



Ľuboslav Lacko

Prakticky úvod do novíniek SQL Serveru 2005 v oblasti relačných databáz

Microsoft®

Vaše možnosti. Naše inspirace.

Prakticky úvod do noviniek SQL Serveru 2005 v oblasti relačných databáz

Obsah:

Kapitola 1: Microsoft SQL Server 2005 – predstavenie	2
Kapitola 2: SQL Server Management Studio	9
Kapitola 3: Novinky pre vysokú dostupnosť databáz	20
Kapitola 4: Bezpečnosť	26
Kapitola 5: Nové črty v jazyku SQL a T-SQL	32
Kapitola 6: Ukladanie a práca s údajmi vo formáte XML	58
Kapitola 7: Rozširovanie funkčnosti SQL serveru v .NET jazykoch	78
Kapitola 8: Nové rysy v ADO.NET 2.0	93
Kapitola 9: Webové služby cez HTTP Endpoint	98
Příloha 1.....	104

Kapitola 1: Microsoft SQL Server 2005 – predstavenie

Táto publikácia venovaná novému databázovému serveru MS SQL Server 2005 bude zameraná viac prakticky, to znamená, že jednotlivé témy budú preberané na cvičných príkladoch. Preto ako prvý úkon odporúčame inštaláciu.

Inštalácia

SQL Server 2005 okrem svojej primárnej úlohy databázového servera poskytuje niekoľko druhov služieb, preto v prvom inštaláčnom dialógu je potrebné špecifikovať o ktoré komponenty máme záujem.

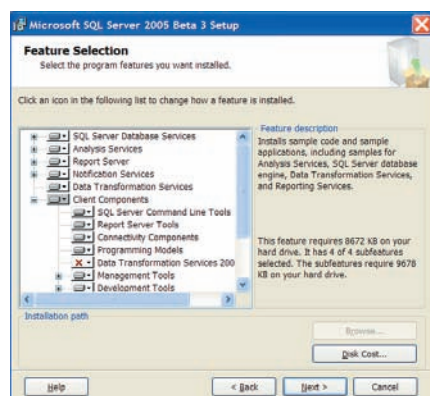
V ponuke sú komponenty:

- databázové služby
- analytické služby
- reportovacie služby
- notifikačné služby
- služby pre transformáciu údajov (DTS – Data Transformation Services)
- administrátorské a vývojárske nástroje, pomocné komponenty, dokumentácia...



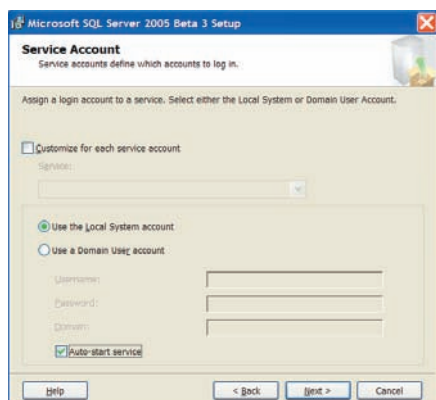
Výber komponentov pre inštaláciu

Pre študijné účely a oboznamovanie sa s SQL Serverom 2005 odporúčame doplniť zoznam komponentov o cvičné databázy AdventureWorks a AdventureWorksDW a cvičné príklady. Do dialógu spresnenia nainštalovaných komponentov sa prepne tlačidlom Advanced. Avizované komponenty nainštalujeme ich označením v zložke Client Components



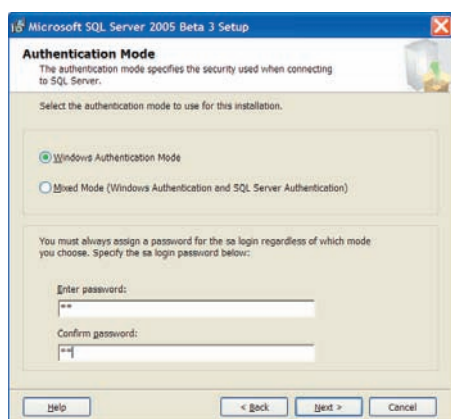
Spresnenie výberu komponentov pre inštaláciu

Nasleduje voľba lokálneho alebo doménového účtu. Vo verzii 2005 môžeme nastaviť tieto parametre pre každú požadovanú službu zvlášť. Zaškrtnutím checkboxu Auto-start sevice nadefinujeme automatický štart služieb pri spustení operačného systému



Vytvorenie prístupového účtu

V okne *Authentication Mode* môžeme zvoliť možnosť *Windows Authentication Mode* alebo *Mixed Mode*. V „Mixed“ móde sa overí buď autentifikácia operačného systému Windows, alebo pomocou prihlasovacieho hesla SQL Serveru. V tejto fáze zadáme heslo, ktoré budeme používať.



Definovanie autentifikačného módu a hesla administrátora

Počas inštalácie reportovacích služieb (ak sme ich inštaláciu požadovali), budú vytvorené dva virtuálne adresáre. Jeden pre reportovací server s URL adresou

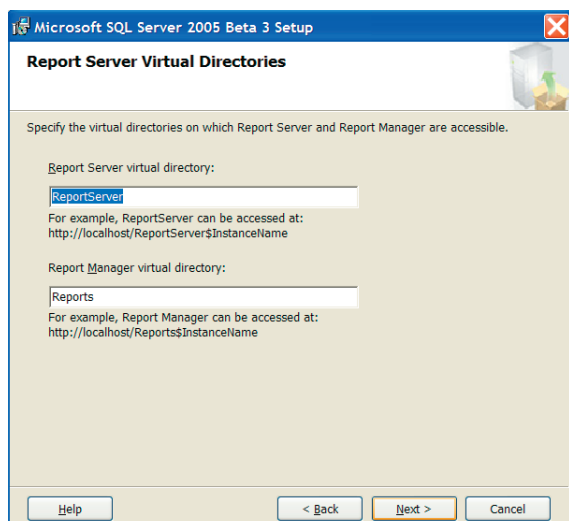
`http://<servername>/ReportServer.`

a jeden pre prístup cez webový server s URL adresou

`http://WebServerName/Reports.`

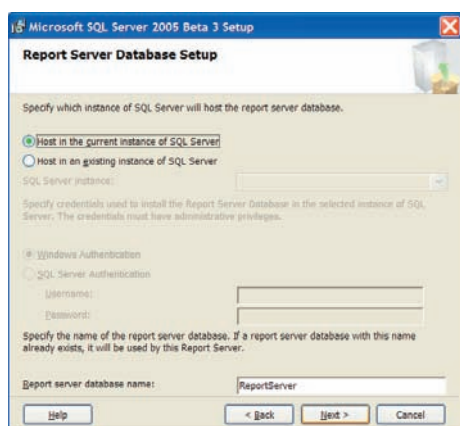
Po nainštalovaní reportovacích služieb si pomocou nástroja IIS Manager (je v zložke administrátorských nástrojov operačného systému dostupnej cez menu START – Control Panel), môžeme overiť jednak vytvorenie týchto virtuálnych adresárov a taktiež môžeme zistiť, ktoré fyzické adresáre zodpovedajú spomínaným dvom virtuálnym adresárom. Implicitne sú to adresáre:

reports	C:\Program Files\Microsoft SQL Server\MSSQL\Reporting Services\app
reportserver	C:\Program Files\Microsoft SQL Server\MSSQL\Reporting Services\ReportServer



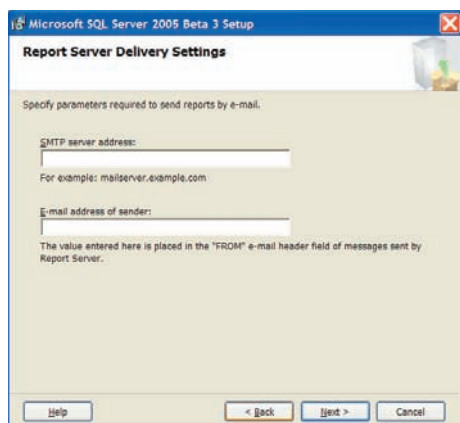
Zadanie názvu virtuálneho adresára pre reportovacie služby

Pre fungovanie reportovacích služieb je potrebná samostatná databáza pre ukladanie pracovných a konfiguračných údajov a metadát reportov. Táto databáza môže byť vytvorená pod databázovým serverom, ktorý práve inštalujeme, alebo môžeme požadovať jej vytvorenie na inej inštancii SQL Servera 2005

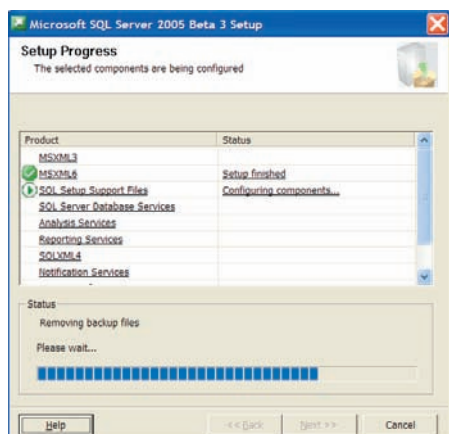


Nastavenie reportovacieho servera

Reportovacie služby umožňujú zasielanie reportov elektronickou poštou. Pre tento účel je potrebné nastaviť adresu SMTP servera a e-mailovú adresu odosielateľa.

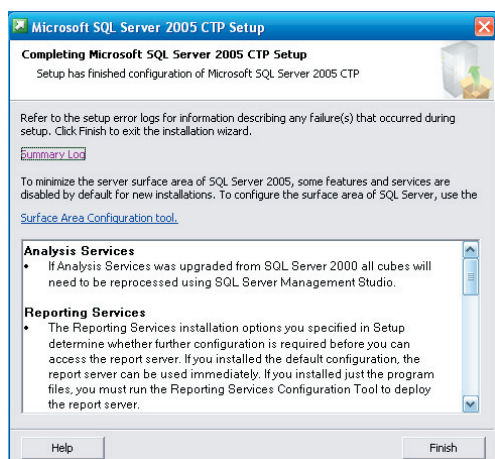


Definovanie parametrov pre zasielanie reportov mailom



Priebeh procesu inštalácie

Po ukončení inštalácie je k dispozícii protokol o jej priebehu, kde je zoznam nainštalovaných komponentov, prípadne sú vypísané problémy sprevádzajúce inštaláciu.



Záverečný dialóg inštalácie

SQL Server 2005 Surface Area Configuration

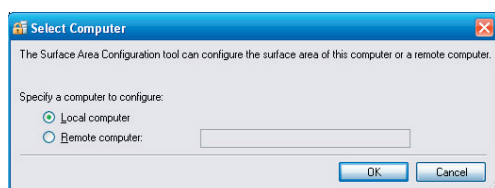
Pri inštalácii serverového softvéru sa zaručene nevyhneme polemike, medzi množstvom nainštalovaných služieb a vlastností a bezpečnosťou. Preto u moderných serverov môžeme vypozerovať dva druhy politik

- pri implicitnej inštalácii prevažnú väčšinu služieb zakázať, s predpokladom, že skúsený administrátor pri ich doinštalovaní alebo povoľovaní ich aj náležite zabezpečí
- zoznam požadovaných komponentov a služieb sa určí počas zadávania parametrov inštalácie, alebo v priebehu samotného inštaláčného procesu

V druhom prípade nainštalujeme služby, ktoré plánujeme využívať, no v prípade ak ich neplánujeme využívať ihneď, je výhodné ich zakázať, aby sa postupom času nestali potenciálnym slabým miestom pre prienik do systému. Po nainštalovaní SQL Servera 2005 máme možnosť v poslednom dialógu inštalácie spustiť nástroj Surface Area Configuration a pomocou neho zakázať všetky služby, ktoré nemienime ihneď po inštalácii využívať

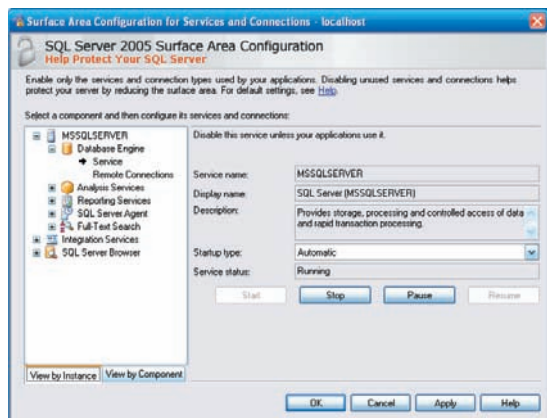


Úvodný dialóg SQL Server 2005 Surface Area Configuration



Dialóg Surface Area Configuration pre výber lokálneho, alebo vzdialeného počítača

V úvodných dialógoch vyberieme, či chceme konfigurovať služby a pripojenia, alebo rozšírené vlastnosti. Program Surface Area Configuration môžeme aplikovať na lokálny, prípadne aj na vzdialený počítač

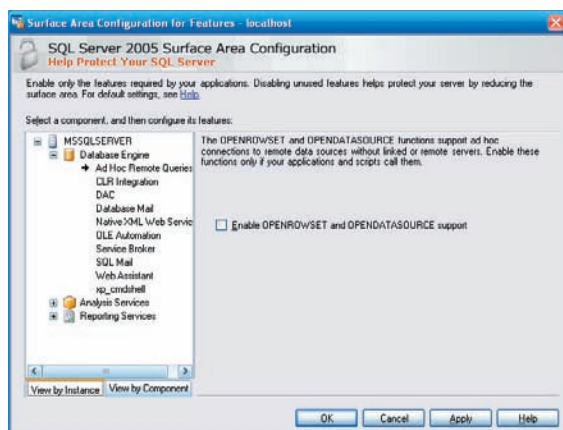


Dialóg Surface Area Configuration pre správu základných služieb a pripojení

Môžeme povoliť, prípadne zakázať

- databázové služby
- Analytické služby
- Reportovacie služby
- SQL Server agenta
- Full textové vyhľadávanie
- Integrované služby
- SQL Server Browser

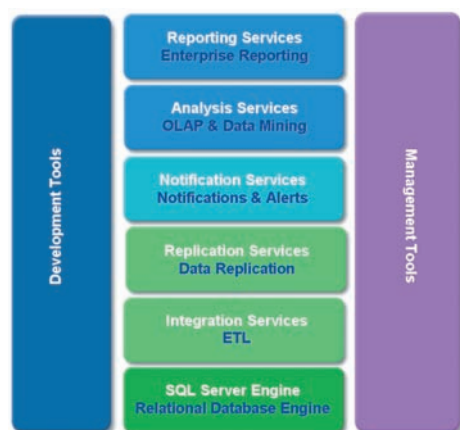
Širokú paletu variant nastavenia umožňuje aj dialóg umožňujúci konfigurovať rozšírené vlastnosti, napríklad CLR, SQL Mail, natívne XML webové služby, Service Broker...



Dialóg Surface Area Configuration pre správu prídavných služieb, vlastností a aplikácií

Architektúra

Globálnu blokovú schému SQL Servera 2005 môžeme znázorniť pomocou schémy:



Bloková schéma SQL Servera 2005

Základným stavebným kameňom architektúry je relačný databázový stroj. Jeho služby majú na starosti správu údajov, ich vkladanie a vymazávanie, vyhľadávanie ...

- **reportovacie služby** (generovanie reportov a zostáv na základe údajov z databáz)
- **analytické služby** (OLAP analýzy, datamining)
- **notifikačné služby**
- **replikačné služby**
- **integračné služby**

K týmto šiestim základným blokom môžeme priradiť bloky pre administráciu a vývoj aplikácií. Databázový server s naznačenou infraštruktúrou nájde uplatnenie nielen pre budovanie rozsiahlych relačných databáz, ale aj analytických databáz a dátových skladov (datawarehouse)

Databázové a analytické služby boli integrované aj do predchádzajúcej verzie SQL Serveru 2000, presnejšie, boli tam integrované od začiatku. Reportovacie služby pre verziu 2000, boli uvádzané na trh ako doplnkový balík. Preto aj rozdiely medzi týmto balíkom a reportovacími službami integrovanými do Yukonu nie sú príliš veľké.

Poznámka autora – otestujte SQL Server 2005 na virtuálnom počítači

Z predchádzajúcej state je zrejmé že databázový server je z hľadiska fungovania na počítači pomerne komplexná záležitosť. Nainštalovanie malého softvérového balíka typu grafický editor, hra a podobne vrátane následného odinštalovania nie je vďaka moderným operačným systémom spravidla problém, no inštalácia databázového servera navyše v kombinácii s vývojovým prostredím už konfiguráciou počítača riadne zamáva. Navyše napríklad vývojár, ktorý musí udržiavať staré produkty nemôže prejsť na novú verziu databázového servera a vývojového prostredia kedy sa mu zachce. Vyhradiť pre testovanie nových produktov samostatný hardvér je síce excelentné riešenie, no pomerne nákladné.

Riešením nielen naznačených problémov, ale aj mnohých iných sú virtuálne počítače. Výhody virtuálnych počítačov sú natoľko zreteľné, že nie je ťažké predikovať im skvelú budúcnosť. Pre zaujímavosť, počas písania tejto publikácie boli vydané tri nové buildy beta verzie SQL Servera, a jedinou možnosťou ako to reálne zvládnuť bolo použitie virtuálneho počítača pod Microsoft Virtual PC 2004.

Kapitola 2: SQL Server Management Studio

Databázový server je koncipovaný ako služba na pozadí. Okrem toho, že túto službu budú využívať aplikácie, je potrebné s databázovým serverom pracovať aj na administrátorskej úrovni. Podobne aj vývojári aplikácií potrebujú nejaký nástroj pre ladenie príkazov jazyka SQL, ktoré potom „vstavajú“ do aplikácie. Mnohé, hlavne podnikové aplikácie pri svojom spustení predpokladajú, že potrebné databázové štruktúry už budú vytvorené v procese inštalácie a konfigurácie.

Taktiež aj začiatočníkov v odbore databáz, ktorí si našťudovali základy teórie databáz a jazyka SQL a budú si chcieť vyskúšať prvý jednoduchý príklad bude určite zaujímať, kam vlastne tie SQL príkazy zadávať a kde a v akej forme získajú výsledok. To všetko sú úlohy pre klientskú konzolovú aplikáciu, pomocou ktorej zadávame databázovému serveru SQL príkazy a v jej okne taktiež vidíme výstupy, ktoré databázový server vygeneruje ako odozvu na naše príkazy. Okrem klientskej konzolovej aplikácie občas potrebujeme ešte nástroj pre správu databázy. Prostredníctvom tejto aplikácie môžeme vytvárať nové databázy, vytvárať a spravovať účty jednotlivých používateľov, pridelovať a rušiť ich oprávnenia pre prácu s jednotlivými objektmi databázy. Pomocou nástroja pre správu databázy je možné nastaviť aj stratégiu údržby a zálohovania údajov v databáze a podobne. Pre prácu s databázovým serverom Microsoft SQL Server 2005 je na spomínané účely k dispozícii SQL Server Management Studio.

SQL Server Management Studio je komplexné integrované prostredie pre správu databázového servera SQL Server 2005. V prvej beta verzii malo toto prostredie pracovný názov SQL Server Workbench. Ak by sme sa snažili určiť pozíciu tohto nástroja voči predchádzajúcej verzii SQL Servera 2000, zjednodušene povedané, mohlo by ísť o zlúčenie nástrojov Enterprise Manager, Query Analyser a do určitej miery aj nástroja Analysis Manager. Management Studio je vybudované na základe unifikovaného vývojového prostredia Microsoft Development Environment, ktoré vychádza z vývojového prostredia Visual Studio 2005. Na tomto základe sú postavené aj iné nástroje, napríklad Business Intelligence Development Studio pre prácu s integračnými službami, OLAP kockami a data miningovými modelmi. Je to významný krok k unifikácii v novej rade vývojárskych produktov a technológií Microsoftu.

Nástroj sa spúšťa štandardným spôsobom z menu operačného systému Windows *Start | All Programs | Microsoft SQL Server 2005 | SQL Server Management Studio*. Pri štandardnej inštalácii SQL Servera 2005 v operačnom systéme Windows 2003 je tento nástroj uložený v adresári `C:\Program Files\Microsoft SQL Server\90\Tools\Binn\VSShell\Common7\IDE\` ako `SqIWb.exe`.

Po spustení nástroja sa zobrazí dialóg pre pripojenie sa k databázovému serveru. Ak robíme prvé pokusy na lokálnom alebo virtuálnom počítači, môžeme nastaviť názov serveru ako `localhost` a ako metódu prihlásenia použiť `Windows Authentication`.



Prihlasovací dialóg SQL Servera 2005

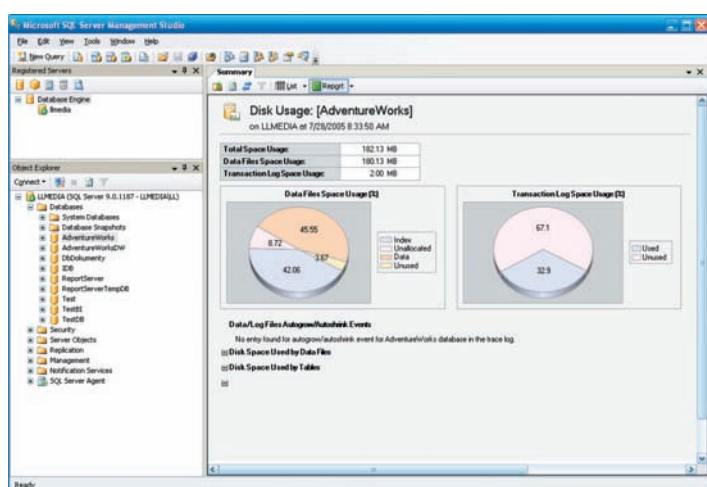
Po zatlačení tlačidla `Options` sa prístupný rozšírený mód prihlasovacieho dialógu so záložkami

- Login
- Connection Properties
- Registered Servers



Záložky prihlasovacieho dialógu v rozšírenom móde

V záložke Connection Properties môžeme nastaviť typ sieťového protokolu a časový limit pre pripojenie a časový limit pre vykonanie príkazu. V záložke Connection Properties si okrem dostupných serverov všimnite možnosť preniesť registračné informácie na iný server



SQL Server Management Studio umožňuje okrem funkcie konzolovej aplikácie aj sledovať štatistiky o prevádzke databázového servera

Pracovná plocha SQL Management Studia je rozdelená na niekoľko častí.

Registered Servers

Okno obsahuje zoznam serverov, na ktoré sa možno pomocou Management Studia pripojiť.

Object Explorer

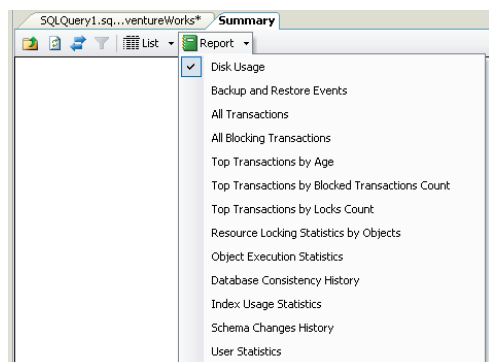
Toto okno je veľmi podobné ľavému panelu Enterprise Managera u staršej verzie SQL Servera 2000 a poskytuje grafický, prehľadný, hierarchický pohľad na SQL komponenty až na úroveň stĺpcov a indexov. Je potrebné si uvedomiť, že objekty sú vo verzii SQL Serveri 2005 asociované so schémou a nie s vlastníkom ako u predchádzajúcej verzie SQL 2000.

Hlavné okno v strede pracovnej plochy

Po spustení aplikácie je možné hlavné okno pracovnej plochy prepnúť do režimov

- List
- Report

V záložke Report môžeme zvoliť, či chceme sledovať štatistiky prístupov na disk, transakcií, použitie indexov a podobne



Možnosti reportov ohľadne štatistiky a monitorovania výkonnosti databázového servera

V záložke List sú všeobecne zobrazené údaje o objekte, ktorý je vybraný v okne Object Explorera, prípadne v móde Query Editor, ktorý nahrádza Query Analyzer z predchádzajúcej verzie je v tomto priestore alokované okno pre zadávanie dopytov a zobrazovanie výsledkov skriptov v jazykoch a skriptovacích systémoch Transact-SQL, XMLA, MDX, DMX alebo XQuery. Môžeme taktiež vytvárať príkazy určené pre Mobile SQL Server.

Pravé zvislé okno je možné prepínať záložkami do módu **Solution Explorer** alebo **Template Explorer**. Pomocou Solution Explorera je možné zoskupovať skripty v databázových jazykoch do projektov. Správu verzií takýchto projektov môžeme vykonávať cez Visual Source Safe

Dopyty v prostredí SQL Server Management Studio

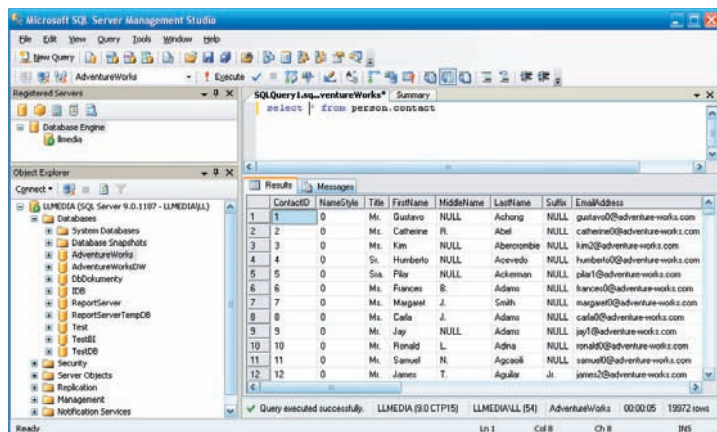
Po aktivovaní tlačidla New Query aktivujeme dopytovací mód Management Studia. Tento mód je nástupcom nástroja Query Analyzer zo staršej verzie SQL Server 2000. Okrem klasických SQL a T-SQL dopytov je možné prepnúť Management Studio do viacerých režimov:

- MDX Query
- DMX Query
- XMLA Query
- SQL Mobile Query

podľa toho či pracujeme s relačnými, multidimenzionálnymi, alebo data miningovými databázami

Po prepnutí do režimu SQLCMD je možné písať v tomto prostredí skripty pre SQLCMD a využívať v nich aj príkazov tejto konzoly. Uvedieme ich zoznam:

```
[:]go [count]
!! <command>
:exit (statement)
:Quit
:r <filename>
:setvar <var> <value>
:connect server[\instance] [-l login_timeout] [-U user [-P password]]
:on error [ignore|exit]
:error <filename>|stderr|stdout
:out <filename>|stderr|stdout
```



SQL Server Management Studio v režime dopytovania

Skôr než začneme zadávať SQL dopyty do niektorej databázy, potrebujeme preskúmať, či je Management Studio prepnuté na správnu databázu. Prepnutie sa realizuje buď pomocou combo boxu na hornom toolbare aplikácie, alebo príkazom

```
use AdventureWorks
```

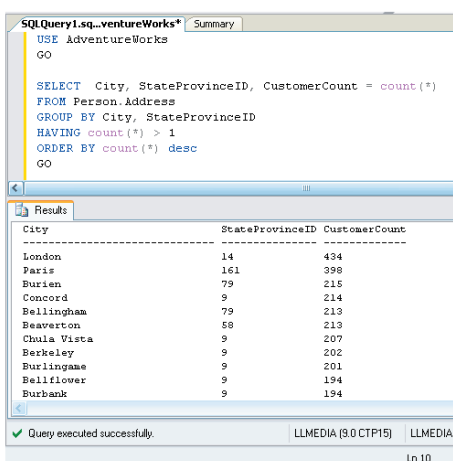
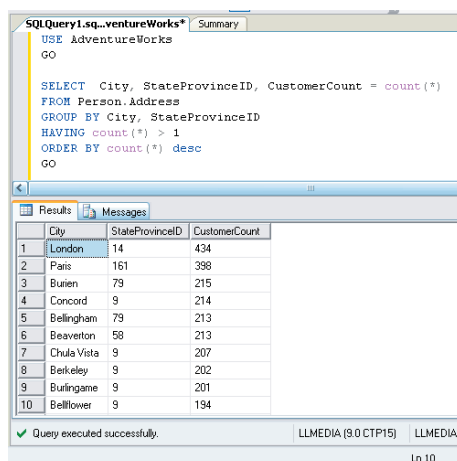
Ak máme správne prepnuté na databázu Adventure Works, môžeme položiť jednoduchý dopyt, napríklad

```
select * from Person.Contact;
```

Príkazy môžeme zlučovať aj do dávok

```
USE AdventureWorks
GO
```

```
SELECT City, StateProvinceID, CustomerCount = count(*)
FROM Person.Address
GROUP BY City, StateProvinceID
HAVING count(*) > 1
ORDER BY count(*) desc
GO
```



SQL dopyt a výsledky dopytovania. Výsledky dopytovania je možné zobrazit vo forme tabuľky alebo textového výpisu, prípadne uložit do súboru.

V okne pre spúšťanie T-SQL dopytov môžeme testovať nielen SQL príkazy ale napríklad aj spúšťať uložené procedúry. Pre tento účel slúži príkaz EXEC. Ukážeme príklad spustenia systémovej uloženej procedúry SP_WHO, ktorá vypíše informácie o používateľoch a procesoch

```
USE AdventureWorks
GO
EXEC sp_who
GO
```

Uložená procedúra vypíše požadované údaje v okne Management Studia (uvádzame v skrátenej forme)

spid	status	loginame	hostname	dbname	cmd
1	background	sa		NULL	RESOURCE MONITOR
2	background	sa		NULL	LAZY WRITER
3	suspended	sa		NULL	LOG WRITER
4	background	sa		NULL	LOCK MONITOR
...					
51	sleeping	LLMEDIA\LL	LLMEDIA	master	AWAITING COMMAND
52	sleeping	NT AUTHORITY\SYSTEM	LLMEDIA	ReportServer	AWAITING COMMAND
53	sleeping	LLMEDIA\LL	LLMEDIA	master	AWAITING COMMAND
54	sleeping	NT AUTHORITY\SYSTEM	LLMEDIA	ReportServer	AWAITING COMMAND
55	sleeping	NT AUTHORITY\SYSTEM	LLMEDIA	msdb	AWAITING COMMAND
56	runnable	LLMEDIA\LL	LLMEDIA	AdventureWorks	SELECT

Uloženú procedúru môžeme volať aj s parametrami. Syntax parametrov pre procedúru SP_WHO

```
sp_who [[@login_name =] ,login']
```

V konzolovej aplikácii voláme procedúru s parametrom v tvare

```
EXEC sp_who ,LLMEDIA\LL`;
```

Tentoraz sa vypíšu len údaje pre konkrétneho používateľa

spid	status	loginame	hostname	dbname	cmd
51	sleeping	LLMEDIA\LL	LLMEDIA	master	AWAITING COMMAND
53	sleeping	LLMEDIA\LL	LLMEDIA	master	AWAITING COMMAND
56	runnable	LLMEDIA\LL	LLMEDIA	AdventureWorks	SELECT

Výpis parametrov a vlastností

Konzolová aplikácia SQL Server Management Studio umožňuje aj výpis parametrov a to na rôznej úrovni. Pomocou klauzuly SERVERPROPERTY môžeme zisťovať niektoré parametre databázového servera, napríklad

```
SELECT SERVERPROPERTY(,ServerName`)
SELECT SERVERPROPERTY(,Edition`)
SELECT SERVERPROPERTY(,ProductVersion`)
SELECT SERVERPROPERTY(,ProductLevel`)
```

Server nám vráti hodnoty požadovaných parametrov v tvare

```
LLMEDIA
Beta Edition
9.00.1187.07
CTP15
```

Pomocou klauzuly DATABASEPROPERTY môžeme zisťovať parametre konkrétnej databázy, napríklad

```
SELECT DATABASEPROPERTYEX(,AdventureWorks`, ,Status`)
```

```

SELECT DATABASEPROPERTYEX(AdventureWorks, Recovery)
SELECT DATABASEPROPERTYEX(AdventureWorks, Collation)
SELECT DATABASEPROPERTYEX(AdventureWorks, Updateability)
SELECT DATABASEPROPERTYEX(AdventureWorks, UserAccess)
SELECT DATABASEPROPERTYEX(AdventureWorks, IsAutoCreateStatistics)
SELECT DATABASEPROPERTYEX(AdventureWorks, IsAutoShrink)

```

Server nám vráti hodnoty v tvare

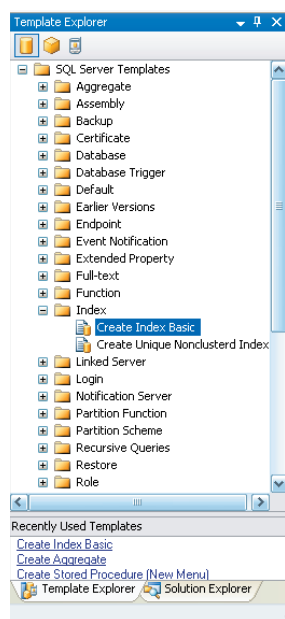
```

ONLINE
SIMPLE
SQL_Latin1_General_CP1_CI_AS
READ_WRITE
MULTI_USER
1
0

```

Využitie šablón skriptov

Ak prepneme pravé okno Management Studia do módu Template Explorer, zobrazí sa zoznam šablón, ktoré sú predpripravené pre najpoužívanejšie druhy skriptov, napríklad pre vytváranie a rušenie objektov a podobne. Šablónu vložíme do skriptu pri jeho vytváraní v menu New File.



Vytvorenie skriptu pomocou šablóny

Napríklad šablóna pre vytvorenie indexu je v tvare

```

-- =====
-- Create index basic template
-- =====
USE <database_name, sysname, AdventureWorks>
GO

CREATE INDEX <index_name, sysname, ind_test>
ON <schema_name, sysname, Person>.<table_name, sysname, Address>
(

```

```

        <column_name1, sysname, PostalCode>
    )
GO

```

Skôr než tento skript spustíme, nahradíme buď ručne, alebo u rozsiahlejších skriptov s väčším opakovaním výrazov pomocou menu Edit | Find and Replace šablóny reťazcov názvov

```
<index_name, sysname, ind_test>
```

názvom, prípadne názvom a parametrami pre príslušný konkrétny objekt. Preddefinované šablóny je možné meniť, prípadne pridávať šablóny nové.

SQLCMD

Pre správu databázového serveru a ladenie databázovej časti aplikácií niekedy využijeme aj jednoduchú interaktívnu textovú konzolovú aplikáciu SQLCMD. Slúži pre zadávanie príkazov jazyka SQL databázovému serveru a okne tej istej aplikácie taktiež vidíme výstupy, ktoré databázový server vygeneruje ako odozvu na naše príkazy, teda napríklad výpis obsahu databázových tabuliek, potvrdenie vykonania našich príkazov, chybové hlásenia a podobne. Konzola SQLCMD nahrádza aplikácie osql a isql známe z predchádzajúcej verzie. Pre pripojenie sa k databázovému serveru využíva SQL Native Client. Ak chceme vykonávať niektoré administrátorské úkony vyžadujúce prioritu zdrojov, je užitočné pripojiť sa cez dedikované spojenie administrátora s parametrom „SQLCMD -A“. Konzola umožňuje zadávať ako parametre aj názvy súborov obsahujúce SQL kód

Nápovedu, obsahujúcu aj parametre pre spustenie aplikácie získame použitím parametra sqlcmd -?

```

C:\>sqlcmd -?
Microsoft (R) SQL Server Command Line Tool
Version 9.00.1187.07 NT INTEL X86
Copyright (C) 2004 Microsoft Corporation. All rights reserved.

usage: Sqlcmd          [-U login id]          [-P password]
      [-S server]      [-H hostname]      [-E trusted connection]
      [-d use database name] [-l login timeout] [-t query timeout]
      [-h headers]     [-s colseparator]   [-w screen width]
      [-a packetsize] [-e echo input]     [-I Enable Quoted Identifiers]
      [-c cmdend]      [-L[c] list servers[clean output]]
      [-q „cmdline query“] [-Q „cmdline query“ and exit]
      [-m errorlevel]  [-V severitylevel]   [-W remove trailing spaces]
      [-u unicode output] [-r[0|1] msgs to stderr]
      [-i inputfile]   [-o outputfile]       [-z new password]
      [-f <codepage> | i:<codepage>[,o:<codepage>]] [-Z new password and exit]
      [-k[1|2] remove[replace] control characters]
      [-y variable length type display width]
      [-Y fixed length type display width]
      [-p[1] print statistics[colon format]]
      [-R use client regional setting]
      [-b On error batch abort]
      [-v var = „value“...] [-A dedicated admin connection]
      [-X[1] disable commands, startup script, environment variables [and exit]]
      [-x disable variable substitution]
      [-? show syntax summary]

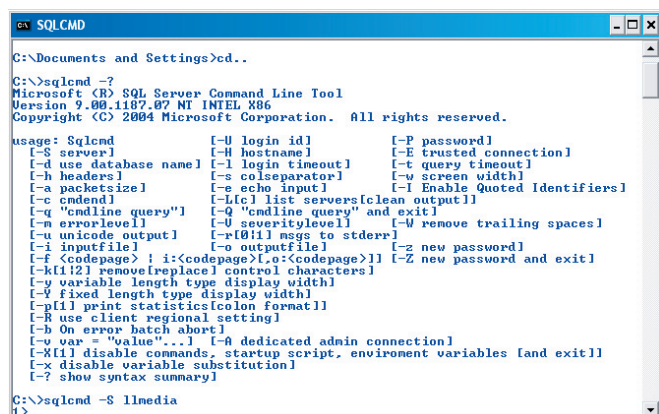
```

Ak sa chceme pripojiť ku konkrétnej inštancii databázového servera, použijeme príkaz

```
sqlcmd -S Server
```


v našom prípade bol názov počítača llmedia takže konkrétny príkaz bol v tvare

```
sqlcmd -S llmedia
```



```

C:\Documents and Settings>cd..
C:\>sqlcmd -?
Microsoft (R) SQL Server Command Line Tool
Version 9.00.1187.07 NT INTEL X86
Copyright (C) 2004 Microsoft Corporation. All rights reserved.

usage: sqlcmd [-U login id] [-P password]
[-S server] [-H hostname] [-E trusted connection]
[-d use database name] [-l login timeout] [-t query timeout]
[-h headers] [-s colseparator] [-w screen width]
[-a packetsize] [-e echo input] [-I Enable Quoted Identifiers]
[-c cndend] [-L[c] list servers[clean output]]
[-q "cmdline query"] [-Q "cmdline query" and exit]
[-n errorlevel] [-O severitylevel] [-W remove trailing spaces]
[-u unicode output] [-r[0|1] msgs to stderr]
[-i inputfile] [-o outputfile] [-Z new password]
[-f <codepage> | i:<codepage>[o:<codepage>]] [-Z new password and exit]
[-k[1|2] remove[replace] control characters]
[-y variable length type display width]
[-Y fixed length type display width]
[-p[1] print statistics[colon format]]
[-R use client regional setting]
[-b On error batch abort]
[-w var = "value"...] [-A dedicated admin connection]
[-M] disable commands, startup script, environment variables [and exit]]
[-x disable variable substitution]
[-? show syntax summary]

C:\>sqlcmd -S llmedia
1>

```

Textová klientská konzolová aplikácie SQLCMD

Klientskú konzolovú aplikáciu SQLCMD90 je možné spustiť aj pomocou menu RUN operačného systému Windows

„C:\Program Files\Microsoft SQL Server\90\Tools\binn\SQLCMD90” -S „llmedia“

Ak máme na vývojárskom počítači nainštalované aj iné SQL servery, je dôležité aby sme boli pripojení k správnej inštancii. Ak sme pripojení k inému serveru ľahko môžu nastať na prvý pohľad nevysvetliteľné situácie, kedy pomocou príkazov z konzoly (v inej databáze pod správou iného servera) vytvoríme a naplníme cvičné databázové tabuľky a z aplikácie ich potom nevidíme. Pre zistenie či sme pripojení k správnej inštancii SQL Servera použijeme príkaz

```
select @@version
go
```

Príkazy potvrdzujeme pomocou GO. V okne klientskej konzolovej aplikácie bude vypísaná informácia typu

```

Microsoft SQL Server 2005 - 9.00.1187.07 (Intel X86)
    May 24 2005 18:22:46
    Copyright (c) 1988-2005 Microsoft Corporation
    Beta Edition on Windows NT 5.1 (Build 2600: Service Pack 2)

```

Ak nie je určené inak konzola sa pripojí k databáze MASTER. Do inej databázy sa prepne príkazom:

```
USE nazov_databazy
Go
```

Pri práci s konzolou môžeme využívať aj rôzne premenné, napríklad

```

:setvar DBName „AdventureWorks“
:setvar TName „HumanResources.Employee“
:setvar ColumnName „EmployeeID“
Use $(DBName)
GO

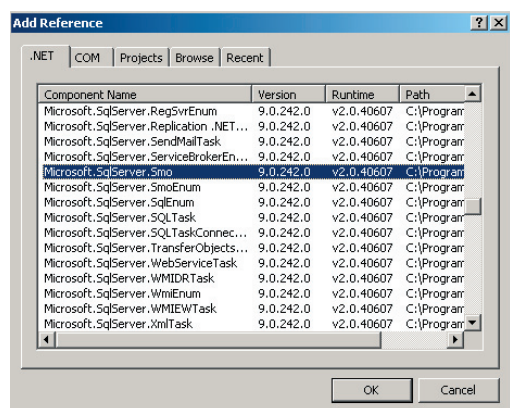
Changed database context to ,AdventureWorks`.

Select $(ColumnName) from $(TName)
GO

```

Server Management Object (SMO)

Pre pripojenie sa k SQL Serveru môžeme použiť aj Server Management Object. Vo vývojovom prostredí Visual Studio 2005 vytvoríme nový projekt, s názvom napríklad ProjSMO. V našom príklade sme použili programovací jazyk Visual Basic. Projekt začneme budovať pridaním referencie na *Microsoft.SqlServer.Smo* a *Microsoft.SqlServer.ConnectionInfo*.



Pridanie referencie na *Microsoft.SqlServer.Smo*

Pripojenie sa k lokálnej databáze cez SMO

Ako prvý krok otestujeme, či sa pomocou technológie Server Management Object (SMO) dokážeme pripojiť k lokálnemu databázovému serveru.

Z vizuálneho návrhového prostredia využijeme pre tento účel len nejaký jednoduchý prvok, pomocou ktorého uvedieme aplikáciu „do akcie“, napríklad tlačidlo s názvom btnConnect. Dvojklikom na symbol tlačidla vytvoríme obslužnú procedúru pre udalosť zatlačenia tlačidla. Na začiatok zdrojového kódu pridáme riadok

```
Imports Microsoft.SqlServer.Management.Smo
a do vnútra obslužnej procedúry kód, ktorým pridáme nový objekt typu Server
```

```
Dim srv As New Server(Environment.MachineName)
```

Následne sa pokúsime vypísať nejaké informácie získané zo serveru, napríklad dátum vytvorenia databázy AdventureWorks.

```
MessageBox.Show(„The AdventureWorks database was created at „ & _
    srv.Databases(„AdventureWorks“).CreateDate.ToString())
```

Kompletný zdrojový kód potom bude (pridané riadky sú vytlačené hrubým fontom)

```
Imports Microsoft.SqlServer.Management.Smo
```

```
Public Class Form1
```

```
Private Sub btnConnect_Click(ByVal sender As System.Object, ByVal e As System.
EventArgs) Handles btnConnect.Click
```

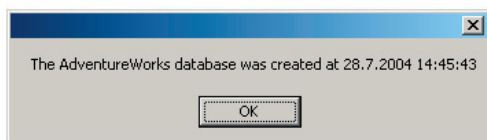
```
Dim srv As New Server(Environment.MachineName)
```

```
MessageBox.Show(„The AdventureWorks database was created at „ & _
    srv.Databases(„AdventureWorks“).CreateDate.ToString())
```

```
End Sub
```

```
End Class
```

Po spustení aplikácie sa v prípade úspechu dozvieme od databázového servera požadované údaje.



Test konektivity na lokálnu databázu cez SMO

Zálohovanie databázy prostredníctvom SMO

Po prvom zoznamovacom pokuse s technológiou Server Management Object, kde sme len testovali, či sa dokážeme pripojiť k lokálnemu databázovému serveru, si vytýčíme ciele oveľa vyššie. Pokúsime sa zazálohovať databázu do súboru. Znovu budeme potrebovať pridať do aplikácie tlačidlo, ktorým odštartujeme zálohovanie, napríklad s názvom btnBackup.

```
Private Sub btnBackup_Click(ByVal sender As System.Object, ByVal e As System.
EventArgs) Handles btnBackup.Click
```

```
    Dim srv As New Server(Environment.MachineName)
    Dim back As New Backup

    back.Database = „AdventureWorks“
    back.DeviceType = DeviceType.File
    back.Devices.Add(„C:\Zaloha\SMO_ZALOHA.bak“)
    back.Action = BackupActionType.Database
    back.SqlBackup(srv)
End Sub
```

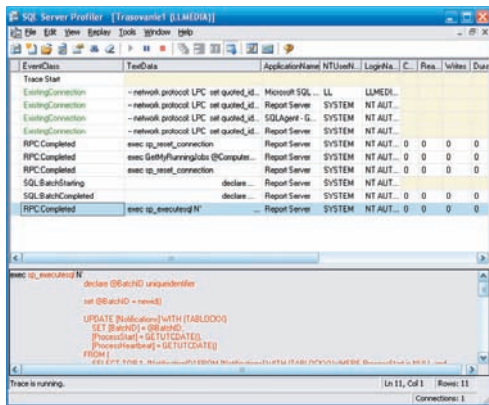
Po aktivovaní zálohovacej procedúry sa môžeme presvedčiť o jej úspešnosti jednoducho tak, že si cez nejaký súborový manažér pozrieme, či súbor zo zálohou údajov vôbec vznikol a akú má prípadne veľkosť

```
Dim rest As New Restore
rest.DeviceType = DeviceType.File
rest.Devices.Add(„C:\Zaloha\SMO_ZALOHA.bak „)

Dim verified As Boolean = rest.SqlVerify(srv)
If verified Then
    MessageBox.Show(„Zalohovanie bolo uspesne“)
Else
    MessageBox.Show(„Chyba pri zalohovani“)
End If
```

SQL Profiler

Ladiaci nástroj Profiler slúži pre monitorovanie inštancií databázového a analytického servera. Pomáha odhaľovať problematické požiadavky a dávky príkazov, ktoré neúmerne zaťažujú databázový server a znižujú tak jeho výkon. Pre zvýšenie komfortu ladenia výkonu a optimalizáciu požiadaviek je možné prehrávať zaznamenané SQL požiadavky. Profiler umožňuje zobrazenie Performance čítačov aj vizualizáciu deadlock-ov. Výsledky trasovania je možné ukladať do XML dokumentov. Administrátor dokáže zistiť informácie o stĺpcoch na ktorých sa robili agregácie, prípadne zistiť akého počtu riadkov sa tá ktorá operácia týkala

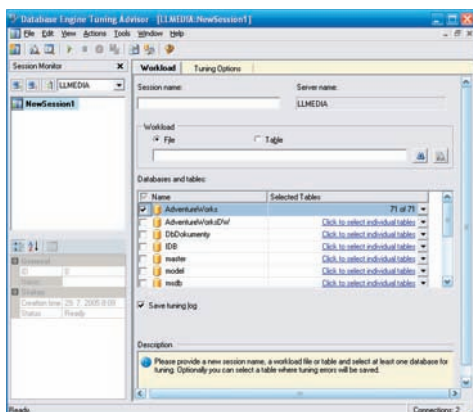


SQL Profiler

Je možné nastaviť šablónu trasovania, alebo zjednodušene povedané na čo sa pri trasovaní zamerať, napríklad na dobu trvania príkazov, spôsob spracovávania agregácií a podobne.

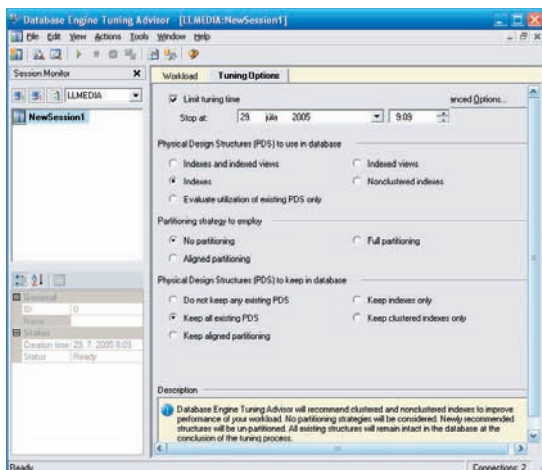
Database Engine Tuning Advisor

Pre časovo ohraničené ladenie výkonu databázy a rôzne analýzy typu „what if?“ („čo ak?“) je možné použiť nástroj Database Engine Tuning Advisor. Pracuje na základe výsledkov z profilera. Ladiť je možné buď celú databázu, prípadne len vybrané tabuľky. Pre výber objektov ladenia slúži záložka Workload



Database Engine Tuning Advisor – záložka Workload

V záložke Tuning Options nastavujeme parametre ladenia



Database Engine Tuning Advisor – záložka Tuning Option

Kapitola 3: Novinky pre vysokú dostupnosť databáz

Na úvod kapitoly venovanej popisu novinek pre vysoký výkon, dostupnosť a zabezpečenie údajov uvidíme prehľad nových vlastností vo forme tabuľky. Nie všetky vlastnosti sú dostupné vo všetkých verziách.

Vlastnosť	Popis
Database Mirroring	Okamžitá náhrada (<3 s), úplná odolnosť voči chybe, možnosť použitia rôznych topológií. Z hľadiska HW sa využívajú štandardné komponenty, Nemá zdieľané úložisko údajov.
Online Restore	Administrátori môžu vykonávať restore operácie počas behu inštancie SQL Servera.
Fast Recovery	Možnosť rýchlej obnovy údajov zvyšuje spoľahlivosť databáz.
Bezpečnosť	Rozšírený bezpečnostný model, dôslednejšie zabezpečenie pri implicitnej inštalácii a nastavení, šifrovanie obsahu databázy, možnosť nastavenia „silných hesiel“, presnejšie špecifikovanie a vymedzenie prístupových práv
SQL Server Management Studio	Integrovaný nástroj pre správu databázy a prácu s databázovými objektmi. Umožňuje dopytovanie v jazykoch T-SQL, MDX a DMX
Dedikované pripojenie administrátora	Umožňuje administrátorovi prístup a spúšťanie diagnostických T-SQL skriptov aj v dobe, keď je SQL Server 2005 pre ostatné prístupy uzamknutý
Database Snapshot	Trvalý snapshot na disku
Snapshot Isolation	Snapshot Isolation umožňuje prístup k poslednému potvrdenému riadku pri zachovaní konzistencie transakcie
Partície	Možnosť rozdelenia tabuľky obsahujúcej veľké množstvo údajov na partície umožňuje rýchlejší prístup k údajom

Niektoré vlastnosti popíšeme podrobnejšie

Snapshot isolation

Nová transakčná izolačná úroveň: Snapshot Isolation (SI) – umožňuje na začiatku transakcie vykonať „snapshot“ (statickú snímku, momentku) databázy. Nasledujúce operácie potom pracujú s takto vytvoreným pohľadom na údaje. Výrazne sa tak zníži doba čakania v dôsledku uzamykania záznamov.

Snapshot vždy číta súvislé dáta, preto nie je potrebné používať „read lock“ ani v prípade ak boli dáta zmenené. Verzia každej hodnoty sa drží podľa požiadaviek. Transakcia číta verziu hodnôt korešpondujúcu s momentom spustenia transakcie.

SQL Server 2005 ponúka dva štýly snapshot isolation, pričom obidva štýly poskytujú detekciu konfliktov

transaction

- konzistentný do začiatku transakcie
- nevidí zmeny spôsobené inými „commit-mi“ počas transakcie
- podobný „serializable“

statement

- konzistentný do posledného príkazu
- vidí zmeny vyvolané inými „commitmi“ počas transakcie
- podobný „read committed“

Prehľad o nastavení SNAPSHOT získame príkazom

```
SELECT name, snapshot_isolation_state, snapshot_isolation_state_desc
FROM sys.databases
```

name	snapshot_isolation_state	snapshot_isolation_state_desc
master	1	ON
tempdb	0	OFF
model	0	OFF
msdb	1	ON
ReportServer	0	OFF
ReportServerTempDB	0	OFF
AdventureWorksDW	1	ON
AdventureWorks	0	OFF
TestBI	0	OFF
TestDB	0	OFF
DbDokumenty	0	OFF
Test	0	OFF
IDB	0	OFF
NovaDatabaza	0	OFF

Snapshot Isolation zapneme pre príslušnú databázu príkazom

```
ALTER DATABASE test SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Nastavením parametra na OFF Snapshot Isolation môžeme zablokovať

```
ALTER DATABASE test SET ALLOW_SNAPSHOT_ISOLATION OFF;
```

Podobne môžeme nastavovať aj vlastnosť READ_COMMITTED_SNAPSHOT

```
ALTER DATABASE test SET READ_COMMITTED_SNAPSHOT ON;
```

Prehľad pre jednotlivé databázy získame príkazom

```
SELECT name, is_read_committed_snapshot_on FROM sys.databases
```

Úroveň Snapshot Isolation nastavujeme príkazom

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
```

Pre demonštráciu Snapshot Isolation urobíme jednoduchý pokus. Vytvoríme tabuľku

```
CREATE TABLE pokus
(
  ID int identity(1,1),
  meno nvarchar(50)
)

INSERT INTO pokus VALUES (,PovodnaHodnota1`);
INSERT INTO pokus VALUES (,PovodnaHodnota2`);
INSERT INTO pokus VALUES (,PovodnaHodnota3`);
```

Nastavíme Snapshot Isolation

```
ALTER DATABASE test SET ALLOW_SNAPSHOT_ISOLATION ON;
```

```
ALTER DATABASE test SET READ_COMMITTED_SNAPSHOT ON;
```

Pre ďalší pokus budeme potrebovať dve Query okná SQL Server management Studia. V prvom okne zadáme príkazy.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
GO
BEGIN TRAN
GO
UPDATE pokus SET meno = ,Hodnota1` WHERE meno = ,PovodnaHodnota1`
GO
```

V druhom následne na to sekvenciu príkazov

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
GO
BEGIN TRAN
GO
Select * from pokus
GO
```

Podľa výsledného výpisu

ID	meno
1	PovodnaHodnota1
2	PovodnaHodnota2
3	PovodnaHodnota3

Vidíme, že rozpracovaná transakcia s ineho pripojenia nijako neovplyvnila pôvodné hodnoty pre iné pripojenia

Môžeme sa pokúsiť potvrdiť transakciu z druhého pripojenia

```
COMMIT TRAN
GO
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
GO
BEGIN TRAN
GO
Select * from pokus
GO
```

No vzhľadom na Snapshot Isolation nebudeme úspešní

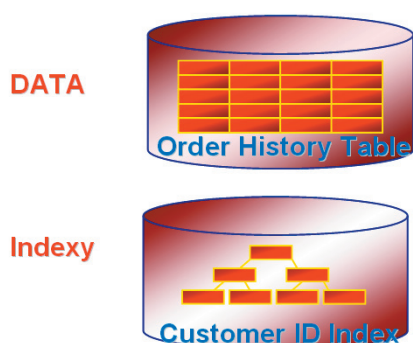
```
Msg 3951, Level 16, State 1, Line 1
Transaction failed in database ,Test` because the statement was run under snapshot
isolation but the transaction did not start in snapshot isolation. You cannot change
the isolation level of the transaction to snapshot after the transaction has started
unless the transaction was originally started under snapshot isolation level.
```

Dedikované administrátorské pripojenie

Administrátorské pripojenie je špeciálny typ pripojenia sa administrátora na SQL server a to za každých okolností. Pre spojenie tohto typu sú zvlášť vyhradené systémové prostriedky, čo umožní jeho fungovanie aj v prípade, že zvyšok databázy je preťažený, alebo nefunkčný. Dedikované administrátorské pripojenie umožňuje administrátorovi pokúsiť sa o ukončenie chybných procesov, prípadne zjednať nápravu v špecifickej situácii pomocou príkazov T-SQL.

Rozdelenie tabuliek na viacero partícií

Ak databázová aplikácia pracuje s veľkým množstvom údajov, je z hľadiska manipulácie s údajmi výhodné ukladať tieto údaje do databázových tabuliek rozdelených na niekoľko častí - partícií. Tabuľku môžeme na partície rozdeliť podľa rôznych kritérií, najčastejšie však podľa časového hľadiska. Ako typický príklad by sme mohli uviesť zber údajov o telefónnych hovoroch účastníkov verejnej telefónnej ústredne, alebo mobilného operátora. Mesačne sa získajú milióny až desiatky miliónov údajov. Preto je v takomto prípade výhodné rozdeliť tabuľku na niekoľko logických oddielov (partícií), v našom prípade bude jeden logický oddiel obsahovať údaje za jeden kalendárny mesiac. Takéto delenie je výhodné nielen pre rýchlejšie dopytovanie v menších tabuľkách, no je logické aj z hľadiska spracovania údajov, pretože reklamácie a fakturácia sa obvykle vykonávajú mesačne. Inokedy pri adekvátnych objemoch údajov je výhodnejšie rozdelenie na partície pre ročné objemy údajov, prípadne môžu byť partície pre jednotlivé filiálky a podobne.



Klasická organizácia – údaje v celistvej databázovej tabuľke, podobne aj celistvú štruktúru majú indexy



Údaje uložené v partiálnych tabuľkách rozdelených podľa rokov

Dopytovanie v tabuľkách je podstatne výkonnejšie na systémoch od 8 procesorov – dopyty sa vykonávajú paralelne cez rôzne partície. Redukuje sa tým čas procesora

Postup vytvorenia databázovej tabuľky rozdelenej na viacero partícií môžeme rozdeliť do troch za sebou nasledujúcich krokov

- vytvorenie partičnej funkcie
- vytvorenie partičnej schémy
- vytvorenie tabuľky

Partičná funkcia slúži pre definovania rozdelenia hodnôt do partícií. Pomocou nej vlastne definujeme kritérium pomocou ktorého zaradíme každú konkrétnu hodnotu do niektorej partície

```
create partition function PF1 (int) as range right for values (1,2,3,4);
```

Následne vytvoríme partičnú schému. Táto mapuje tabuľku alebo index rozdelený na partície na príslušné fyzické súbory


```
create partition scheme PS2 as partition PF1 all to ([PRIMARY]);
```

Databázový server nám potvrdí vytvorenie partičnú schému. Táto bude využívať „filegroupu“ PRIMARY

```
Partition scheme ,PS2` has been created successfully.
,PRIMARY` is marked as the next used filegroup in partition scheme ,PS2`.
```

Ak už máme vytvorenú partičnú funkciu, môžeme vytvoriť tabuľku, kde ju využijeme

```
create table tabulka1
(
    a int,
    b int
) on PS2 (a);
```

Následne tabuľku naplníme údajmi. V tomto simulovanom prípade použijeme pre generovanie údajov generátor náhodných čísel v cykle. Pre demonštráciu nám postačí 1000 hodnôt

```
declare @i int;
set nocount on;
set @i=0;
while @i<1000
begin set @i=@i+1;
    insert into tabulka1 values (5*rand(),100*rand())
end;
```

Pre ilustráciu si môžeme nechať vypísať obsah tabuľky príkazom

```
select * from tabulka1
```

Keby sme nevedeli, že tabuľka je rozdelená na viac partičí, vôbec by sme nepostrehli, že výpis, ktorý sme získali pomocou dopytu je vlastne spojený výpis z piatich partičí. Môžeme koncipovať SQL dotazy tak, aby smerovali len do niektorých partičí. Napríklad tento dopyt bude smerovať len do štvrtej a piatej partičie

```
select * from tabulka1 where a>3
```

Nasledujúci dopyt smeruje len do jednej partičie, smerovanie pomocou podmienky je nepriame – na základe znalosti ako sú údaje v partičiách organizované

```
select * from tabulka1 where a=4
```

Ak chceme nasmerovať dotaz do konkrétnej partičie môžeme použiť dopyt v tvare

```
select * from tabulka1 where $partition.PF1(a)=3
```

Ak chceme nasmerovať dotaz do konkrétnej partičie môžeme použiť dopyt v tvare

```
select * from tabulka1 where $partition.PF1(a)=3
```

Dokážeme taktiež zistiť aj počet záznamov v každej neprázdnej partiči

```
select $partition.PF1(a), count(*) from tabulka1
group by $partition.PF1(a)
order by 1;
```

V tomto prípade, keďže sme pre naplnenie tabuliek použili generátor náhodných čísel, je výsledok aj nepriamym potvrdením jeho kvality, keďže vygenerované čísla sú do jednotlivých partičí rozdelené zhruba rovnomerne

```
-----
1          215
2          192
3          182
4          222
5          189
```

Viac informácií zistíme, ak povolíme generovanie štatistického profilu

```
set statistics profile on
select * from tabulka1 where $partition.PF1(a)>3
```

Po ukončení testovania sledovanie štatistiky vypneme

```
set statistics profile off
```

Záznamy je možné zadeľovať do partícií len na základe hodnoty jedného stĺpca. V prvom príklade sme ich zaraďovali podľa číselnej hodnoty. V nasledujúcom príklade budeme zaraďovať podľa mailovej adresy, teda podľa textového reťazca. V príklade budú vytvorené 3 partície. Prvá pre adresy medzi null a 'F', druhá medzi 'G' a 'M', a posledná od 'N' do konca abecedy.

vytvoríme partičnú funkciu

```
CREATE PARTITION FUNCTION emailPF (nvarchar(30))
AS RANGE RIGHT FOR VALUES (,G', ,N')
```

vytvoríme partičnú schému

```
CREATE PARTITION SCHEME emailPS
AS PARTITION emailPF all to ([PRIMARY])
```

A nakoniec vytvoríme tabuľku

```
CREATE TABLE CustomerEmail
(CustID int, email nvarchar(30))
ON EMailPS (email)
```

Kapitola 4: Bezpečnosť

Nová verzia databázového serveru obsahuje okrem významných vylepšení v oblasti funkcionality a výkonu aj nemenej dôležité vylepšenia v oblasti bezpečnosti. Pokryť tému bezpečnosti komplexne v publikácii tohto rozsahu nie je možné, preto sme z množstva nových črt a funkcií sme vybrali dve – schémy a šifrovanie údajov.

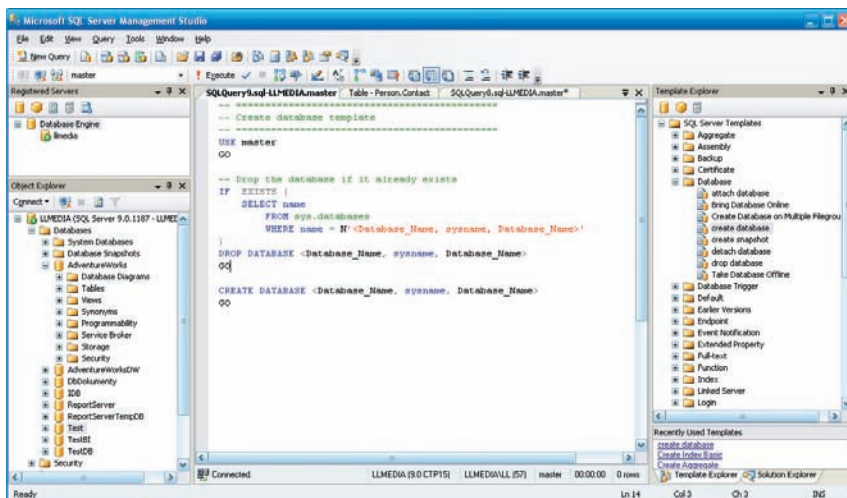
Schémy

Jednou z najvýznamnejších novinek, ktorú prináša SQL Server 2005 sú schémy. SQL Server 2005 umožňuje použitie viacerých schém v jednej databáze. Schémy existujú nezávisle od užívateľov, pričom každý užívateľ má prednastavenú schému. Pre použitie „non-default“ schémy musí byť použité dvojdielne meno. Meno schémy uľahčuje správu databázy môže nahrádzať meno užívateľa v objekte pri migrácii osôb. Napríklad

```
CREATE TABLE marketing.prehľad (...)  
GO  
ALTER SCHEMA marketing  
    AUTHORIZATION JozefNovak
```

Pričom MARKETING je schéma a nie užívateľ, ako by sa mohlo zdať na prvý pohľad. Druhým príkazom určíme, že vlastníkom schémy je JozefNovak

Pre demonštrovanie použitia schém na praktických príkladoch vytvoríme novú databázu. Bude to užitočné, pretože zároveň ukážeme použitie šablón. V okne Template Explorer vyberieme šablónu create database. Nachádza sa v zložke Database.

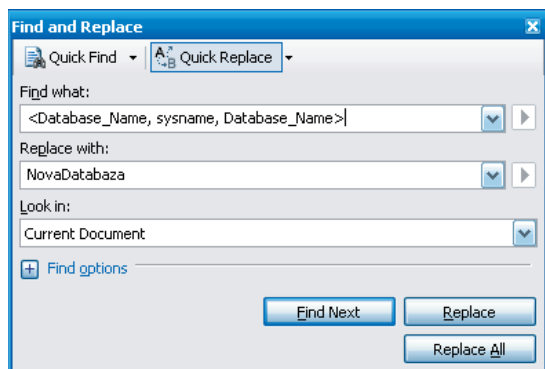


Využitie šablóny pre vytvorenie novej databázy

```
-- =====  
-- Create database template  
-- =====  
USE master  
GO  
  
-- Drop the database if it already exists  
IF EXISTS (  
    SELECT name  
        FROM sys.databases  
        WHERE name = N'<Database_Name, sysname, Database_Name>'  
)  
DROP DATABASE <Database_Name, sysname, Database_Name>  
GO
```

```
CREATE DATABASE <Database_Name, sysname, Database_Name>
GO
```

V šablóne je potrebné nahradiť text <Database_Name, sysname, Database_Name> konkrétnym názvom databázy, v našom prípade napríklad NovaDatabaza. Veľmi ľahko to dosiahneme pomocou menu Edit | Find and Replace | Quick Replace.



Nahradenie parametrov v šablóne konkrétnymi hodnotami

Po nahradení parametrov (a vynechaní komentárov) bude príkaz pre vytvorenie novej databázy vyzerat' nasledovne

```
USE master

IF EXISTS (
    SELECT name
    FROM sys.databases
    WHERE name = N'NovaDatabaza'
)
DROP DATABASE NovaDatabaza
CREATE DATABASE NovaDatabaza
```

O vytvorení novej databázy sa môžeme jednoducho presvedčiť v okne Server Explorer. Možno že bude potrebné pomocou kontextového menu aktivovať funkciu Refresh na zložku Database.

Vytvorenie nových používateľov

Prvým krokom k vytvoreniu prehľadného systému objektov v databáze s využitím schém je vytvorenie nových používateľov

```
EXEC sp_addlogin Pouzivatel1, he$$1o1, NovaDatabaza
EXEC sp_addlogin Pouzivatel2, he$$1o1, NovaDatabaza
```

O vytvorení používateľov sa môžeme presvedčiť buď pomocou vizuálnych nástrojov Management Studia, alebo príkazom

```
EXEC sp_helplogins
```

LoginName	DefDBName	DefLangName	AUser	ARemote
...				
LLMEDIA\ASPNET	master	us_english	yes	no
NT AUTHORITY\SYSTEM	master	us_english	yes	no
Pouzivatel1	NovaDatabaza	us_english	NO	no
Pouzivatel2	NovaDatabaza	us_english	NO	no
sa	master	us_english	yes	no
...				

Novovytvorené účty nakonfigurujeme ako používateľov v novovytvorenej databáze, pričom Používateľ1 bude asociovaný so schémou NasaSchema.

```
USE NovaDatabaza
GO
CREATE USER Pouzivatel1
WITH DEFAULT_SCHEMA = NasaSchema
GO
CREATE USER Pouzivatel2
GO
```

Role

```
USE NovaDatabaza
GO
CREATE ROLE CvicnyPouzivatel
GO
CREATE SCHEMA CvicnyPouzivatel AUTHORIZATION CvicnyPouzivatel
GO
EXEC sp_addrolemember CvicnyPouzivatel, Pouzivatel1
EXEC sp_addrolemember CvicnyPouzivatel, Pouzivatel2
EXEC sp_addrolemember db_datareader, CvicnyPouzivatel
GO
```

Vytvoríme ďalšiu schému s názvom CvicnaSchema

```
CREATE SCHEMA CvicnaSchema AUTHORIZATION CvicnyPouzivatel
```

Zoznam aplikovaných schém môžeme vypísať pomocou príkazu

```
SELECT * FROM sys.schemas
```

name	schema_id	principal_id
dbo	1	1
guest	2	2
INFORMATION_SCHEMA	3	3
sys	4	4
CvicnaSchema	5	7
CvicnyPouzivatel	7	7
db_owner	16384	16384
db_accessadmin	16385	16385
db_securityadmin	16386	16386
db_ddladmin	16387	16387
db_backupoperator	16389	16389
db_datareader	16390	16390
db_datawriter	16391	16391
db_denydatareader	16392	16392
db_denydatawriter	16393	16393

Po vytvorení schém a nadefinovaní prístupových práv môžeme v schémach vytvárať nové objekty, napríklad databázové tabuľky. Aby sme ukázali zapuzdrenie tabuliek v schémach, vytvoríme dve tabuľky s rovnakým názvom, pričom jedna bude v schéme CvicnaSchema a druhá v schéme dbo (Database Owner). Do tabuliek vložíme jeden textový reťazec so špecifickým názvom, aby sme neskôr dokázali tabuľky v jednotlivých schémach identifikovať

```
CREATE TABLE CvicnaSchema.CvicnaTabulka
(
    verzia varchar(20)
)
INSERT INTO CvicnaSchema.CvicnaTabulka SELECT ,CvicnaSchema `
GO
```

```
CREATE TABLE dbo.CvicnaTabulka
(
    verzia varchar(20)
)
INSERT INTO dbo.CvicnaTabulka SELECT ,DBO schema `
```

Ak položíme SQL dopyt v tvare

```
SELECT * FROM CvicnaTabulka
```

```
verzia
-----
DBO schema
```

Z výsledkov dotazovania je jasné, že dopyt smeruje do default schémy daného používateľa. Implicitne je to schéma DBO (Database Owner)

Ak presne špecifikujeme schému, bude dopyt smerovať do tabuľky vo vybranej schéme

```
SELECT * FROM CvicnaSchema.CvicnaTabulka
```

```
verzia
-----
CvicnaSchema
```

Šifrovanie údajov s stĺpcoch databázovej tabuľky

SQL Server 2005 podporuje symetrické šifrovanie údajov v stĺpcoch databázovej tabuľky. Pre tento účel má implementované funkcie:

- EncryptByKey
- DecryptByKey
- EncryptByPassPhrase
- DecryptByPassPhrase
- Key_ID
- Key_GUID

Pre vytvorenie symetrických kľúčov a certifikátov sú k dispozícii príkazy:

- CREATE SYMMETRIC KEY
- CREATE CERTIFICATE

Zoznam kľúčov môžeme vypísať príkazom

```
SELECT * FROM sys.symmetric_keys
```

Šifrovanie stĺpcov ukážeme na príklade v cvičnej databáze AdventureWorks. Najskôr vytvoríme MASTER KEY. Toto heslo je dôležité pre prenosu databázy na iný server. Je šifrovaný kľúčom uloženým v DAPI, takže ho nie je nutné zadávať pri každom štarte databázy.

```
CREATE MASTER KEY ENCRYPTION BY
    PASSWORD = ,HrdzavaKlucKaNaKancelariiCislo999`
```

a certifikát, ktorý slúži pre ochranu symetrických kľúčov databázy

```
CREATE CERTIFICATE Sales09
    WITH SUBJECT = ,Customer Credit Card Numbers`;
```

Po vytvorení nového certifikátu budeme upozornení na skutočnosť, že zatiaľ nie je platný

Warning: The certificate you created is not yet valid; its start date is in the future.

Vytvoríme symetrický kľúč

```
CREATE SYMMETRIC KEY CreditCards_Key11
    WITH ALGORITHM = DES
    ENCRYPTION BY CERTIFICATE Sales09;
```

Šifrovanie údajov budeme realizovať v tabuľke CreditCard.

CreditCardID	CardType	CardNumber	ExpMonth	ExpYear	ModifiedDate
1	SuperiorCard	33332664695310	11	2006	2003-08-30
2	Distinguish	55552127249722	8	2005	2004-01-06
3	ColonialVoice	77778344838353	7	2005	2004-02-15
4	ColonialVoice	77774915718248	7	2006	2003-06-21
5	Vista	11114404600042	4	2005	2003-03-05
6	Distinguish	55557132036181	9	2006	2004-05-11
...					

Do tejto tabuľky pridáme stĺpec pre ukladanie zašifrovaných čísel kreditných kariet

```
ALTER TABLE Sales.CreditCard
    ADD CardNumber_Encrypted varbinary(128);
```

Pre stĺpec otvoríme symetrický kód

```
OPEN SYMMETRIC KEY CreditCards_Key11
    DECRYPTION BY CERTIFICATE Sales09;
```

Pomocou symetrického kódu zašifrujeme údaje zo stĺpca CardNumber a uložíme ich do stĺpca CardNumber_Encrypted

```
UPDATE Sales.CreditCard
SET CardNumber_Encrypted = EncryptByKey(Key_GUID(,CreditCards_Key11`),
    , CardNumber, 1, CONVERT( varbinary, CreditCardID));
```

O zasifrovaní sa môžeme presvedčiť výpisom (náš výpis je vodorovne skrátenej)

CardNumber	CardNumber_Encrypted
33332664695310	0x00C9264CBEC44B48B65CFD6DD6713BA20100000032B1BB9AB86918B0273...
55552127249722	0x00C9264CBEC44B48B65CFD6DD6713BA201000000AE534B178DC9EA7F35C...
77778344838353	0x00C9264CBEC44B48B65CFD6DD6713BA2010000007E172EC5EA704B32BD2...
77774915718248	0x00C9264CBEC44B48B65CFD6DD6713BA2010000009A47642E6C213139B14...
11114404600042	0x00C9264CBEC44B48B65CFD6DD6713BA201000000E73962DA312A83BB5FD...
...	

Pre opačný proces – dešifrovanie je potrebné otvoriť symetrický kód a použiť funkciu pre dešifrovanie

```
OPEN SYMMETRIC KEY CreditCards_Key11
    DECRYPTION BY CERTIFICATE Sales09;
GO
```

```
SELECT CardNumber, CardNumber_Encrypted
    AS „Encrypted card number“, CONVERT(nvarchar,
    DecryptByKey(CardNumber_Encrypted, 1 ,
    CONVERT(varbinary, CreditCardID)))
    AS „Decrypted card number“ FROM Sales.CreditCard;
GO
```

V reálnej aplikácii by sa samozrejme po zašifrovaní stĺpec CardNumber, obsahujúci nezašifrované údaje z databázovej tabuľky odstránil.

Kapitola 5: Nové črty v jazyku SQL a T-SQL

Ak by sme sa snažili nájsť oblasť v ktorej by mali byť databázové servery a to nielen po sebe idúce verzie toho istého výrobcu, ale aj databázové servery rôznych firiem najviac kompatibilné, mal by to byť práve jazyk SQL, ktorý je normalizovaný podľa štandardov SQL 92 (pre túto normu sa vžil skrátený názov SQL-2). Vývoj v oblasti normalizácie pokračoval verziami SQL 1999, (SQL-3) a najnovšou verziou SQL 2003. Žiadna z noriem však neobmedzuje črty a funkcie, ktoré sa v snahe o vylepšenie svojho produktu každý výrobca databázového servera snaží implementovať. Predovšetkým je to procedurálna nadstavba jazyka SQL. U všetkých verzií Microsoft SQL Servera je implementovaná procedurálne nadstavba s názvom Transact SQL (T-SQL).

Po zoznámení sa so schémami si priblížme najvýznamnejšie novinky novej verzie SQL Servera 2005. Pracovať budeme s cvičnou databázou Adventure Works.

Príkaz TOP n

Príkaz TOP n pomocou ktorého je možné vypísať z množiny vybraných záznamov ktoré vyhovujú podmienkam výberu len požadovaný počet, napríklad desať „naj“, pričom o tom či sa vyberalo z oblasti najväčších alebo najmenších hodnôt sa rozhodovalo klauzulou ORDER BY pre utriedenie. V novej verzii 2005 však parametrom n určujúcim počet vybraných hodnôt môže byť aj premenná, čím môžeme tento počet flexibilne meniť. V nasledujúcom príklade vytvoríme premennú @n, naplníme ju požadovanou hodnotou a túto premennú použijeme ako parameter v príkaze TOP (n)

```
DECLARE @n int
SET @n=5
SELECT TOP(@n) EmployeeID, ManagerID, Title, SickLeaveHours FROM HumanResources.
Employee ORDER BY SickLeaveHours DESC
```

Výstupom bude zoznam piatich pracovníkov, ktorí mali najviac vymeškaných hodín z dôvodu nemoci

EmployeeID	DepartmentID	ManagerID	Title	SickLeaveHours
4	2	3	Senior Tool Designer	80
72	15	85	Stocker	69
109	16	NULL	Chief Executive Officer	69
141	7	38	Production Technician - WC50	69
179	7	38	Production Technician - WC50	69

Ak by sme chceli aj mená zamestnancov, potrebujeme ešte aj údaje z tabuľky Contact v schéme Person. Necháme si vypísať prvých troch najväčších absentérov

```
DECLARE @n int
SET @n=3
SELECT TOP(@n) Em.EmployeeID, LastName, FirstName, SickLeaveHours
FROM HumanResources.Employee AS Em
JOIN Person.Contact AS Co
ON Co.ContactID = Em.ContactID
ORDER BY SickLeaveHours DESC
```

Výstup bude

EmployeeID	LastName	FirstName	SickLeaveHours
4	Walters	Rob	80
72	Ralls	Kim	69
109	Sánchez	Ken	69

Príkaz TOP (n) je možné použiť aj vo všetkých DML (Data Manipulation Language) príkazoch

Funkcie RANK, DENSE RANK, ROW_NUMBER a NTILE

Príklady Rank a DenseRank, môžeme použiť pre ohodnotenie záznamov podľa niektorého stĺpca. DENSE_RANK neinkrementuje počítadlo ohodnotenia pri viacerých rovnakých hodnotách.

Význam funkcií je v tabuľke

Funkcia	Popis
RANK	Vracia „rank“ pre každý riadok v špecifikovanej partícii resultsetu
DENSE_RANK	Vracia po sebe idúci „rank“ pre každý riadok v špecifikovanej partícii v resultsete
ROW_NUMBER	Vracia poradové číslo riadku v skupine resultsetu
NTILE	Rozdelí riadky v každej partícii resultsetu na špecifikovaný počet tried podľa zadanej hodnoty

Pre všetky vymenované funkcie je typická funkcia OVER, ktorej syntax je

```
OVER ( [ PARTITION BY < value_expression > , ... [ n ] ]
      ORDER BY <column> [ ASC | DESC ] [, ...[ n ] ] )
```

Príkaz **RANK** slúži k vyjadreniu ohodnotenia jednotlivých záznamov podľa určených kritérií. Ak máme napríklad tabuľku hokejových klubov, môžeme ich vyhodnotiť raz podľa počtu vyhraných zápasov, inokedy podľa počtu nastrielaných gólov, prípadne podľa počtu vylúčených hráčov. Pre praktické precvičenie príkazu RANK použijeme ako príklad veľmi zjednodušenú databázu firmy, ktorej predmetom podnikania je poskytovanie spotrebného úveru. V databáze bude tabuľka UVER naplnená niekoľkými záznamami

```
CREATE TABLE uver
(
    zakaznik VARCHAR(20),
    splatit DECIMAL(9,2),
    dlh DECIMAL(9,2)
);

INSERT INTO uver VALUES('Tichy Radek', 350000, 1000)
INSERT INTO uver VALUES('Majer Jan', 0, 5500.20)
INSERT INTO uver VALUES('Setrna Lucia', 500, 200)
INSERT INTO uver VALUES('Toman Jiri', 10000, 3000)
INSERT INTO uver VALUES('Suchy Ignac', 19500, 68800)
INSERT INTO uver VALUES('Tupy Simon', 185300, 2000)
```

Predstavme si reálnu situáciu, keď naša firma poskytujúca spotrebné úvery má už tisíce zákazníkov a ocitla sa v situácii, kedy naozaj potrebuje rýchly prístup k predspracovaným údajom. Na spotrebnom veľtrhu si firma zriadi malý stánok, kde bude mať jedného pracovníka s notebookom, na ktorom bude replika firemnej databázy. Okrem prijímania zákaziek firma urobí reklamnú kampaň v ktorej 100 najlepších zákazníkov získa darček ak na veľtrhu využijú jej služby. Informáciu o tom, či zákazník, ktorého práve obsluhujeme pri stánku firmy patrí medzi 100 najlepších alebo nie zistíme jednoduchým dotazom do databázy. Problém však môže nastať ak sa notebook v stánku pokazí, alebo sa preruší napájanie. Preto potrebujeme vytlačiť čo najjednoduchšiu tabuľku, ktorá bude obsahovať predspracované údaje. Tabuľka musí samozrejme obsahovať zoznam zákazníkov, zoradený v abecednom poradí, aby sme mohli každého zákazníka rýchle nájsť. Ďalej potrebujeme údaje o tom či patrí medzi 100 najlepších zákazníkov. Pre istotu si vytlačíme poradie podľa výšky splatenej sumy pre všetkých zákazníkov, ved' čo ak by konkurencia prišla s lepšou ponukou a my by sme ich mohli tromfnúť len oznamom, že až 200 najlepších zákazníkov získa darček.

Pre tento účel využijeme príkaz RANK()

```
SELECT zakaznik, splatil, dlh,
       RANK() OVER (ORDER BY splatil DESC) AS P_splatil
FROM uver
ORDER BY zakaznik
```

zakaznik	splatil	dlh	P_splatil
Majer Jan	0.00	5500.20	6
Setrna Lucia	500.00	200.00	5
Suchy Ignac	19500.00	68800.00	3
Tichy Radek	350000.00	1000.00	1
Toman Jiri	10000.00	3000.00	4
Tupy Simon	185300.00	2000.00	2

V stĺpci P_SPLATIL máme poradové číslo zákazníka podľa výšky doterajších splátok.

Použitie funkcie RANK() ukážeme aj na jednoduchom príklade v cvičnej databáze AdventureWorks.

```
SELECT SalesOrderID, CustomerID, RANK() OVER (ORDER BY CustomerID) AS RANK
FROM Sales.SalesOrderHeader
ORDER BY CustomerID
```

SalesOrderID	CustomerID	RANK
43860	1	1
44501	1	1
45283	1	1
46042	1	1
46976	2	5
47997	2	5
49054	2	5
...		

Vidíme, že absolútne poradie sa zachováva a po štyroch jedničkách nasleduje päťka. Funkcia DENSE_RANK() „zahusťí“ výpis, tak že sa rovnaké poradie nepreskakuje

```
SELECT SalesOrderID, CustomerID, DENSE_RANK() OVER (ORDER BY CustomerID) AS RANK
FROM Sales.SalesOrderHeader
ORDER BY CustomerID
```

SalesOrderID	CustomerID	RANK
43860	1	1
44501	1	1
45283	1	1
46042	1	1
46976	2	2
47997	2	2
49054	2	2
...		

Ešte raz poukážeme na rozdiel medzi funkciami RANK() a DENSE_RANK(). Zatiaľ čo RANK dokáže produkovať nesequenčné pozície. napr. ak štyri riadky vrátia rovnaku ranking hodnotu a táto je = 1, potom ďalší ranking row bude mať hodnotu 5 (pretože pred ním sú štyri riadky väčšej hodnoty).

DENSE_RANK vypisuje hodnoty sekvencne ak štyri riadky majú hodnotu 1, ďalší ranking riadok bude mať hodnotu 2.

Funkcia **ROW_NUMBER()** vracia poradové číslo riadku v resultsete podľa zadaného kritéria. Je dosť podobná funkcii RANK(), no pre každý záznam sa inkrementuje, takže je v sekvencnom poradí

Syntax:

```
ROW_NUMBER () OVER ( [ <partition_by_clause> ] <order_by_clause> )
```

Použitie funkcie ukážeme na príklade v cvičnej databáze AdventureWorks. Najskôr pre kontrolu vypíšeme poradie podľa stĺpca na ktorý aplikujeme ROW_NUMBER()

```
SELECT SalesOrderID, CustomerID, ROW_NUMBER() OVER (ORDER BY SalesOrderID) AS
Poradie
FROM Sales.SalesOrderHeader
ORDER BY SalesOrderID
```

SalesOrderID	CustomerID	Poradie
43659	676	1
43660	117	2
43661	442	3
43662	227	4
43663	510	5
43664	397	6
43665	146	7
43666	511	8
43667	646	9
...		

Je logické, že údaje boli do databázy postupom času ukladané v takom poradí ako rástlo poradové číslo objednávok. Teraz usporiadame záznamy podľa poradového čísla zákazníkov a prostredníctvom funkcie ROW_NUMBER() zistíme poradie zákazok príslušného zákazníka

```
SELECT SalesOrderID, CustomerID, Row_Number() OVER (ORDER BY SalesOrderID) AS
Poradie
FROM Sales.SalesOrderHeader
ORDER BY CustomerID
```

SalesOrderID	CustomerID	Poradie
43860	1	202
44501	1	843
45283	1	1625
46042	1	2384
46976	2	3318
47997	2	4339
49054	2	5396
50216	2	6558
...		

Môžeme si vybrať len určitý interval z množiny záznamov, v našom prípade medzi 100 a 105.

```

WITH Poradie AS
(
    SELECT SalesOrderID, CustomerID, Row_Number()
        OVER (ORDER BY SalesOrderID) AS Poradie
    FROM Sales.SalesOrderHeader
)
SELECT * FROM Poradie
WHERE Poradie BETWEEN 100 AND 105
ORDER BY SalesOrderID

```

SalesOrderID	CustomerID	Poradie
43758	27646	100
43759	13257	101
43760	16352	102
43761	16493	103
43762	27578	104
43763	16525	105

(6 row(s) affected)

Funkcia **NTILE()** – rozdelí riadky v každej partícii resultsetu na špecifikovaný počet tried podľa zadanej hodnoty. Napríklad akrozdelíme

```

SELECT SalesOrderID, CustomerID, NTILE(10000)
    OVER (ORDER BY CustomerID) AS Poradie
FROM Sales.SalesOrderHeader
WHERE SalesOrderID > 10000
ORDER BY CustomerID

```

SalesOrderID	CustomerID	Poradie
43860	1	1
44501	1	1
45283	1	1
46042	1	1
46976	2	2
47997	2	2
49054	2	2
50216	2	2
51728	2	3
..		
45199	29476	9998
60449	29477	9998
60955	29478	9999
49617	29479	9999
62341	29480	9999
45427	29481	10000
49746	29482	10000
49665	29483	10000

Aby sme to zhrnuli, ukážeme príklad na porovnanie „ranking“ funkcie

```

SELECT SalesOrderID, CustomerID,
    ROW_NUMBER() OVER (ORDER BY CustomerID) AS ROW_NUMBER,
    RANK() OVER (ORDER BY CustomerID) AS RANK,

```

```

DENSE_RANK() OVER (ORDER BY CustomerID) AS DENSE_RANK,
NTILE(10000) OVER (ORDER BY CustomerID) AS NTILE
FROM Sales.SalesOrderHeader
WHERE SalesOrderID > 10000
ORDER BY CustomerID

```

SalesOrderID	CustomerID	ROW_NUMBER	RANK	DENSE_RANK	NTILE
43860	1	1	1	1	1
44501	1	2	1	1	1
45283	1	3	1	1	1
46042	1	4	1	1	1
46976	2	5	5	2	2
47997	2	6	5	2	2
49054	2	7	5	2	2
50216	2	8	5	2	2
51728	2	9	5	2	3
57044	2	10	5	2	3
...					

Ošetrenie chýb pomocou konštrukcie TRY ... CATCH

Jednou z významných novínok SQL Servera 2005 je nový spôsob ošetrenia výnimiek. V predošlých verziách SQL serverov síce výnimky bolo možné ošetrovať pomocou @@ERROR, no túto premennú bolo potrebné nastavovať pre každý príkaz, ktorý mohol potenciálne viesť k vzniku výnimky. Nová verzia SQL Server 2005 umožňuje zapúzdrovať príkazy do podmienkových blokov. Pomocou prídavných funkcií je možné získať informácie o chybe, prípadne si vyžiadať stav transakcie

Z pohľadu syntaxe by sme to mohli zapísať nasledovne:

```

BEGIN TRY
{ sql_statement | statement_block }
END TRY
BEGIN CATCH
{ sql_statement | statement_block }
END CATCH
[ ; ]

```

Pokúsme sa ošetrenie výnimiek demonštrovať na niekoľkých príkladoch. Najskôr do bloku BEGIN-END vložme príkaz, ktorý určite vyvolá výnimku

```

USE AdventureWorks
SET XACT_ABORT ON

BEGIN TRY
    TRUNCATE TABLE Production.Category
END TRY
BEGIN CATCH
    PRINT N'Nevyšlo to, číslo chyby je ,+CONVERT(nvarchar,@@ERROR)
END CATCH

```

Konzolová aplikácia nám v prípade výskytu výnimky vráti oznam

```
Nevyšlo to, číslo chyby je 1088
```

V databáze AdventureWorks vytvorme dve tabuľky

```
CREATE TABLE VynimkaT1
(
    hodnota int NOT NULL PRIMARY KEY
)
CREATE TABLE VynimkaT2
(
    hodnota int NOT NULL REFERENCES VynimkaT1(hodnota)
)
```

Tabuľku VynimkaT1 naplníme niekoľkými hodnotami

```
INSERT INTO VynimkaT1 VALUES (1)
INSERT INTO VynimkaT1 VALUES (3)
INSERT INTO VynimkaT1 VALUES (4)
INSERT INTO VynimkaT1 VALUES (6)
```

Nastavením parametra XACT_ABORT na hodnotu OFF zabezpečíme aby v rámci transakcie bol zrušený len príkaz, ktorý spôsobil výnimku. V nasledujúcom reťazci príkazov v transakcii to bude stredný príkaz, nakoľko v tabuľke VynimkaT1, číslo 2 ako cudzí kľúč nemáme. Výnimku zatiaľ ošetrovať nebudeme

```
SET XACT_ABORT OFF
BEGIN TRAN
    INSERT INTO VynimkaT2 VALUES (1)
    INSERT INTO VynimkaT2 VALUES (2) /* !!! chyba !!! cudzi kluc */
    INSERT INTO VynimkaT2 VALUES (3)
COMMIT TRAN
GO
```

Konzolová aplikácia reaguje na túto chybu oznamom

```
(1 row(s) affected)
Msg 547, Level 16, State 0, Line 5
The INSERT statement conflicted with the FOREIGN KEY constraint „FK__VynimkaT2__
_hodno__414EAC47“. The conflict occurred in database „AdventureWorks“, table
„VynimkaT1“, column ‚hodnota‘.
The statement has been terminated.
(1 row(s) affected)
```

Všimnite si najskôr, že prvý a posledný príkaz transakcie bol vykonaný, zrušený bol len stredný príkaz, ktorý spôsobil chybu

Aby sme mali komplexný prehľad o tom, čo sa vlastne „prihodilo“ vypíšme obsahy tabuliek VynimkaT1 a Vynimka T2

```
SELECT * FROM VynimkaT1
SELECT * FROM VynimkaT2
```

```
hodnota
-----
1
3
4
6
```

```

hodnota
-----
1
3

```

Teraz urobíme rovnaký pokus s vyvolaním chyby, no vznik potenciálnej výnimky ošetríme tak, že v prípade akéhokoľvek neúspechu celú transakciu zrušíme

```

SET XACT_ABORT OFF
BEGIN TRY
    BEGIN TRAN
        INSERT INTO VynimkaT2 VALUES (4)
        INSERT INTO VynimkaT2 VALUES (5) /* !!! cudzi kluc */
        INSERT INTO VynimkaT2 VALUES (6)
    COMMIT TRAN
    PRINT ,Transakcia potvrdena `
END TRY
BEGIN CATCH
    ROLLBACK
    PRINT ,Transakcia odvolana `
END CATCH

```

Konzolová aplikácia nám priebeh akcie popisuje takto

```

(1 row(s) affected)
Transakcia odvolana

```

Po týchto pokusoch odporúčame „upratať“ cvičnú databázu AdventureWorks do pôvodného stavu, teda vymazať obidve tabuľky, ktoré sme v nej vytvorili

```

DROP TABLE VynimkaT1
DROP TABLE VynimkaT1

```

Operátor APPLY

Operátor APPLY by sa najjednoduchšie mohli prirovnať k operátoru JOIN pre spájanie viacerých tabuliek. Na rozdiel od tohto operátora však APPLY nemá žiadnu klauzulu ON. Pravým operandom môže byť ľubovoľná tabuľka, ale pôvodný zámer je pre používateľom definované funkcie nad tabuľkou. Parametre pre používateľom definovanú funkciu môžu byť získané zo stĺpcov ľavého operandu. Podobne ako JOIN aj operátor APPLY umožňuje spájanie typu „cross“ aj „outer“

CROSS APPLY vypíše len riadky sediace s výsledkami funkcie
 OUTER APPLY vypíše všetky riadky bez ohľadu na výsledky funkcie

```

CREATE TABLE Tab1
(
    ID int
)

CREATE TABLE Tab2
(
    ID int
)

INSERT INTO Tab1 VALUES (1)
INSERT INTO Tab1 VALUES (2)
INSERT INTO Tab2 VALUES (3)
INSERT INTO Tab2 VALUES (4)

```



```
SELECT COUNT(*) FROM Tab1 CROSS APPLY Tab2
```

4

Ak aplikujeme CROSS APPLY alebo OUTER bez použitia funkcie bude rozdiel iba v poradí záznamov

```
SELECT * FROM Tab1 CROSS APPLY Tab2
```

ID	ID
1	3
2	3
1	4
2	4

```
SELECT * FROM Tab1 OUTER APPLY Tab2
```

ID	ID
1	3
1	4
2	3
2	4

Druhý príklad bude nad tabuľkami cvičnej databázu AdventureWorks.

```
CREATE FUNCTION dbo.Najnemocnejši
(@n int, @DeptID AS int) RETURNS TABLE AS
RETURN
  SELECT TOP(@n) EmployeeID,SickLeaveHours
  FROM HumanResources.Employee
  WHERE DepartmentID = @DeptID
  ORDER BY SickLeaveHours DESC
```

Funkciu otestujeme

```
SELECT * FROM dbo.Najnemocnejši(5,1)
```

EmployeeID	SickLeaveHours
11	23
270	22
9	22
3	21
267	21

Pomocou operátora APPLY vypíšeme troch najväčších absentérov z dôvodu nemocnosti z každého oddelenia

```
SELECT DepartmentID,Name,Nemocni.* FROM HumanResources.Department
  CROSS APPLY dbo.Najnemocnejši(3,DepartmentID) Nemocni
  ORDER BY DepartmentID
```

(výpis je skrátený)

DepartmentID	Name	EmployeeID	SickLeaveHours
1	Engineering	11	23
1	Engineering	270	22
1	Engineering	9	22
2	Tool Design	4	80
2	Tool Design	265	24
2	Tool Design	5	24
3	Sales	275	39
...			

Operátory PIVOT a UNPIVOT,

Pomocou operátorov PIVOT a UNPIVOT dokážeme prevádzať klasickú databázovú tabuľku na kontingenčnú a naopak. Kontingenčná tabuľka má na rozdiel od klasickej tabuľky niekoľko špeciálnych vlastností. Napríklad umožňuje určitú rotáciu, teda výmenu riadkov a stĺpcov, kombináciu a hierarchickú štruktúru riadkov a stĺpcov.

Zjednodušený syntaktický predpis príkazu PIVOT:

```
<pivoted_table> ::=
table_source PIVOT <pivot_clause> table_alias
<pivot_clause> ::=
( aggregate_function(value_column)
FOR pivot_column
IN ( <column_list>)
```

Zjednodušený syntaktický predpis príkazu UNPIVOT:

```
<unpivoted_table> ::=
table_source UNPIVOT <unpivot_clause> table_alias
<unpivot_clause> ::=
( value_column FOR pivot_column
IN ( <column_list> )
)
<column_list> ::=
column_name [, ...]
```

Najlepšie princípy a použitie vysvetlíme na klasickej databázovej tabuľke. V databáze AdventureWorks vytvoríme novú tabuľku mesačných súhrnov

```
CREATE TABLE SalesOrderTotalsMonthly
(
    CustomerID int NOT NULL,
    OrderMonth int NOT NULL,
    SubTotal money NOT NULL
)
```

a naplníme ju údajmi z tabuľky SalesOrderHeader

```
INSERT SalesOrderTotalsMonthly
SELECT CustomerID, DATEPART(m, OrderDate), SubTotal
FROM Sales.SalesOrderHeader
WHERE CustomerID IN (1,2,4,6)
```

Tabuľka teda bude obsahovať tieto údaje

CustomerID	OrderMonth	SubTotal
4	1	97037,566
4	10	119637,8301
4	7	118394,1258
2	2	1574,1247
2	11	4949,8589
2	8	9216,3596
1	5	31423,5209
...		

Ak však potrebujeme výstup, kde budú riadky a stĺpce usporiadané trochu inak, napríklad pre jednotlivé kvartály príslušného roku, musíme pre vygenerovanie takehoto výstupu použiť pomerne zložitý SQL dotaz

```
SELECT CustomerID,
       SUM(CASE OrderMonth WHEN 1 THEN SubTotal ELSE 0 END) AS ,1`,
       SUM(CASE OrderMonth WHEN 2 THEN SubTotal ELSE 0 END) AS ,2`,
       SUM(CASE OrderMonth WHEN 3 THEN SubTotal ELSE 0 END) AS ,3`,
       SUM(CASE OrderMonth WHEN 4 THEN SubTotal ELSE 0 END) AS ,4`,
       SUM(CASE OrderMonth WHEN 5 THEN SubTotal ELSE 0 END) AS ,5`,
       SUM(CASE OrderMonth WHEN 6 THEN SubTotal ELSE 0 END) AS ,6`,
       SUM(CASE OrderMonth WHEN 7 THEN SubTotal ELSE 0 END) AS ,7`,
       SUM(CASE OrderMonth WHEN 8 THEN SubTotal ELSE 0 END) AS ,8`,
       SUM(CASE OrderMonth WHEN 9 THEN SubTotal ELSE 0 END) AS ,9`,
       SUM(CASE OrderMonth WHEN 10 THEN SubTotal ELSE 0 END) AS ,10`,
       SUM(CASE OrderMonth WHEN 11 THEN SubTotal ELSE 0 END) AS ,11`,
       SUM(CASE OrderMonth WHEN 12 THEN SubTotal ELSE 0 END) AS ,12`
FROM SalesOrderTotalsMonthly GROUP BY CustomerID
```

Požadované hodnoty získame v tvare (výpis obsahuje len údaje za prvých päť mesiacov)

CustomerID	1	2	3	4	5
1	0,00	34066,1881	0,00	0,00	31423,5209
2	0,00	5242,8583	0,00	0,00	2573,6077
4	164452,3584	0,00	0,00	156606,4713	0,00
6	0,00	0,00	559,0108	0,00	0,00

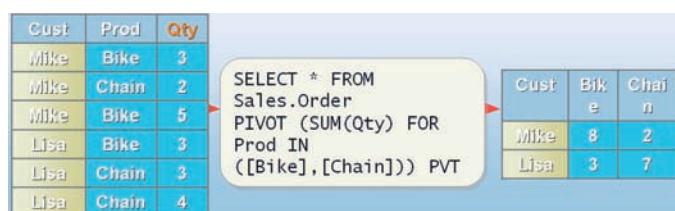
Pomocou operátora PIVOT to dokážeme oveľa jednoduchšie

```
SELECT * FROM SalesOrderTotalsMonthly
PIVOT (SUM(SubTotal) FOR OrderMonth IN
([1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12])) AS a
```

Požadované hodnoty získame v tvare (výpis obsahuje len údaje za prvých päť mesiacov)

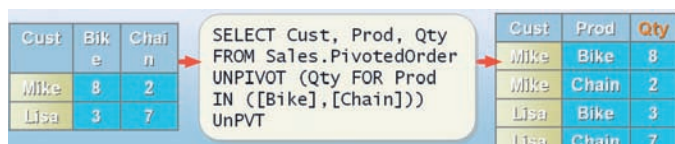
CustomerID	1	2	3	4	5
1	NULL	34066,1881	NULL	NULL	31423,5209
2	NULL	5242,8583	NULL	NULL	2573,6077
4	164452,3584	NULL	NULL	156606,4713	NULL
6	NULL	NULL	559,0108	NULL	NULL

Zjednodušene povedané operátory PIVOT a UNPIVOT sú inverzné. Operátor PIVOT konvertuje hodnoty na stĺpce, presnejšie povedané presúva riadky do stĺpcov a agreguje hodnoty.



Operátor PIVOT konvertuje hodnoty na stĺpce

Naopak operátor UNPIVOT rotuje stĺpce do riadkov



Operátor UNPIVOT konvertuje stĺpce na hodnoty

Problematika kontingenčných tabuliek nie je pre mnohých ľudí príliš pochopiteľná na prvé počutie, preto uvedieme ešte jeden podobný príklad. Tentoraz pôjde o ročné súhrny

Najskôr v databáze AdventureWorks vytvoríme novú tabuľku

```
CREATE TABLE YearlySalesPivot
(
    OrderYear int NOT NULL,
    [1] money NULL,
    [2] money NULL,
    [4] money NULL,
    [6] money NULL
)
```

ktorú naplníme údajmi z tabuľky SalesOrderHeader

```
INSERT SalesOrderTotalsYearly
SELECT CustomerID, YEAR(OrderDate), SubTotal
FROM Sales.SalesOrderHeader
WHERE CustomerID IN (1,2,4,6,35)
```

V novovytvorenej tabuľke budú tieto údaje:

CustomerID	OrderYear	SubTotal
1	2001	13216,0537
1	2001	23646,0339
1	2002	34066,1881
1	2002	31423,5209
2	2002	9216,3596
2	2002	4949,8589
2	2003	1574,1247

Na túto tabuľku môžeme aplikovať niekoľko príkazov SELECT s operátorom PIVOT, napríklad pre súhrny podľa rokov

```
SELECT * FROM SalesOrderTotalsYearly
PIVOT (SUM(SubTotal) FOR OrderYear IN ([2002], [2003], [2004])) AS a
```

Dopyt vráti údaje:

CustomerID	2002	2003	2004
1	65489,709	NULL	NULL
2	14166,2185	10966,5406	4490,7426
4	238031,9559	337994,8408	129887,4224
6	NULL	604,9648	2696,2418

Alebo súhrny podľa zákazníkov

```
SELECT * FROM SalesOrderTotalsYearly
PIVOT (SUM(SubTotal) FOR CustomerID in ([1], [2], [4], [6])) AS a
```

OrderYear	1	2	4	6
2001	36862,0876	NULL	NULL	NULL
2002	65489,709	14166,2185	238031,9559	NULL
2003	NULL	10966,5406	337994,8408	604,9648
2004	NULL	4490,7426	129887,4224	2696,2418

Použitie operátora UNPIVOT ukážeme na príklade tabuľky ročných súhrnov.

V databáze AdventureWorks vytvoríme novú tabuľku, ktorá bude v tvare kontingenčnej tabuľky

```
CREATE TABLE YearlySalesPivot
(
    OrderYear int NOT NULL,
    [1] money NULL,
    [2] money NULL,
    [4] money NULL,
    [6] money NULL
)
```

a naplníme ju údajmi z tabuľky SalesOrderHeader, pričom využijeme konštrukciu INSERT INTO... SELECT s operátorom PIVOT

```
INSERT YearlySalesPivot
SELECT * FROM SalesOrderTotalsYearly
PIVOT (SUM(SubTotal) FOR CustomerID IN ([1], [2], [4], [6])) AS a
```

Tabuľka bude obsahovať rovnaké údaje ako vo výsledku výpisu príkazu SELECT s operátorom PIVOT

OrderYear	1	2	4	6
2001	36862,0876	NULL	NULL	NULL
2002	65489,709	14166,2185	238031,9559	NULL
2003	NULL	10966,5406	337994,8408	604,9648
2004	NULL	4490,7426	129887,4224	2696,2418

Na túto jednoznačne kontingenčnú tabuľku aplikujeme operátor UNPIVOT

```
SELECT * FROM YearlySalesPivot
UNPIVOT (SubTotal FOR CustomerID IN ([1], [2], [4], [6])) AS a
ORDER BY CustomerID
```

Takto sme z kontingenčnej údaje získali výpis kde sú prekonvertované stĺpce na hodnoty.

OrderYear	SubTotal	CustomerID
2001	36862,0876	1
2002	65489,709	1
2002	14166,2185	2
2003	10966,5406	2
2004	4490,7426	2
2004	129887,4224	4
2003	337994,8408	4
2002	238031,9559	4
2003	604,9648	6
2004	2696,2418	6

Ak chceme po týchto pokusoch v databáze AdventureWorks upratať, odstránime tabuľky, ktoré sme sami vytvorili

```
DROP TABLE YearlySalesPivot
DROP TABLE SalesOrderTotalsYearly
DROP TABLE SalesOrderTotalsMonthly
```

Common Table Expressions (CTE)

Common Table Expressions (CTE) by sme mohli charakterizovať ako dočasnú pomenovanú množinu výsledkov „resultset“. Túto množinu špecifikujeme štartovacím dopytom, ktorý obsahuje kľúčové slovo WITH. CTE môže byť zaujímavou alternatívou vnorených dotazov. Môžeme ho použiť spolu s SELECT/INSERT/UPDATE/DELETE a využívať aj v pohľadoch.

Zjednodušený syntaktický predpis by sme mohli zapísať v tvare:

```
SELECT/INSERT/UPDATE/DELETE
WITH <cte>
<cte>::= expression_name
[( column_name [ ,...n ])]
AS
( CTE_query_definition )
```

Ak si pozrieme príkaz pre CTE, napríklad

```
WITH SalesCTE(ProductID, SalesOrderID)
AS
(
    SELECT ProductID, COUNT(SalesOrderID)
    FROM Sales.SalesOrderDetail
    GROUP BY ProductID
)
SELECT * FROM SalesCTE
```

Výsledkom príkazu bude množina údajov

ProductID	SalesOrderID
707	3083
708	3007
709	188
710	44

```
711          3090
712          3382
...
```

Podobne ako u vnorených dotazov môžeme rozpoznať vnútorný

```
SELECT ProductID, COUNT(SalesOrderID)
FROM Sales.SalesOrderDetail
GROUP BY ProductID
```

a vonkajší dopyt

```
SELECT * FROM SalesCTE
```

Pomocou CTE však môžeme riešiť náročnejšie úlohy. Vonkajší dopyt môže obsahovať napríklad podmienku

```
WITH SalesCTE(ProductID, SalesOrderID)
AS
(
    SELECT ProductID, COUNT(SalesOrderID)
    FROM Sales.SalesOrderDetail
    GROUP BY ProductID
)
SELECT * FROM SalesCTE
WHERE SalesOrderID > 50
```

prípade agregačné funkcie

```
WITH SalesCTE(ProductID, SalesOrderID)
AS
(
    SELECT ProductID, COUNT(SalesOrderID)
    FROM Sales.SalesOrderDetail
    GROUP BY ProductID
)
SELECT AVG(SalesOrderID)
FROM SalesCTE
WHERE SalesOrderID > 50
```

Vrcholom možností CTE je podpora rekurzívnych kalkulácií.

Pre príklad rekurzívnej kalkulácie vytvoríme tabuľku, zameranú napríklad na komponenty osobných automobilov

```
CREATE TABLE AutoSuciatky
(
    autoID int NOT NULL,
    diel varchar(15),
    suciastka varchar(15),
    mnozstvo int
)
```

Tabuľku naplníme niekoľkými údajmi, napríklad:

```
INSERT AutoSuciatky VALUES (1, ,Karoseria`, ,Dvere`, 4)
INSERT AutoSuciatky VALUES (1, ,Karoseria`, ,Veko motora`, 1)
INSERT AutoSuciatky VALUES (1, ,Karoseria`, ,Veko kufra`, 1)
INSERT AutoSuciatky VALUES (1, ,Dvere`, ,Klucka`, 1)
```

```

INSERT AutoSuciatky VALUES (1, ,Dvere`, ,Zamok`, 1)
INSERT AutoSuciatky VALUES (1, ,Dvere`, ,Okno`, 1)
INSERT AutoSuciatky VALUES (1, ,Karoseria`, ,Nity`, 1000)
INSERT AutoSuciatky VALUES (1, ,Dvere`, ,Nity`, 100)
INSERT AutoSuciatky VALUES (1, ,Dvere`, ,Zrkadlo`, 1)

```

Všimnite si, že tabuľka je odrazom určitej reality vzťahov vonkajšieho sveta, v tomto prípade častí automobilu. Táto štruktúra je hierarchická. Automobil obsahuje dvere, ktoré pozostávajú z ďalších súčiastok. Pre túto tabuľku vytvoríme rekurzívny dopyt s využitím CTE

```

WITH AutoSuciatkyCTE(suciastka, mnozstvo)
AS
(
    -- UKOTVENIE dopyt nevracia hodnty pre AutoSuciatkyCTE
    SELECT suciastka, mnozstvo FROM AutoSuciatky WHERE diel = ,Karoseria`
    UNION ALL
    -- REKURZIVNY ČLEN dopyt vracia hodnoty AutoSuciatkyCTE
    SELECT AutoSuciatky.suciastka, AutoSuciatkyCTE.mnozstvo * AutoSuciatky.
mnozstvo
    FROM AutoSuciatkyCTE
    INNER JOIN AutoSuciatky ON AutoSuciatky.suciastka = AutoSuciatky.Diel
    WHERE AutoSuciatky.autoID = 1
)
-- vonkajsi dopyt
SELECT suciastka, SUM(mnozstvo) as mnozstvo FROM AutoSuciatkyCTE
GROUP BY suciastka

```

Skôr než si pozrieme výsledky, popíšeme si tento dopyt podrobnejšie. Podľa komentárov vidíme, že rekurzívna „common table expression“ má tri časti

- ukotvenie, nasledované UNION ALL; robí inicializáciu
- rekurzívny členo po UNION ALL; rekurzia až kým už nie sú výsledky
- „vonkajší outer select“; vyberá výsledky pre vrátenie z požiadavky

Rekurzívny CTE dopyt vráti tieto výsledky

suciastka	mnozstvo
Dvere	4
Nity	1000
Veko kufra	1
Veko motora	1

Rozšírenie dátových typov

V novej verzii SQL Server 2005 boli zavedené nové dátové typy pre veľké hodnoty:

- **varchar(max)**
- **nvarchar(max)**
- **varbinary(max)**

a dátový typ **xml** pre ukladanie XML dát voliteľne so podľa schémy.

Dátové typy **varchar(max)**, **nvarchar(max)**, **varbinary(max)** na rozdiel od typov **text**, **ntext** a **image** prelomujú dvojgigabajtovú hranicu pre ukladanie veľkých blokov údajov. Umožňujú uložiť až 2³¹-1 bajtov. Ako vyplýva z názvov, prvé dva dátové typy **varchar(max)**, **nvarchar(max)** slúžia pre ukladanie textových údajov a dátový typ **varbinary(max)** pre ukladanie binárnych údajov, napríklad obrázkov a multimédií.

Ukladanie binárnych dokumentov do databázy sa v praxi pomerne často používa, hlavne obrázkov, prípadne dokumentov vytvorených vo formátoch rôznych kancelárskych programov, napríklad textového editora Microsoft Word.

Pre ilustráciu použitia typu VARBINARY(MAX) vytvoríme jednoduchú tabuľku pre ukladanie wordových dokumentov. Dva stĺpce typu NVARCHAR(60) budú obsahovať údaje o názve súboru a jeho typu, teda prípone a v stĺpci Dokument, ktorý je typu VARBINARY(MAX) bude tento dokument fyzicky uložený.

```
CREATE TABLE Dokumenty
(
  ID int primary key,
  NazovSuboru nvarchar(60),
  TypSuboru nvarchar(60),
  Dokument varbinary(max)
)
```

Dokumenty do databázovej tabuľky ukladáme pomocou funkcie OPENROWSET, pričom stream binárnych údajov vytvoríme pomocou funkcie BULK

```
INSERT INTO Dokumenty(ID, NazovSuboru, TypSuboru, Dokument)
  SELECT 1, ,O sliepocke a kohutikovi.doc` AS NazovSuboru, ,.doc` AS TypSuboru, *
FROM
OPENROWSET(BULK N'C:\O sliepocke a kohutikovi.doc`, SINGLE_BLOB) AS Dokument;
```

```
INSERT INTO Dokumenty(ID, NazovSuboru, TypSuboru, Dokument)
  SELECT 2, ,Pes a zajac.doc` AS NazovSuboru, ,.doc` AS TypSuboru, * FROM
OPENROWSET(BULK N'C:\Pes a zajac.doc`, SINGLE_BLOB) AS Dokument;
```

```
INSERT INTO Dokumenty(ID, NazovSuboru, TypSuboru, Dokument)
  SELECT 3, ,Tahal dedko repu.doc` AS NazovSuboru, ,.doc` AS TypSuboru, * FROM
OPENROWSET(BULK N'C:\Tahal dedko repu.doc`, SINGLE_BLOB) AS Dokument;
```

Do takto univerzálne navrhutej tabuľky môžeme samozrejme ukladať aj iné typy dokumentov, napríklad obrázky, dokumenty v PDF, a podobne.

Pri výpise údajov z tabuľky zistíme obsah stĺpcov so všetkými dátovými typmi s výnimkou varbinary(MAX).

```
SELECT * Dokumenty;
```

ID	NazovSuboru	TypSuboru	Dokument
1	O sliepocke a kohutikovi.doc	.doc	0xD0CF11E0A1B11AE1...
2	Pes a zajac.doc	.doc	0xD0CF11E0A1B11AE1...
3	Tahal dedko repu.doc	.doc	0xD0CF11E0A1B11AE1...

Z typu varbinary(MAX) sa vypíše len niekoľko bajtov v hexadecimálnom vyjadrení. Inverzná funkcia k BULK neexistuje, preto dokumenty, ktoré sme uložili do databázy používame priamo v aplikáciách.

Len pre zaujímavosť ukážeme, že v takto organizovaných dokumentoch je možné aplikovať aj fulltextové vyhľadávanie.

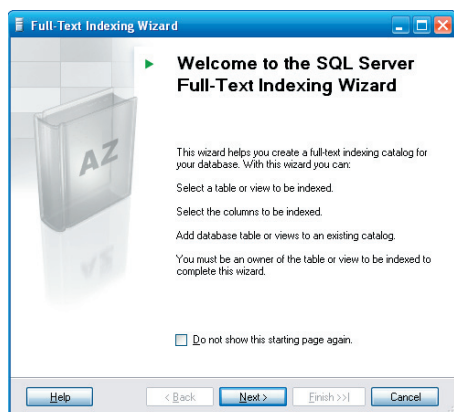
Ak chceme full - textové vyhľadávanie využívať v databáze, ktorá je už vytvorená použijeme pre povolenie tejto funkcionality uložení procedúru sp_fulltext_database.

```
USE Test;
GO
EXEC sp_fulltext_database ,enable`;
GO
```

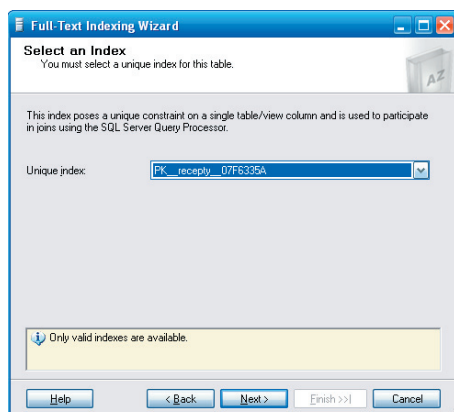
Fulltextové indexovanie môžeme aj kedykoľvek zakázať použitím tejto uloženej procedúry s parametrom

```
EXEC sp_fulltext_database ,disable`;
```

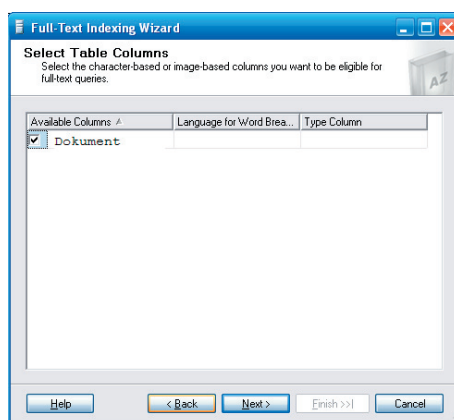
Potom pomocou sprievodcu SQL Server Full-Text Indexing Wizard vytvoríme index pre fulltext. Aktivujeme ho v kontextovom menu databázovej tabuľky v okne Object Explorer. Postup je dostatočne zrejmy zo série obrázkov



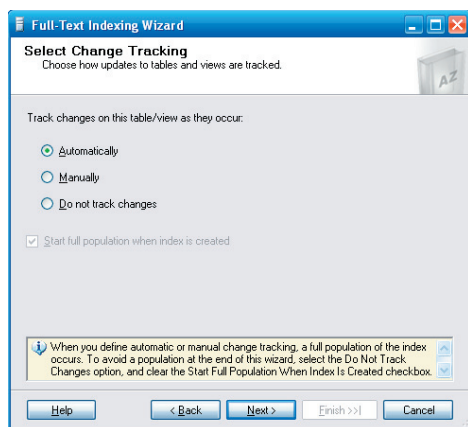
Sprievodca SQL Server Full-Text Indexing Wizard



Full-Text Indexing Wizard – výber unikátneho identifikátora



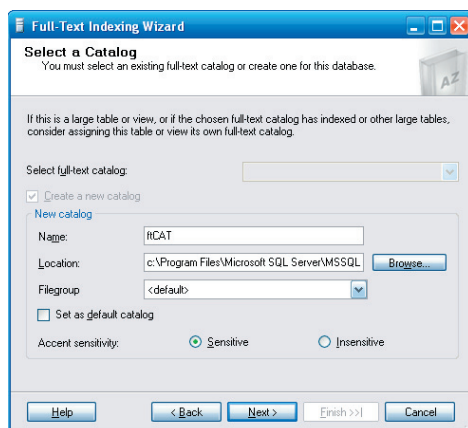
Full-Text Indexing Wizard – výber stĺpcov na ktoré chceme aplikovať full – textové vyhľadavania



Full-Text Indexing Wizard – Select Change Tracking

Indexový katalóg sme nazvali ftCAT. Nebude uložený v databáze, ale v súborovom systéme, v našom prípade pri implicitnej inštalácii konkrétne v adresári

c:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\FTData



Full-Text Indexing Wizard – Výber Full – textoveho katalogu

V predposlednom dialógu sprievodcu sú zhrnuté dosiaľ zadané parametre. Posledný dialóg je akčný, zobrazuje priebeh vytvárania indexu.

Aby sme sa presvedčili, že full text skutočne vyhľadáva v obsahu dokumentov a nie v ich názvoch, nedáme vyhľadávať slová, ktoré sa vyskytujú v názvoch rozprávok (dedko, repa, pes zajac), ale niečo čo sa vyskytuje v dokumente, v našom prípade v rozprávke. Určite si z detstva spomínate, že v „obchodnom reťazci“, v ktorom zháňal kohútik vodu pre sliepocku figurovala aj hus, ktorá mala dať pierko a chcela za to tuším trávu. Nuž poďme to zistiť.

```
SELECT ID, NazovSuboru FROM Dokumenty WHERE CONTAINS(Dokument, , "hus*" );
```

```
ID          NazovSuboru
-----
1           O sliepocke a kohutikovi.doc
```

Hus nefiguruje v žiadnom nadpise. No a tu je ako dôkaz citát z tejto rozprávky

- Kosec, kosec, daj trávy, - prosil ho.
- Komu trávy? - opýtal sa kosec.
- Husi trávu, aby hus pero dala, - vraví kohútik.
- Komu pero? - chcel vedieť kosec....

Možnosti full textového vyhľadávania SQL Servera 2005 sú oveľa širšie, syntax a použitie ostatných príkazov (CONTAINSTABLE, FREETEXT a FREETEXTTABLE...) si prípadní záujemcovia nájdu v nápovede. K SQL Serveru 2000 a 2005 existuje tzv. stemmer,

[\(http://www.microsoft.com/cze/windowsserverssystem/FullTextSearchModules/\)](http://www.microsoft.com/cze/windowsserverssystem/FullTextSearchModules/)

ktorý umožňuje vyhľadávať slová vo všetkých jazykových modifikáciách a gramaticky definovaných tvaroch. Bohužiaľ v poslednej dostupnej beta verzii SQL Serveru 2005 (júnové CTP) sa ešte nedal použiť. Podľa informácií z Microsoftu bude tento problém v ďalšej verzii opravený.

Triggery (spúšte) na DDL funkcie

V predchádzajúcej verzii SQL Servera 2000 bolo možné definovať AFTER trigger, teda spúšte, ktoré sa aktivujú po vykonaní príslušného úkonu len pre DML (Data Manipulation Language) príkazy teda INSERT, UPDATE, a DELETE. Verzia 2005 umožňuje definovať spúšte aj pre príkazy z množiny DDL (Data Definition Language).

Z pohľadu syntaxe by sme to mohli zapísať nasledovne:

```
CREATE TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
{ FOR | AFTER } { event_type | event_group } [ ,...n ]
AS { sql_statement [ ...n ] | EXTERNAL NAME < method specifier > }
[ ; ]
<method_specifier> ::=
assembly_name.class_name[.method_name]
```

V prvom príklade ukážeme vytvorenie bezpečnostného triggera, ktorý zabráni odstráneniu tabuľky

```
CREATE TABLE TestovaciaTabulka
(
    hodnota int NOT NULL
)
GO

CREATE TRIGGER poistka
ON DATABASE
FOR DROP_TABLE
AS
    PRINT ,Pre vymazanie tabuľky je potrebné zakázať spust DROP_TABLE !\
    ROLLBACK
GO
```

Pri pokuse o vymazanie tabuľky bude táto akcia pomocou funkcie ROLLBACK stornovaná.

```
DROP TABLE TestovaciaTabulka
```

Databázový server nám oznámi nasledovné skutočnosti.

```
Pre vymazanie tabuľky je potrebné zakázať spust DROP_TABLE !
Msg 3609, Level 16, State 2, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

V zozname objektov (po refreshi), prípadne dopytom zistíme, že tabuľka stále existuje a teda spúšť skutočne zabránila jej nechcenému odstráneniu.

Najskôr pre monitorované údaje vytvoríme databázovú tabuľku

```
CREATE TABLE EventLog
(
    cas datetime,
    pouzivatel nvarchar(100),
    udalost nvarchar(100),
    TSQL nvarchar(2000)
)
GO
```

a zmeníme trigger POISTKA

```
ALTER TRIGGER poistka
ON DATABASE
FOR DDL_DATABASE_LEVEL_EVENTS
AS
DECLARE @data xml
SET @data = eventdata()
INSERT EventLog (cas, pouzivatel, udalost, TSQL)
VALUES (GETDATE(), CONVERT(nvarchar(100), CURRENT_USER),
CONVERT(nvarchar(100), @data.query(, data(//EventType) `)),
CONVERT(nvarchar(2000), @data.query(, data(//TSQLCommand) `)))
GO
```

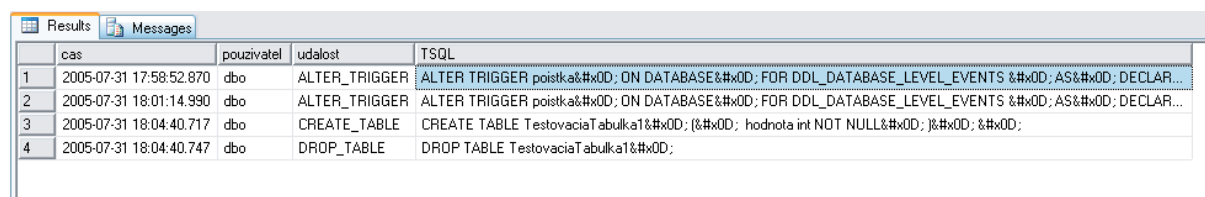
Teraz zopakujeme predchádzajúci pokus s vytvorením a odstránením tabuľky

```
CREATE TABLE TestovaciaTabulka1
(
    hodnota int NOT NULL
)

DROP TABLE TestovaciaTabulka1
GO
```

V monitorovacej tabuľky sú údaje ktoré tam uložiť trigger

```
SELECT * FROM EventLog
```



	cas	pouzivatel	udalost	TSQL
1	2005-07-31 17:58:52.870	dbo	ALTER_TRIGGER	ALTER TRIGGER poistka ON DATABASE FOR DDL_DATABASE_LEVEL_EVENTS  AS DECLAR...
2	2005-07-31 18:01:14.990	dbo	ALTER_TRIGGER	ALTER TRIGGER poistka ON DATABASE FOR DDL_DATABASE_LEVEL_EVENTS  AS DECLAR...
3	2005-07-31 18:04:40.717	dbo	CREATE_TABLE	CREATE TABLE TestovaciaTabulka1 ( hodnota int NOT NULL)
4	2005-07-31 18:04:40.747	dbo	DROP_TABLE	DROP TABLE TestovaciaTabulka1

Výpis monitorovacej tabuľky udalostí spúšťa

Klauzula OUTPUT

Zatiaľ čo SQL príkaz vracia len počet aktualizovaných riadkov, niekedy sa chceme o vykonávanej akcii dozvedieť viac, hlavne ktoré riadky boli zmenené. Samozrejme pomocou kurzora a premennej @@ identity. Klauzula OUTPUT popri hlavnom príkaze generuje ďalší výstup a vracia hodnoty cez premenné TABLE. Nie je potrebné pridávať dodatočnú programovú logiku, môžeme s výhodou použiť logické tabuľky na vrátenie stavov „pred/po“. Výsledky klauzuly OUTPUT môžete vložiť do tabuliek

Zjednodušený syntaktický predpis pre klauzulu OUTPUT:

```
<OUTPUT_CLAUSE> ::=
{
OUTPUT <dml_select_list> [ ,...n ]
INTO @table_variable
}
<dml_select_list> ::=
{ <column_name> | scalar_expression }
<column_name> ::=
{ DELETED | INSERTED | from_table_name } . { * | column_name }
```

Klauzulu OUTPUT je možné pridať ku každému príkazu okrem INSERT. Pre demonštráciu klauzuly OUTPUT si pripravíme tabuľku a naplníme niekoľkými údajmi.

```
CREATE TABLE PokusnaTabulka
(
    id INT IDENTITY,
    hodnota VARCHAR(15)
)
```

```
INSERT INTO PokusnaTabulka VALUES (,riadok_1`)
INSERT INTO PokusnaTabulka VALUES (,riadok_2`)
INSERT INTO PokusnaTabulka VALUES (,riadok_5`)
INSERT INTO PokusnaTabulka VALUES (,riadok_6`)
INSERT INTO PokusnaTabulka VALUES (,riadok_7`)
INSERT INTO PokusnaTabulka VALUES (,riadok_8`)
INSERT INTO PokusnaTabulka VALUES (,riadok_9`)
INSERT INTO PokusnaTabulka VALUES (,riadok_10`)
```

Najskôr budeme monitorovať vymazávanie. Vymažeme nejaké záznamy a pomocou klauzule OUTPUT uložíme vymazávané údaje do dočasnej tabuľky @DEL

```
DECLARE @del AS TABLE (vymazaneId INT, vymazanaHodnota VARCHAR(15))
DELETE PokusnaTabulka OUTPUT DELETED.id, DELETED.hodnota INTO @del
WHERE id < 3
SELECT * FROM @del
```

V tabuľke @DEL budú uložené vymazané údaje

```
vymazaneId  vymazanaHodnota
-----
1           riadok_1
2           riadok_2
```

Podobne môžeme monitorovať modifikáciu záznamov

```
DECLARE @zmenene TABLE
(id INT, StaraHodnota VARCHAR(15), NovaHodnota VARCHAR(15))
UPDATE PokusnaTabulka
SET hodnota = ,ZMENENE`
OUTPUT inserted.id, deleted.hodnota, inserted.hodnota
INTO @zmenene
WHERE id < 5
SELECT * FROM @zmenene
```

V tabuľke @ZMENENE budú uložené záznamy o zmenách

id	StaraHodnota	NovaHodnota
3	riadok_5	ZMENENE
4	riadok_6	ZMENENE

Service Broker

Nie je to tak dávno, čo sa najskôr nemeslo, a potom hromadne začal presadzovať pojem XML. Podobný osud zažívajú webové služby. Od možnosti hromadného nasadenia platformovo nezávislých webových služieb a formátu XML pre výmenu údajov v heterogénnom prostredí je už len krôčik k architektúre informačných systémov orientovanej na služby. Označujeme ju skratkou SOA (Service-Oriented Architecture)

SOA architektúru by sme mohli voľne definovať ako systém voľne previazaných asynchrónne bežiacich webových služieb. Ich prepojením získame vlastne akúsi formu komplexnejšej aplikácie. Hlavnou výhodou takto koncipovanej architektúry je značná flexibilita. Modulárnosť riešenia prináša aj jednoduchú a centralizovanú správu, upgrade a taktiež podporuje aj integráciu heterogénnych systémov. Veď služby konverzných a integračných modulov pre prispôbenie komunikácie s iným systémom môžu používať všetky aplikácie bežiacie na príslušnom systéme. To umožňuje napríklad flexibilné prispôbenie sa informačným systémom obchodných partnerov a znižovanie nákladov na integráciu. Vzhľadom na využívanie služieb podnikových informačných systémov (databázových a aplikačných serverov, systémov pre messaging a kolaboráciu) a postupne aplikovanú modularitu riešení, znižuje postupne rozsah vyvíjaných častí produktov a modulov, skraca sa doba vývoja a znižujú sa vývojárske tímy, ktoré tým pádom dokážu efektívnejšie spolupracovať. Pomocou modularity zvýšime aj flexibilitu systému. Moduly je možné v rámci riešenia preskupiť, prípadne nahradiť novšími. Ak nastane požiadavka prispôsobiť riešenie novým požiadavkám, alebo operačným podmienkam nezasiahne to celú aplikáciu, ale len niektoré moduly, ktorých sa zmeny priamo dotýkajú.

Základné princípy SOA

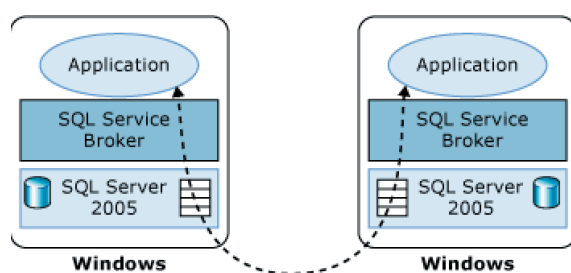
Pri implementácii architektúry orientovanej na služby vstupujú do hry tri strany

- poskytovateľ
- register služieb
- odberateľ

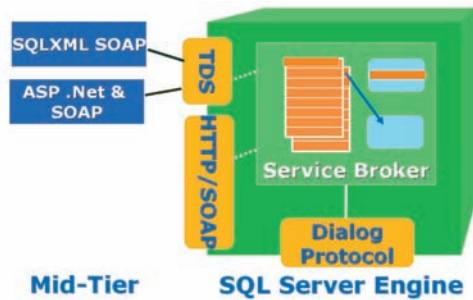
Poskytovateľom je sieťová aplikácia, ktorá má implementované potrebné sieťové funkcie. Bez **registra služieb** by sa o službe prakticky nikto nedozvedel. Preto aplikácia ako poskytovateľ služby posiela do registra informácie, čo príslušná služba vykonáva a kým spôsobom sa na ňu môže **odberateľ** napojiť. Odberateľom je spravidla iná aplikácia a táto sa o existencii služby dozvie práve z registra. Ako centrálny register pre webové služby slúži UDDI (Universal, Description, Discovery and Integration), prípadne môžeme zoznam služieb uložiť do nejakého dokumentu

Service Broker

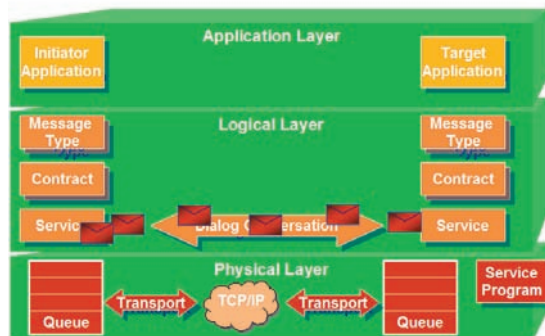
SQL Server 2005 prináša ako novinku nové distribuované rozhranie založené na službách – Service Broker. Uplatnenie nájde hlavne v podnikových databázových aplikáciách, kde nastáva potreba komunikácie rôznych blokov realizovaných spravidla na úrovni uložených procedúr.



SQL Princíp SQL Server Service Broker



SQL 2005 SOA architektúra



Architektúra Service Broker

Na schéme je architektúra Service Broker rozdelená do troch vrstiev

- aplikačná vrstva
- logická vrstva
- fyzická vrstva

Na najvyššej úrovni aplikačnej vrstvy sú dve aplikácie, prípadne aplikácia a uložená procedúra. Jedna z nich je iniciátorom posielania správy, pričom táto správa smeruje k druhej, cieľovej aplikácii. Na úrovni logickej vrstvy prebieha komunikácia medzi službami. Na tejto úrovni je definovaný „kontrakt“, to znamená spôsob reakcie na určitý typ správy. Napríklad ak nám zákazník pošle objednávku, tak mu potvrdíme jej prevzatie. Definovaný je aj typ správy, ktorá bude odoslaná, alebo ktorá sa očakáva. Na najnižšej úrovni fyzickej vrstvy sa organizujú fronty správ a zabezpečí sa ich doručovanie pomocou protokolov nižšej úrovne. Na rozdiel od logickej vrstvy, kde komunikácia prebieha pomocou dialógového protokolu sa na fyzickej vrstve komunikuje prostredníctvom TCP/IP protokolu.

Fungovanie Service Brokera budeme demonštrovať na jednoduchom ilustračnom príklade. Pre jeho realizáciu budeme potrebovať dve databázy. Môžeme si vytvoriť napríklad databázy s názvami Trader a Broker

Vytvoríme procedúry, ktoré budú spracovávať udalosti, v našom prípade sme vytvorili dve procedúry, ktoré pre jednoduchosť nerobia nič.

```
USE TraderDB
GO
CREATE PROCEDURE ProcPotvrdenie
AS
RETURN 0
GO
```

```
USE BrokerDB
GO
CREATE PROCEDURE ProcVstup
AS
RETURN 0
GO
```


Pomocou XML schemy alebo URL adresy definujeme typ správy a potvrdenia

```
create message type
[//www.firma.com/Broker/Vstup]
VALIDATION = WELL_FORMED_XML
```

```
create message type
[//www.firma.com/Broker/Sluzba]
VALIDATION = WELL_FORMED_XML
```

Distribúované rozhranie začneme budovať od najnižšej – fyzickej vrstvy. V databázach TraderDB a BrokerDB vytvoríme fronty

```
USE TraderDB
GO
CREATE QUEUE PotvrdenieQueue WITH STATUS = ON,
ACTIVATION (PROCEDURE_NAME = ProcPotvrdenie,MAX_QUEUE_READERS = 5,EXECUTE AS SELF)
```

```
USE BrokerDB
GO
CREATE QUEUE VstupQueue WITH STATUS = ON,
ACTIVATION (PROCEDURE_NAME = ProcVstup,MAX_QUEUE_READERS = 5,EXECUTE AS SELF)
```

Položka ACTIVATION je nepovinná, fronta bude fungovať aj bez uloženej procedúry. Následne je potrebné vytvoriť služby (URL adresy musia byť samozrejme platné, inak služba nebude vytvorená).

```
USE TraderDB
GO
CREATE SERVICE sVstup
ON QUEUE PotvrdenieQueue
([//www.firma.com/Broker/Vstup])
```

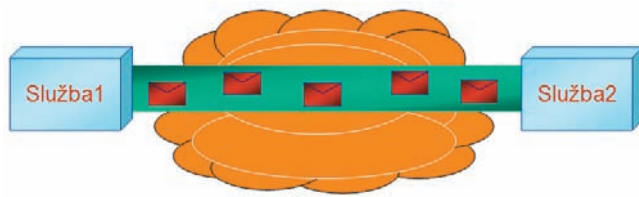
```
USE Broker DB
GO
CREATE SERVICE
[//www.firma.com/Broker/Sluzba]
ON QUEUE VstupQueue
([//www.firma.com/Broker/Vstup])
)
```

Po prípravných úkonoch sa môžeme presunúť na vyššiu, logickú vrstvu a zahájiť konverzáciu. Musíme definovať unikátny identifikátor pre komunikačnú skupinu

```
USE TraderDB
GO

DECLARE @id uniqueidentifier;
BEGIN DIALOG CONVERSATION @id
FROM SERVICE sVstup
TO SERVICE
, //www.firma.com/Broker/Sluzba `
ON CONTRACT
[//www.firma.com/Broker/Vstup];
```

Ak si to zrekapitulujeme, vytvorili sme komunikáciu medzi dvoma službami na úrovni správ. Situácia je znázornená na obrázku.



Posielanie správ medzi službami

Takáto komunikácia zatiaľ nemá veľký zmysel, je úplne samoúčelná. Ak si pozrieme na schéme architektúry logickú vrstvu, vytvorili sme len služby, ale zatiaľ neboli definované „kontrakty“, inými slovami, zatiaľ nie je definované na aké správy má aplikácia reagovať a samozrejme ani to ako má reagovať. Kontrakty na oboch stranách definujeme pomocou príkazu

```
CREATE CONTRACT
[...URL...]
( [...URL...]
SENT BY INITIATOR,
[...URL...]
SENT BY TARGET
)
```

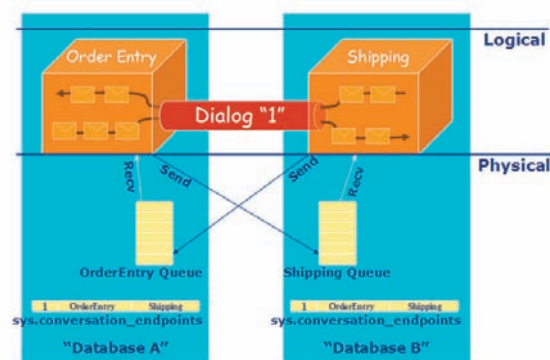
Správy sa posielajú pomocou konštrukcie

```
SEND ON CONVERSATION @id MESSAGE TYPE [...URL...] (@ExpenseRpt);
```

Pre ich prijímanie máme viac možností

```
RECEIVE * FROM PotvrdenieQueue;
WAITFOR(RECEIVE * FROM PotvrdenieQueue);
WAITFOR(RECEIVE * FROM PotvrdenieQueue) TIMEOUT 500;
```

Fungovanie Service Brokera na logickej úrovni medzi dvomi databázami pravdepodobne najlepšie pochopíme zo schématického obrázku



Posielanie správ na fyzickej a logickej vrstve

V porovnaní s inými komunikačnými technológiami prebieha komunikácia medzi aplikáciami a uloženými procedúrami prostredníctvom SSB na úrovni databázovej vrstvy. Naproti tomu MSMQ využíva ukladanie komunikačných frontov v operačnej pamäti. O perspektívach SSB do blízkej budúcnosti svedčí možnosť spolupráce na úrovni nového komunikačného rozhrania Indigo v novej generácii operačných systémov.

Kapitola 6: Ukladanie a práca s údajmi vo formáte XML

Pri hľadani vhodného formátu pre výmenu údajov hlavne medzi rôznymi aplikáciami a na rôznych platformách pre e-business sa zistilo, že týmto požiadavkám najviac vyhovuje jazyk XML (Extensible Markup Language). Pre komplexné prostredie informačných systémov zameraných na e-business sa stali typickými databázové a aplikačné prostredia a XML formát pre výmenu údajov. Podobne ako u ostatných moderných databázových serverov aj SQL Server vo verzii 2005 dokáže pracovať s XML dokumentom ako natívnym databázovým typom.

Výstup údajov z databázovej tabuľky v XML formáte

Základný príkaz SELECT pre výpis obsahu databázových tabuliek je

```
SELECT * FROM tabulka FOR XML mode [, XMLDATA] [, ELEMENTS][, BINARY BASE64]
```

pričom

- mode .. určuje tvar výsledku,
- XMLDATA .. určuje, že súčasťou dokumentu bude aj XML schéma.
- ELEMENTS .. v móde AUTO sa určuje, že údaje budú dávané ako elementy. Inak sú údaje v hodnotách atribútov.
- BINARY BASE64 .. určuje, že binárne dáta budú reprezentované vo formáte base64-encoded.

Za klauzulou FOR XML môžu nasledovať v parametre mode 4 typy modifikátorov

- RAW
- AUTO
- EXPLICIT
- PATH

Fungovanie modifikátorov vysvetlíme na dopyte

```
SELECT P.ProductID, P.Name, PSC.Name AS Subcategory
FROM Production.ProductSubcategory PSC
INNER JOIN Production.Product P
ON PSC.ProductSubCategoryID=P.ProductSubCategoryID
FOR XML RAW
```

V ktorom budeme postupne vymieňať modifikátory

RAW

Neumožňuje definovať formátovanie riadku, elementy sú radené v reťazci za sebou

Výsledkom dopytu je množina elementov, výpis sme pre prehľadnosť upravili tak aby v každom riadku bol len jeden element

```
<row ProductID="680" Name="HL Road Frame - Black, 58" Subcategory="Road Frames" />
<row ProductID="706" Name="HL Road Frame - Red, 58" Subcategory="Road Frames" />
<row ProductID="707" Name="Sport-100 Helmet, Red" Subcategory="Helmets" />
...
```

Všimnite si použitie aliasu PSC.Name AS Subcategory, pretože ak by dva stĺpce pochádzajúce z dvoch rôznych tabuliek mali rovnaký názov boli by vygenerované dva rovnaké atribúty v tom istom XML tagu, čo je neprípustné, takže by došlo k chybe

```
Msg 6810, Level 16, State 1, Line 4
Column name ,Name` is repeated. The same attribute cannot be generated more than
once on the same XML tag.
```

AUTO rešpektuje aliasy tabuliek aj hierarchiu pripojení typu v JOIN podľa poradia stĺpcov. Ak sa nad tým hlbšie nezamyslíme a zostavíme dopyt mechanicky

```
SELECT P.ProductID, P.Name, PSC.Name AS Subcategory
FROM Production.ProductSubcategory PSC
INNER JOIN Production.Product P
ON PSC.ProductSubCategoryID=P.ProductSubcategoryID
FOR XML AUTO
```

Výsledok bude trochu nelogický, pretože subkategória nie je podelementom produktu ale naopak produkt je podelementom subkategórie

```
<P ProductID="680" Name="HL Road Frame - Black, 58">
  <PSC Subcategory="Road Frames" />
</P>
<P ProductID="706" Name="HL Road Frame - Red, 58">
  <PSC Subcategory="Road Frames" />
</P>
<P ProductID="707" Name="Sport-100 Helmet, Red">
  <PSC Subcategory="Helmets" />
</P>
```

Z hľadiska aplikačnej logiky to bude správne takto:

```
SELECT PSC.Name AS Subcategory, P.ProductID, P.Name
FROM Production.ProductSubcategory PSC
INNER JOIN Production.Product P
ON PSC.ProductSubCategoryID=P.ProductSubcategoryID
FOR XML AUTO
```

```
<PSC Subcategory="Road Frames">
  <P ProductID="680" Name="HL Road Frame - Black, 58" />
  <P ProductID="706" Name="HL Road Frame - Red, 58" />
</PSC>
<PSC Subcategory="Helmets">
  <P ProductID="707" Name="Sport-100 Helmet, Red" />
  <P ProductID="708" Name="Sport-100 Helmet, Black" />
</PSC>
<PSC Subcategory="Socks">
  <P ProductID="709" Name="Mountain Bike Socks, M" />
  <P ProductID="710" Name="Mountain Bike Socks, L" />
</PSC>
<PSC Subcategory="Helmets">
  <P ProductID="711" Name="Sport-100 Helmet, Blue" />
</PSC>
<PSC Subcategory="Caps">
  <P ProductID="712" Name="AWC Logo Cap" />
</PSC>
```

Vidíme, že elementy pripojenej tabuľky, napríklad <P ProductID="711" Name="Sport-100 Helmet, Blue" /> sú vnorené v hierarchii hlavného elementu

Ak za modifikátormi RAW, AUTO, EXPLICIT alebo PATH použijeme ešte modifikátor ELEMENTS

FOR XML AUTO, ELEMENTS

bude výstup obsahovať namiesto atribútov elementy a teda bude v tvare

```
<PSC>
  <Subcategory>Road Frames</Subcategory>
  <P>
    <ProductID>680</ProductID>
    <Name>HL Road Frame - Black, 58</Name>
  </P>
  <P>
    <ProductID>706</ProductID>
    <Name>HL Road Frame - Red, 58</Name>
  </P>
</PSC>
<PSC>
  <Subcategory>Helmets</Subcategory>
  <P>
    <ProductID>707</ProductID>
    <Name>Sport-100 Helmet, Red</Name>
  </P>
  <P>
    <ProductID>708</ProductID>
    <Name>Sport-100 Helmet, Black</Name>
  </P>
</PSC>
```

Ak použijeme modifikátor ROOT, bude výstup odpovedať správne formátovanému XML dokumentu, ktorý musí mať práve jeden koreňový element

FOR XML AUTO, ELEMENTS, ROOT('KatalogProduktov')

bude výstup obsahovať okrem elementov <P> pre jednotlivé produkty obsahovať aj koreňový element

```
<KatalogProduktov>
  <PSC>
    <Subcategory>Road Frames</Subcategory>
    <P>
      <ProductID>680</ProductID>
      <Name>HL Road Frame - Black, 58</Name>
    </P>
    <P>
      <ProductID>706</ProductID>
      <Name>HL Road Frame - Red, 58</Name>
    </P>
  </PSC>
  ...
</KatalogProduktov>
```

Za modifikátormi RAW, AUTO, EXPLICIT alebo PATH môžeme použiť ďalšie dva modifikátory XMLSCHEMA - tento generuje inline XSD schému dokumentu a XSINIL ktorý generuje výstup aj pre null elementy.

EXPLICIT tento zastaralý klasifikátor s veľmi zložitou konštrukciou je v novej verzii databázového servera nahradený novou alternatívou PATH, takže sa ním ani nebudeme zaoberať

PATH pri použití tohto klasifikátora mená stĺpcov určujú XPATH polohu v XML stromovej štruktúre

```
SELECT PSC.ProductSubcategoryID ,@SubCategoryID`,PSC.Name ,Name`,
```

```
P.ProductID ,Product/@ProductID`,P.ProductNumber ,Product/Number`,P.Name ,Product/
Name`,P.ReorderPoint ,Product/ReorderPoint `
FROM Production.ProductSubcategory PSC
INNER JOIN Production.Product P
ON P.ProductSubCategoryID=PSC.ProductSubcategoryID
FOR XML PATH(,SubCategory`),ROOT(,Katalog`)
```

Výsledok bude v tvare

```
<Katalog>
  <SubCategory SubCategoryID="14">
    <Name>Road Frames</Name>
    <Product ProductID="680">
      <Number>FR-R92B-58</Number>
      <Name>HL Road Frame - Black, 58</Name>
      <ReorderPoint>375</ReorderPoint>
    </Product>
  </SubCategory>
  <SubCategory SubCategoryID="14">
    <Name>Road Frames</Name>
    <Product ProductID="706">
      <Number>FR-R92R-58</Number>
      <Name>HL Road Frame - Red, 58</Name>
      <ReorderPoint>375</ReorderPoint>
    </Product>
  </SubCategory>
```

Na prvý pohľad vidíme nevýhodu takéhoto výpisu. Neakceptuje hierarchickú štruktúru subkategórií a produktov. Pre vnorené elementy je výhodnejšie použiť upravený dopyt, v ktorom bude koreňovým atribútom na najvyššej úrovni hierarchie ,Katalog`, Na nižšom stupni hierarchie budú subkategórie produktov

```
SELECT ProductSubcategoryID ,@SubCategoryID`,Name ,Name`,
      (SELECT P.ProductID ,@ProductID`,P.ProductNumber ,Number`,P.Name ,Name`,P.
ReorderPoint ,ReorderPoint `
      FROM Production.Product P
      WHERE P.ProductSubCategoryID=PSC.ProductSubcategoryID
      FOR XML PATH(,Product`),TYPE)
FROM Production.ProductSubcategory PSC
FOR XML PATH(,SubCategory`),ROOT(,Katalog`)
```

Potom výsledok bude v hierarchicky správnom tvare podľa kategórií

```
<Katalog>
  <SubCategory SubCategoryID="18">
    <Name>Bib-Shorts</Name>
    <Product ProductID="855">
      <Number>SB-M891-S</Number>
      <Name>Men`s Bib-Shorts, S</Name>
      <ReorderPoint>3</ReorderPoint>
    </Product>
    <Product ProductID="856">
      <Number>SB-M891-M</Number>
      <Name>Men`s Bib-Shorts, M</Name>
      <ReorderPoint>3</ReorderPoint>
    </Product>
    <Product ProductID="857">
      <Number>SB-M891-L</Number>
```

```

    <Name>Men`s Bib-Shorts, L</Name>
    <ReorderPoint>3</ReorderPoint>
  </Product>
</SubCategory>
<SubCategory SubCategoryID="26">
  <Name>Bike Racks</Name>
  <Product ProductID="876">
    <Number>RA-H123</Number>
    <Name>Hitch Rack - 4-Bike</Name>
    <ReorderPoint>3</ReorderPoint>
  </Product>
</SubCategory>

```

XML dátový typ

Po oboznámení sa s možnosťou výstupu údajov z bežných tabuliek vo formáte XML podrobnejšie predstavíme natívny XML dátový typ. Môžeme ho použiť ako:

- typ stĺpca v tabuľke
- typ parametra v uloženej procedúre
- typ návratovej hodnoty v „user-defined function“
- typ premennej

Aj keď je XML dokument vo formáte textu, XML dátový typ nie je svojou povahou textový, preto niektoré operácie bežné u textových dátových typov nepodporuje. Je rozložený do vnútorných relačných tabuliek databázy. Okrem toho, že stĺpec XML dátového typu nie je možné použiť ako primárny kľúč, nepodporuje textové porovnávanie, operátory GROUP BY a ORDER BY a obmedzenie UNIQUE.

Pre prvé pokusy s dátovým typom XML si vytvoríme pre tento dátový typ premennú.

```

DECLARE @premenna xml
SET @premenna = '<A></A><B>bbb</B>'
SELECT @premenna AS premenna
SELECT CONVERT(nvarchar(1000), @premenna) AS premenna_ako_retazec

```

Všimnite si, že reprezentácia prázdneho tagu je odlišná od vstupného reťazca

```

premenna
-----
<A /><B>bbb</B>

```

```

premenna_ako_retazec
-----
<A/><B>bbb</B>

```

Môžeme sa pokúsiť databázovému serveru podvrhnúť aj non-well formatted XML dokument, kde počiatočný a koncový element nie sú rovnaké

```

DECLARE @premenna xml
SET @premenna = '<A></AB>'

```

no skončí to samozrejme nezdarom

```

Msg 9436, Level 16, State 1, Line 2
XML parsing: line 1, character 8, end tag does not match start tag

```

Pre demonštráciu základných možností práce s XML dokumentmi a natívnym XML dátovým typom vytvoríme jednoduchú cvičnú tabuľku, ktorá obsahuje dva stĺpce. V jednom je ID záznamu ako dátový typ integer, druhý stĺpec bude obsahovať natívny XML dátový typ

```
CREATE TABLE xml_tabulka
(
  id INT,
  xml_stlpec XML
)
```

To takto navrhutej tabuľky vložme jeden záznam. Všimnite si, že do dátového typu XML vkladáme textový reťazec. Je to v poriadku, však XML dokument je dokument textovej povahy. V priebehu vkladania prebehne validácia a auto konverzia textového reťazca na XML dokument.

```
INSERT INTO xml_tabulka VALUES (1, ,<doc/>`)
```

Ak textový reťazec neobsahuje platný XML dokument (tagy X1 a X2 sú vo vnútri dokumentu navzájom prehodené), autokonverzia neprebehne a príkaz pre vkladanie záznamov skončí chybovým hlásením.

```
INSERT INTO xml_tabulka VALUES (2, ,<doc><x1><x2></x1></x2></doc>`)
```

```
Msg 9436, Level 16, State 1, Line 1
XML parsing: line 1, character 18, end tag does not match start tag
```

XML stĺpec môže obsahovať iba „well-formed“ XML podľa špecifikácie XML 1.0. Môže obsahovať dokumenty alebo ich časti

Pri výpise pomocou príkazu SELECT sa výsledok zobrazí v tvare

```
SELECT * FROM xml_tabulka;

id          xml_stlpec
-----
1          <doc />
```

Prípadne môžeme použiť explicitnú konverziu pomocou príkazu CAST

```
SELECT CAST(xml_stlpec AS VARCHAR(MAX)) FROM xml_tabulka
```

Stĺpec dátového typu nemôže byť primárnym kľúčom. Pokus o vytvorenie takéhoto stĺpca sa skončí chybou

```
CREATE TABLE xml_tabulka
(
  xml_stlpec XML PRIMARY KEY
)
```

```
Msg 1919, Level 16, State 1, Line 1
Column ,xml_stlpec` in table ,xml_tabulka` is of a type that is invalid for use as
a key column in an index.
```

V XML stĺpci môžeme definovať aj DEFAULT hodnoty napríklad

```
create table faktury
(
  id INT,
  dokument XML DEFAULT ,<invoice/>`
)
```


Potom pri vkladaní použijeme kľúčové slovo DEFAULT

```
INSERT faktury VALUES (1, DEFAULT)
```

Už z doterajších pokusov je zrejmé, že XML stĺpec nie je len bežný textový stĺpec, v pozadí je podporovaná široká paleta XML technológií. Obsah stĺpca môže byť validovaný voči XML Schéme, podporuje XML indexy, XQuery a XPath 2.0. Obsah stĺpca je konzistentný s XML funkcionalitou databázy FOR XML a OpenXML

```
SELECT CAST(xml_stlpec AS VARCHAR(MAX)) FROM xml_tabulka
```

```
-----
<doc/>
<doc><x1><x2/></x1></doc>
```

Naplnenie premennej dátového typu XML z databázovej tabuľky

Po predchádzajúcich ukázkach si už celkom dobre dokážeme predstaviť možnosti dátového typu XML v SQL Serveri 2005. Jednou z častých úloh pred ktoré budeme možno postavení je aj naplnenie premennej dátového typu XML z databázovej tabuľky. Ukážeme si to na príklade, kedy do XML dokumentu, presnejšie do premennej XML dátového typu uložíme údaje z databázovej tabuľky Person.Address cvičnej databázy Adventure Works. Tabuľka obsahuje údaje v tvare (vo výpise sú len niektoré stĺpce)

```
SELECT * FROM Person.Address
```

AddressID	AddressLine1	City	StateProvinceID	PostalCode
1	1970 Napa Ct.	Bothell	79	98011
2	9833 Mt. Dias Blv.	Bothell	79	98011
3	7484 Roundtree Drive	Bothell	79	98011
4	9539 Glenside Dr	Bothell	79	98011
5	1226 Shoe St.	Bothell	79	98011
6	1399 Firestone Drive	Bothell	79	98011
7	5672 Hale Dr.	Bothell	79	98011

Vytvoríme premennú X dátového typu XML a naplníme ju údajmi z tabuľky.

```
DECLARE @x XML
SET @x = (SELECT AddressLine1, City, StateProvinceID, PostalCode
FROM Person.Address WHERE AddressID = 1 FOR XML AUTO, ELEMENTS, TYPE)
SELECT @x
```

Premenná X bude obsahovať jeden záznam vo forme XML dokumentu

```
<Person.Address>
  <AddressLine1>1970 Napa Ct.</AddressLine1>
  <City>Bothell</City>
  <StateProvinceID>79</StateProvinceID>
  <PostalCode>98011</PostalCode>
</Person.Address>
```

Vloženie záznamov do stĺpca tabuľky XML dátového typu

Po úvodnej rozcvičke si trúfneme aj na zložitejší príklad. Pre praktické overenie práce s údajmi vo formáte XML si v databáze AdventureWorks vytvoríme tabuľku, ktorá bude obsahovať jeden stĺpec s dátovým typom XML. Nakoľko všetky názvy v databáze AdventureWorks sú v angličtine, budú aj názvy objektov vytvorených v cvičeniach tejto publikácie v angličtine.

```
CREATE TABLE ProductDocs
(
    ID INT IDENTITY PRIMARY KEY,
    ProductDoc XML NOT NULL
)
```

Do tabuľky vložíme záznam

```
INSERT INTO ProductDocs VALUES
(,
    <Product>
        <ProductID>1</ProductID>
        <ProductName>Chai</ProductName>
        <SupplierID>1</SupplierID>
        <CategoryID>1</CategoryID>
        <QuantityPerUnit>10 boxes x 20 bags</QuantityPerUnit>
        <UnitPrice>18.0000</UnitPrice>
        <UnitsInStock>39</UnitsInStock>
        <UnitsOnOrder>0</UnitsOnOrder>
        <ReorderLevel>10</ReorderLevel>
        <Discontinued>0</Discontinued>
    </Product>
,)
```

V Management Studiu si môžeme nechať zobrazíť vlastnosti stĺpca ProductDoc. Vložené údaje môžeme nechať vypísať SQL dopytom

```
SELECT ID, ProductDoc FROM ProductDocs WHERE ID = 1
```

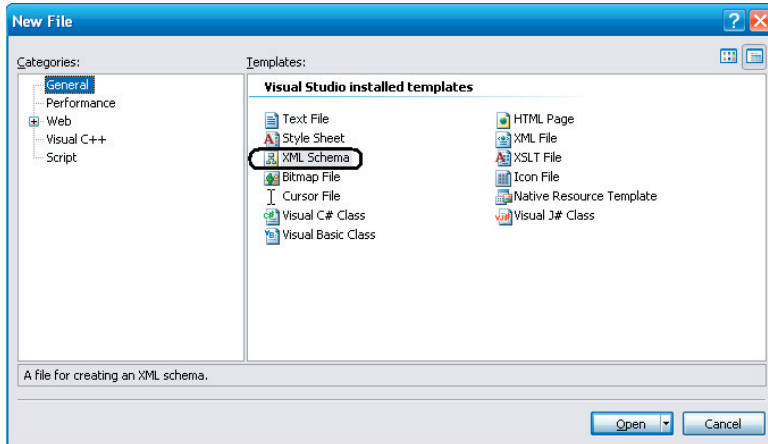
Výsledok dopytu v textovom režime bude v tvare

ID	ProductDoc
1	<pre><Product> <ProductID>1</ProductID> <ProductName>Chai</ProductName> <SupplierID>1</SupplierID> <CategoryID>1</CategoryID> <QuantityPerUnit>10 boxes x 20 bags</QuantityPerUnit> <UnitPrice>18.0000</UnitPrice> <UnitsInStock>39</UnitsInStock> <UnitsOnOrder>0</UnitsOnOrder> <ReorderLevel>10</ReorderLevel> <Discontinued>0</Discontinued> </Product></pre>

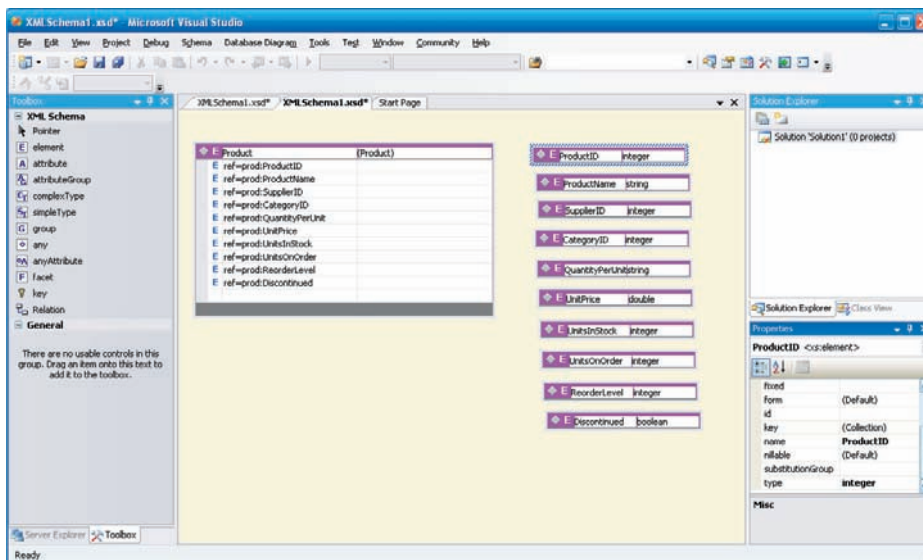
XML schéma

XML schéma opisuje usporiadanie súčastí XML dokumentu a definuje ich typy. XML schémy môžu byť asociované s XML typom. Pomocou nich verifikujeme či údaje vložené do stĺpca dátového typu XML sedia so schémou. Sadu schém označujeme ako „schema collection“

Schémy môžeme vytvárať napríklad vo vývojovom prostredí Visual Studio 2005

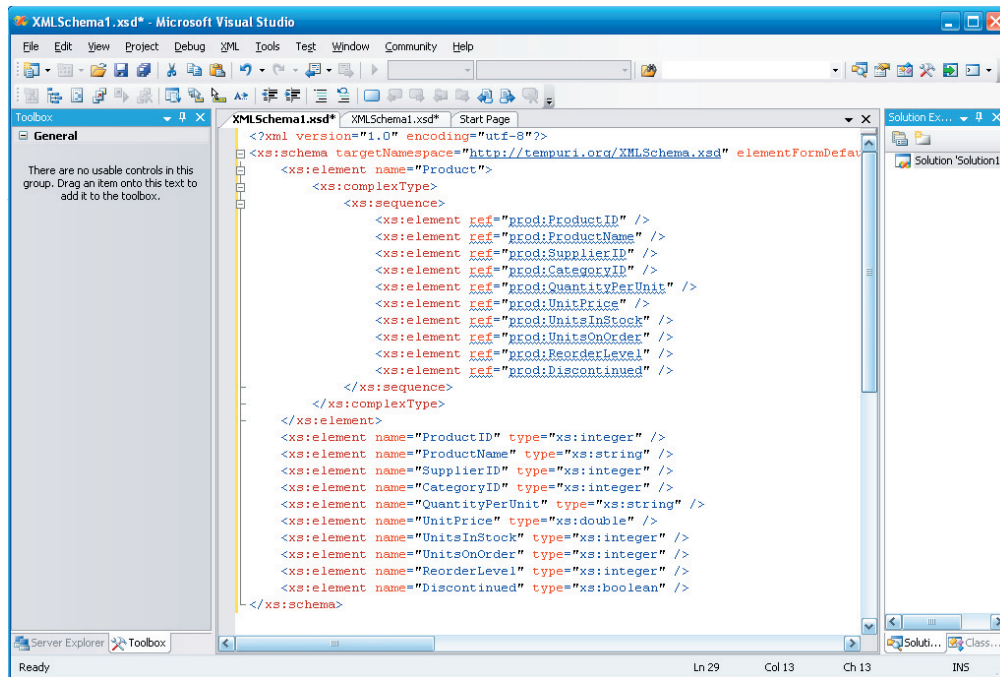


Vytvorenie nového súboru typu XML schéma vo vývojovom prostredí Visual Studio 2005



XML schéma v grafickom návrhovom prostredí Visual Studio 2005

V pracovnom priestore vývojového prostredia sú tri okná – paleta nástrojov (Toolbox), okno pohľadu na XML schému a okno vlastností (Properties). Z palety nástrojov sa dajú ponúknuté komponenty umiestniť v okne pohľadu a tak schému rozširovať. V okne vlastností je možné definovať vlastnosti prvku, ktorý je označený v okne pohľadu na XML schému. Pri pohľade text XML schémy si všimnite, že názvy prvkov začínajú trojicou znakov „xs:“. Znamená to, že sa používa priestor mien (Namespace) xs.



Zdrojový text XML schémy v prostredí Visual Studio 2005

XML schéma pre verifikáciu musí byť uložená v databáze. V našom príklade schému (vytvorenú napríklad vo vývojovom prostredí Visual Studio 2005) v databáze vytvoríme pomocou príkazu CREATE XML SCHEMA COLLECTION, v ktorom je celý text definície schémy

```
CREATE XML SCHEMA COLLECTION ProductSchema AS ,
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.microsoft.com/schemas/adventure-works/products"
xmlns:prod="http://www.microsoft.com/schemas/adventure-works/products">
  <xs:element name="Product">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="prod:ProductID" />
        <xs:element ref="prod:ProductName" />
        <xs:element ref="prod:SupplierID" />
        <xs:element ref="prod:CategoryID" />
        <xs:element ref="prod:QuantityPerUnit" />
        <xs:element ref="prod:UnitPrice" />
        <xs:element ref="prod:UnitsInStock" />
        <xs:element ref="prod:UnitsOnOrder" />
        <xs:element ref="prod:ReorderLevel" />
        <xs:element ref="prod:Discontinued" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="ProductID" type="xs:integer" />
  <xs:element name="ProductName" type="xs:string" />
  <xs:element name="SupplierID" type="xs:integer" />
  <xs:element name="CategoryID" type="xs:integer" />
  <xs:element name="QuantityPerUnit" type="xs:string" />
  <xs:element name="UnitPrice" type="xs:double" />
  <xs:element name="UnitsInStock" type="xs:integer" />
  <xs:element name="UnitsOnOrder" type="xs:integer" />
  <xs:element name="ReorderLevel" type="xs:integer" />
  <xs:element name="Discontinued" type="xs:boolean" />
</xs:schema>
```

```

    <xs:element name="ReorderLevel" type="xs:integer" />
    <xs:element name="Discontinued" type="xs:boolean" />

</xs:schema>
,

SELECT * FROM sys.xml_namespaces

```

Teraz, keď máme vytvorenú schému vytvoríme novú tabuľku, pričom túto schému využijeme. Starú tabuľku odstránime príkazom DROP TABLE

```

DROP TABLE ProductDocs
GO

CREATE TABLE ProductDocs
(
    ID INT IDENTITY PRIMARY KEY,
    ProductDoc XML(ProductSchema) NOT NULL
)
GO

```

Do novovytvorenej tabuľky vložíme údaje, no vynechajme hrubo vytlačený riadok

```

INSERT INTO ProductDocs VALUES
(,
    <Product>
        <ProductID>1</ProductID>
        <ProductName>Chai</ProductName>
        <SupplierID>1</SupplierID>
        <CategoryID>1</CategoryID>
        <QuantityPerUnit>10 boxes x 20 bags</QuantityPerUnit>
        <UnitPrice>18.0000</UnitPrice>
        <UnitsInStock>39</UnitsInStock>
        <UnitsOnOrder>0</UnitsOnOrder>
        <ReorderLevel>10</ReorderLevel>
        <Discontinued>0</Discontinued>
    </Product>
, )

```

Dôsledkom čoho je, že vo vkladaných údajoch chýba element **<ProductName>**.

Nakoľko štruktúra vkladaných údajov nezodpovedá XML schéme dôjde k chybe, ktorá sa prejaví chybovým hlásením

```

Msg 6913, Level 16, State 1, Line 1
XML Validation: Declaration not found for element ,Product`. Location: /*:Product[1]

```

Ak ponecháme v príkaze INSERT INTO ProductDocs... všetky elementy predpísané schémou, príkaz sa vykoná bez chýb.

XML indexy

Nad XML stĺpcami môžeme vytvoriť špeciálne XML indexy, ktoré optimalizujú XML požiadavky nad príslušným stĺpcom. Tabuľka alebo pohľad, v ktorej indexujeme XML dátový typ musí mať primárny kľúč typu „clustered“. Najskôr musí byť vytvorený primárny XML index. Kompozitný XML index nie je podporovaný.

```

CREATE TABLE xml_tab
(
  id integer primary key,
  doc xml
)
GO
CREATE PRIMARY XML INDEX xml_idx on xml_tab (doc)
GO

```

SQL Server 2005 podporuje 3 špecializované typy XML indexov: VALUE, PATH a PROPERTY. Uvádzame len príklady pre ich vytvorenie. Podrobnosti k jednotlivým typom indexov sú v dokumentácii

```

-- value
CREATE XML INDEX xmlv1 ON xml_tab(doc)
USING XML INDEX xml_idx FOR VALUE
GO
-- path
CREATE XML INDEX xmls1 ON xml_tab(doc)
USING XML INDEX xml_idx FOR PATH
GO
-- property
CREATE XML INDEX xmlp1 ON xml_tab(doc)
USING XML INDEX xml_idx FOR PROPERTY
GO

```

XQuery

Klasický databázový jazyk SQL nie je pre dopytovanie vo vnútri XML dátových typov príliš vhodný. Pre tento účel je v SQL Serveri 2005 implementovaný jazyk XQuery. Jeho základom metódy XML typu

- xml.exist
- xml.value
- xml.query
- xml.modify
- xml.nodes

spolu v kombinácii s FLOWR príkazmi a konštrukciami. FLOWR je akronym od hlavných kľúčových slov XQuery výrazov: FOR, LET, WHERE, ORDER BY a RETURN Ak sa tieto XQuery metódy použijú s SQL príkazom SELECT môžu vracaať stĺpce v „rowset-och“.

FLOWR príkazy

Príkaz	Popis
for	Iteruje cez „súrodenecké“ uzly
let	Priradenie (nie je podporované)
order by	kritérium pre utriedenie
where	Filtrovacie kritérium pre iteráciu
return	špecifikuje návratové XML

Využijeme podobnú tabuľku ako v predchádzajúcej stati a uložíme do nej tri záznamy s XML dokumentmi obsahujúcimi po jednom elemente a jeden záznam, ktorý má v XML dokumente štyri elementy,

```
DROP TABLE xml_tabulka
```

```
CREATE TABLE xml_tabulka
```

```
(
  id INT IDENTITY PRIMARY KEY,
  xml_stlpec XML
)
```

```
INSERT INTO xml_tabulka
```

```
VALUES (, <tovar><naradie><nazov>hoblík</nazov></naradie></tovar>')
```

```
INSERT INTO xml_tabulka
```

```
VALUES (, <tovar><naradie><nazov>vodováha</nazov></naradie></tovar>')
```

```
INSERT INTO xml_tabulka
```

```
VALUES (, <tovar><naradie><nazov>píla</nazov></naradie></tovar>')
```

```
INSERT INTO xml_tabulka VALUES
```

```
(
  , <tovar>
    <naradie nazov = "klieste" />
    <naradie nazov = "kladivo" />
    <naradie nazov = "pilník" />
    <naradie nazov = "vrtačka" />
  </tovar> `
)
```

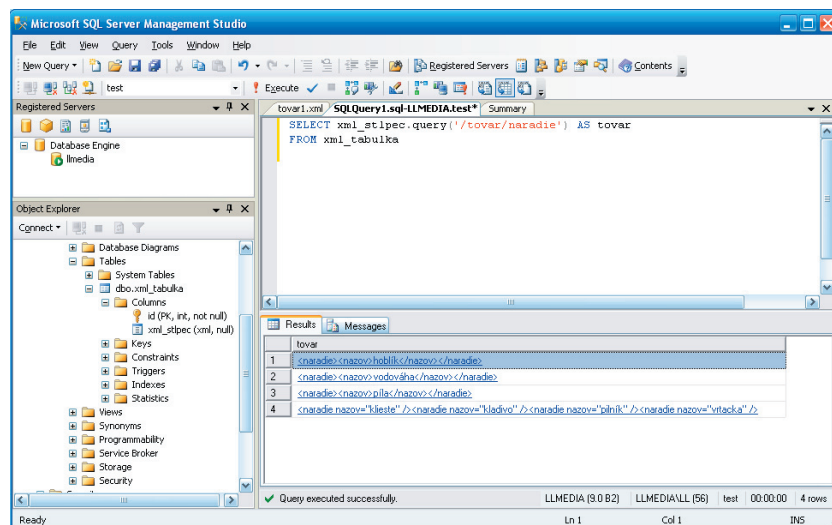
xml.query

Najskôr ukážeme jednoduchý dopyt s využitím metódy xml.query bez programových konštrukcií

```
SELECT xml_stlpec.query(/tovar/naradie`) AS tovar
FROM xml_tabulka
```

```
tovar
```

```
-----
<naradie><nazov>hoblík</nazov></naradie>
<naradie><nazov>vodováha</nazov></naradie>
<naradie><nazov>píla</nazov></naradie>
<naradie nazov="klieste" /><naradie nazov="kladivo" /><naradie nazov="pilník" />
</naradie nazov="vrtačka" />
```



Zobrazenie XML v tabuľke vo forme odkazu

Ak necháme zobrazíť výsledky v tabuľke, zobrazia sa odkazy. Po kliknutí na ne sa v novom okne zobrazia elementy XML dokumentu, v našom prípade pre odkazy s jedným elementom

```
<naradie>
  <nazov>hoblík</nazov>
</naradie>
```

Pre záznam obsahujúci viac elementov bude tento výpis v tvare

```
<naradie nazov="klieste" />
<naradie nazov="kladivo" />
<naradie nazov="pilník" />
<naradie nazov="vrtacka" />
```

Aby sme postupne prenikli do tajov vytvárania kódov s použitím FLOWR príkazov napíšeme jednoduchý skript s využitím premennej a FLOWR príkazov FOR a RETURN

```
SELECT xml_stlpec.query
(
  for $x in /tovar/naradie
  return ($x)
,
)
FROM xml_tabulka
```

tovar

```
-----
<naradie><nazov>hoblík</nazov></naradie>
<naradie><nazov>vodováha</nazov></naradie>
<naradie><nazov>píla</nazov></naradie>
<naradie nazov="klieste" /><naradie nazov="kladivo" /><naradie nazov="pilník"
/><naradie nazov="vrtacka" />
```

V XQuery dopyte môžeme zostrojiť aj podmienku s využitím klauzuly WHERE

```
SELECT id, xml_stlpec.query
(
  for $x in /tovar/naradie
  where $x/@nazov = „klieste“
  return ($x)
,
) AS naradie
FROM xml_tabulka
```

```
id          naradie
-----
1
2
3
4          <naradie nazov="klieste" />
```

Požadovaný element môžeme vypísať aj pomocou XPath predikátu

```
SELECT id, xml_stlpec.query(/tovar/naradie[@nazov = „klieste“]) AS naradie
FROM xml_tabulka
```



```

id          naradie
-----
1
2
3
4          <naradie nazov="klieste" />

```

xml.exist

Metóda nám vráti, či daný element v stĺpci s XML dátovým typom existuje alebo nie. V prípade existencie elementu vráti príkaz hodnotu 1, v opačnom prípade hodnotu 0. Najskôr overíme, či máme v XML dokumente záznam o kladive.

```

SELECT xml_stlpec.exist(/tovar/naradie[@nazov="kladivo"]) AS je_naradie
FROM xml_tabulka

```

```

je_naradie
-----
0
0
0
1

```

V tomto prípade hodnota 1 znamená úspech a teda existenciu elementu. Skúsme sa opýtať na element, ktorý v dokumente nemáme

```

SELECT xml_stlpec.exist(/tovar/naradie[@nazov="vodováha"]) AS je_naradie
FROM xml_tabulka

```

```

je_naradie
-----
0
0
0
0

```

Dopyt s využitím metódy `xml.exist` môžeme formulovať aj inak. Napríklad chceme zistiť ID väznamu, ktorý v XML dátovom type obsahuje príslušnú hodnotu

```

SELECT id FROM xml_tabulka
WHERE xml_stlpec.exist(/tovar/naradie[@nazov="kladivo"]) = 1

```

```

id
-----
4

```

xml.value

Metóda nám vráti, hodnotu príslušného prvku. Prvok je určený indexami v hranatých zátvorkách.

```

SELECT id,
xml_stlpec.value(/tovar[1]/naradie[3]/@nazov', , varchar(15)) AS nazov
FROM xml_tabulka

```

id	nazov
1	NULL
2	NULL
3	NULL
4	pilník

xml.nodes

Pomocou metódy `xml.nodes` vyberáme príslušné uzly. Môžeme použiť klauzulu `CROSS APPLY` alebo `OUTER APPLY`. V prvom prípade použijeme klauzulu `CROSS APPLY`. Zobrazia sa len záznamy obsahujúce dokumenty s jedným elementom

```
SELECT id, x.value('text()[1]', ,VARCHAR(15)') AS nazov
FROM xml_tabulka
CROSS APPLY
xml_stlpec.nodes('/tovar/naradie/nazov') AS result(x)
```

id	nazov
1	hoblík
2	vodováha
3	píla

Ak použijeme klauzulu `OUTER APPLY`, zobrazia sa všetky elementy

```
SELECT id, x.value('text()[1]', ,VARCHAR(15)') AS nazov
FROM xml_tabulka
OUTER APPLY
xml_stlpec.nodes('/tovar/naradie/nazov') AS result(x)
```

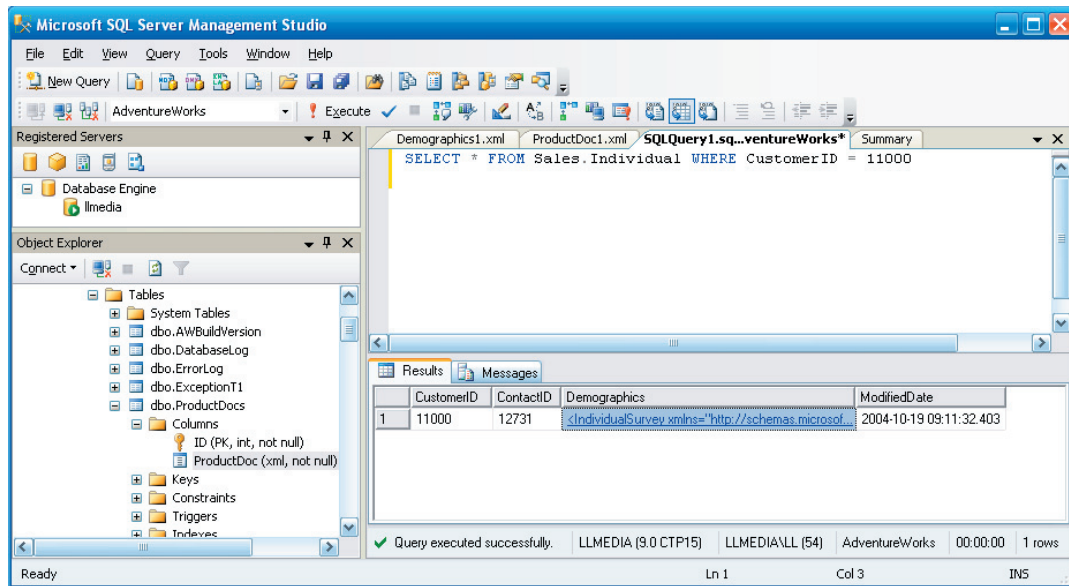
id	nazov
1	hoblík
2	vodováha
3	píla
4	NULL

Dopytovanie v tabuľkách cvičnej databázy AdventureWorks obsahujúcich XML dátový typ

Všimnime si v databáze AdventureWorks pozornejšie tabuľku `Sales.Individual`. V stĺpci `Demographics`, ktorý je dátového typu XML sú demografické informácie získané z dotazníkov vrátane informácií o hobby a podobne. Do tejto tabuľky budú smerovať naše dopyty.

Príklad výsledkov individuálneho dotazníka pre zákazníka s ID 11000 získame SQL dopytom

```
SELECT * FROM Sales.Individual WHERE CustomerID = 11000
```



Vlastnosti stĺpca dátového typu XML

Ak si necháme výsledok zobrazit' vo forme tabuľky, v stĺpci Demographic, ktorý je navrhnutý pre ukladanie dátového typu XML bude odkaz na zobrazenie XML dokumentu v XML editore

```
IndividualSurvey xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/IndividualSurvey">
```

```
  <TotalPurchaseYTD>8248.99</TotalPurchaseYTD>
  <DateFirstPurchase>2001-07-22Z</DateFirstPurchase>
  <BirthDate>1966-04-08Z</BirthDate>
  <MaritalStatus>M</MaritalStatus>
  <YearlyIncome>75001-100000</YearlyIncome>
  <Gender>M</Gender>
  <TotalChildren>2</TotalChildren>
  <NumberChildrenAtHome>0</NumberChildrenAtHome>
  <Education>Bachelors </Education>
  <Occupation>Professional</Occupation>
  <HomeOwnerFlag>1</HomeOwnerFlag>
  <NumberCarsOwned>0</NumberCarsOwned>
  <Hobby>Golf</Hobby>
  <Hobby>Watch TV</Hobby>
  <CommuteDistance>1-2 Miles</CommuteDistance>
</IndividualSurvey>
```

Podobne vo formáte XML sú uložené aj inštrukcie v niektorých záznamoch v tabuľke Production.ProductModel. Pre zoznámenie sa s tabuľkou vypíšme jeden záznam

```
SELECT * FROM Production.ProductModel WHERE ProductModelID=7
```

Pre overenie možností XQuery skúste si v tabuľke Production.ProductModel tieto dopyty

```
SELECT Instructions.query(,declare namespace AWMI="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManuInstructions";
  /AWMI:root/AWMI:Location[@LocationID=10]
,) AS Result
FROM Production.ProductModel
WHERE ProductModelID=7
```

```

SELECT CatalogDescription.query(,
declare namespace PD="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelDescription";
    /PD:ProductDescription/PD:Summary
, ) as Result
FROM Production.ProductModel
where ProductModelID=19

```

```

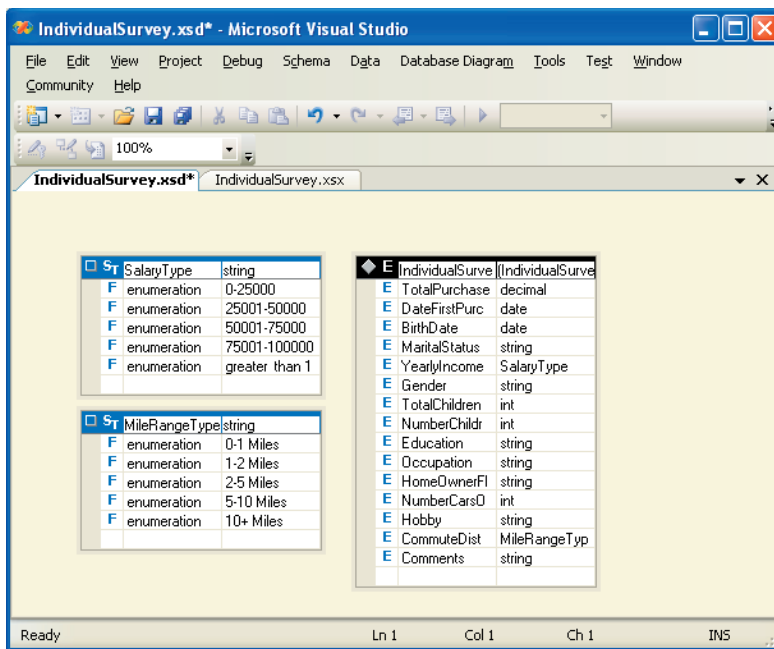
SELECT Instructions.query(,
    <step1> Step 1 description goes here</step1>,
    <step2> Step 2 description goes here </step2>
, ) AS Result
FROM Production.ProductModel
WHERE ProductModelID=7

```

V tomto prípade vzhľadom na to, že s cvičnou databázou bude pracovať veľa záujemcov o zoznámenie sa s SQL Serverom 2005 sú na URI adresách jednoznačne definujúcich menný priestor aj skutočné XSD schémy.

<http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/IndividualSurvey/>

Pre lepšiu orientáciu môžeme nechať schému zobrazit' vo vývojovom prostredí Visual Studio 2005



XSD schéma IndividualSurvey zobrazená schématicky vo Visual Studiu

schéma tabuľky ProductModel bola v dobe písania publikácie na URI adrese <http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ProductModelManuInstructions/ProductModelManuInstructions.xsd>

pre prehľad uvádzame podstatnú časť výpisu schémy bez namespace a anotácie

```

<xsd:annotation>
  <xsd:documentation>SetupHour is the time it takes to set up the machine.
  MachineHour is the time the machine is busy manufacturing LaborHour is the labor
  hours in the manu process LotSize is the minimum quantity manufactured. For example,

```

```

no. of frames cut from the sheet metal</xsd:documentation>
  </xsd:annotation>
- <xsd:complexType name="StepType" mixed="true">
- <xsd:choice minOccurs="0" maxOccurs="unbounded">
  <xsd:element name="tool" type="xsd:string" />
  <xsd:element name="material" type="xsd:string" />
  <xsd:element name="blueprint" type="xsd:string" />
  <xsd:element name="specs" type="xsd:string" />
  <xsd:element name="diag" type="xsd:string" />
</xsd:choice>
</xsd:complexType>
- <xsd:element name="root">
- <xsd:complexType mixed="true">
- <xsd:sequence>
- <xsd:element name="Location" minOccurs="1" maxOccurs="unbounded">
- <xsd:complexType mixed="true">
- <xsd:sequence>
  <xsd:element name="step" type="StepType" minOccurs="1" maxOccurs="unbounded" />
</xsd:sequence>
  <xsd:attribute name="LocationID" type="xsd:integer" use="required" />
  <xsd:attribute name="SetupHours" type="xsd:decimal" use="optional" />
  <xsd:attribute name="MachineHours" type="xsd:decimal" use="optional" />
  <xsd:attribute name="LaborHours" type="xsd:decimal" use="optional" />
  <xsd:attribute name="LotSize" type="xsd:decimal" use="optional" />
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>

```

Kombinácia SQL a XQuery

V stati venovanej XML dátovému typu sme v jednom príklade naplnili premennú dátového typu XML z databázovej tabuľky. Person.Address cvičnej databázy Adventure Works.

V príklade v uvedenej stati sme nechali do XML premennej uložiť len jeden záznam (podmienka WHERE AddressID = 1). V stati venovanej XQuery na tento príklad naviažeme, no v tomto príklade naplníme XML premennú celým obsahom tabuľky a následne z tejto premennej vypíšeme zoznam miest. Príslušné elementy vyberieme pomocou XQuery metódy xml.query

```

DECLARE @x XML
SET @x = (SELECT TOP 10 City FROM Person.Address FOR XML AUTO, ELEMENTS, TYPE)
SELECT @x.query(,
  <Cities>
  {
    for $city in /Person.Address/City
    return $city
  }
  </Cities>
, )

```

Mestá budú vypísané v tvare

```
<Cities>
  <City>Ottawa</City>
  <City>Burnaby</City>
  <City>Dunkerque</City>
  <City>Verrieres Le Buisson</City>
  <City>Verrieres Le Buisson</City>
  <City>Saint-Denis</City>
  <City>Seattle</City>
  <City>Les Ulis</City>
  <City>Miami</City>
  <City>Portland</City>
</Cities>
```

Kapitola 7: Rozširovanie funkčnosti SQL serveru v .NET jazykoch

Ukladanie údajov do databáz pod správu databázových serverov je len prostriedkom. Cieľom je ich efektívne využívanie, napríklad prostredníctvom rôznych aplikácií. Staršie verzie databázových serverov síce umožňovali vytváranie uložených procedúr, funkcií a triggerov v procedurálnej nadstavbe jazyka SQL. U produktovej rady SQL Server je takýmto jazykom T-SQL (Transact SQL). Výhodou uložených procedúr je ich tesné spojenie s údajmi, no boli tu aj určité nevýhody. Pre tvorbu samotných aplikácií, či už webových, intranetových alebo klasických spustiteľných programov sa T-SQL nehodí. Pre tieto účely používajú vývojári na platforme Microsoft .NET programovacie jazyky Visual C#, Visual Basic, Visual J#, prípadne iné .NET jazyky. Nevýhoda je okamžite jasná. Okrem „svojho“ obľúbeného programovacieho jazyka pre tvorbu aplikácií musel vývojár poznať aj T-SQL pre tvorbu uložených procedúr, funkcií a spúští. Taktiež porovnanie možností pre vizuálny návrh, tvorbu a ladenie kódu by pre T-SQL by nedopadlo v porovnaní s Visual Studiom a .NET jazykmi práve najlepšie.

Porovnanie T-SQL s .NET programovacím jazykom (C#)

Ak sme postavení pred riešenie aj relatívne jednoduchej úlohy, napríklad konverzie binárnej numerickej do hexadecimálneho formátu, v procedurálnom jazyku T-SQL to dokážeme pomocou značne dlhého kódu

```

if @bin is null return null
declare @len int, @b tinyint, @lowbyte tinyint, @hibyte tinyint, @index int, @str
nchar(2), @result nvarchar(4000)
set @len = datalength(@bin)
set @index = 1
set @result = ,0x`
while @index <= @len
begin
    set @b = substring(@bin, @index, 1)
    set @index = @index + 1
    set @lowbyte = @b & 0xF
    set @hibyte = @b & 0xF0
    if @hibyte > 0
        set @hibyte = @hibyte / 0xF
    set @result = @result +
        ((case
            when @hibyte < 10 then convert(varchar(1), @hibyte)
            when @hibyte = 10 then ,A`
            when @hibyte = 11 then ,B`
            when @hibyte = 12 then ,C`
            when @hibyte = 13 then ,D`
            when @hibyte = 14 then ,E`
            when @hibyte = 15 then ,F`
            else ,Z`
        end)
        +
        (case
            when @lowbyte < 10 then convert(varchar(1), @lowbyte)
            when @lowbyte = 10 then ,A`
            when @lowbyte = 11 then ,B`
            when @lowbyte = 12 then ,C`
            when @lowbyte = 13 then ,D`
            when @lowbyte = 14 then ,E`
            when @lowbyte = 15 then ,F`
            else ,Z`
        end))
end

```

Je zrejmé, že na úlohy tohto typu programovací jazyk T-SQL príliš vhodný nie je. V tomto konkrétnom prípade pätriadkový kód v C# dokáže nahradiť predchádzajúci zložitý polstránkový algoritmus v T-SQL

```
if (value == null) return null;
    StringBuilder sb = new StringBuilder();
    foreach (byte b in value)
        sb.Append(b.ToString("X2"));
    return sb.ToString();
```

Zdalo by sa, že použitie vyššieho programovacieho jazyka bude v porovnaní s T-SQL vždy výhodnejšie. No nie je tomu tak. Dokázali by sme nájsť veľa príkladov, kedy je to presne naopak.

Program v jazyku T-SQL

```
SELECT * FROM T_CUSTOMERS WHERE C_CITY = @name
```

Program v jazyku C#

```
using(SqlConnection conn = new SqlConnection("context connection=true"))
{
    conn.Open();
    SqlCommand cmd = new SqlCommand("SELECT * FROM T_CUSTOMERS WHERE C_CITY = @name", conn);

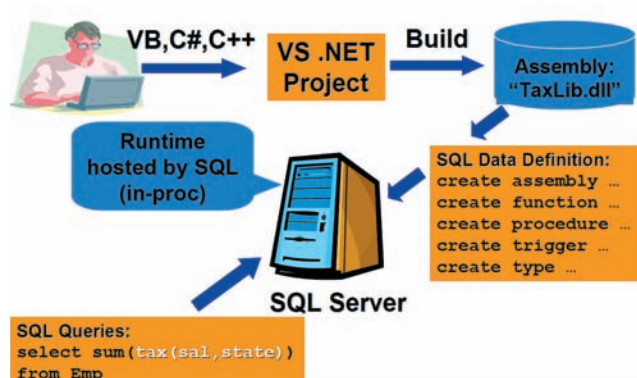
    SqlDataReader rdr = cmd.ExecuteReader();
    SqlContext.GetPipe().Send(rdr);
}
```



Princíp integrácie CLR vrstvy

Výhody riadeného (managed) kódu sú zhrnuté v tabuľke

Vlastnosť	Popis
Programovací jazyk	Možnosť výberu zo všetkých .NET jazykov
Zabezpečenie typov	Zabezpečenie typov cez „common type system“
Bezpečnosť	Bezpečnosť cez „code access security“ a „role-based security“
Produktivita vývoja	Produktivita podložená šírkou knižnice tried v .NET Frameworku
Interoperabilita	Interoperabilita s „nemanážovaným“ kódom a sieťovým/operačným prostredím



Princíp funkcie CLR vrstvy

„CLR“ uložené procedúry

Uložené procedúry sú tak ako vyplýva z ich názvu uložené priamo v databáze spolu s ostatnými objektmi. V praxi to teda znamená, že do databázy je možné umiestniť okrem údajov aj časť, prípadne aj celú aj aplikačnú logiku pre spracovanie týchto údajov. To umožňuje nielen jednoduchšiu distribúciu aplikácie, ale prispieva aj k spoľahlivosti, keďže aplikačná logika je zabezpečená a zálohovaná rovnako spoľahlivo ako samotné údaje. Uložené procedúry sa ukladajú do databázy spravidla v predkompilovanej podobe. V príslušnej triede vytvoríme „public static“ metódu. Pre nasadenie uloženej procedúry je potrebné špecifikovať atribút [SqlProcedure (Name=„GetContactNames“)]

Príklad kódu „manažovanej“ uloženej procedúry

```
public class ContactCode
{
    [SqlProcedure (Name=„GetContactNames“)]
    public static void GetContactNames()
    {
        SqlCommand cmd = SqlContext.GetCommand();
        cmd.CommandText = „SELECT FirstName + , , + LastName“ +
        „ AS [Name] FROM Person.Contact“;
        SqlDataReader rdr = cmd.ExecuteReader();
        SqlPipe sp = SqlContext.GetPipe();
        sp.Send(rdr);
    }
}
```

„CLR“ užívateľsky definované funkcie

Funkcia je špeciálnym prípadom uloženej procedúry, ktorá však na rozdiel od procedúry vráti bloku, v ktorom bola zavolaná nejakú hodnotu. Táto hodnota je získaná alebo vypočítaná v tele procedúry. Vytvoríme „public static“ metódu. Pre nasadenie UDF je potrebné špecifikovať atribút [SqlFunction (Name=„GetLongDate“)] V zásade ľubovoľná funkcia .NET jazyka môže byť použitá ako T-SQL funkcia. Príslušná trieda musí byť typu public, pričom sa vyžaduje aby funkcia bola typu public static. CLR assembly obsahujúca triedu a funkcia musí byť katalogizovaná

Príklad kódu „manažovanej“ UDF

```
public class MyFunctions
{
    [SqlFunction (Name=„GetLongDate“ )]
    public static SqlString GetLongDate (SqlDateTime DateVal)
    {
```

```

        // Return the date as a long string
        return DateVal.Value.ToLongDateString();
    }
}

```

„CLR“ spúšte (trigger)

Pod pojmom spúšť (trigger) rozumieme v podstate uložený procedúru, ktorá sa automaticky vykoná v prípade určitej predtým definovanej udalosti, ktorá nastáva pri manipulácii s údajmi, napríklad pri vkladaní alebo vymazávaní údajov v tabuľke a podobne. Nikdy sa teda nespúšťajú priamo, ale sú naviazané na príkazy pre modifikáciu údajov (INSERT, DELETE, UPDATE). Ich cieľom nie je vracanie výsledkov. Spúšte sa môžu použiť na kontrolu zadávaných údajov, zaistenie dátovej integrity a podobne. Dôležité je samozrejme aj to, kedy sa má trigger aktivovať. Niekedy je potrebné, aby sa trigger aktivoval pred príkazom, ktorý jeho aktiváciu vyvolá, napríklad ak chceme upraviť a skontrolovať vkladané údaje, inokedy po úspešnom vykonaní príkazu, napríklad ak po vmazaní nejakého záznamu chceme vymazať, alebo upraviť záznamy, ktoré s ním z hľadiska aplikačnej logiky súvisia. A niekedy trigger úplne nahradí príkaz, ktorý ho aktivoval. Podobne ako u „manažovanej“ uloženej procedúre au u spúšte v príslušnej triede vytvoríme „public static“ metódu. Pre nasadenie spúšte je potrebné špecifikovať atribút

```
[SqlTrigger (Name="ContactUpdTrg", Target="Person.Contact", Event="FOR UPDATE")]
```

Príklad kódu „manažovanej“ spúšte

```

public class ContactCode
{
    [SqlTrigger (Name="ContactUpdTrg", Target="Person.Contact", Event="FOR UPDATE")]
    public static void ChangeEmail()
    {
        SqlTriggerContext trg = SqlContext.GetTriggerContext();
        if (trg.TriggerAction == TriggerAction.Update)
        {
            if (trg.ColumnsUpdated[7] == true)
                //... kód ktorý napríklad odošle e-mail na každý nový kontakt
        }
    }
}

```

„CLR“ agregácie

Na rozdiel od štandardných agregačných funkcií môžeme u CLR agregácií stanoviť pravidlá sami. Napríklad pre výpočet známky krasokorčuliara, kde pravidlá vyžadujú vyškrtnúť najhoršie a najlepšie hodnotenie a až potom na základe zvyšných hodnôt sa vypočíta priemer známok. Ešte zložitejšie pravidlá je potrebné naprogramovať pre výpočet koeficientu ELO, pomocou ktorého sa hodnotí výkonnosť šachistov.

Po vytvorení „public“ triedy pridáme atribúty `[Serializable, SqlUserDefinedAggregate (...)]`

Pre agregáciu vytvoríme kód metód `Init`, `Accumulate`, `Merge` a `Terminate`

Príklad kódu „manažovanej“ agregácie

```

[Serializable, SqlUserDefinedAggregate (...)]
public class CommaDelimit
{
    public void Init()
    { ... }

    public void Accumulate (SqlString Value)
    { ... }
}

```

```

public void Merge(CommaDelimit Group)
{...}

public SqlString Terminate()
{...}
}

```

Projekt typu SQL Server Project – používateľsky definovaná funkcia

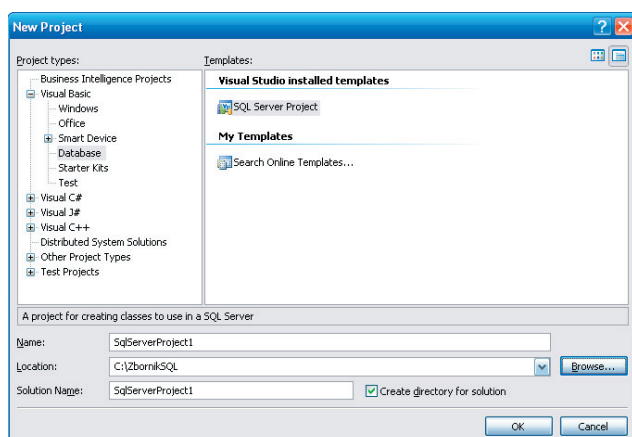
Do vývojového prostredia Visual Studio 2005 boli pridané šablóny pre vytváranie SQL Server projektov, ktoré umožňujú tvorbu a efektívne ladenie blokov riadeného kódu. V projekte typu SQL Server Project sú predpripravené referencie

- Microsoft.VisualStudio.DataTools.SqlAttributes.dll
- sqlaccess.dll
- System.Data.dll

Projekt obsahuje šablóny pre typy objektov potrebné pre aplikácie v SQL Serveri

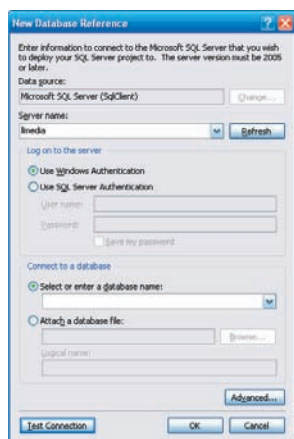
- Stored procedure
- Trigger
- User-defined function
- User-defined type
- Aggregate

Vytvoríme nový projekt typu SQL Server Project, napríklad v jazyku Visual Basic.



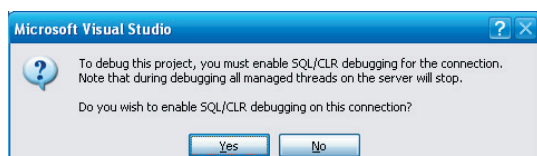
Vytvorenie nového projektu typu „SQL Server project“

Keďže náš projekt je typu SQL Server Project, prvou starosťou sprievodcu vytvorení projektu je vytvorenie pripojenia na databázový server. Preto po úvodnom dialógu nasleduje výber databázového servera



Vytvorenie referencie na databázový server

Tretím krokom pri vytváraní nového projektu je povolenie SQL/CLR ladenia.



Povolenie SQL/CLR ladenia

Po vytvorení projektu si všimnite zložku Test Scripts, ktorá obsahuje testovací SQL skript. Všetky príkazy sú v komentári. Tento súbor môžete modifikovať a uložiť tam SQL príkazy pre aplikáciu

```
-- Examples for queries that exercise different SQL objects implemented by this
assembly
```

```
-----
-- Stored procedure
-----
-- exec StoredProcedureName
```

```
-----
-- User defined function
-----
-- select dbo.FunctionName()
```

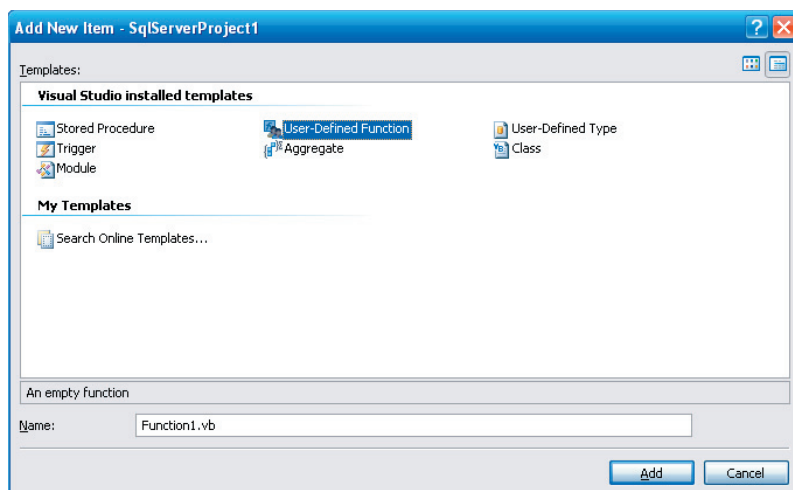
```
-----
-- User defined type
-----
-- CREATE TABLE test_table (col1 UserType)
-- go
--
-- INSERT INTO test_table VALUES (convert(uri, ,Instantiation String 1`))
-- INSERT INTO test_table VALUES (convert(uri, ,Instantiation String 2`))
-- INSERT INTO test_table VALUES (convert(uri, ,Instantiation String 3`))
--
-- select col1::method1() from test_table
```

```
-----
-- User defined type
-----
-- select dbo.AggregateName(Column1) from Table1
```

V našej aplikácii do tohto súboru vložíme volanie používateľsky definovanej funkcie

```
select dbo.GetTodaysDate()
```

V kontextovom menu projektu použijeme voľbu „Add New Item“ a vyberieme User-Defined Function. Funkcii ponecháme implicitný názov Function1.vb



Dialóg „Add Existing Item“

V procese vytvárania používateľsky definovanej funkcie bol vygenerovaný kód

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server

Partial Public Class UserDefinedFunctions
    <Microsoft.SqlServer.Server.SqlFunction()> _
    Public Shared Function Function1() As SqlString
        , Add your code here
        Return New SqlString(„Hello“)
    End Function
End Class
```

Funkciu

```
<Microsoft.SqlServer.Server.SqlFunction()> _
    Public Shared Function Function1() As SqlString
        , Add your code here
        Return New SqlString(„Hello“)
    End Function
```

Nahradíme funkciou, ktorú potrebujeme do našej aplikácie,

```
<Microsoft.SqlServer.Server.SqlFunction()> _
    Public Shared Function GetTodaysDate() As Date
        Return Date.Today
    End Function
```

Aplikáciu zostavíme. Pri pokuse o deploy používateľsky definovanej funkcie na SQL Server 2005 však dôjde k chybe

```
Auto-attach to process , [976] [SQL] mnbeta2\ on machine , mnbeta2\ succeeded.
Debugging script from project script file.
```

```
The thread , mnbeta2 [55] \ (0x5b4) has exited with code 0 (0x0).
The thread , mnbeta2 [55] \ (0x5b4) has exited with code 0 (0x0).
The thread , mnbeta2 [55] \ (0x5b4) has exited with code 0 (0x0).
```

```
Execution of user code in the .NET Framework is disabled. Use sp_configure „clr
enabled“ to enable execution of user code in the .NET Framework.
The thread ,mnbeta2 [55]` (0x5b4) has exited with code 0 (0x0).
The program ,[976] [SQL] mnbeta2: mnbeta2` has exited with code 0 (0x0).
```

V SQL Serveri 2005 preto pomocou nástroja SQL Server Management Studio povolíme CLR príkazom

```
sp_configure ,clr enabled`, 1
```

Reakciou je výzva SQL servera na rekonfiguráciu

```
Configuration option ,clr enabled` changed from 0 to 1. Run the RECONFIGURE statement
to install.
```

V konzolovej aplikácii zadáme príkaz RECONFIGURE

```
reconfigure;
```

Ak rekonfigurácia prebehne úspešne SQL Server nám to potvrdí

```
Command(s) completed successfully.
```

Teraz by malo prebehnúť úspešne aj zavedenie používateľsky definovanej funkcie do SQL Servera 2005

```
Auto-attach to process ,[976] [SQL] mnbeta2` on machine ,mnbeta2` succeeded.
Debugging script from project script file.
```

```
...
```

```
,Managed`: Loaded ,SqlServerProject1`, No symbols loaded.
```

```
The thread ,mnbeta2 [55]` (0x1c0) has exited with code 0 (0x0).
```

```
Column1
```

```
-----
13.8.2005
```

```
No rows affected.
```

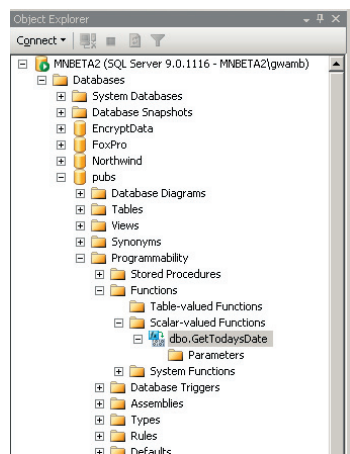
```
(1 row(s) returned)
```

```
Finished running sp_executesql.
```

```
The program ,[976] sqlservr.exe: MNBETA2;.Net SqlClient Data Provider;1516` has
exited with code 259 (0x103).
```

```
The program ,[976] [SQL] mnbeta2: mnbeta2` has exited with code 0 (0x0).
```

Zo strany vývojového prostredia Visual Studio 2005 sa zdá byť všetko v poriadku, podme sa však presvedčiť, či bola príslušná používateľsky definovaná funkcia uložená do databázy SQL Servera



Object Explorer CLR používateľsky definovaná funkcia v databáze

Prípadne môžeme funkciu pokusne spustiť cez konzolu a presvedčiť sa, že úspešne prebehne

```
SELECT dbo.GetTodaysDate ()
```

```
-----  
2005-08-13 00:00:00.000
```

```
(1 row(s) affected)
```

U zložitejšej funkcie si môžeme vyskúšať aj ladenie. Na problémový riadok umiestnime breakpoint, na ktorom sa aplikácia zastaví

V nasledujúcej stati vytvoríme aplikáciu, ktorá bude túto funkciu využívať

Import existujúcej CLR assembly

Existujúcu CLR assembly (súbor s príponou DLL) je možné importovať aj pomocou klientskej konzolovej aplikácie SQL Servera 2005 pomocou príkazu CLR ASSEMBLY. Príkaz vyžaduje zadanie mena objektu v databáze pre assembly, zadanie referencie alebo cesty na DLL súbor a nastavenie oprávnení. Môžeme nastaviť tri úrovne oprávnení: SAFE, EXTERNAL_ACCESS, a UNSAFE. Popíšeme aspoň heslovite správanie sa assembly pre jednotlivé úrovne nastavení

SAFE

- právo na spustenie a prístup k dátam
- žiadny prístup k zdrojom mimo SQL Servera
- žiadne „nemanážované“ volania
- musí byť verifikovateľný

EXTERNAL_ACCESS: READ E_A & WRITE E_A

Tie isté oprávnenia ako SAFE, plus navyše

- prístup k externým zdrojom (súbory, siete)
- prístup k externým zdrojom cez „manažované“ API
- SQL Server impersonifikuje volajúceho pre externý prístup (EXECUTE AS)
- Musí byť verifikovateľný
- vyžaduje vytvorenie nového práva EXTERNAL ACCESS

UNSAFE

- môže volať „nemanážovaný“ kód, môže byť neverifikovateľný
- môže vytvoriť len Sysadmin

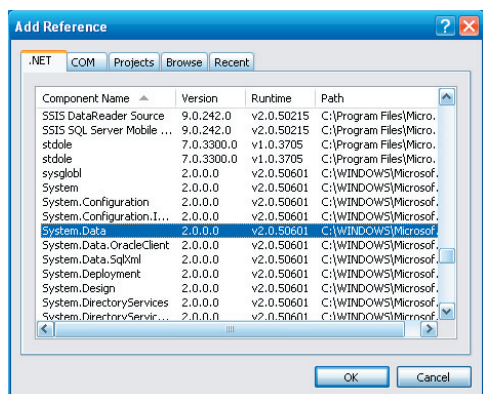
Pre funkciu vytvorenú v predchádzajúcej stati by sme použili príkaz v tvare

```
CREATE ASSEMBLY SqlServerProject1  
FROM ,C:\ZbornikSQL\SqlServerProject1\SqlServerProject1\bin\SqlServerProject1.dll\  
WITH PERMISSION_SET = EXTERNAL_ACCESS
```

Ak nepoužijeme nastavenie úrovne oprávnení, táto úroveň bude automaticky nastavená ako SAFE.

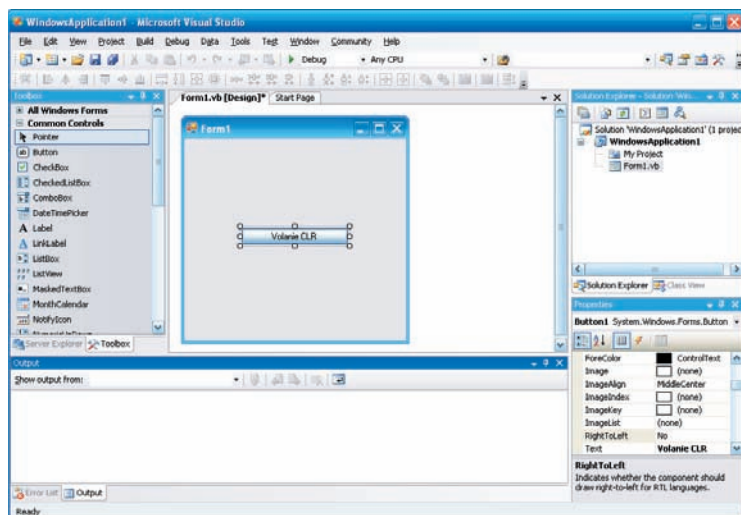
Windows aplikácia pre prístup k CLR kódu

V predchádzajúcej stati sme vytvorili CLR assembly, obsahujúcu používateľsky definovanú funkciu. Aby bol náš prístup k CLR komplexný, vytvoríme aj konzumenta, teda Windows aplikáciu, ktorá bude túto funkciu volať. Vo vývojovom prostredí Visual Studio 2005 vytvoríme Windows aplikáciu napríklad v jazyku Visual Basic. Ako prvý krok pridáme do projektu referenciu na System.Data



Dialóg pre pridanie referencie na System.Data

Aplikačná logika bude pozostávať z jediného tlačidla, pomocou ktorého budeme aktivovať CLR funkciu



Formulár aplikácie pre prístup k CLR kódu

Do kódu pridáme referenciu na namespace `System.Data.SqlClient` a napíšeme kód pre obsluhu udalosti zatlačenie tlačidla

```
Imports System.Data.SqlClient
```

```
Public Class Form1
```

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.
EventArgs) Handles Button1.Click
```

```
Dim connectionString As String = „Data Source=localhost;” & _
„Initial Catalog=Test;” & „Integrated Security=True”
```

```
Using conn As New SqlConnection(connectionString)
conn.Open()
```

```
Dim cmd As New SqlCommand(„SELECT dbo.GetTodaysDate()”, conn)
```

```
Dim result As Object = cmd.ExecuteScalar()
Dim d As Date = CDate(result)
```



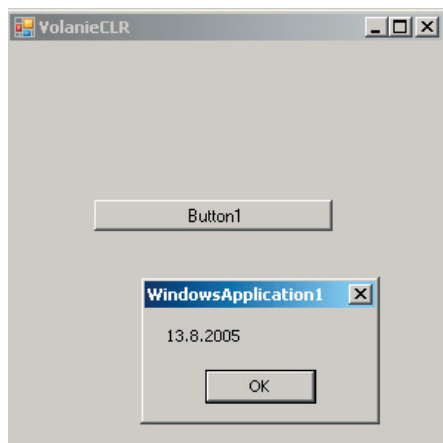
```

        MsgBox(d.ToString („d“))
    End Using

End Sub
End Class

```

Teraz môžeme aplikáciu zostaviť a vyskúšať. V prípade úspechu sa zobrazí dátum,



Windows aplikácia využívajúca CLR používateľsky definovanú procedúru

v opačnom prípade ak v SQL serveri nemáme príslušný kód alebo aplikácia sa s SQL serverom nedokáže spojiť dospejeme k chybovému hláseniu

```

Cannot find either column „dbo“ or the user-defined function or aggregate „dbo.
GetTodaysDate“, or the name is ambiguous.

```

Kontrola regulárneho výrazu pomocou funkcie `Regex.IsMatch`

„Manažovaný“ CLR kód v SQL Serveri 2005 nám umožňuje riešiť mnohé užitočné záležitosti, o ktorých vývojári vo verzii SQL Servera 2000 mohli len snívať. Jednou z takýchto črt sú regulárne výrazy. Pomocou regulárnych výrazov môžeme veľmi efektívne pracovať s textom, napríklad vyhľadávať a vzájomne nahradzovať presne určené časti textu, prípadne kontrolovať správnosť zadania textových reťazcov pred ich ďalším strojovým spracovaním. Tieto časti definujeme pomocou vzorov. Najjednoduchšími regulárnymi výrazmi sú písmená, zhľuky písmen a slová. Vtedy sa pri testovaní textu zisťuje, či overovaný text obsahuje ono písmeno, zhľuk písmen alebo slovo, ktoré je definované pomocou regulárneho výrazu. Takéto porovnávanie nám však neposkytuje dostatočnú variabilitu. Preto v regulárnych výrazoch používame rôzne zástupné znaky – žolíky.

. (bodka) slúži pre nahradenie jedného ľubovoľného znaku. Pomocou regulárneho výrazu `.lyn` môžeme nájsť slová `plyn` a `mlyn`.

ak chceme nahradzovať na pozícii zástupného znaku len určité znaky, vymenujeme ich v hranatých zátvorkách, napríklad `[mp]lyn`. Hranaté zátvorky umožňujú nájsť len jeden znak zo znakov v nich vymenovaných. Ak chceme hľadať viac znakov použijeme viac hranatých zátvoriek. Napríklad všetky typy automobilov od firmy Peugeot nájdeme pomocou jednoduchého regulárneho výrazu, kde prvú a poslednú číslicu typového označenia nahradíme hranatými zátvorkami s vymenovaním všetkých prípustných znakov, teda číslic 1 až 9.

```
Peugeot [123456789]0[123456789]
```

Automobilka totiž pre označenie jednotlivých modelov používa trojmiestne číslo v ktorom je v strede číslica nula. Takže na našich cestách sa stretávame s automobilmi Peugeot 205, 309, 405....

Ten istý regulárny výraz môžeme zapísať aj ako interval znakov [1-9] teda

Peugeot [1-9]0[1-9]

Pomocou intervalov v hranatých zátvorkách potom ľahko nájdeme napríklad dátum v tvare dd.mm.rrrr. Stačí použiť regulárny výraz

[0-3] [0-9]. [0-1] [0-9]. [1-2] [0-9] [0-9] [0-9]

Samozrejme takto len nájdeme dátum, ale neoveríme jeho správnosť, nakoľko uvedenému regulárnemu výrazu vyhovuje aj dátum

38. 19. 2004

Znakom ? (otáznik) definujeme, že predchádzajúci znak tam môže, alebo nemusí byť. Napríklad regulárnemu výrazu

prst?

vyhovuje aj prs aj prst.

Zatiaľ sa stále venujeme nahradeniu jedného znaku znakom zástupným.

Sila programovania je v opakovaní kódu, preto máme aj zástupný znak pre opakovanie, ktorým je * (hviezdička).

Ak chceme nájsť v texte všetky čiary zložené z pomlčiek opakovane vypísaných jedna za druhou, použijeme regulárny výraz

._*

Samozrejme prípustná je aj kombinácia s hranatými zátvorkami, takže regulárny výraz

[0-9] *

nájde všetky prirodzené čísla.

Ako znak opakovania môžeme použiť aj znak + (plus), no s tým rozdielom že kým hviezdička znamenala ľubovoľný počet opakovaní teda aj nulu, znak plus vyžaduje aspoň jedno opakovanie.

Ľubovoľný textový reťazec potom nájdeme pomocou bodky ako zástupného znaku pre ľubovoľný znak a hviezdičky ako zástupného znaku pre ľubovoľný počet opakovaní.

.*

V regulárnych výrazoch môžeme definovať aj pozíciu v rámci riadku. Pomocou zástupného znaku ^ (strieška) definujeme začiatok riadku a pomocou zástupného znaku \$ (dolár) definujeme koniec riadku.

Výpis všetkých modifikátorov by bol dosť rozsiahly, preto sa ich použitie pokúsime demonštrovať na praktickom príklade regulárneho výrazu pre kontrolu mailovej adresy. Tento by mohol byť v tvare.

/^[^@\s]+@[(-a-z0-9)+\.]+[a-z]{2,}\$ /i

Takýto výraz si samozrejme zasluhuje podrobnejšie vysvetlenie

```

/
^          # značka pre začiatok textu
[^\s]+    # meno môže pozostávať z ľubovoľných znakov okrem @ a medzery
@         # zavináč @ pre oddelenie mena od domény
(
  [-a-z0-9]+ # názov subdomény
  \.         # bodka ako oddelovač
)+        # subdomén oddelených bodkou môže byť viac

  [a-z]{2,} # najmenej dvojpísmenová doména
$         # značka pre koniec textu
/ix      # nerozlišovať veľké a malé písmená

```

Podobne ako v predchádzajúcom riešení vytvoríme nový projekt typu SQL Server Project a v ňom použijeme definovanú funkciu, ktorá bude pomocou regulárneho výrazu

```
@“^\s*(\d{5}|(\d{5}-\d{4}))\s*$“
```

Kontrolovať „zip.code“ – americkú obdobu našich poštových smerovacích čísel. Pre zmenu ukážeme projekt typu SQL Server Project v jazyku C#

```

using System;
using System.Data;
using System.Data.Sql;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class UserDefinedFunctions
{
    [Microsoft.SqlServer.Server.SqlFunction]
    public static SqlString Function1()
    {
        // Put your code here
        return new SqlString(„Hello“);
    }
};

```

Ilustračnú funkciu, ktorú vytvoril sprievodca nahradíme novou funkciou, ktorá bude kontrolovať „zip kódy“

```

public partial class UserDefinedFunctions
{
    [Microsoft.SqlServer.Server.SqlFunction]
    public static bool IsValidZipCode(string ZipCode)
    {
        // Put your code here
        return System.Text.RegularExpressions.Regex.IsMatch(
            ZipCode, @“^\s*(\d{5}|(\d{5}-\d{4}))\s*$“);
    }
};

```

Pre zaujímavosť si skúste vytvoriť Windows aplikáciu, ktorá bude využívať funkciu pre kontrolu regulárnych výrazov

```

using System.Data.SqlClient;

string connectionString = „Data Source=localhost;“ +

```

```

        „Initial Catalog=SQLCLRIntegration;“ +
        „Integrated Security=True“;

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

    SqlCommand cmd = new SqlCommand(
        „SELECT dbo.IsValidZipCode(@ZipCode)“, conn);

    cmd.CommandType = CommandType.Text;
    cmd.Parameters.Add(„@ZipCode“, SqlDbType.VarChar);
    cmd.Parameters[„@ZipCode“].Value = textBox1.Text;

    Object result = cmd.ExecuteScalar();
    MessageBox.Show(result.ToString());
}

```

Používateľsky definované typy

Umožňujú rozšíriť systém typov o rôzne napríklad skalárne typy, ktorých potreba vyplýva z aplikačnej logiky. Ich použitie má význam u veličín, ktoré sú dané všeobecnými pravidlami z niektorého vedného odboru a nebudú sa meniť. Napríklad vektorové veličiny, súradnice bodov, geometrických obrazcov, zemepisné súradnice, časový údaj zahrňujúci časovú zónu a podobne. Jednoducho údaje, ktorých štruktúra sa ani po storočiach nemení. Naopak rôzne štruktúry, ktoré sa vzťahujú na konkrétny prípad súvisiaci s ľudskou činnosťou kde sú možné operatívne zmeny, napríklad pre štruktúru objednávky, alebo faktúry sú tieto typy nevhodné, pretože pri každej operatívnej zmene štruktúry faktúry je potrebné predefinovávať dátové typy. V takýchto prípadoch aj po úplnom odladení aplikácie stačí malá legislatívna zmena a celú štruktúru je potrebné predefinovať.

Pre používateľsky definované typy (UDT User Defined Type) je možné napríklad definovať pravidlá pre triedenie, agregácie, alebo vlastné aritmetické kalkulácie. Operácie s UDT sú napísané v kompilovaných jazykoch. UDT môžu byť použité ako stĺpce v databázových tabuľkách, premenné a parametre a návratové hodnoty pre používateľsky definované funkcie (UDF) UDT Používateľsky definované typy podporujú implicitnú konverziu z varchar, je možné použiť aj explicitnú konverzia, prípadne si môžeme napísať vlastnú konverznú funkciu

To že sme UDT v tejto publikácii zaradili chronologicky za CLR assembly nie je náhoda. SQL Server 2005 UDT sú uložené práve v .NET assembly. Ak ich chceme používať, musíme ich samozrejme najprv v CLR kóde vytvoriť a následne najskôr katalogizovať assembly (CREATE ASSEMBLY) a potom katalogizovať typ (CREATE TYPE).
Například

```

CREATE ASSEMBLY Point
    FROM ,c:\types\Point.dll\
GO

CREATE TYPE PointCls
EXTERNAL NAME Point.PointCls
GO

```

Externé meno (external name) sa musí zhodovať s menom .NET triedy, symbolické meno môže byť akékoľvek. Nasledujúci SQL kód demonštruje použitie UDT v databázovej tabuľke

```

CREATE TABLE point_tab(
    id int primary key,
    thepoint PointCls)

```

```
INSERT INTO point_tab  
values(1, ,100:200`)
```

Prípadne môžeme definovať aj premennú používateľsky definovaného typu

```
DECLARE @p PointCls  
SET @p = convert(PointCls, ,300:400`)
```

Kapitola 8: Nové rysy v ADO.NET 2.0

Databázu chápeme ako úložisko údajov, ktoré sú uložené a spracovávané nezávisle od aplikačných programov. Databázy zapuzdrujú jednak vlastné údaje, ale aj relačné vzťahy medzi jednotlivými prvkami a objektmi v databáze, schémy popisujúce štruktúry údajov a integritné obmedzenia.

Pri použití databázy nám stačí definovať požadovanú operáciu nad množinou údajov v databáze pomocou príkazu jazyka SQL (Structured Query Language) a ostatné už vykoná databázový server sám. Aplikácie, či už typu tenký, „rich“ alebo „smart“ klient cez vhodné rozhranie dokážu efektívne pracovať prakticky s každým databázovým serverom. Prakticky v každej databázovej aplikácii využívajúcej ADO.NET musíme vytvoriť pripojenie na databázový server a toto pripojenie otvoriť.

Vytvorenie objektu pre pripojenie

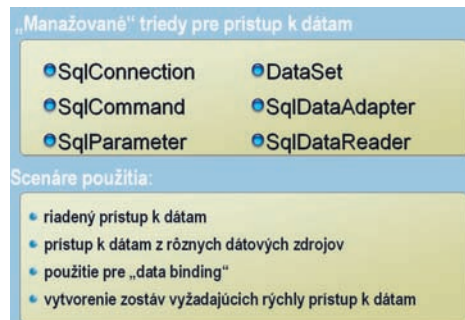
Základom pre vybudovanie pripojenia je vytvorenie príslušného objektu ADO.NET, ktorý pripojenie určitým spôsobom sprostredkuje. Ako parameter konštruktora sa vyžaduje pripojovací reťazec, ktorý obsahuje údaje o databázovom serveri, databáze, prípadne aj ďalšie parametre pre vytvorenie pripojenia

Otvorenie pripojenia

Po vytvorení objektu pre pripojenie sa k údajom v databáze nasleduje ďalší logický krok – otvorenie pripojenia. V tomto okamihu sa už vytvorí aj fyzické pripojenie k databáze. Od tejto chvíle vlastne môže aplikácia komunikovať s databázovým serverom.

V zásade rozoznávame dva spôsoby pripojenia sprostredkované cez príslušné objekty DataReader a DataSet. Zatiaľ čo pripojenie cez DataReader je živé, a poskytuje prístup k aktuálnemu stavu dátového zdroja, DataSet predstavuje vo svojej podstate odpojený zdroj údajov, obsahujúci snapshot vybraných údajov v okamihu načítania, pričom údaje sú uložené v pamäti. DataSet sa správa ako kontajner do ktorého môžeme uložiť jeden alebo viac objektov a to aj rôzneho typu

Podstatu ADO.NET veľmi dobre vystihuje nasledujúca schéma, obsahujúca akési zhrnutie tried a scenárov pre prístup k údajom v databázach.



Využitie šablóny pre vytvorenie novej databázy

SQL Server 2005 umožňuje prístup aplikácií k údajom prostredníctvom data providera System.Data.SqlClient.

Pripojenie k viacerým dátovým zdrojom – providers factory

Ak sa aplikácia potrebuje pripájať k viacerým nehomogénnym zdrojom údajov, je potrebné využívať viacero providerov pre pripojenia a podľa aktuálnej potreby pripojenia vybrať príslušný adaptér

```
enum provider {sqlserver, oracle, oledb, odbc};
// determine provider from configuration
provider prov = GetProviderFromConfigFile();
IDbConnection conn = null;
switch (prov) {
```

```

case provider.sqlserver:
    conn = new SqlConnection(); break;
case provider.oracle:
    conn = new OracleConnection(); break;
case provider.oledb:
    conn = new OleDbConnection(); break;
case provider.odbc:
    conn = new OdbcConnection(); break;
// ...
}

```

Ten istý efekt v ADO.NET dosiahneme pomocou oveľa jednoduchšieho kódu s využitím objektu DbProviderFactory

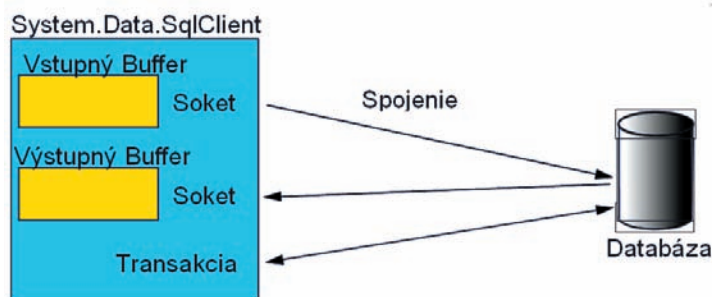
```

string provstring = GetProviderInvariantString();
DbProviderFactory fact = DbProviderFactories.GetFactory(provstring);
IDbConnection = fact.CreateConnection();

```

SQL.Client – kód mimo databázy

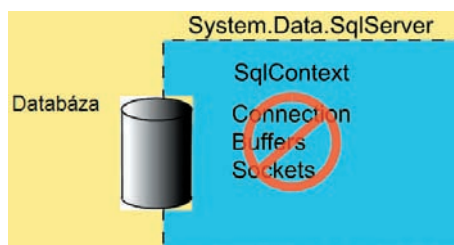
Kód mimo databázy musí pre prístup k údajom v databáze nadviazať fyzické spojenie s databázovým serverom. Pre tento účel slúži objekt SqlClient používa SqlConnection. Po nadviazaní spojenia sa na úrovni vrstvy System.Data. SqlClient vytvorí vstupný a výstupný vyrovnávací bafer a následne sa v rámci transakcie ktorú si pripojenie vytvorilo alebo na ktorej participuje spustí dávka príkazov. Transakcie môžeme v rámci pripojenia vytvárať aj explicitne, zjednodušene povedané príkaz Connection.BeginTransaction vyvolá v SQL Servero SQL príkaz „BEGIN TRAN“



Kód mimo databázy System.Data.SqlClient

Kód vo vnútri databázy

Pri tomto type pripojenia je kód vo vnútri databázy a explicitné pripojenie sa vytvára so špeciálnym Connection Stringom. Využíva sa objekt SqlContext. SqlServer pre tento typ prístupu k údajom nevyžaduje spojenie. Kód beží ako časť databázy pričom príkazy sú časťou jednej dávky. Na tejto úrovni nie je potrebné ani vytvárať bufre

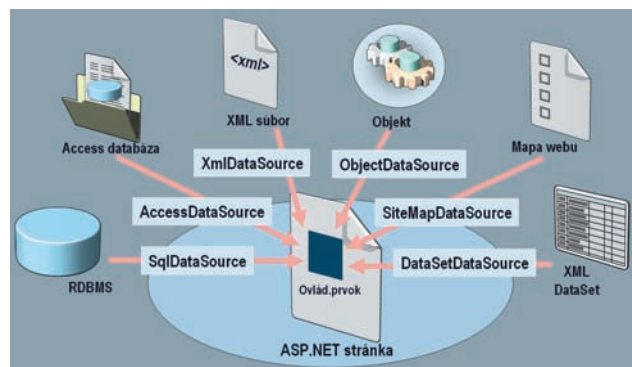


Kód vo vnútri databázy System.Data.SqlServer

Jednotlivé triedy Namespace System.Data.SqlServer sú v tabuľke.

Trieda	Popis
SqlConnection	Otvorenie spojenia s SQL Server databázou
SqlCommand	Odoslanie príkazu na databázový server
SqlParameter	Dodáva parametre pre objekt SqlCommand
SqlDataReader	Čítanie dát po riadkoch (od prvého po posledný)
SqlResultSet	Trieda na prácu so serverovými kurzormi
SqlTransaction	Trieda na transakčné spracovanie
SqlTriggerContext	Kontextové informácie o akcii triggeru

Aj napriek odlišnostiam vyplývajúcim zo spôsobu pripojenia tabuľka je v princípe spoločná pre obidva datové providery – SQL Client a SQL Server – zdieľajú SQLTypes,



Využitie šablóny pre vytvorenie novej databázy

Štatistika o prístupe údajov

Ak prístupujeme k databázovému serveru cez konzolovú aplikáciu môžeme pomocou monitorovania rôznych štatistických parametrov ladiť výkonnosť databázovej vrstvy. ADO.NET 2.0 poskytuje túto možnosť monitorovania štatistiky aj na aplikačnej úrovni

```

...
string connect_string = GetConnectionStringFromConfigFile();
SqlConnection conn = new SqlConnection(connect_string);
conn.Open();

// Povolenie štatistiky
conn.StatisticsEnabled = true;
//
// ... monitorovane akcie
//
SqlCommand cmd = new SqlCommand("select * from authors", conn);
SqlDataReader rdr = cmd.ExecuteReader();
Hashtable stats = (Hashtable)conn.RetrieveStatistics();

// štatistika
IDictionaryEnumerator e = stats.GetEnumerator();
while (e.MoveNext())
Console.WriteLine("{0} : {1}", e.Key, e.Value);
conn.ResetStatistics();
...

```


Connection-specific statistics

```

BuffersReceived      : 1
BuffersSent         : 1
BytesReceived       : 220
BytesSent           : 72
ConnectionTime      : 149
CursorFetchCount    : 0
CursorFetchTime     : 0
CursorOpens         : 0
CursorUsed          : 0
ExecutionTime       : 138
IduCount            : 0
IduRows             : 0
NetworkServerTime   : 79
PreparedExecs       : 0
Prepares            : 0
SelectCount         : 0
SelectRows          : 0
ServerRoundtrips    : 1
SumResultSets       : 0
Transactions        : 0
UnpreparedExecs     : 1

```

Na záver kapitoly zhrnieme nové črty ADO.NET v prehľadnej tabuľke v závislosti od providera

	All Providers	SQL Server 2000	SQL Server 2005
Provider Factories	X	X	X
Runs w/Partial Trust	X	X	X
Server Enumeration	X	X	X
Connection String Builder	X	X	X
Metadata Schemas	X	X	X
Batch Update Support	X	X	X
Provider-Specific Types	X	X	X
Conflict Detection	X	X	X
Tracing Support	X	X	X
Pooling Enhancements	SqlClient and OracleClient	X	X
MARS			X
SqlNotificationRequest			X
SqlDependency			X
IsolationLevel.Snapshot			X
Asynch Commands		X	X
Client Failover			X
Bulk Import		X	X
Password Change API			X
Statistics		X	X
New Datatypes			X
Promotable Tx		X	X
AttachDbFileName		X	X

Druhá tabuľka obsahuje odporúčanie pre niektoré typické scenáre

Úloha	Doporučenia
Spojenie	Optimalizácia použitia spojenia (otvor neskoro, zatvor skoro) Použite „connection pooling“ Použite „Windows authentication“ Bezpečne ukladajte „connection string-y“
Príkazy	Používajte „stored procedures“ Vhodne voľte transakcie
Načítanie dát	DataSet alebo SqlDataReader na vrátenie viac riadkov SqlReader na jeden riadok ExecuteScalar pre jednu hodnotu
Modifikácia dát	ExecuteNonQuery pre napojený/pesimistický scenár DataSet pre odpojený/optimistický scenár

Kapitola 9: Webové služby cez HTTP Endpoint

Webová služba je v podstate softvérová aplikácia identifikovaná prostredníctvom URI (Uniform Resource Identifier) Interfejsy a väzby webovej služby je možné definovať, opísať a vyhľadávať ako artefakty XML. Podporuje priamu interakciu s inými softvérovými aplikáciami prostredníctvom správ zapísanými v jazyku XML a prenášanými protokolmi

Webová služba ponúka aplikačnú logiku dostupnú cez Intranet/Internet. Web služby komunikujú použitím Simple Object Access Protocol (SOAP)

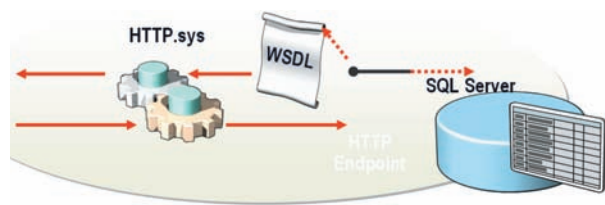
V skratke vymenujeme základné výhody webových služieb:

- Interoperabilita
- multijazyčná podpora
- využitie existujúcich aplikácií
- založené na štandardoch

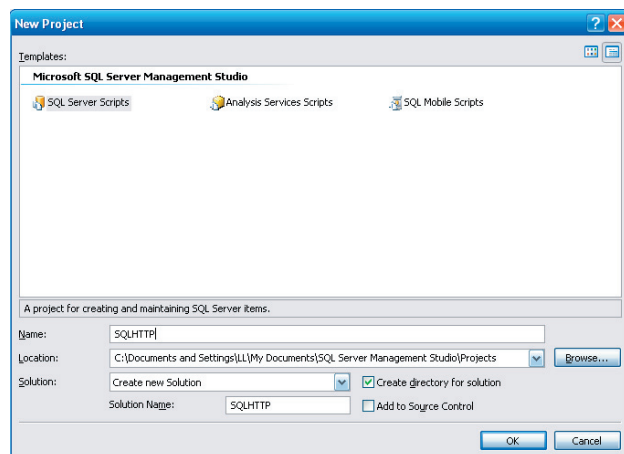
Prístup cez HTTP Endpoint

SQL Server 2005 umožňuje vytváranie prístupových bodov k dátovým objektom, čím podstatne rozširuje možnosti klientského prístupu. K údajom uloženým v databázach je možné pristupovať prostredníctvom webových služieb využívajúcich technológie HTTP, WSDL a SOAP, pričom údaje sú poskytované vo formáte XML. Pomocou http Endpointu môžeme cez HTTP „vystaviť“ uložené procedúry, používateľom definované funkcie, prípadne dávky Transact-SQL príkazov. Zabezpečenie údajov je pritom úplne v správe SQL Servera. Tento prístup môžeme použiť napríklad vtedy, ak máme na firewallle obmedzený počet portov, alebo ak predpokladáme heterogénny prístup k údajom.

Princíp fungovania HTTP Endpointu je na obrázku. Zo schémy je zrejmé, že natívny prístup nevyžaduje IIS, no nutnosťou je serverový operačný systém Windows Server 2003. Proces začína tým, že modul HTTP.sys prijme požiadavku a posunie ju HTTP endpoint-u SQL Servera. SQL Server generuje WSDL dokumentáciu na popisanie metód webovej služby. Metódy webovej služby potom poskytnú prístup k databázovým objektom



Vytvorenie HTTP Endpointu ukážeme na jednoduchom príklade. Pomocou SQL Server Management Studia vytvoríme nový projekt typu SQL Server Scripts, napríklad s názvom SQLHTTP.



Vytvorenie projektu SQLHTTP

Pomocou kontextového menu Solution Explorera v záložke Query vytvoríme nový súbor dopytov s názvom http.sql. V ňom najskôr vytvoríme funkciu, ktorá vráti počet objednávok, ktoré spracováva konkrétny zamestnanec. Parametrom funkcie je ID zamestnanca

```
USE AdventureWorks

GO
CREATE FUNCTION PocetObjednavok(@EmpID INT) RETURNS INT AS
BEGIN
    RETURN
    (
        SELECT COUNT(*) AS ,Pocet objednavok u zamestnanca `
        FROM Purchasing.PurchaseOrderHeader WHERE EmployeeID = @EmpID
        GROUP BY EmployeeID
    )
END
```

Funkciu môžeme v procese ladenia operatívne vyskúšať cez klientskú konzolovú aplikáciu, aby sme v nasledujúcich fázach mali istotu, že sme vybudovali správny základ pre následné vytvorenie HTTP Endpointu.

```
SELECT dbo.PocetObjednavok (231);

-----
360
```

Po overení správnej činnosti funkcie pridáme skript pre vytvorenie endpointu. V tejto etape špecifikujeme aj transportný protokol SOAP - Simple Object Access Protocol. Je to protokol založený na XML. Je určený pre výmenu údajov v decentralizovanom distribuovanom prostredí.

```
CREATE ENDPOINT PocetEndpoint
STATE = STARTED
AS HTTP (
    PATH = ,/sql`,
    AUTHENTICATION = (INTEGRATED),
    PORTS = ( CLEAR )
)
FOR SOAP (
    WEBMETHOD ,http://tempUri.org/`. `DajPocetObjednavok`
        (name=`AdventureWorks.dbo.PocetObjednavok`,
        schema=STANDARD ),
    BATCHES = ENABLED,
    WSDL = DEFAULT,
    DATABASE = ,AdventureWorks`,
    NAMESPACE = ,http://AdventureWorks/Customer`
)
GO
```

V krátkosti vysvetlíme význam parametrov

Pomocou parametra STATE môžeme definovať stav endpointu

STARTED—počúva a odpovedá

DISABLED—nepočúva a samozrejme ani neodpovedá

STOPPED—počúva, no neodpovedá, klientove požiadavky končia chybou

Parameter Path obsahuje virtuálnu URL adresu, kde bude webová služba sídliť. Pomocou parametra Authentication nastavíme typ autentifikácie

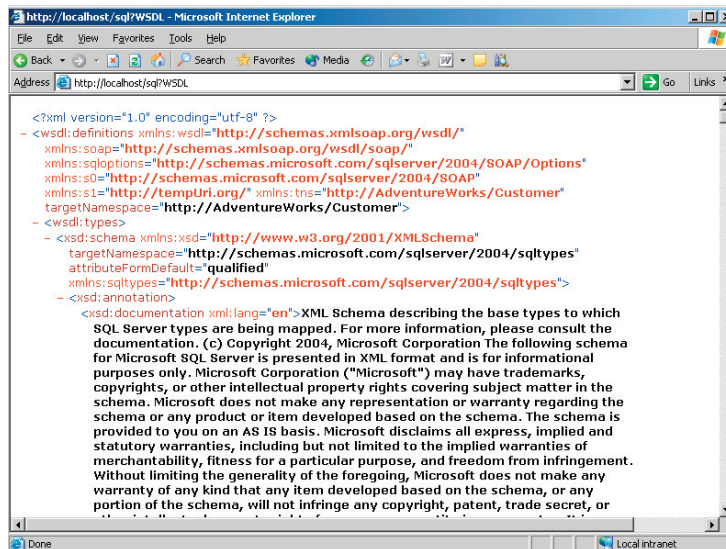
INTEGRATED – najbezpečnejšie nastavenie, ak je to možné použije sa autentifikácia na báze protokolu Kerberos alebo NTLM.

DIGEST - poskytuje menšiu úroveň zabezpečenia ako INTEGRATED. Použije sa vtedy ak nie je možné použiť úroveň INTEGRATED

BASIC – je pri použití protokolu HTTP najmenej bezpečná autentifikácia. Odporúča sa použiť metódy INTEGRATED alebo DIGEST. Pri použití komunikácie cez HTTPS je úroveň zabezpečenia aj pri použití tejto metódy dostatočná

Parameter Ports môžeme nastaviť na hodnoty CLEAR (HTTP – implicitný port 80) alebo SSL (HTTPS - port 443 by default)

Fungovanie HTTP Endpointu môžeme vyskúšať prostredníctvom prehliadača webových stránok, zadaním URL adresy <http://localhost/sql?WSDL>.



Zobrazenie údajov cez WSDL vo formáte XML

V zobrazenom dokumente môžeme nájsť v schéme metódy webovej služby (náš výpis obsahuje len krátky fragment)

```
<xsd:element name="DajPocetObjednavok">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="1" maxOccurs="1" name="EmpID" type="sqltypes:int"
nillable="true" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
- <xsd:element name="DajPocetObjednavokResponse">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="1" maxOccurs="1" name="DajPocetObjednavokResult"
type="sqltypes:int" nillable="true" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Pomocou dopytu (do databázy Master) zistíme informácie o http endpointoch

```
SELECT * FROM sys.endpoints
```

	name	endpoint_id	principal_id	protocol	protocol_desc	type	type_desc	state	state_desc
1	Dedicated Admin ...	1	1	2	TCP	2	TSQL	0	STARTED
2	TSQL Local Mac...	2	1	4	SHARED_MEMO...	2	TSQL	0	STARTED
3	TSQL Named Pipes	3	1	3	NAMED_PIPES	2	TSQL	0	STARTED
4	TSQL Default TCP	4	1	2	TCP	2	TSQL	0	STARTED
5	TSQL Default VIA	5	1	5	VIA	2	TSQL	0	STARTED
6	PocetEndpoint	65536	261	1	HTTP	1	SOAP	0	STARTED

Výsledok SQL dopytu v tabuľke

Prípadne môžeme použiť aj iné typy dopytov. Pomocou posledného dopytu zistíme aj informácie o metódach webových služieb

```
SELECT * FROM sys.http_endpoints
GO
SELECT * FROM sys.soap_endpoints
GO
SELECT * FROM sys.endpoint_webmethods
GO
```

	name	endpoint_id	principal_id	protocol	protocol_desc	type	type_desc	state	state_desc
1	PocetEndpoint	65536	261	1	HTTP	1	SOAP	0	STARTED

endpoint_id	namespace	method_alias	object_name	result_schema	result_schema_de...	result_format
65536	http://localhost/	DajPocetObjednavok	[AdventureWorks:][dbo].[PocetObjednavok]	1	STANDARD	1

Výsledok SQL dopytu v tabuľke

Webová služba je dynamická zložitosť, preto môžeme nastaviť privilégia pre jej zmenu,

```
GRANT ALTER ON ENDPOINT::PocetEndpoint TO AdminAW
```

Prípadne pre zmenu všetkých endpointov

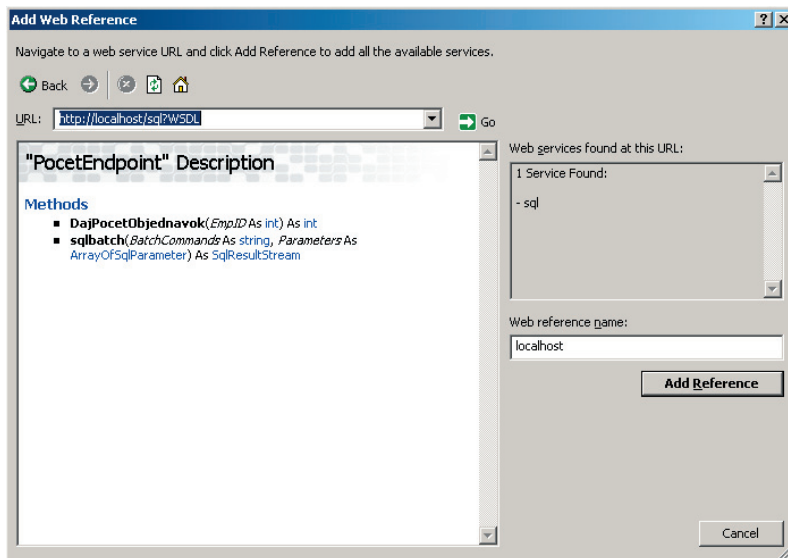
```
GRANT ALTER ANY ENDPOINT TO AdminAW
```

Klient webovej služby

Ukážeme príklad klientskej aplikácie, ktorá bude využívať metódu webovej služby. Postup pri vytvorení aplikácie ktorá bude konzumentom webovej služby http endpoint by sme mohli zhrnúť do niekoľkých bodov

- Vytvorenie projektu v Visual Studiu .NET
- Vytvorenie proxy triedy použitím *Add Web Reference*
- Vytvorenie inštancie proxy
- Nastavenie autentizačných „credentials“
- Zavolanie metódy web.sluzby cez proxy triedu
- Spracovanie odozvy z webovej služby

Do projektu pridáme webovú referenciu na URL adresu HHTP Endpointu *http://localhost/sql?WSDL*.



Pridanie WSDL referencie do projektu

V dialógu zistíme informáciu o metódach webovej služby

Methods

DajPocetObjednavok(EmpID As int) As int

sqlbatch(BatchCommands As string, Parameters As ArrayOfSqlParameter) As
SqlResultStream

Aby bola aplikácia čo najjednoduchšia, metódu webovej služby budeme volať ihneď pri spustení aplikácie v obsluhu udalosti PageLoad. Pre úspešné spustenie programu je potrebné nastaviť prístupové privilégia pre prístup k údajom v databáze Adventure Works

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Net;
using System.Windows.Forms;

namespace MonitObjednavok
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            localhost.PocetEndpoint ws = new localhost.PocetEndpoint();

            ws.Credentials =
                new System.Net.NetworkCredential("AdminAW", "pas@word");
        }
    }
}
```

```
        MessageBox.Show(„Pocet objednavok od zamestnanca ID 231: „ +  
            ws.DajPocetObjednavok(231).ToString());  
    }  
}
```

Pre úspešné spustenie programu je potrebné nastaviť prístupové privilégia pre prístup k údajom v databáze Adventure Works. Ak chceme použiť Windows účet súčasne prihláseného používateľa môžeme nastaviť Default Credentials

```
private void Form1_Load(object sender, EventArgs e)  
{  
    localhost.PocetEndpoint ws = new localhost.PocetEndpoint();  
    ws.Credetials=CredentialCache.DefaultCredentials;  
    MessageBox.Show(„Pocet objednavok od zamestnanca ID 231: „ +  
        ws.DajPocetObjednavok(231).ToString());  
}
```


Príloha č. 1: Zoznámenie sa s databázou Adventure Works

Spolu s SQL Serverom 2005 je dodávaná aj cvičná databáza fiktívnej firmy vyrábajúcej bicykle a bicyklové príslušenstvo. Ak by sme sa pokúsili o popis všetkých takmer sedemdesiatich tabuliek, ktoré tvoria túto databázu, aj napriek veľmi jasne pomenovaným tabuľkám v takejto podobe by sme len veľmi ťažko identifikovali, aké sú vzájomné väzby a filozofia návrhovej štruktúry. Predstaviť firemnú databázu, či už reálnej, alebo v tomto prípade fiktívnej firmy sa nám pravdepodobne najlepšie podarí tak, že predstavíme firemné procesy. Podľa týchto procesov sú objekty rozdelené do schém

Schema	Popis objektov	Príklad tabuliek v schéme
HumanResources	Zamestnanci spoločnosti Adventure Works Cycles.	Employee, Department
Person	Mená a adresy zákazníkov, predajcov a zamestnancov.	Contact, Address, StateProvince
Production	Produkty vyrábané a predávané spoločnosťou Adventure Works Cycles.	BillOfMaterials, Product, ProductCategory, ProductSubcategory, WorkOrder
Purchasing	Dodávatelia súčiastok.	PurchaseOrderDetail, PurchaseOrderHeader, Vendor
Sales	Údaje týkajúce sa obchodu a zákazníkov	Customer, Individual, SalesOrderDetail, SalesOrderHeader, Store, StoreContact

Na základe skúseností s cvičnými databázami NORTHWIND a PUBS z predchádzajúcej verzie SQL servera (viď porovnávacie tabuľky na konci state) by sa mohlo zdať, že nám stačí pracovať so schémou Human Resources, ktorá obsahuje tabuľky Employee,

EmployeeID	NationalIDNumber	ContactID	LoginID	ManagerID	Title	BirthDate	MaritalStatus	Gender	HireDate
1	14417807	1209	adventure-work...	16	Production Technician - WC60	15. 5. 1972 0:00:00	M	M	31. 7. 1996 0:00:00
2	253022876	1030	adventure-work...	6	Marketing Assistant	3. 6. 1977 0:00:00	S	M	26. 2. 1997 0:00:00
3	509647174	1002	adventure-work...	12	Engineering Manager	13. 12. 1964 0:00:00	M	M	12. 12. 1997 0:00:00
4	112457891	1290	adventure-work...	3	Senior Tool Designer	23. 1. 1965 0:00:00	S	M	5. 1. 1998 0:00:00
5	480168528	1009	adventure-work...	263	Tool Designer	29. 8. 1949 0:00:00	M	M	11. 1. 1998 0:00:00
6	24756624	1028	adventure-work...	109	Marketing Manager	19. 4. 1965 0:00:00	S	M	20. 1. 1998 0:00:00
7	309738752	1070	adventure-work...	21	Production Supervisor - WC60	16. 2. 1946 0:00:00	S	F	26. 1. 1998 0:00:00
8	690627818	1071	adventure-work...	185	Production Technician - WC10	6. 7. 1946 0:00:00	M	F	6. 2. 1998 0:00:00
9	695256908	1005	adventure-work...	3	Design Engineer	29. 10. 1942 0:00:00	M	F	6. 2. 1998 0:00:00
10	912265825	1076	adventure-work...	185	Production Technician - WC10	27. 4. 1946 0:00:00	S	M	7. 2. 1998 0:00:00
11	998320692	1006	adventure-work...	3	Design Engineer	11. 4. 1949 0:00:00	M	M	24. 2. 1998 0:00:00

Tabuľka Employee zo schémy HumanResources

a Department

Table - HumanResources.Department	Table - HumanResources.Employee	SQLQuery	
DepartmentID	Name	GroupName	ModifiedDate
1	Engineering	Research and D...	1. 6. 1998 0:00:00
2	Tool Design	Research and D...	1. 6. 1998 0:00:00
3	Sales	Sales and Marke...	1. 6. 1998 0:00:00
4	Marketing	Sales and Marke...	1. 6. 1998 0:00:00
5	Purchasing	Inventory Mana...	1. 6. 1998 0:00:00
6	Research and D...	Research and D...	1. 6. 1998 0:00:00
7	Production	Manufacturing	1. 6. 1998 0:00:00
8	Production Control	Manufacturing	1. 6. 1998 0:00:00
9	Human Resources	Executive Gener...	1. 6. 1998 0:00:00
10	Finance	Executive Gener...	1. 6. 1998 0:00:00
11	Information Servi...	Executive Gener...	1. 6. 1998 0:00:00
12	Document Control	Quality Assurance	1. 6. 1998 0:00:00
13	Quality Assurance	Quality Assurance	1. 6. 1998 0:00:00
14	Facilities and Mai...	Executive Gener...	1. 6. 1998 0:00:00
15	Shipping and Re...	Inventory Mana...	1. 6. 1998 0:00:00
16	Executive	Executive Gener...	1. 6. 1998 0:00:00

Tabuľka Department zo schémy HumanResources

No vidíme, že tabuľka zamestnancov neobsahuje ani ich mená ani ďalšie osobné údaje.

Tieto sú v tabuľke Contact v schéme Person. Tabuľka Contact obsahuje údaje o všetkých osobách vstupujúcich do firemných procesov, a teda aj zamestnancov a zákazníkov.

Na základe naznačených väzieb získame údaje o zamestnancoch napríklad pomocou SQL dotazu do dvoch tabuliek

```
SELECT Em.EmployeeID, LastName, FirstName, EmailAddress, Phone
FROM HumanResources.Employee AS Em
     JOIN Person.Contact AS Co
        ON Co.ContactID = Em.ContactID
ORDER BY LastName, FirstName ;
```

EmployeeID	LastName	FirstName	EmailAddress	Phone
288 0182	Abbas	Syed	syed0@adventure-works.com	926-555-
235 0114	Abercrombie	Kim	kim1@adventure-works.com	208-555-
200 0125	Abolrous	Hazem	hazem0@adventure-works.com	869-555-
85 0185	Ackerman	Pilar	pilar0@adventure-works.com	577-555-
208 0165	Adams	Jay	jay0@adventure-works.com	407-555-
117 0110	Ajenstat	François	francois0@adventure-works.com	785-555-

Ak by sme chceli pracovať s údajmi v XML, všimnime si pozornejšie tabuľku Sales.Individual. V stĺpci Demographics, ktorý je dátového typu XML obsahuje podrobné demografické informácie vrátane informácií o hobby a podobne.

```

<IndividualSurvey xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/IndividualSurvey">
  <TotalPurchaseYTD>8248.99</TotalPurchaseYTD>
  <DateFirstPurchase>2001-07-22Z</DateFirstPurchase>
  <BirthDate>1966-04-08Z</BirthDate>
  <MaritalStatus>M</MaritalStatus>
  <YearlyIncome>75001-100000</YearlyIncome>
  <Gender>M</Gender>
  <TotalChildren>2</TotalChildren>
  <NumberChildrenAtHome>0</NumberChildrenAtHome>
  <Education>Bachelors </Education>
  <Occupation>Professional</Occupation>
  <HomeOwnerFlag>1</HomeOwnerFlag>
  <NumberCarsOwned>0</NumberCarsOwned>
  <Hobby>Golf</Hobby>
  <Hobby>Watch TV</Hobby>
  <CommuteDistance>1-2 Miles</CommuteDistance>
</IndividualSurvey>

```

Porovnanie cvičnej databázy Adventure Works s cvičnými databázami PUBS a NORTHWIND z SQL Servera 2000

Pre čitateľov, ktorí boli zvyknutí používať v predchádzajúcej verzii SQL Servera 2000 jeho cvičné databázy PUBS a NORTHWIND uvádzame zaujímavú približnú analógiu ktorým tabuľkám z týchto databáz vydavateľstiev (PUBS) a fiktívnej firmy (NORTHWIND) sa približne svojou štruktúrou a použitím podobajú tabuľky novej databázy

pubs	AdventureWorks
authors	Purchasing.Vendor
discounts	Sales.SpecialOffer
employee	HumanResources.Employee
jobs	HumanResources.Employee
pub_info	Production.ProductPhoto Production.ProductDescription
publishers	Sales.Store Person.Address Sales.CustomerAddress Person.CountryRegion Person.StateProvince
roysched	Sales.SpecialOffer
sales	Sales.SalesOrderHeader Sales.SalesOrderDetail
stores	Sales.Store
titleauthor	Production.ProductVendor
titles	Production.Product

Northwind	AdventureWorks
Categories	Production.ProductCategory
Customers	Sales.Customer Join with Sales.Individual and Sales.Store
Customer Demographics	Sales.Individual Sales.Store
Employees	HumanResources.Employee Join with Person.Contact
Employee Territories	Sales.SalesPerson
Orders	Sales.SalesOrderHeader
Order Details	Sales.SalesOrderDetail
Products	Production.Product
Region	Sales.SalesTerritory
Shippers	Purchasing.ShipMethod
Suppliers	Purchasing.Vendor
Territories	Sales.SalesTerritory

Zoznam použitej a odporúčanej literatúry:

[1] Luboslav Lacko: SQL Hotová řešení, Computer Press, Praha, 2004,

[2] Bob Beauchemin, Niels Berglund, Dan Sullivan: A First Look at Microsoft SQL Server 2005 for Developers

[3] Michael Otey: Microsoft SQL Server 2005 New Features



Vydal:

Microsoft s.r.o., BB Centrum, budova Alpha, Vyskočilova 1461/2a, 140 00 Praha 4
tel.: +420-261 197 111 , fax: +420-261 197 100, <http://www.microsoft.com/cze>