се появява чрез *Add Existing Fields* . С влачене на мишката могат да се добавят контроли, свързани със съответната колона.

Работа с макроси

Макросът е множество от едно или повече действия, всяко от които извършва част от операция (например отделно действие е отварянето на форма, затваряне на форма, извеждане на съобщение и други). Макросите позволяват да се автоматизира някаква по-обща задача, която да се изпълни при определено събитие на даден контрол (отваряне на форма, затваряне на форма, натискане на бутон, двойно щракване на левия бутон на мишката върху контрол и други). Някои действия, които могат да бъдат включени в макроси, са:

- **OpenForm** отваря форма и има следните аргументи: *Form Name* – име; *View* – начин, по който да се отвори формата – Form, Design, Print Preview, Datasheet; *Filter Name* – филтър за определяне на сортиране и ограничаване на редовете за форма; *Where Condition* – израз, който избира редовете за формата от заявката или таблицата; *Data Mode* – начин на работа с данните във формата – Add, Edit, Read Only; *Window Mode* – дали прозорецът на формата да е в нормален (Normal), скрит (Hidden), минимизиран (Icon) или диалогов (Dialog) изглед.
- **OpenTable** отваря таблица;
- **OpenView** отваря изглед;
- **OpenReport** отваря отчет; аргументите на това действие са аналогични на тези за отваряне на форма.
- **Close** затваря определен прозорец или активния прозорец, ако няма определен.
- **MsgBox** извежда съобщение.
- **CancelEvent** отказ от MS Access събитието, което е стартирало макроса, съдържащо това действие. Например, ако събитието *Before Update* предизвика изпълнение на макрос за валидност и тя не е изпълнена, действието ще откаже актуализирането на данните.

Условия в макрос

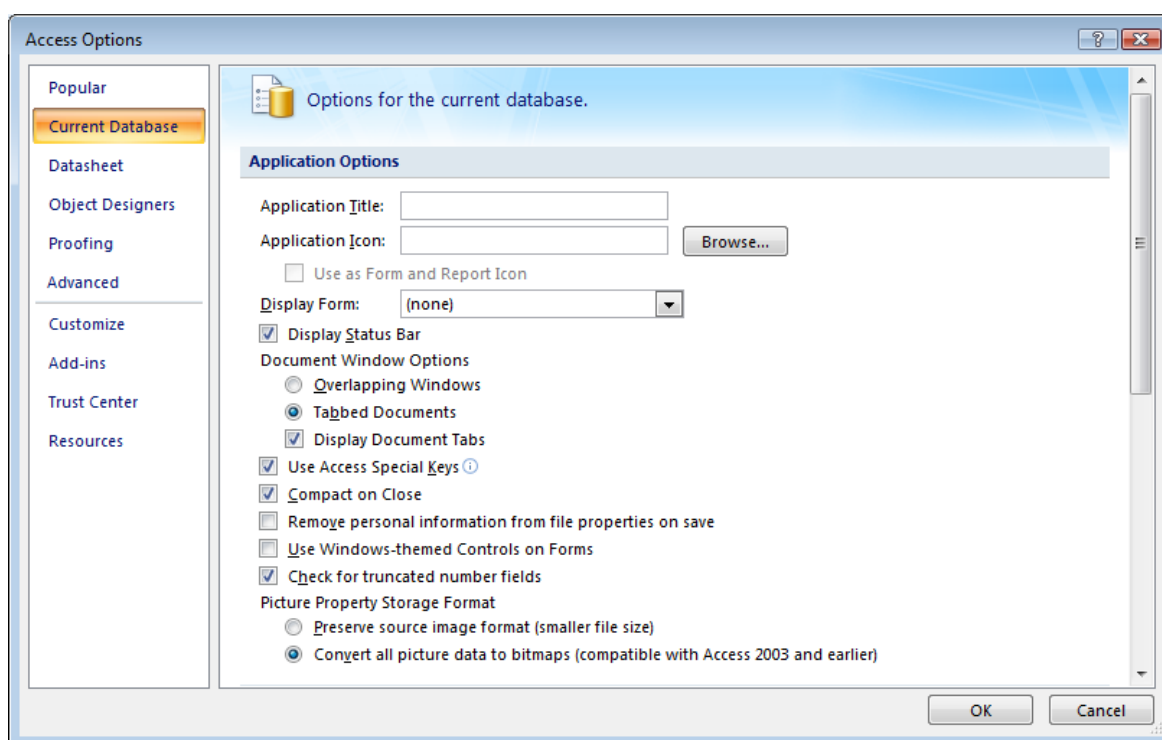
В някои случаи е необходимо действие или последователност от действия в макрос да се изпълнят само ако определено условие е изпълнено. Условието се въвежда в колоната *Condition* на прозореца на макрос в проектен изглед. За визуализирането на тази колона се използва командата от менюто *Conditions* . Ако условието е изпълнено, MS Access изпълнява действието в този ред. Ако е необходимо да се изпълнят последователно повече действия (при изпълнено условие) в колоната *Condition* се пише многоточие (...) на редовете, които непосредствено следват условието. Ако условието не е изпълнено, MS Access игнорира действията, които се предхождат от многоточие в колоната *Condition* и се прехвърлят към редовете със следващото условие или празно място в колоната *Condition*.

Асоцииране на макроси към събития

Дадено събитие се свързва с макрос, т.е. когато дадено събитие се случи, макросът се стартира, чрез страницата *Event* на прозореца *Property Sheet* на съответния контрол.

Макросът **AutoExec** е специален макрос, който предизвиква действията в него да се изпълняват първи при отваряне на базата от данни. Когато MS Access намери този макрос при отваряне на базата от данни, го стартира автоматично. Преди макроса *AutoExec* се изпълняват само опциите за стартиране *Startup*. За да не се стартира този макрос, както и да се пренебрегнат опциите за стартиране, трябва да се държи натиснат клавиш Shift, докато се отваря базата от данни.

Опциите за стартиране се задават от диалоговия прозорец *Access Options*, който се отваря с *Office Button* | *Access Options* | *Current Database* (фиг. 4).



Фиг. 4 Опциите за стартиране

- **Application Title** – заглавие, което се появява в заглавната лента на прозореца на MS Access;
- **Application Icon** – *bitmap* (.bmp) или *icon* (.ico) файл, както и пълния път до него, съдържащ изображение на икона на приложението в заглавната лента;
- **Display Form** определя форма, която да се появи при отваряне на базата от данни;
- **Display Navigation Pane** – при включена опция прозорецът *Navigation Pane* се появява при отваряне на базата от данни; при изключване на опцията – се предотвратява визуализирането на прозореца *Navigation Pane* при отваряне на базата от данни;
- **Display Status Bar** – тази опция се включва, за да се визуализира лента на състоянието (*Status Bar*);
- **Menu Bar** – избор на лента с меню от списъка за визуализиране на собствено меню като меню по подразбиране за текущата база от данни;

- **Shortcut Menu Bar** – избор на контекстно меню по подразбиране за текущата база от данни;
- **Allow Full Menus** – опцията се избира, за да позволи използване на всички MS Access команди от менюто;
- **Allow Default Shortcut Menus** – избира се опцията, за да се позволи използването на контекстното меню на MS Access по подразбиране;
- **Allow Built-in Toolbars** – избира се опцията, за да се позволи визуализирането и използването на лентата с инструменти на MS Access по подразбиране;
- **Use Access Special Keys** избира се опцията, за да се позволи използването на специалния клавиш F11 за отваряне на прозореца *Navigation Pane*. Ако опцията е изключена, както и опцията **Display Navigation Pane**, потребителят още може да отвори прозореца *Navigation Pane*, като задържи Shift при отваряне на базата от данни.

Пример 1 Форма *ProductsInfo Form*, осигуряваща достъп до данните за продуктите за преглед и променяне.

Свойства на формата *ProductsInfo Form*

- Източник на записи, т.е. **Record Source** е изгледа, създаден със следната T-SQL команда:

```
CREATE VIEW ProductsInfo
AS
    SELECT p.ProductName, p.CategoryID,
           p.Price, p.Stock,
           p.SupplierID, s.Address, s.City
    FROM Suppliers s
    INNER JOIN Products p
        ON s.SupplierID = p.SupplierID
```

- Свойството **Unique Table** е установено на **Products**.
- На свойството **Resync Command** е зададен като стойност следният израз:

```
SELECT p.ProductName, p.CategoryID,
       p.Price, p.Stock,
       p.SupplierID, s.Address, s.City
FROM Suppliers s
INNER JOIN Products p
    ON s.SupplierID = p.SupplierID
WHERE p.ProductID = ?
```

Свойства на контролите във формата *ProductsInfo Form*

- За контрол, свързан с колоната **SupplierID** от таблицата **Products**, се използва **Combo Box** със следните свойства: на **Row Source** е зададен SQL изразът:

```
SELECT SupplierID, CompanyName FROM Suppliers
```

Column Count: 2; Column Widths: 0 cm; 3 cm
- За контрол, свързан с колоната **CategoryID** от таблицата **Products**, се използва **Combo Box** със следните свойства: на **Row Source** е зададен изразът:

```
SELECT CategoryID, CategoryName FROM Categories
```

Column Count: 2; Column Widths: 0 cm; 3 cm
- Бутон *Save Record* за запис във формата на промените в данните.

При избор на доставчик от списъка на **Combo Box** се актуализират и съответните адрес и град след запис на промените.

Пример 2 Форма *EmployeesInfo Form*, осигуряваща достъп до данните за служителите за преглед и променяне.

Свойства на формата *EmployeesInfo Form*

- Източник на записи, т.е. **Record Source** е изгледа, създаден със следната T-SQL команда:

```
CREATE VIEW ProductsInfo
AS
    SELECT e.FirstName, e.LastName,
           e.Title, e.StoreID,
           e.HireDate, e.TerminationDate,
           s.Address, s.City
    FROM Employees e
    INNER JOIN Stores s
        ON e.StoreID = s.StoreID
```

- Свойството **Unique Table** е установено на *Employees*.
- На свойството **Resync Command** е зададен като стойност следният израз:

```
SELECT e.FirstName, e.LastName,
       e.Title, e.StoreID,
       e.HireDate, e.TerminationDate,
       s.Address, s.City
FROM Employees e
INNER JOIN Stores s
    ON e.StoreID = s.StoreID
WHERE e.EmployeeID = ?
```

Свойства на контролите във формата *EmployeesInfo Form*

- За контрол, свързан с колоната *StoreID* от таблицата *Employees*, се използва **Combo Box** със следните свойства: на **Row Source** е зададен изразът:

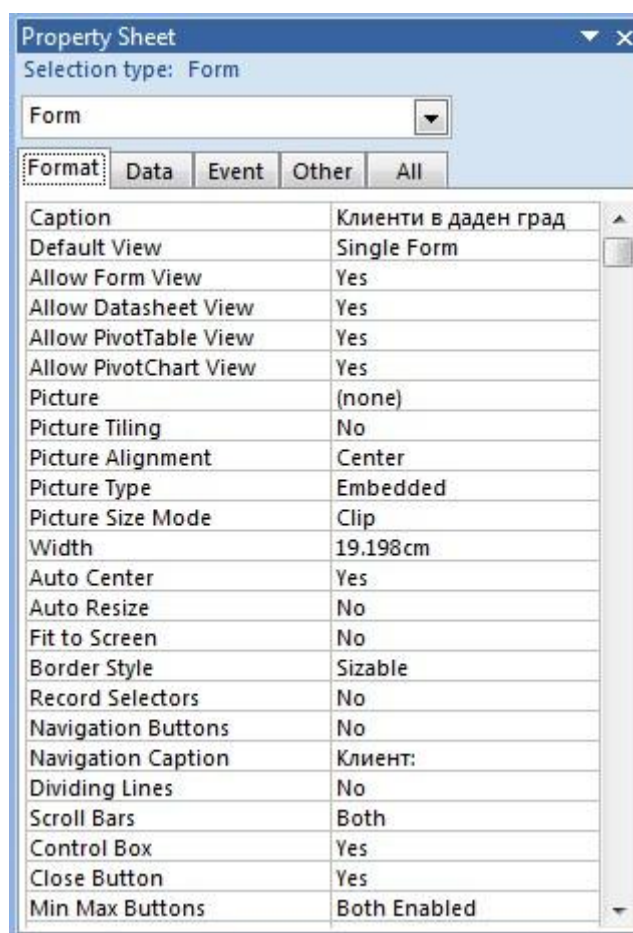
```
SELECT StoreID, StoreName FROM Stores
```

Column Count: 2; Column Widths: 0 cm; 3 cm

- Бутон *Save Record* за запис във формата на промените в данните.

При избор на магазин от списъка на **Combo Box** се актуализират и съответните адрес и град след запис на промените.

Пример 3 Форма *CustomersInfo Form*, която извежда данните на клиентите в даден град (фиг. 5).



Фиг. 6 Форматиращи свойства на *CustomersInfo Form*

Свойства на контролите във формата *CustomersInfo Form*

Формата съдържа два контрола – един Combo Box за въвеждане на град със следните свойства: Name: *City*; Row Source Type: *Table/View/SourceProc*; Row Source:

```
SELECT DISTINCT city FROM Customers
```

и един контрол Subform за извеждане на данните на клиентите със следните свойства Source Object: *CustomersInfo subform*; Link Child Fields: *City*; Link Master Fields: *City*.

Свойствата на формата *CustomersInfo subform* са:

- Record Source е таблицата *Customers*.
- Default View: *Datasheet*

Формата *CustomersInfo subform*, която е източник на записи за субформата, съдържа контроли Text Box, свързани с колоните от таблицата *Customers*.

Пример 4 Форма *Saled Quantity*, която извежда продадените количества по продукти за зададен период от време (фиг. 7).

Фиг. 7 Примерна форма *Saled Quantity*

Свойства на формата *Saled Quantity*

Формата не е свързана с източник на данни и има форматиращи свойства, показани на фигура 8.

Property Sheet	
Selection type: Form	
Form	
Format	Data
Caption	Продадени количества за определен период от време
Default View	Single Form
Allow Form View	Yes
Allow Datasheet View	Yes
Allow PivotTable View	Yes
Allow PivotChart View	Yes
Picture	(none)
Picture Tiling	No
Picture Alignment	Center
Picture Type	Embedded
Picture Size Mode	Clip
Width	16.598cm
Auto Center	Yes
Auto Resize	No
Fit to Screen	No
Border Style	Sizable
Record Selectors	No
Navigation Buttons	No
Navigation Caption	
Dividing Lines	No
Scroll Bars	Both
Control Box	Yes
Close Button	Yes
Min Max Buttons	Both Enabled

Фиг. 8 Форматиращи свойства на *Saled Quantity*

Свойства на контролите във формата *Saled Quantity*

Формата съдържа три контрола – два **Text Box** за въвеждане на начална и крайна дата (със свойство **Format** – *Long Date* и с имена **Name** съответно *BeginDate*, *EndDate*) и един контрол **Subform** за извеждане на данните със свойство **Source Object**: *Saled products quantity Subform*.

Някои свойства на формата *Saled products quantity Subform* са:

- **Record Source** е съхранената процедура *SaledProductsQuantity*, която е създадена със следната команда от T-SQL:

```
CREATE PROCEDURE SaledProductsQuantity
    @BeginDate char(10), @EndDate char(10)
AS
    DECLARE @d1 datetime, @d2 datetime

    SET DATEFORMAT dmy
    SET @d1 = CAST(@BeginDate AS datetime)
    SET @d2 = CAST(@EndDate AS datetime)

    SELECT ProductName,
           SUM(quantity) AS SaledQuantity
    FROM Products p
    INNER JOIN SaleDetails sd
        ON p.ProductID = sd.ProductID
    INNER JOIN Sales s
        ON s.SaleID = sd.SaleID
    WHERE SaleDate BETWEEN @d1 AND @d2
    GROUP BY ProductName
```

- **Input Parameters**: *BeginDate*, *EndDate*. Използват се имената (стойностите на **Name**) на контролите **Text Box** от главната форма.
- **Default View**: *Datasheet*

Формата *Saled products quantity Subform*, която е източник на записи за субформата, съдържа два контрола **Text Box**, свързани с колоните *ProductName* и *SaledQuantity*.

Задачи

Задача 1. Да се създадат форми, осигуряващи достъп до данните за категориите, клиентите, доставчиците, магазините и продажбите за преглед и променяне.

Задача 2. Да се създаде форма, която извежда данните на магазините в даден град.

Задача 3. Да се създаде форма, която извежда данните на служителите, назначени през даден период от време.

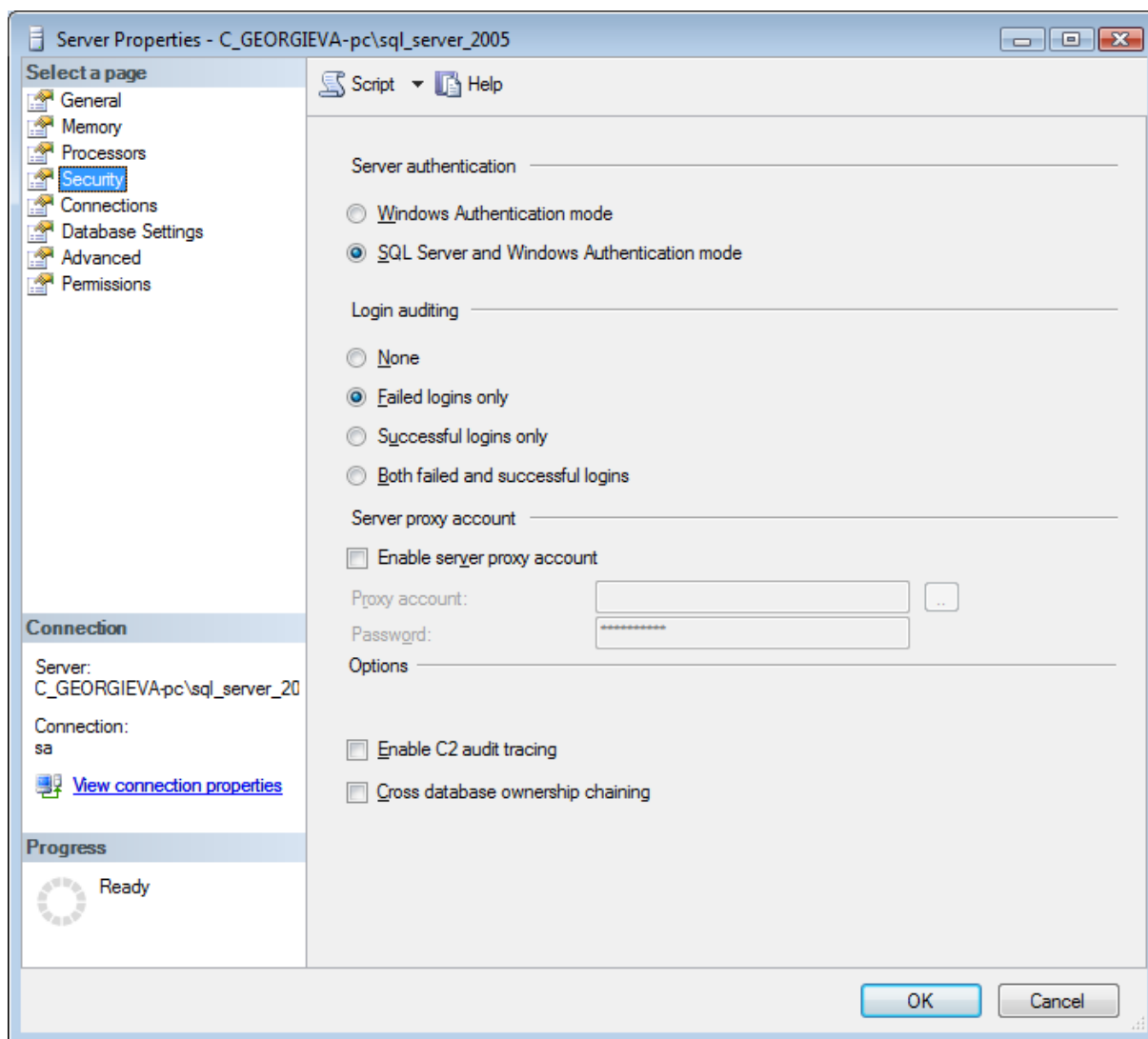
Системата за сигурност на Microsoft SQL Server

SQL Server предоставя възможност за управление на сигурността с помощта на инструмента Management Studio. Контролирането на достъпа до базите от данни се извършва на две стъпки:

- I. свързване със SQL Server (*автентикация*);
- II. осъществяване на достъп до някоя база от данни и нейните обекти (*оторизация*).

Режими за автентикация на Microsoft SQL Server

SQL Server поддържа два режима на автентикация: Windows Authentication и SQL Server Authentication. При Windows Authentication се използват потребителски и групови акаунти, които се намират в Windows домейн, за установяване на конекция със SQL Server; при SQL Server Authentication се използва идентификационен номер за логин на SQL Server за свързване със SQL Server. Конфигурирането на сървъра за даден метод за автентикация се осъществява по време на инсталацията на SQL Server или по-късно от диалоговия прозорец *Server Properties* (фиг. 1).



Фиг. 1 Задаване на режим на автентикация

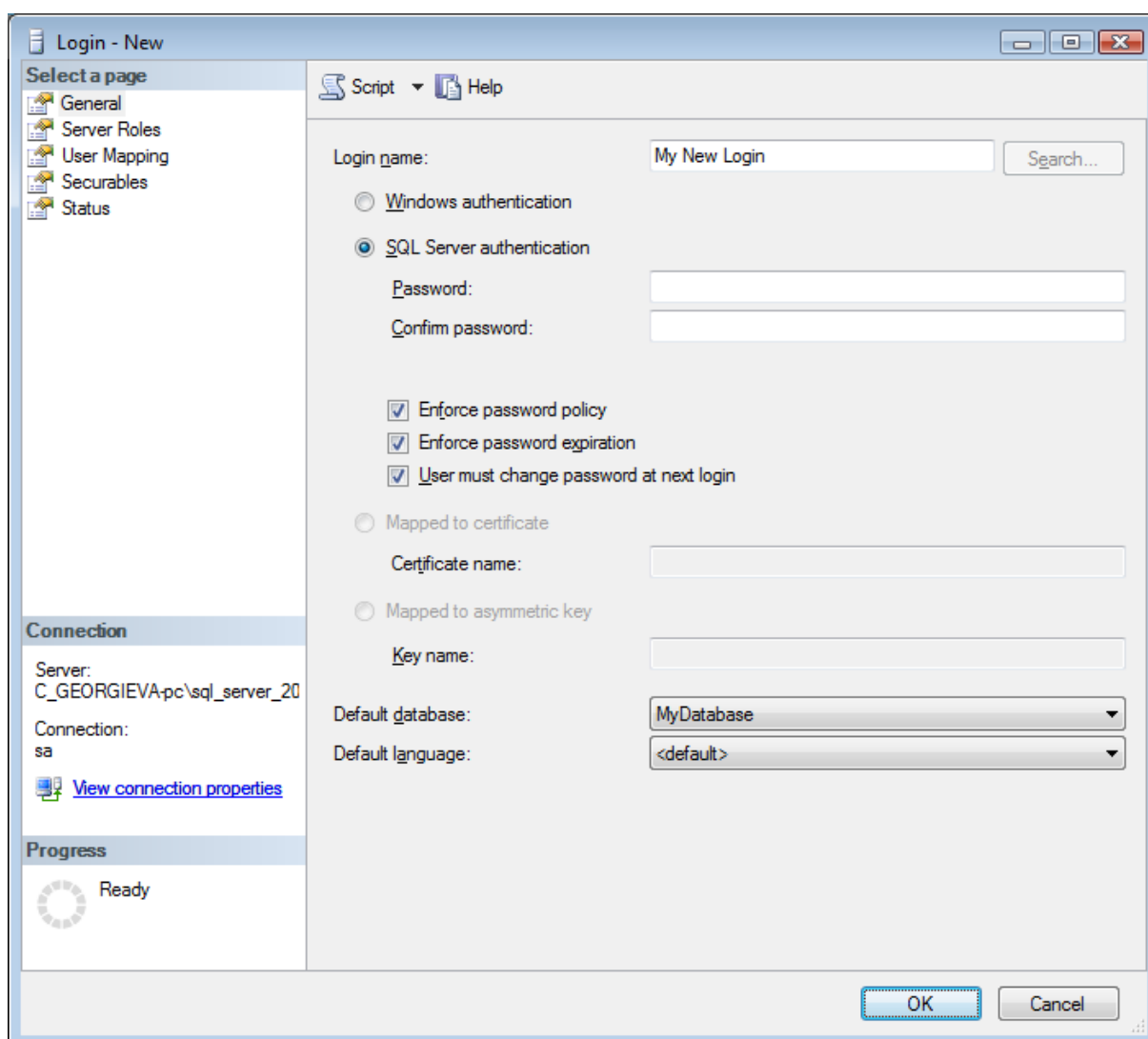
Опцията *SQL Server and Windows Authentication Mode* позволява използването на двата режима за автентикация (нарича се още смесен режим – *Mixed Mode*).

Microsoft SQL Server логины

След като се избере режима на автентикация, може да се започне с осигуряване на достъп на потребителите до SQL Server чрез добавяне на акаунт (логин – *Login*) на системата за сигурност. Възможно е създаването на два типа логины – стандартни логины и логины на Windows. Ако сървърът е конфигуриран за сигурност от смесен тип, могат да се използва и двата типа логины, в противен случай – само логины на Windows.

Стандартни логины

В Management Studio стандартни логины се създават, като се маркира *Security / Logins* и от контекстното меню се избере *New Login....* В диалоговия прозорец *Login – New* се въвежда името на акаунта, избира се SQL Server Authentication за създаване на нов SQL Server логин, въвежда се парола, определя се подразбиращата се база от данни и подразбиращия се език за логина (фиг. 2).



Фиг. 2 Създаване на логин

Създаването на нов логин може да се осъществи и чрез използването на системната съхранена процедура:

```
sp_addlogin [@loginame =] 'login'
```

```
[, [@passwd =] 'password']
[, [@defdb =] 'database']
[, [@deflanguage =] 'language']
[, [@sid =] 'sid']
[, [@encryptopt =] 'encryption_option']
```

Параметърът @sid е уникален идентификационен номер на сигурността. Има подразбираща се стойност NULL. Ако е NULL, системата генерира номер за новия логин. Използва се например, когато се генерира скрипт или се преместват SQL Server логици от един сървър на друг и е необходимо логините да имат същия SID на различните сървъри.

Параметърът @encryptopt определя дали паролата се съхранява криптирана в системните таблици. Подразбиращата се стойност е NULL, което означава, че паролата се съхранява криптирана. Другата възможна стойност е *skip_encryption* – паролата се съхранява без да е криптирана. Например:

```
EXEC sp_addlogin @loginame = 'MyLogin1',
               @passwd = 'MyPassword',
               @defdb = 'MyDatabase'
```

Изтриването на логин се извършва чрез изпълнението на системната съхранена процедура:

```
sp_droplogin [@loginame =] 'login'
```

Например:

```
EXEC sp_droplogin @loginame = 'MyLogin1'
```

Необходимо е първо да се премахне потребителският идентификатор, чрез който логинът получава достъп до някоя база от данни (чрез *sp_dropuser*). Не е възможно да се премахне логинът *sa* на системния администратор; логин, притежаващ база от данни; логин, който в момента се използва.

Смяна на парола на SQL Server логици освен чрез диалоговата рамка *Login Properties*, може да се извърши и чрез процедурата:

```
sp_password [[@old =] 'old_password', ]
           {[@new =] 'new_password'}
           [,[@loginame =] 'login']
```

Не е допустимо да се използва с логици на Windows, тъй като те се управляват от Windows. Всеки потребител може да промени своята собствена парола. Например:

```
EXEC sp_password NULL, 'NewPassword'
```

В този случай не се използва параметърът @loginame, но задължително се посочва старата (текущата) парола. Само член на *sysadmin* може да промени парола на друг потребителски логин, като се включи параметърът @loginame, но не е задължително да се посочва старата парола (@old). Например, когато се е логнал член на *sysadmin*, изпълнението на процедурата:

```
EXEC sp_password @new = NULL, @loginame = 'student'
```

задава стойност NULL на паролата на логин student.

Следното изпълнение на процедурата променя паролата на текущия потребител, тъй като не е посочен друг логин:

```
EXEC sp_password @new = 'NewPassword', @old = NULL
```

Логини на Windows

Един Windows логин може да бъде съпоставен с потребител или групови акаунти на Windows домейни. В диалоговия прозорец *Login – New* се въвежда името на акаунта, избира се Windows Authentication, определят се Windows домейн, подразбиращи се база от данни и език (фиг. 2).

Определянето на подразбираща се база от данни не предоставя на логина позволение за достъп до базата от данни.

За разрешаване на Windows потребител или група да се свързва със SQL Server може да се използва и системната съхранена процедура:

```
sp_grantlogin [@loginame =] 'login'
```

Например:

```
EXEC sp_grantlogin @loginame = 'MyComputer\MyLogin'
```

Премахването на такъв логин се извършва чрез системната съхранена процедура:

```
sp_revokelogin [@loginame =] 'login'
```

Например:

```
EXEC sp_revokelogin @loginame = 'MyComputer\MyLogin'
```

По този начин не се предотвратява изрично свързването на Windows логините със SQL Server. Те все още биха могли да се свържат, ако членуват в Windows група, на която е предоставен достъп до SQL Server чрез sp_grantlogin.

За отказване на достъп до сървъра на даден логин независимо от членството му в Windows групи се използва:

```
sp_denylogin [@loginame =] 'login'
```

Например:

```
EXEC sp_denylogin @loginame = 'MyComputer\MyLogin'
```

С помощта на системната съхранена процедура

```
sp_helplogins [[@LoginNamePattern =] 'login']
```

може да се получи информация за даден логин (стандартен или на Windows) и присъединените потребителски идентификатори за съответните бази от данни. Ако не се зададе стойност на параметъра, се дава списък с всички валидни логини. Например:

```
EXEC sp_helplogins 'student'
```

```
EXEC sp_helplogins 'MyComputer\MyLogin'
```

Роли

За да се опрости администрирането на много потребители, SQL Server използва роли (*roles*). Ролята е административна единица в SQL Server, която съдържа SQL Server логини, Windows логини или други роли. Настройките за сигурността, дефинирани за дадена роля, се прилагат за всичките нейни членове.

Механизмът за реализиране на системата за сигурност в SQL Server включва някои роли с права по подразбиране – фиксирани роли за сървър и фиксирани роли за база от данни. Предварително дефинираните роли не могат да бъдат членове на други роли.

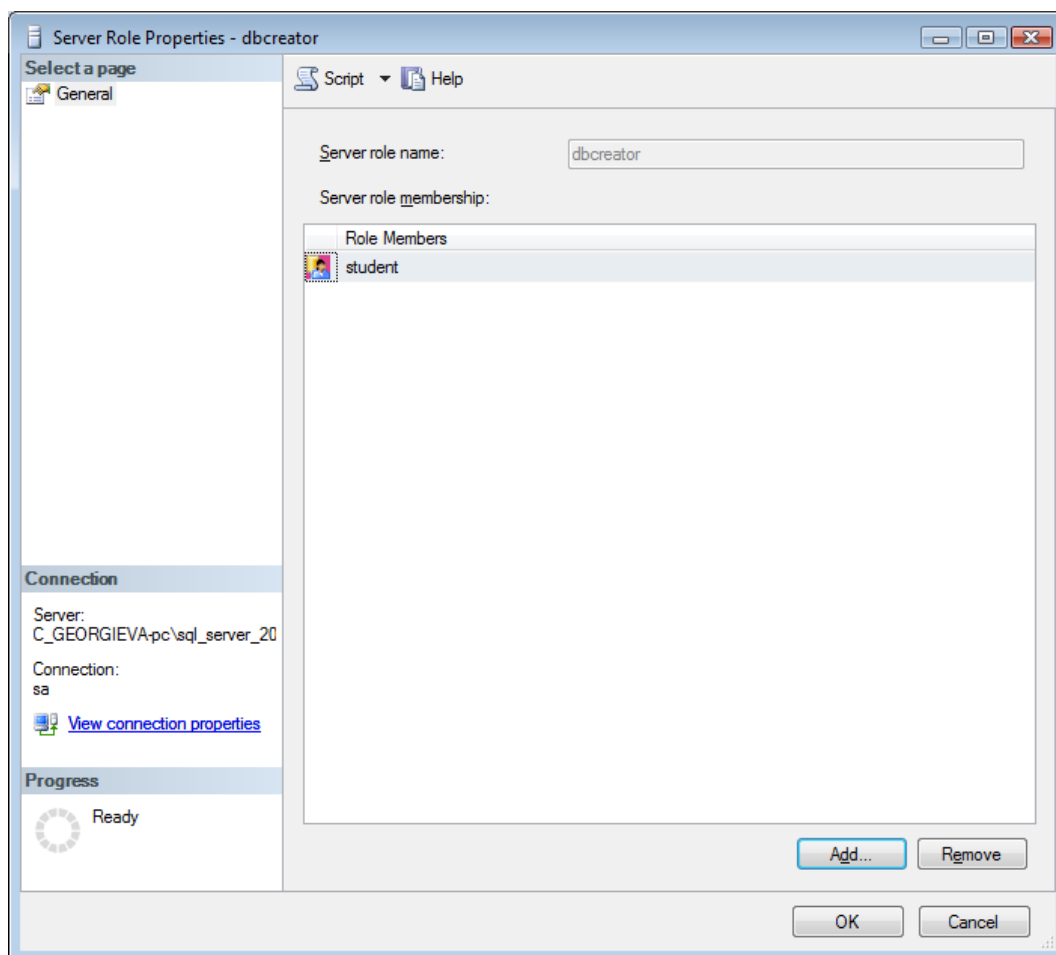
Фиксирани роли за сървър

Използват се за ограничаване на административния достъп, който един потребител има, след като се е свързал със SQL Server. Някои потребители например могат да имат пълен контрол, докато на други да е позволено само да извършват управление на сигурността. Предварително дефинираните роли за сървър са:

- **Sysadmin:** Членове на ролята *sysadmin* са упълномощени да извършат всякакви дейности в SQL Server.
- **Serveradmin:** Тези потребители могат да задават опциите на конфигурацията.
- **Setupadmin:** Членовете на тази роля могат да добавят и премахват свързани сървъри и да контролират процедурите по инсталирането.

- **Securityadmin:** Тези потребители изпълняват всички задачи, свързани с реализиране на системата за сигурност, като например създаване и изтриване на логици, предоставяне на потребителите на право за създаване на бази от данни, четене на дневника на грешките.
- **Processadmin:** Предназначена е за потребители, които трябва да управляват процесите в SQL Server. Членовете на тази роля могат да прекратят всеки процес.
- **Dbcreator:** Тези потребители могат да създават, променят и изтриват бази от данни.
- **Diskadmin:** Предназначена за потребители, които трябва да управляват файлове на диска. Могат да създават огледални бази от данни и да добавят устройства за архивиране.
- **Bulkadmin:** Членовете на тази роля могат да изпълняват конструкцията BULK INSERT.

Не е възможно да се създаде нова роля на ниво сървър, нито променянето или изтриването на съществуващите. Добавянето на член към фиксирана роля за сървър се осъществява чрез Management Studio, като се маркира *Security / Server Roles* и от контекстното меню на съответната роля се избира *Properties*. От диалоговата рамка *Server Roles Properties* се избира бутона *Add* (фиг. 3).



Фиг. 3 Добавяне на член към фиксирана роля за сървър

Ако е необходимо даден потребител да няма никакви административни права, трябва да не принадлежи на никоя фиксирана роля за сървър.

Конфигурирането на ролите за сървър може да се осъществи и чрез Transact-SQL. Следната съхранена процедура добавя логин към роля за сървър:

```
sp_addsrvrolemember [@loginame =] 'login',
                    [@rolename =] 'role'
```

Например:

```
EXEC sp_addsrvrolemember @loginame = 'student',
                        @rolename = 'dbcreator'
```

По аналогичен начин се използва съхранената процедура

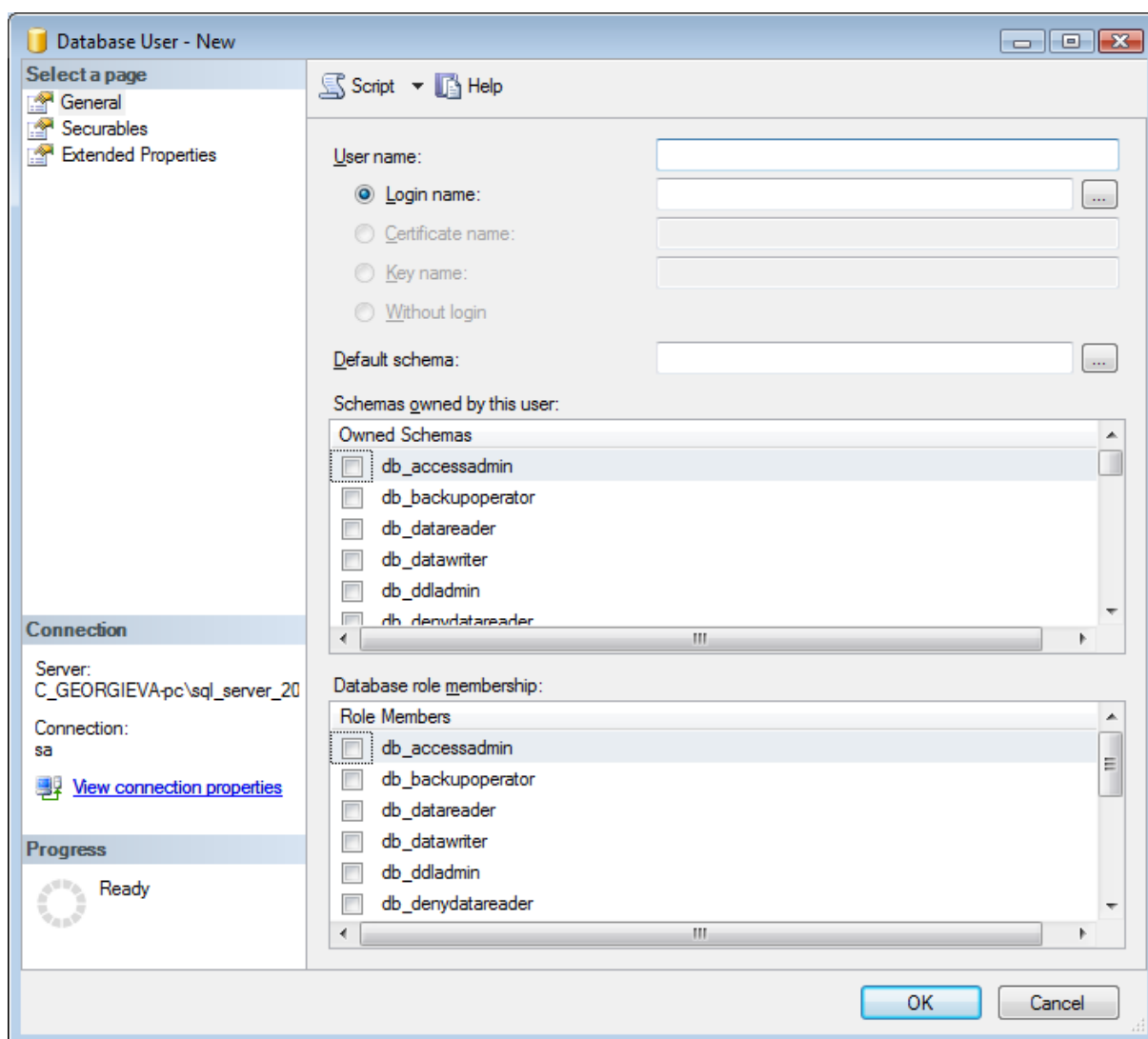
```
sp_dropsrvrolemember [@loginame =] 'login',
                    [@rolename =] 'role'
```

за премахване на логин от роля за сървър.

Чрез `sp_helpsrvrole` [[@srvrolename =] 'role'] се извежда списък с всички фиксирани роли за сървър и техните описания. За извличане на информация за членовете на фиксирана роля за сървър е предназначена процедурата `sp_helpsrvrolemember` [[@srvrolename =] 'role'].

Създаване на потребителски идентификатори за база от данни

За да се осъществи достъп до база от данни в SQL Server, е необходимо да се създаде потребителски идентификатор за базата от данни. След като на даден потребител е позволен достъп до SQL Server с използване на неговия логин, се изисква потребителски идентификатор за всяка база от данни, за която потребителят има нужда от достъп. Това изискване предотвратява при свързване на потребителите със SQL Server да имат достъп до всяка база от данни на сървъра. Потребителски идентификатор за база от данни може да се създаде от Management Studio, като се маркира *Users* на съответната база от данни и от контекстното меню се избере *New User....* От диалоговия прозорец *Database User – New User* се избира някой от вече създадените логици, тъй като могат да се добавят само съществуващите логици (фиг. 4).



Фиг. 4 Създаване на потребителски идентификатор за база от данни

Създаването на потребителски идентификатор в текущата база от данни може да се осъществи и чрез T-SQL с помощта на системната съхранена процедура:

```
sp_adduser [@loginame =] 'login'
          [, [@name_in_db =] 'user']
          [, [@grpname =] 'group']
```

Параметърът @loginame е име на съществуващ логин; @name_in_db е потребителско име, което ако се пропусне, се използва името на съответния логин; @grpname е роля (потребителски дефинирана или фиксирана роля за база от данни), на която новият потребителски идентификатор автоматично става член. Например:

```
EXEC sp_adduser @loginame = 'student',
               @name_in_db = 'dbuser',
               @grpname = 'db_owner'
```

Премахване на потребителски идентификатор от текущата база от данни може да се извърши чрез системната съхранена процедура:

```
sp_dropuser [@name_in_db =] 'user'
```

Например:

```
EXEC sp_dropuser 'MyLogin'
```

Тези две процедури са включени за съвместимост с предишна версия. За същата цел се препоръчва използването на описаните по-долу.

За предоставяне на достъп на даден логин до дадена база от данни се използва съхранената процедура:

```
sp_grantdbaccess [@loginame =] 'login'
[, [@name_in_db =] 'name_in_db' [OUTPUT]]
```

Например:

```
EXEC sp_grantdbaccess @loginame = 'MyLogin',
                        @name_in_db = 'MyName'
EXEC sp_grantdbaccess 'guest'
```

За премахване на потребителски акаунт от текущата база от данни се изпълнява процедурата:

```
sp_revokedbaccess [@name_in_db =] 'name'
```

Например:

```
EXEC sp_revokedbaccess @name_in_db = 'MyName'
EXEC sp_revokedbaccess 'guest'
```

Помощна информация за потребителски идентификатор и ролите в текущата база от данни може да се получи чрез:

```
sp_helpuser [[@name_in_db =] 'security_account']
```

Например:

```
EXEC sp_helpuser @name_in_db = 'MyLogin'
EXEC sp_helpuser
EXEC sp_helpuser @name_in_db = 'MyRole'
```

Потребителят *dbo (database owner)* – собственикът на базата от данни е специален потребител във всяка база от данни. Членовете на фиксираната роля *sysadmin* автоматично стават *dbo* потребители във всяка база от данни в системата и могат да извършват всички административни функции в базите от данни (като например добавяне на потребители и създаване на обекти). Промяна на собственика на текущата база от данни може да се осъществи чрез системната съхранена процедура:

```
sp_changedbowner [@loginame =] 'login'
[, [@map =] remap_alias_flag]
```

Първият параметър определя кой е новия собственик на текущата база от данни. Не може новият собственик вече да е потребител на базата от данни, затова предварително трябва да се изтрие като потребител. Вторият параметър приема стойност *true* или *false* (по подразбиране *true*) и определя дали съществуващите псевдоними (потребителски идентификатори) на стария собственик да се прехвърлят на новия или да се изтрият. След изпълнението на процедурата новият собственик е вече потребител *dbo* в базата от данни. Тази процедура може да се изпълнява само от членове на *sysadmin* и от собственика на базата от данни. Собственикът *sa* на системните бази от данни (*master*, *model*, *msdb*, *tempdb*) не може да се променя.

Например:

```
EXEC sp_changedbowner 'MyLogin'
```

Потребителят *guest* позволява на всеки, притежаващ валиден SQL Server логин, да има достъп до базата от данни. Ако една база от данни няма потребителски акаунт *guest*, никой SQL Server логин не може да има достъп до тази база от данни, освен ако е асоцииран с валиден потребителски акаунт. Потребителят *guest* трябва да има ограничени разрешения, тъй като всеки със SQL Server логин може да го използва.

След създаване на потребителски идентификатори за всички, за които е необходимо да имат достъп до базата от данни, трябва да се зададе какви действия е допустимо да извършват спрямо обектите в базата от данни. Това се осъществява чрез дефиниране на привилегии директно на потребителите или чрез добавяне на потребителите към роли за бази от данни с предварително дефинирани привилегии.

Привилегии

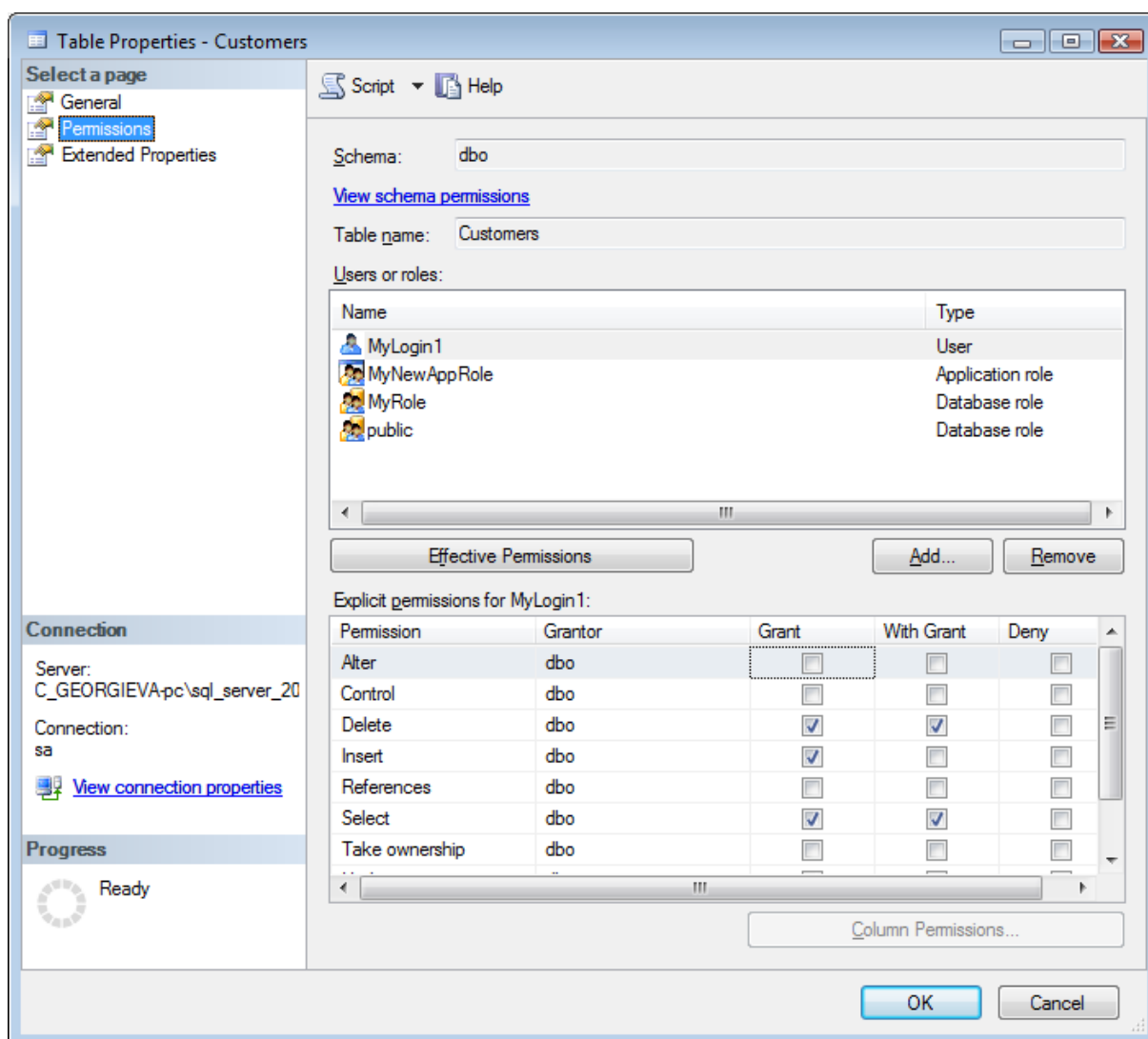
Потребителите имат множество от привилегии, които определят какви действия могат да изпълняват в SQL Server или в една база от данни. Използват се два типа привилегии: за обекти – контролират достъпа до таблици, изгледи, съхранени процедури, дефинирани от потребителя функции и за конструкции – контролират администраторските дейности като създаване на база от данни и обекти в нея.

Привилегии за обекти

Различни са за различните видове обекти и чрез тях се контролира кой кои данни може да чете, добавя, изтрива или променя. Обектните позволения са:

- **Select:** Когато е разрешено, потребителят може да чете данните от таблица или изглед. Това позволение може да се предоставя на ниво колона, в резултат на което се позволява четене само на определени колони.
- **Insert:** Позволява на потребителите да добавят нови редове в таблица.
- **Update:** Позволява на потребителите да променят съществуващите редове в таблица. Тази привилегия може да се предоставя на ниво колона, в резултат на което се позволява променяне само на определени колони.
- **Delete:** Позволява да се изтриват редове от таблица.
- **References:** Позволява на потребителите да създават ограничение външен ключ, който сочи към колони от таблицата, за която се прилага привилегията. Тази привилегия може да се предоставя на ниво колона, в резултат на което се позволява рефериране само на определени колони.
- **Execute:** позволява на потребителите да изпълняват съхранената процедура или дефинираната от потребителя функция, за която се прилага привилегията.

Предоставянето на привилегии за обекти от Management Studio се извършва, като се маркира съответния обект и от контекстното меню се избере *Properties | Permissions* (фиг. 5).



Фиг. 5 Предоставяне на привилегии за обекти

Привилегии за конструкции

Само членовете на *sysadmin* и собственикът на базата от данни могат да предоставят привилегии за конструкции, които са:

- Create Database
- Create Table
- Create View
- Create Procedure
- Create Index
- Create Rule
- Create Default
- Create Function
- Backup Database

Предоставянето на привилегии за конструкции от Management Studio се извършва, като се маркира съответната база от данни и от контекстното меню се избере *Properties | Permissions*.

Роли за бази от данни

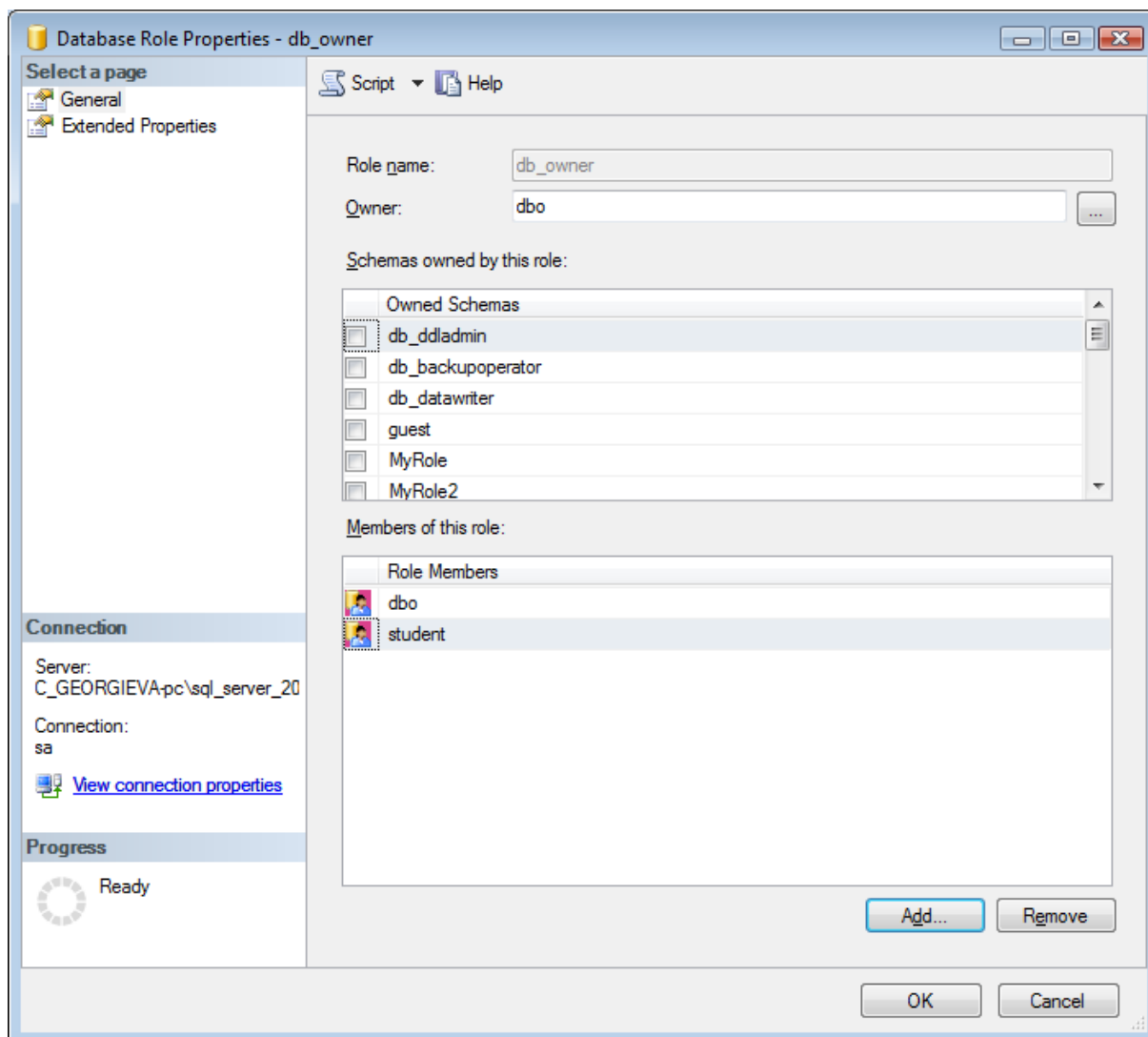
В SQL Server, когато няколко потребителя имат нужда от едни и същи привилегии в дадена база от данни, е много по-удобно те да им бъдат предоставени наведнъж като група, отколкото да се контролира достъпа на всеки потребител поотделно. Това се реализира чрез дефиниране на роли за всяка отделна база от данни. SQL Server поддържа три типа роли за бази от данни – фиксирани роли, потребителски дефинирани стандартни роли, потребителски дефинирани роли за приложения.

Фиксирани роли за бази от данни

Предварително дефинираните роли за бази от данни са вградени и имат привилегии, които не могат да бъдат променени. Използват се за присвояване на привилегии за администриране на базата от данни, като за целта е необходимо потребителят да бъде добавен към съответната роля. Фиксираните роли за бази от данни, които могат да бъдат използвани за предоставяне на привилегии, са:

- **Db_owner:** Членовете на тази роля имат пълен контрол върху всички данни в базата от данни.
- **Db_accessadmin:** Потребителите имат привилегията да определят кой да има достъп до базата от данни чрез добавяне и премахване на потребителски идентификатори.
- **Db_datareader:** Членовете на тази роля могат да четат данни от всички таблици в базата от данни.
- **Db_datawriter:** Членовете на тази роля могат да добавят, променят и изтриват данни от всички таблици в базата от данни.
- **Db_ddladmin:** Членовете на тази роля могат да изпълняват всички конструкции от езика за дефиниране на данни – Data Definition Language (DDL). Това им позволява да създават, променят и изтриват обекти в базата от данни.
- **Db_securityadmin:** Членовете на тази роля могат да добавят и премахват потребители от ролите за бази от данни и да управляват привилегиите за обекти и за конструкции.
- **Db_backupoperator:** Тези потребители могат да архивират базата от данни.
- **Db_denydatareader:** Членовете на тази роля не могат да получават позволение да четат данните в базата от данни, но могат да им се предоставят привилегии да контролират позволенията за *Select* за обектите в базата от данни, разрешения за *Insert*, *Update* и *Delete* за обектите в базата от данни и привилегии за конструкции.
- **Db_denydatawriter:** Членовете на тази роля не могат да получават позволение да променят данните в базата от данни, но могат да им се предоставят привилегии да контролират позволенията за *Insert*, *Update* и *Delete* за обектите в базата от данни, разрешения за *Select* за обектите в базата от данни и привилегии за конструкции.
- **Public:** Има за цел да предостави на потребителите подразбиращ се набор от позволения в базата от данни. Всички потребители в базата от данни автоматично се присъединяват към тази роля и не могат да бъдат отстранени от нея.

За добавяне на потребители към някоя от изброените фиксирани роли за бази от данни може да се използва Management Studio, като се избере базата от данни и се маркира съответната роля от списъка в *Roles*. От контекстното меню се избира *Properties* и от диалоговата рамка *Database Role Properties* се избира бутона *Add* (фиг. 6).



Фиг. 6 Добавяне на потребители към фиксирана роля за бази от данни

За извеждане на привилегиите на фиксирана роля за бази от данни се използва съхранената процедура:

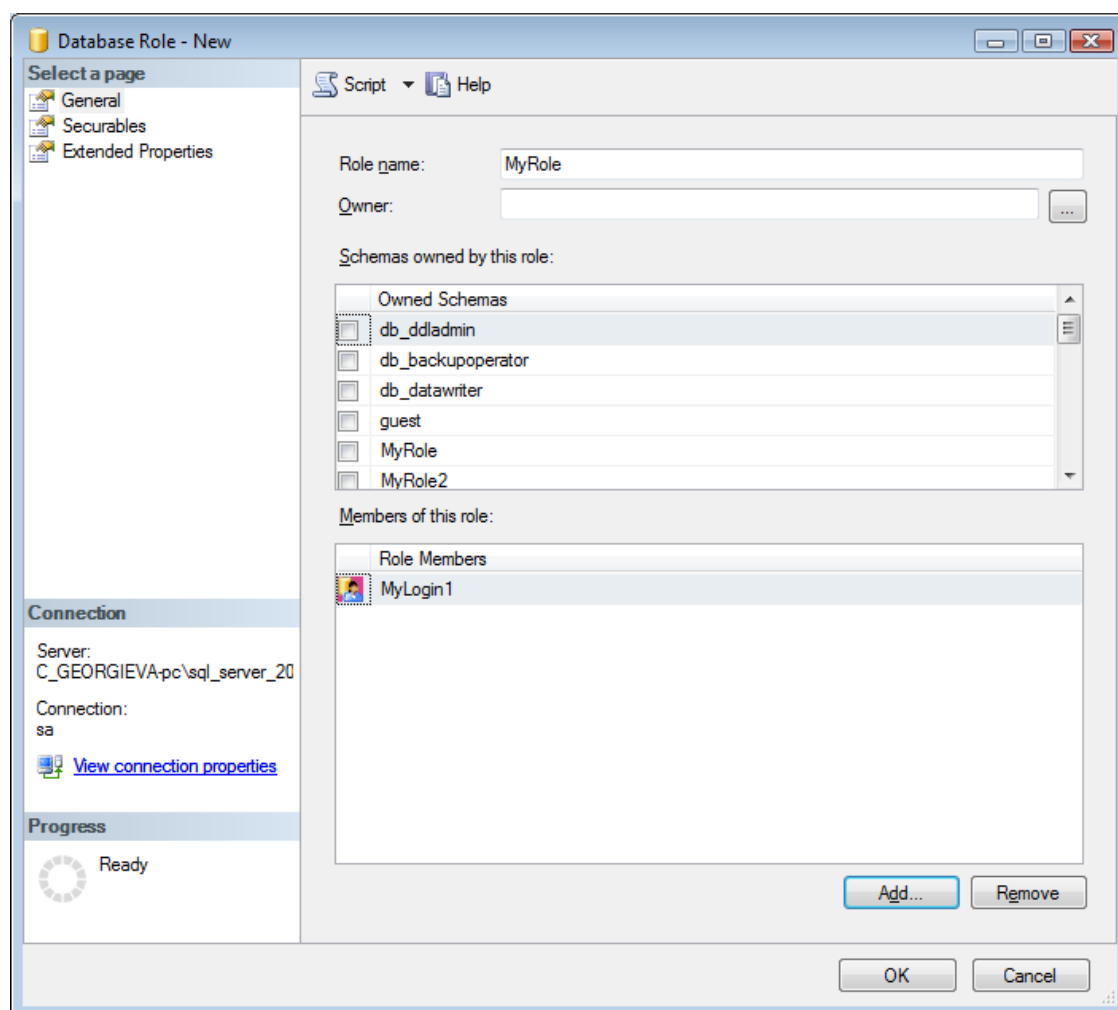
```
sp_dbfixedrolepermission [[@rolename =] 'role']
```

Например:

```
EXEC sp_dbfixedrolepermission  
    @rolename = 'db_securityadmin'
```

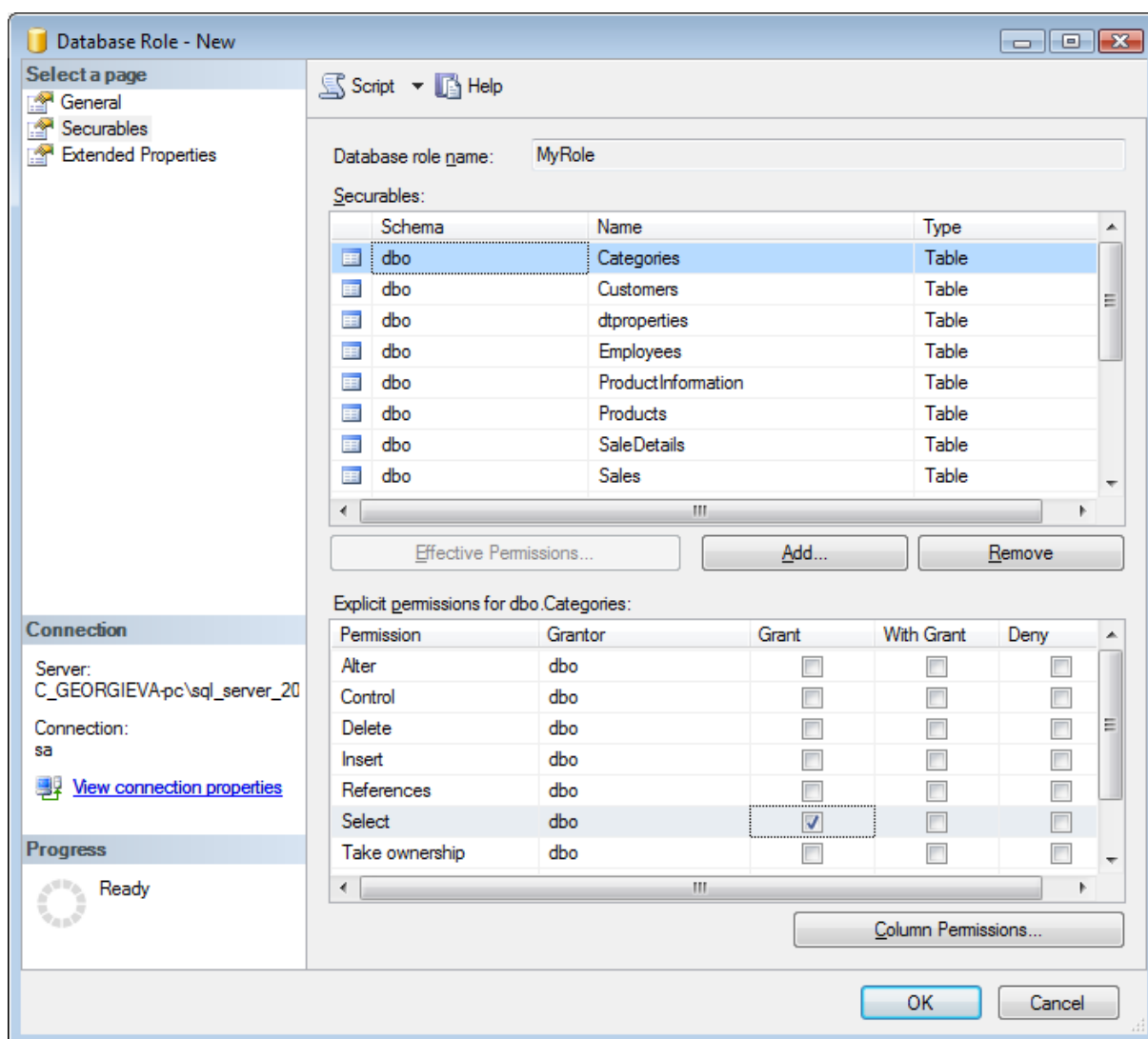
Потребителски дефинирани стандартни роли за бази от данни

Когато се създава нова роля, се задават привилегиите, които потребителите наследяват при присъединяване към тази роля (за разлика от фиксираните роли, където привилегиите са предварително дефинирани). Възможно е дефинирани от потребителя роли за бази от данни да се членове на други роли. Създаването на роля от Management Studio се извършва, като се маркира *Roles* на съответната база от данни. От контекстното меню се избира *New Database Role...* и в диалоговата рамка *Database Role – New* се въвежда името на новата роля и чрез бутона *Add* се добавят потребители към нея (фиг. 7).



Фиг. 7 Създаване на роля

Позволението за роля могат да се дефинират в страницата *Securables* (фиг. 8).



Фиг. 8 Задаване на позволения на роля

Чрез диалоговата рамка *Column Permissions* се уточняват колоните, които могат да бъдат четени или актуализирани за позволенията *Select* и *Update*.

Системната съхранена процедура за създаване на нова стандартна роля е:

```
sp_addrole [@rolename =] 'role'
[, [@ownername =] 'owner']
```

Параметърът @ownername определя собственика на ролята и трябва да бъде име на потребител в текущата база от данни. Подразбиращата стойност е *dbo*.

Например:

```
EXEC sp_addrole @rolename = 'MyRole'
```

За премахване на съществуваща стандартна роля се използва:

```
sp_droprole [@rolename =] 'role'
```

Например:

```
EXEC sp_droprole @rolename = 'MyRole'
```

За получаване на помощна информация за ролята в текущата база от данни:

```
sp_helprole [[@rolename =] 'role']
```

Добавянето на член на роля (фиксирана или стандартна) за текущата база от данни се извършва чрез процедурата:

```
sp_addrolemember [@rolename =] 'role',
[@membername =] 'security_account'
```

Например:

```
EXEC sp_addrolemember
    @rolename = 'db_securityadmin',
    @membername = 'MyLogin'
EXEC sp_addrolemember
    @rolename = 'MyRole',
    @membername = 'MyLogin'
```

Премахването на член от роля (фиксирана или стандартна) за текущата база от данни се осъществява чрез:

```
sp_droprolemember [@rolename =] 'role',
                  [@membername =] 'security_account'
```

Например:

```
EXEC sp_droprolemember
    @rolename = 'db_securityadmin',
    @membername = 'MyLogin'
EXEC sp_droprolemember
    @rolename = 'MyRole',
    @membername = 'MyLogin'
```

За получаване на помощна информация за членовете на роля (фиксирана или стандартна) в текущата база от данни се изпълнява:

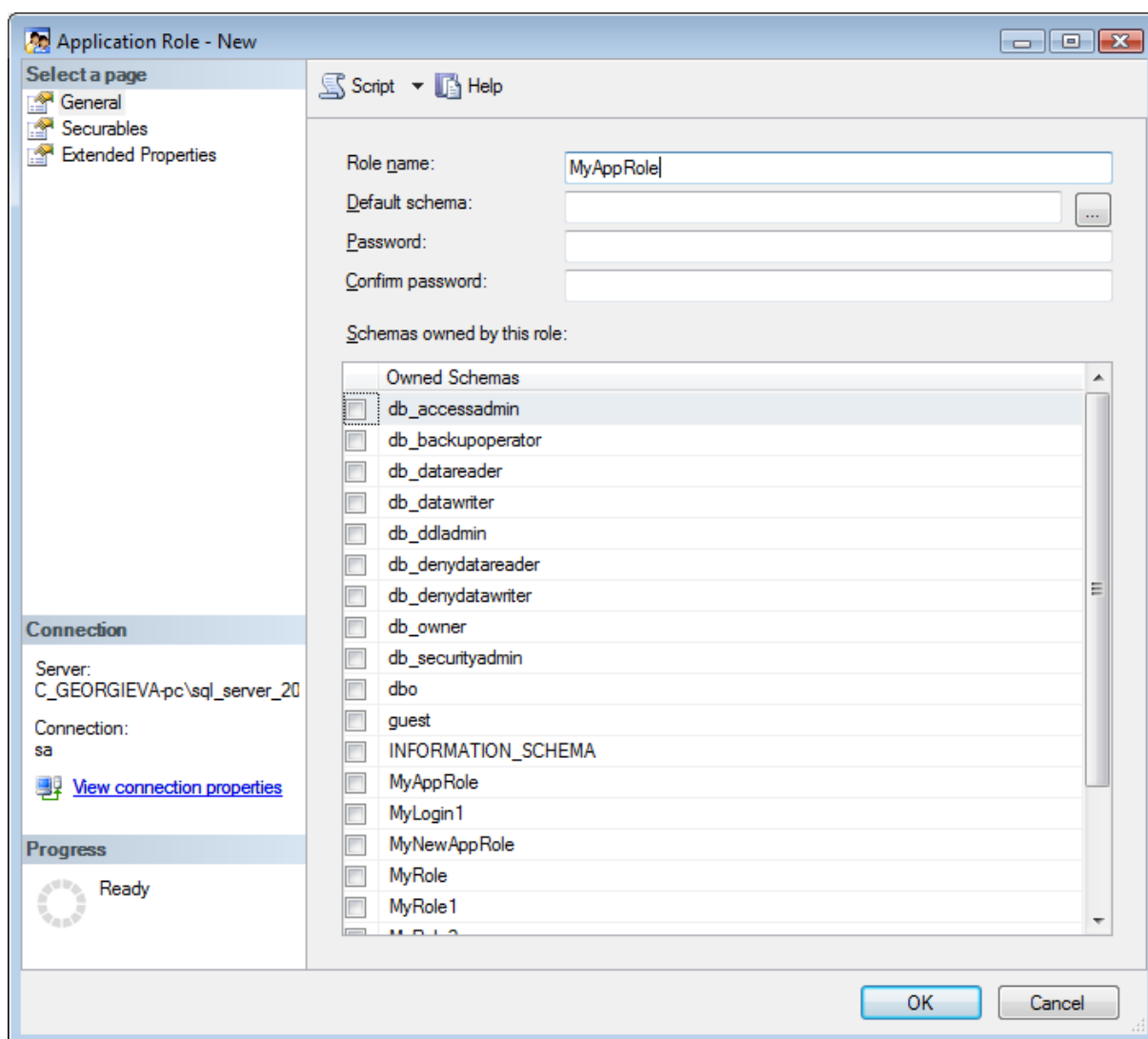
```
sp_helprolemember [[@rolename =] 'role']
```

Например:

```
EXEC sp_helprolemember @rolename = 'db_securityadmin'
EXEC sp_helprolemember @rolename = 'MyRole'
```

Потребителски дефинирани роли за приложения за бази от данни

Позволяват да се създадат защитени чрез парола роли за определени приложения. Потребители и други роли не могат да бъдат присвоени на роля за приложение. Роля за приложение се активира, когато приложението се свързва с базата от данни. Предназначени са да се използват от приложения, които осъществяват достъп до базата от данни, нямат асоциирани с тях логини. Потребителите нямат достъп до данните извън приложението (като например Microsoft Access). Създаването на роли за приложения от Management Studio се извършва, като се маркира *Roles* на съответната база от данни. От контекстното меню се избира *New Application Role...* и в диалоговата рамка *Application Role – New* се въвежда името на новата роля и парола (фиг. 9).



Фиг. 9 Създаване на роля за приложения

Друг начин за създаване на роля за приложение е да се използва съхранената процедура:

```
sp_addapprole [@rolename =] 'role',
               [@password =] 'password'
```

Например:

```
EXEC sp_addapprole @rolename = 'MyNewAppRole',
                   @password = 'password'
```

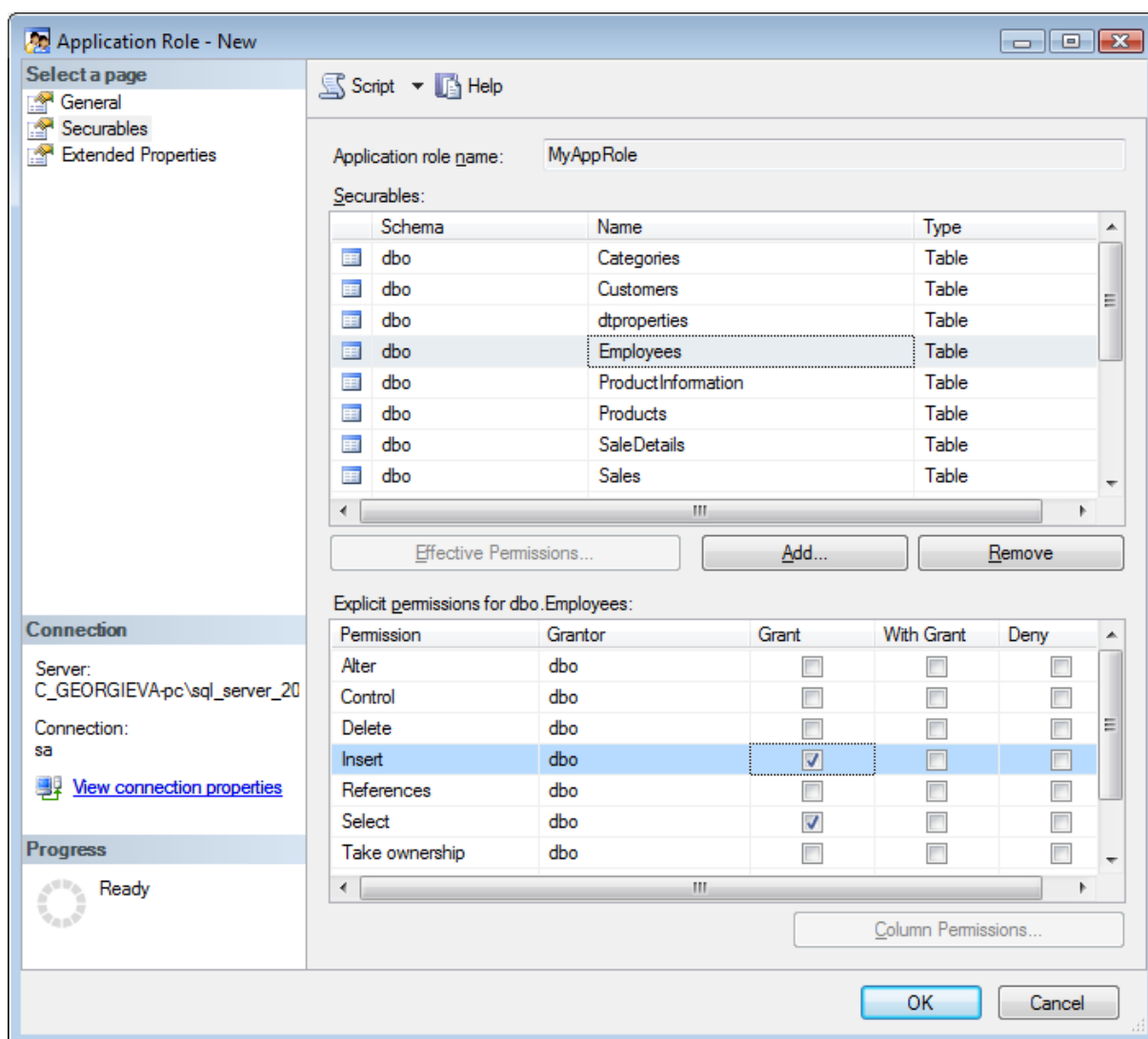
Премахване на такава роля за приложение се осъществява чрез:

```
sp_dropapprole [@rolename =] 'role'
```

Например:

```
EXEC sp_dropapprole @rolename = 'MyNewAppRole'
```

Позволението на роля за приложения могат да се дефинират в страницата *Securables* (фиг. 10).



Фиг. 10 Задаване на позволения на роля за приложения

За активиране на ролята за приложение се използва системната съхранена процедура:

```
sp_setapprole [@rolename =] 'role',
               [@password =] {Encrypt N'password'} | 'password'
               [, [@encrypt =] 'encrypt_style']
```

Паролата може да бъде криптирана чрез ODBC функцията Encrypt. Когато се прилага тази функция паролата трябва да е конвертирана в Unicode низ, затова се предшества от N. Третият параметър определя стил на криптиране и може да има стойност: *none* – паролата не се криптира и се изпраща към MS SQL Server като чист текст (това е подразбиращата се стойност) или *odbc* – паролата се криптира чрез използване на ODBC функцията Encrypt преди да бъде изпратена към SQL Server (поддържа се от ODBC и OLE DB клиенти).

Например:

```
EXEC sp_setapprole @rolename = 'MyNewAppRole',
                  @password = 'password'
```

Всеки потребител може да изпълни тази съхранена процедура чрез осигуряване на валидна парола за приложението. В Microsoft Access например, може да се създаде макрос с действие RunSQL, аргументите на което са зададени по следния начин:

SQL Statement	EXEC sp_setapprole @rolename = 'I
Use Transaction	No

Стартирането на този макрос може да се предизвика автоматично при отваряне на *.adp* файла чрез добавяне на действие RunMacro в макрос с име AutoExec.

Промяна на паролата на роля за приложение в текущата база от данни може да се извърши чрез:

```
sp_approlepassword [@rolename =] 'role',
                  [@newpwd =] 'password'
```

Възможно е да се изпълни само от членове на фиксирани роли за бази от данни *db_securityadmin* и *db_owner*. Последните две разгледани системни съхранени процедури не могат да бъдат извиквани като част от дефинирана от потребителя транзакция (затова вторият аргумент на макроса трябва да има стойност No).

Състояния на привилегиите

Съществуват три състояния на привилегиите:

- предоставяне (Grant) – дава се позволение за извършване на съответната задача; ако се отнася за роли, всички членове на ролята наследяват позволението;
- отнемане (Revoke) – премахва се вече дадено позволение, но не се забранява явно извършването на съответната задача от потребителя или ролята; въпреки отнемането на дадена привилегия от даден потребител или дадена роля, те могат да наследят тази привилегия от друга роля;
- отказване (Deny) – явно се отказва позволение за изпълнение на дадена задача от потребителя или ролята и забранява наследяването му от друга роля; отказването има приоритет над всички предоставени привилегии.

Действителните права на един логин на системата за сигурност, на който е предоставен достъп до база от данни, представляват обединението от привилегиите, предоставени на ролята *Public*, привилегиите, които потребителят получава като член на други роли, и всички привилегии, които са предоставени директно на съответния потребителски идентификатор.

Собственик на обект в базата от данни

Обектите на базата от данни са таблици, индекси, изгледи, съхранени процедури, дефинирани от потребителя функции, тригери. Първоначално потребителят, който е създал обект в базата от данни, е негов собственик (*database object owner*). Собственикът на базата от данни или системният администратор първо трябва да предостави на потребителя привилегия да създава отделни типове обекти. Собственикът на съответния обект на базата от данни може (освен да създава обект) и да предоставя на други потребители позволения да използват този обект.

Собствениците на обекти на базата от данни нямат специални логици. Създателят на обект в базата от данни е с безусловно предоставени всички права, но изрично трябва да предостави права на други потребители, преди те да могат да имат достъп до обекта.

Когато потребителите имат достъп до обекти, създадени от други потребители, обектът трябва да бъде квалифициран с името на собственика на обекта, в противен случай няма да е еднозначно определено кой обект да се използва, тъй като може да има няколко обекта с едно и също име, всеки от които да е собственост на различни потребители. Когато към един обект се прави обръщение, без да се посочи името на собственика му, например *MyTable* вместо *owner.MyTable*, SQL Server търси обекта в базата от данни в следния ред: 1) измежду обектите, които са собственост на текущия потребител; 2) измежду обектите, които принадлежат на собственика на базата от данни *dbo*.

Ако един потребител е собственик на обекти в базата от данни и трябва да бъде отстранен от базата от данни, първо ще е необходимо всички обекти, които са негова собственост или да бъдат изтрети, или да се прехвърли собствеността им на друг потребител.

Прехвърлянето на собственост на обект в базата от данни може да се осъществи чрез системната съхранена процедура:

```
sp_changeobjectowner [@objname =] 'object',
                    [@newowner =] 'owner'
```

Процедурата може да бъде изпълнявана само от членове на *sysadmin* и от собственика на базата от данни.

Косвени привилегии

Само членове на фиксираните роли за сървър, фиксирани роли за бази от данни или собствениците на бази от данни, собствениците на обекти в бази от данни могат да използват косвени привилегии. Косвените привилегии, получени от членство във фиксирана роля, не могат да бъдат променяни или да се предоставят поотделно на други логици, т.е. придобиват се, като даден логин стане член на фиксирана роля. Някои от привилегиите могат да бъдат придобити само чрез членство във фиксирана роля (например тези, които позволяват администриране на ниво сървър).

Собствениците на бази от данни и на обекти в база от данни също имат косвени привилегии, които им позволяват да извършват всички действия спрямо базата от данни или спрямо обекта, който притежават. Потребител, който притежава обект например таблица, освен да извлича и модифицира данните в нея, може да променя дефиницията на таблицата и да контролира привилегиите за тази таблица на другите потребители в базата от данни.

Косвените привилегии не могат да бъдат отнети от собствениците или фиксираните роли.

Привилегии на роли за приложения

Ролята за приложения съдържа парола и няма членове. Тази специална роля е създадена с цел да контролира какви привилегии се предоставят на потребителите, които осъществяват достъп до базата от данни от едно конкретно приложение. На ролята за приложения се предоставят привилегии за една база от данни и обектите в нея. Всеки потребител с валиден логин може да активира ролята за приложения (чрез *sp_setapprole*). След като една роля за приложения бъде активирана, всички други привилегии за потребители се отнемат до приключване на съответната сесия или приложение. Ако едно приложение трябва да осъществи достъп до друга база от данни, когато е активна конкретна роля за приложения, разрешението за друга база от данни може да се получи само чрез логина за потребител *guest*.

Конфигуриране на привилегии за обекти чрез Transact-SQL

Конструкции *GRANT*, *REVOKE*, *DENY* на Transact-SQL се използват за администриране на привилегиите за обекти и конструкции.

Предоставяне на привилегии за обекти

Общият вид на конструкцията за предоставяне на привилегии за обекти е:

```
GRANT {ALL [PRIVILEGES] | permission [,... n]}
{
    [(column_name [,... n])] ON {table | view}
    | ON {table | view}[(column_name [,... n])]
```

```

        | ON {stored_procedure | extended_procedure}
    }
TO security_account [... n]
[WITH GRANT OPTION]
[AS {group | role}]

```

При използване на тази конструкция трябва да се укаже:

- Коя е привилегията (или привилегиите) и на кого се предоставят.
- Върху какво се прилагат привилегиите за обекти, т.е. задава се еднозначно името на конкретния обект.
- Списък от акаунти на системата за сигурност, т.е. потребителски идентификатори и/или роли, на които се предоставят съответните привилегии.
- Опцията WITH GRANT OPTION (която е незадължителна) се включва, за да се позволи на съответния потребител да дава привилегията на други.
- Ключовата дума AS се използва, когато привилегията за обект е предоставена на дадена роля, след което е необходимо тази привилегия за обект да бъде предоставяна на потребители, които не са членове на ролята. Тъй като само потребител (не и роля) може да изпълнява конструкцията GRANT, определен член от дадената роля предоставя привилегия за обект под пълномощието на ролята.

Когато се използва някоя от диалоговите рамки, които предлага Management Studio за конфигуриране на привилегии, трябва да се има предвид, че наличието на отметка в полето *Grant* означава, че позволенията е предоставено.

Пример 1

```

USE MyDatabase
GO
GRANT SELECT ON Customers TO Public

```

Пример 2

```

GRANT Insert, Update, Delete
ON Customers
TO MyLogin, MyRole

```

Пример 3

```

GRANT SELECT
ON Employees (FirstName, LastName, Title)
TO MyLogin1

```

Пример 4 Предоставяне на привилегии за добавяне, променяне и изтриване на редове в таблицата Sales, както и за предоставяне на тези привилегии на други:

```

GRANT Insert, Update, Delete
ON Sales
TO MyRole3
WITH GRANT OPTION

```

Нека MyLogin3 е член на MyRole3. За да предостави MyLogin3 привилегията за добавяне, променяне и изтриване на редове в таблицата Sales на друг потребител, който не е член на MyRole3, трябва да се използва ключовата дума AS по следния начин:

```

-- изпълнява се от конекция на потребителя MyLogin3:
GRANT Insert, Update, Delete
ON Sales

```



```
TO MyLogin
AS MyRole3
```

Пример 5

```
GRANT EXEC
ON stored_proc_name
TO MyRole1
```

Пример 6 Нека MyLogin1 е член на фиксираната роля за бази от данни *db_denydatareader*:

```
EXEC sp_addrolemember
    @rolename = 'db_denydatareader',
    @membername = 'MyLogin1'
```

Тогава чрез

```
GRANT SELECT ON Customers TO MyLogin1
WITH GRANT OPTION
```

се предоставя позволение на MyLogin1 да дава тази привилегия SELECT на други, но продължава да няма позволение за четене на данните в таблицата Customers.

Отнемане на привилегии за обекти

Общият вид на конструкцията за отнемане на привилегии за обекти е:

```
REVOKE [GRANT OPTION FOR]
    {ALL [PRIVILEGES] | permission [,... n]}
    {
        [(column_name [,... n])] ON {table | view}
        | ON {table | view}[(column_name [,... n])]
        | ON {stored_procedure | extended_procedure}
    }
    {TO | FROM} security_account [,... n]
    [CASCADE]
    [AS {group | role}]
```

Синтаксисът на REVOKE е подобен на синтаксиса на конструкцията GRANT с някои изключения:

- Опционалните ключови думи GRANT OPTION FOR се използват за отнемане на WITH GRANT OPTION, предоставено на съответния акаунт на системата за сигурност.
- Опционалната ключова дума CASCADE се използва за отнемане на привилегиите, които са предоставени на зададения акаунт на системата за сигурност и на всеки друг акаунт, на който зададеният акаунт е предоставил привилегиите. Може да се включат ключовите думи GRANT OPTION FOR и CASCADE, за да се отнеме състоянието на разрешение GRANT, предоставено чрез акаунта на системата за сигурност на други акаунти на тази система.

Когато се използва някоя от диалоговите рамки, които предлага Management Studio за конфигуриране на привилегии, трябва да се има предвид, че премахването на отметката от полето *Grant* показва, че позволение е отнето.

Пример 7 Отнема се привилегията на MyLogin1 за добавяне на редове в таблицата Sales (или се отменя отказ върху предоставената изрично привилегия):

```
USE MyDatabase
GO
REVOKE INSERT ON Sales TO MyLogin1
```

Пример 8 Запазва се привилегията INSERT на MyLogin1 върху таблицата Sales, но се отнема привилегията му да предоставя на други потребители привилегията INSERT върху тази таблица:

```
REVOKE GRANT OPTION FOR INSERT ON Sales TO MyLogin1
```

Пример 9 Отнемане на разрешението да се изпълнява съхранената процедура *stored_proc_name* от MyLogin1 и от всички други потребители, на които MyLogin1 е предоставил разрешение EXEC:

```
REVOKE EXEC
ON stored_proc_name
FROM MyLogin1
CASCADE
```

Отказване на привилегии за обекти

Общият вид на конструкцията за отказване на привилегии за обекти е:

```
DENY {ALL [PRIVILEGES] | permission [,... n]}
{
    [(column_name [,... n])] ON {table | view}
    | ON {table | view}[(column_name [,... n])]
    | ON {stored_procedure | extended_procedure}
}
TO security_account [,... n]
[CASCADE]
```

Синтаксисът на конструкцията DENY е аналогичен на синтаксиса на конструкциите GRANT и REVOKE. Конструкцията DENY може да включва ключовата дума CASCADE за явно отказване на привилегиите, които са били предоставени от съответния акаунт на други акаунти.

Когато се използва някоя от диалоговите рамки, които предлага Management Studio за конфигуриране на привилегии, трябва да се има предвид, че наличието на отметка в полето *Deny* показва, че позволеният е отказан.

Пример 10 Нека MyLogin1 е член на ролята MyRole1.

-- *Предоставя привилегия на ролята:*

```
USE MyDatabase
GO
GRANT SELECT ON Sales
TO MyRole1
```

-- *Отказва привилегията на потребителя MyLogin1:*

```
DENY SELECT ON Sales TO MyLogin1
/* Отнема привилегията от MyLogin1, следователно MyLogin1 вече може
да изпълнява SELECT върху Sales, поради членството си в ролята MyRole1,
на която тя е предоставена: */
REVOKE SELECT ON Sales
TO MyLogin1
```

Пример 11 Отказване на привилегията INSERT на MyLogin1 върху таблицата Sales:

```
DENY INSERT ON Sales TO MyLogin1
```

Конфигуриране на привилегии за конструкции чрез Transact-SQL

Предоставяне на привилегии за конструкции

Общият вид на конструкцията за предоставяне на привилегии за конструкции е:

```
GRANT {ALL | statement [,... n]}  
TO security_account [,... n]
```

Пример 12

```
USE MyDatabase  
GO  
GRANT CREATE TABLE, CREATE VIEW, BACKUP DATABASE  
TO MyLogin1, MyRole2
```

Пример 13

```
USE master  
GO  
GRANT CREATE DATABASE  
TO MyLogin1
```

Отнемане на привилегии за конструкции

Общият вид на конструкцията за отнемане на привилегии за конструкции е:

```
REVOKE {ALL | statement [,... n]}  
FROM security_account [,... n]
```

Синтаксисът на REVOKE е подобен на синтаксиса на конструкцията GRANT с изключение на това, че ключовата дума TO се заменя с FROM.

Пример 14

```
USE MyDatabase  
GO  
REVOKE CREATE TABLE, CREATE VIEW, BACKUP DATABASE  
FROM MyLogin2
```

Отказване на привилегии за конструкции

Общият вид на конструкцията за отказване на привилегии за конструкции е:

```
DENY {ALL | statement [,... n]}  
TO security_account [,... n]
```

Синтаксисът на REVOKE също е подобен на синтаксиса на конструкцията GRANT.

Пример 15

```
USE MyDatabase  
GO  
DENY CREATE TABLE  
TO MyLogin1
```

Помощната информация за привилегиите за обекти или за конструкции на потребителите в текущата база от данни може да се получи чрез системната съхранена процедура sp_helprotect.

Създаване на план на сигурността

Първата стъпка от реализирането на системата за сигурност винаги е създаването на план за сигурността. Основните въпроси, които трябва да се разгледат при планиране на сигурността, са:

- **Тип на потребителите:** Ако потребителите поддържат доверени конекции, могат да се използват Windows акаунти. В противен случай е необходимо да се зададе смесен режим на автентикация и да се създадат стандартни логици.
- **Фиксирани роли за сървър:** След като се осигури достъп на потребителите до SQL Server, трябва да се реши какви административни функции да им бъдат предоставени. Ако е необходимо потребителите да извършват административни задачи, те се добавят към съответните фиксирани роли за сървър, в противен случай ако не бива да имат никакви административни пълномощия, не се добавят към никоя от тези роли.
- **Достъп до база от данни:** Определя се до коя база от данни има достъп всеки потребител, който има валиден логин.
- **Тип на достъпа:** След като потребителят има акаунт за достъп до дадена база от данни, трябва да се определи какви привилегии са му необходими в тази база от данни. Например, установява се дали всички потребители е необходимо да четат и променят данни или има подмножество от потребители, на които трябва да е позволено само четене.
- **Групови привилегии:** Обикновено е най-добре да се предоставят привилегии на роли за бази от данни и да се добавят потребители в тези роли. Във всяка система съществуват някои изключения, поради което ще е необходимо да се зададат някои привилегии директно към потребителите, особено за тези, за които трябва да бъде отказан достъп до ресурс.
- **Създаване на обект:** Определя се кой трябва да има позволение за създаване на обекти – таблици, изгледи и т.н. и се добавят в ролята *db_owner* или *db_dlladmin*.
- **Привилегии на ролята *public*:** Всички привилегии, които има ролята *public*, биват предоставяни на всички потребители, което налага тяхното ограничаване.
- **Достъп чрез *guest*:** Преценява се дали потребителите, които нямат потребителски идентификатор, трябва да имат възможност да осъществяват достъп до базата от данни чрез акаунта *guest* и евентуално да се конфигурират привилегиите им чрез *guest*.

Задачи

Задача 1. Да се напише код за създаване на роля *SalesPersons* за базата от данни и предоставяне на позволение за четене, добавяне и променяне на данни в таблицата *Sales*.

Задача 2. Да се напише код за създаване на логици *SalesPerson1*, *SalesPerson2* и *SalesPerson3*, за предоставянето им на достъп до базата от данни и за добавяне на потребителите на базата от данни към създадената вече роля *SalesPersons* за базата от данни.

Приложение 1

В задачите, включени в настоящия практикум, е използвана примерната база от данни, описана по-долу.

Нека е дадена база от данни, която да съхранява информацията, необходима на една фирма за продажба на определени продукти, за да може по-лесно и по-ефикасно да се управляват наличностите и да се следят продажбите. За всеки *продукт* се съхранява информация за наименованието му; вид (*категория*); доставчик; доставна цена; налично количество; дали продажбата на този продукт е преустановена или не; количество, което се счита за критично за съответния продукт (т.е. показва необходимост от нова доставка). За *магазините* на фирмата се съхранява информация за име; адрес; град; телефонен номер; факс; e-mail; Web адрес. За *служителите* се поддържа информация за имена; длъжност; ЕГН; магазина, в който работи; дата на постъпване; дата на напускане; телефонен номер; адрес; град; e-mail. За *доставчиците* на стоки се съхранява информация за името на фирмата-доставчик; име за контакт; длъжност на служителя, чието име е съхранено; адрес; телефонен номер; факс; e-mail; Web адрес. За *клиентите* се поддържа информация за име на фирмата-клиент; име за контакт; длъжност; адрес; телефонен номер; факс; e-mail; Web адрес; Bulstat; данъчен номер (за да е възможно издаване на фактури). За *продажбите* се поддържа информация за клиент; служител; дата на продажба; обща сума на продажба; продукти, закупени с дадена продажба; продадено количество от съответния продукт; продажна цена; отстъпка като процент от продажната цена за покупка на голямо количество от даден продукт; отстъпка като процент от общата сума за покупка на стойност над дадена сума или за клиенти с годишна сума на покупките над дадена сума.

Нека таблиците, които са необходими, са Products, Categories, Suppliers, Employees, Customers, Sales, SaleDetails. Колоните, от които се състоят, са представени в таблица 1. Колоните, съставлящи първичния ключ в съответната таблица на базата от данни, са подчертани.

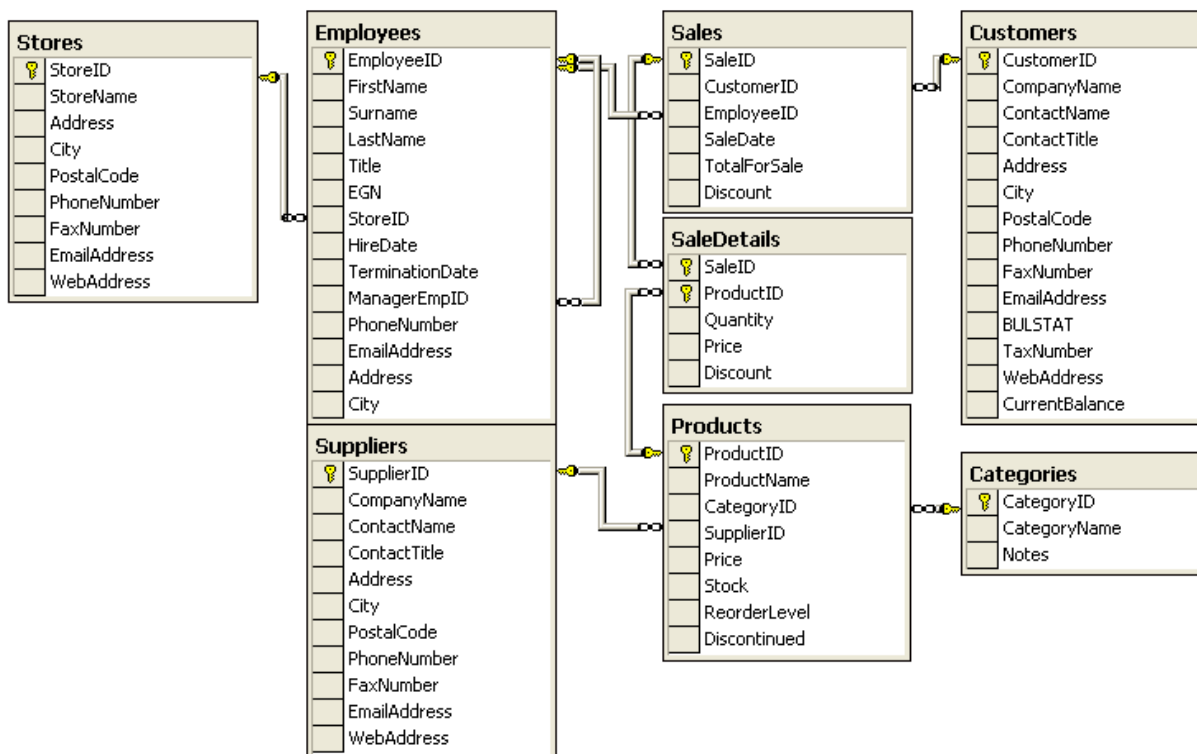
Таблица	Колони
Products	<u>ProductID</u> , ProductName, CategoryID, SupplierID, Price, Stock, ReorderLevel, Discontinued
Categories	CategoryID, CategoryName, Notes
Suppliers	<u>SupplierID</u> , CompanyName, ContactName, ContactTitle, Address, City, PostalCode, PhoneNumber, FaxNumber, EmailAddress, WebAddress
Employees	<u>EmployeeID</u> , FirstName, Surname, LastName, Title, EGN, StoreID, HireDate, TerminationDate, ManagerEmpID, PhoneNumber, Address, City, EmailAddress
Stores	<u>StoreID</u> , StoreName, Address, City, PostalCode, PhoneNumber, FaxNumber, EmailAddress, WebAddress
Customers	<u>CustomerID</u> , CompanyName, ContactName, ContactTitle, Address, City, PostalCode, PhoneNumber, FaxNumber, EmailAddress, WebAddress, Bulstat, TaxNumber, CurrentBalance
Sales	<u>SaleID</u> , CustomerID, EmployeeID, SaleDate, TotalForSale, Discount
SaleDetails	<u>SaleID</u> , <u>ProductID</u> , Quantity, Price, Discount

Таблица 1 Колоните, съдържащи се в съответните таблици

Забележка Приемаме, че един продукт се доставя от точно един доставчик, т.е. релационната връзка между таблиците *Suppliers* и *Products* е “едно към много”. Освен това в базата от данни не е осигурена възможност за подробно проследяване на отделните доставки на различни продукти. Таблиците в разглежданата база от данни са използвани в примерите на настоящия практикум, като целта е в тях да се наблегне на основния смисъл, което изисква да се избегне излишното усложняване на модела на базата от данни.

Релационните връзки между таблиците в базата от данни са показани на фигура

1.



Фиг. 1 Обект-релационна връзка диаграма, представяща връзките между таблиците

Литература

1. Вискас Дж., *Всичко за Access 2000*, СофтПрес, София, 2000
2. Гарсиа-Молина Г., Дж. Ульман, Дж. Уидом, *Системы баз данных*, ИК “Вильямс”, 2002
3. Георгиев Е., *Научете сами SQL, Експрес дизайн*, София, 1998
4. Грубер М., *SQL – Професионално издание*, I и II том, СофтПрес, София, 2001
5. Дебета П., *Въведение в Microsoft SQL Server 2005 за разработчици*, СофтПрес, София, 2005
6. Ернандес М., *Проектиране на бази от данни*, СофтПрес, София, 2004
7. Крѐнке Д., *Теория и практика построения баз данных*, Питер, 2003
8. Рей Д., Е. Рей, *Access 2000 за Windows*, ИнфоДАР, София, 2000
9. Станек У., *Microsoft SQL Server 2000 Наръчник на администратора*, СофтПрес, София, 2001
10. Сукъп Р., К. Дилейни, *Microsoft SQL Server 7.0 Поглед отвътре – I и II том*, СофтПрес, София, 2000
11. Форт С., Т. Хоу, Дж. Ралстън, *Microsoft Access 2000 /Професионално – I и II том*, ИнфоДАР, София, 1999
12. Хулет Ф., *SQL Ръководство на програмиста*, СофтПрес, София, 2001
13. Шапиро Дж., *SQL Server 2000 Пълно ръководство за администриране*, ИнфоДАР, София, 2001
14. Шапиро Дж., *SQL Server 2000 Пълно ръководство за програмиране*, ИнфоДАР, София, 2001
15. Microsoft Corporation, *MCSE Training: Microsoft SQL Server 2000 – Проектиране и реализация на бази данни*, I и II том, СофтПрес, София, 2001
16. Bieniek D., Dyess R., Hotek M., Loria J., Machanic A., Soto A., Wiernik A., *Microsoft SQL Server 2005 Implementation and Maintenance – Training Kit*, Microsoft Press, 2006

Адреси в Интернет

17. <http://free.techno-link.com/database>
18. <http://www.georgehernandez.com/xDatabases/MD/MDX.htm>
19. <http://www.databasejournal.com>
20. <http://www.datawarehouse.com>
21. <http://www.datawarehousing.com>
22. <http://www.datawarehousingonline.com>
23. <http://www.dmreview.com>
24. <http://www.dw-institute.com>
25. <http://www.dwinfolcenter.org>
26. <http://www.informit.com>
27. <http://www.microsoft.com/technet>
28. <http://www.microsoft.com/sql>
29. <http://www.olapcouncil.org>
30. <http://www.olapreport.com>
31. <http://www.olapinfo.de>
32. <http://www.patternwarehouse.com/bridge.htm>
33. <http://www.research.microsoft.com/datamine>
34. <http://www.searchdatabase.com>

35. <http://www.sqlmag.com>
36. <http://www.sqlwarehouse.com>
37. <http://www.sql-server-performance.com>
38. <http://www.w3.org/XML>
39. <http://www.w3schools.com/xml>

ПРАКТИКУМ ПО БАЗИ ОТ ДАННИ

ЧАСТ II

Автор:

гл. ас. д-р Цветанка Любомирова Георгиева-Трифенова

Българска
Първо издание

Рецензент:

ст.н.с. I ст. д.м.н Петър Любомиров Станчев

Цветанка Георгиева-Трифенова

ПРАКТИКУМ ПО БАЗИ ОТ ДАННИ

ЧАСТ II

Основни теми:

- Програмиране с Transact-SQL
- Създаване на:
 - ❖ изгледи
 - ❖ съхранени процедури
 - ❖ дефинирани от потребителя функции
 - ❖ тригери
- Работа с:
 - ❖ XML данни
 - ❖ транзакции
 - ❖ курсори

Включените в темите примери са достъпни от:

- <http://practicum.host22.com>