Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky

Práce s databází SQL SERVER 2005 v prostředí Framework.NET a programovacím jazyku C++

Bakalářská práce

Aplikovaná informatika

vedoucí bakalářské práce
Ing. Miroslav Máče, CSc., Ph.D.

vypracoval
Ondřej Sýkora

České Budějovice, 2007

Annotation

The aim of this dissertation is offer to overview cardinal procedures in development of software in environment of .NET development. Especially in language C++/CLI. Dissertation includes large amount of space about SQL Server 2005. In relation of these two development enviroments come up new applications based on .NET Framework.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně, a že jsem veškeré použité zdroje uvedl v Seznamu použitých zdrojů.

Ondřej Sýkora

1. Úvod

Cílem bakalářské práce je popsat techniky a technologie pro vytváření vícevrstvých aplikací na platformě Windows z pohledu vývojáře. Popisovanými nástroji pro vývoj aplikací jsou z pohledu dat databáze Microsoft SQL Server 2005, z pohledu prezentační a aplikační vrstvy (klient i server) jazyk C++ z vývojové platformy Microsoft .NET Framework (Visual C++ .NET). Nástroj pro vytváření prezentační aplikace je použito vývojové prostředí Visual Studio 2005.

Dokument je po obsahové stránce strukturován do dvou hlavních oddílů.

1. SQL Server 2005 – detailní popis částí databáze pro vývojáře.
2. Visual C++ .NET – popis částí pro vytváření aplikace (formuláře, vybrané knihovny), detailní popis knihoven (tříd a jejich vlastností) pro práci s databází (ADO.NET) a popis vytvoření aplikace vzdáleného řízení (.NET Remoting)

2. Microsoft SQL Server 2005

2.1. Přehled o databázi SQL Server 2005

SQL Server 2005 je datová platforma – všestranné, integrované ale komplexní řešení pro data. Datová platforma SQL Serveru zahrnuje nástroje:

- Relační databáze
- Služba Replication Services – replikace dat pro aplikace zpracovávající distribuovaná data
- Služba Notification Services – funkce zasílání upozornění pro vývoj a nasazení aplikací
- Služba Integration Services – funkce extrakce, transformace a načítání dat pro datové sklady
- Služba Analysis Services – funkce OLAP (Online Analytical Processing) pro analýzu datových sad.
- Službu Reporting Services – řešení pro vytváření, správu a zasílání webových sestav
- Nástroje pro správu – sem patří podpora webových služeb
- Nástroje pro vývojáře

Nelze tedy mluvit pouze o relačně databázovém systému – SQL Server 2005 pokrývá mnohem větší oblast využití.

2.2. Historie vývoje produktu SQL Server

SQL Server se původně zrodil ze Sybase SQL Serveru. Firma Microsoft vstoupila v roce 1989 se Sybase do spolupráce, jejímž cílem bylo vyvinout SQL Server pro systém OS/2. S verzí 4.2 byl převeden na platformu Windows.
Stručný přehled historie vývoje až do verze SQL Server 2000 je zobrazen v tabulce 1.1

Rok	**Verze**	**Stručný popis**
1993	SQL Server 4.2	Funkčně omezená databáze – fungující jako desktop. Schopnost pojmout a udržet data malého obsahu. Avšak pro svou integraci s platformou Windows a jednoduché přístupové rozhraní byla tato verze značně populární
1995	SQL Server 6.5	Zásadní přepracování databázového stroje. První významná verze SQL Serveru (business řešení). Vylepšení

		výkonu a důležitá rozšíření.
1998	SQL Server 7.0	Další změny v přepracování databázového stroje. Tato verze, svým výkonem a možnostmi, vyplnila místo mezi desktop databázemi (např.: Microsoft Access) a výkonnými podnikovými databázemi (např.: Oracle, DB2).
2000	SQL Server 2000	První skutečná podniková databáze od společnosti Microsoft. Stává se již platformou – obsahuje kromě databázového stroje další nástroje – správa, vývoj, analýza, a další

Tabulka 2.1 – Přehled edicí

2.3. Přehled existujících edicí SQL Server 2005

SQL Server 2005 je vydáván v několika různých edicích.

- Enterprise – nejvýkonnější a nejdražší edice s možností největší škálovatelnosti. Zahrnuje všechny uvedené nástroje. Možnost škálovatelnosti ji předurčuje pro prostředí s extrémním zatížení
- Developer – od Enterprise edice se odlišuje v omezení licencí
- Standard – levnější než předchozí uvedené edice; nasazována do prostředí malých a středních podniků; nepodporuje všechny uvedené nástroje (chybí např.: Analysis Services, Integration Services, databáze mirroring a další)
- Workgroups – navržena pro malé a střední podniky – kromě omezení, která jsou uvedena u předchozí edice je tato edice omezena také hardwarově
- Express – volně dostupná edice; její největší nevýhodou je omezení v prostoru pro databázi

Hardwarová omezení pro jednotlivé edice jsou uvedeny v tabulce 1.2

	Enterprise/ Developer	Standard	Workgroup	Express
Max. CPUs	Neomezeno	4	2	1
Max. RAM	Neomezeno	Neomezeno	3GB	2GB
Podpora 64-bit	Ano	Ano		

Max. prostor	Neomezeno	Neomezeno	Neomezeno	4GB

Tabulka 2.2 – Přehled hardwarových omezení pro edice

2.4. Vývojové prostředí pro práci s SQL Server 2005

Pro práci vývojáře je důležité mít správné vývojové prostředí. K SQL Server 2005 se hodí Management Studio. Je dodávané k jednotlivým edicím a splňuje i vyšší požadavky (vysoká míra nastavení si prostředí podle svých představ). Jednotlivé nástroje jsou integrované a po nainstalování Management Studia už není třeba žádných dalších přídavných instalací. Obsahuje nástroje např.:

- Object explorer – průzkumník udrží přehled nad objekty databáze(možnost připojení několika databází najednou)
- Nástroj parsování kódu – identifikace chybné syntaxe
- Nástroj vypočtení tzv. „Execution plan“ – ladění databázových dotazů podle optimizeru
- Nástroj navrhování databázových dotazů
- Vypočítávání statistik
- Různé výstupy výsledků z databázových dotazů
- Nápověda – od lokální nápovědy až po hledání příspěvků na odborných fórech bez nutnosti vyhledávat přes webový prohlížeč

2.5. Transact-SQL alias T-SQL

2.5.1. Úvod do T-SQL

T-SQL nebo-li Transact-SQL je zvláštním dialektem obecného jazyka SQL v SQL Serveru. To znamená, že jazyk T-SQL slouží výhradně potřebám SQL Serveru. Na druhou stranu, ne všechny příkazy, které se v něm zapisují, jsou tak speciální. Jazyk T-SQL do jisté úrovně odpovídá normě ANSI SQL-92, odpovídá široce otevřenému standardu. Pro vývojáře z toho vyplývají pozitiva, že může značnou část jazyka SQL použitou v SQL Serveru použít beze změny i v prostředí jiného databázového serveru založeného na standardu SQL (Sybase, Oracle, Informix, DB2). Odlišnosti od standardu SQL vznikají z důvodu konkrétních potřeb pro daný databázový systém (např.: zvýšení výkonu), proto je dobré využít právě těchto odlišností.

2.5.2. Datové typy v T-SQL

T-SQL tak jako každý jiný jazyk používá ve svém systému své vlastní datové typy. Tabulka 2.3 uvádí přehled datových typů a jejich rozsah hodnota

Datový typ	Rozsah hodnot
`bigint`	Celé číslo od -2^{63} až do 2^{63}.
`binary`	Binární data pevné délky, nejvýše 8000 bajtů.
`bit`	Celé číslo, 0 nebo 1.
`char`	Znaková data pevné délky, nejvýše 8000 znaků.
`datetime`	Datum a čas od 1. ledna 1753 až do prosince 9999.
`float`	Číslo v pohyblivé čárce od -1.79E + 38 do -2.23E – 38 a 2.23E – 38 do 1.79E + 38.
`image`	Binární data proměnlivé délky od 0 do $2^{31} – 1$. Tento datový typ bude v budoucí verzi SQL Serveru odstraněn. Je doporučeno místo něho používat datový typ varbinary(max).
`int`	Celé číslo od -2^{31} do $2^{31} -1$.
`money`	Peněžní částka od -2^{63} do $2^{63} – 1$.
`nchar`	Znaková data Unicode pevné délky, nejvýše 4 000 znaků.
`ntext`	Znaková data Unicode proměnlivé délky, nejvýše 1 073 741 823 znaků. Tento datový tyb bude v budoucí verzi SQL Serveru odstraněn. Je doporučeno místo něho používat datový typ nvarchar(max).
`nvarchar`	Znaková data Unicode proměnlivé délky, nejvýše 4000znaků.
`real`	Číslo v pohyblivé řádové čárce od -1.18E – 38, 0 a 1.18E – 38 do 3.40E – 38.

`smalldatetime`	Datum a čas od 1. ledna 1900 do 6.června 2079.
`smallint`	Celé číslo od -32 768 do 32 767.
`sql_variant`	Datový typ, do něhož se dá uložit hodnota jakéhokoliv datového typu kromě `text, ntext,timestamp, varchar(max), nvarchar(max), varbinary(max), xml, image.`
`table`	Datový typ `table` slouží pro tabulkové proměnné nebo jako úložiště řádků nějaké tabulkové funkce.
`text`	Data proměnlivé délky, nejvýše 2 147 483 647 znaků. Tento datový tyb bude v budoucí verzi SQL Serveru odstraněn. Je doporučeno místo něho používat datový typ `varchar(max).`
`timestamp`	Jedinečné číslo v rámci celé databáze, které se aktualizuje, když se modifikuje řádek.
`tinyint`	Celé číslo od 0 do 255.
`uniqueidentifier`	Globálně jedinečný identifikátor – 16 bajtů.
`varbinary`	Data proměnlivé délky, nejvýše 8000 bajtů. S volbou max umožňuje ukládat až 2^{31} -1 bajtů.
`varchar`	Znaková sada proměnlivé délky, nejvýše 8000 bajtů. S volbou max umožňuje ukládat až 2^{31} -1 bajtů.
`xml`	Tento datový typ umožňuje ukládat nativní data v xml.

Tabulka 2.3 – Přehled datových typů

2.5.3. Rozdělení T-SQL

Jazyk T-SQL lze z pozice vývojáře rozdělit do tří skupin.

- DDL – jazyk pro definici dat
- DML – jazyk pro manipulování s daty

- To ostatní - uložené procedury a uživatelsky definované funkce – zvláštní kapitola patřící výlučně Microsoft SQL Serveru

2.5.3.1. DDL

DDL (Data Definition Language) je jazyk pro definici dat. Tyto příkazy jazyka SQL určují strukturu databáze a jejích objektů (tabulky, indexy). Příkazy DDL uvedu opravdu jen ve zkratce – s mírnou dávkou nadsázky se dá říct, že DDL příkazy nejsou každodenním chlebem vývojáře na rozdíl od příkazů DML. Důvodem okrajového dotčení DDL příkazů je také skutečnost, že Management Studio práci vytváření databáze, tabulek a indexů do značné míry umožní vývojářovi „naklikat“ aniž by byl nucen dlouze vypisovat příkazy ručně.

2.5.3.1.1. Příkaz CREATE

CREATE DATABASE – Příkaz pro vytvoření databáze

Databáze je základním stavebním objektem pro vytváření databázových systémů – bez databáze by to opravdu nešlo. Příklad uvádí syntaxi vytvoření databáze. Databáze je nejvyšším celkem uchovávající data v databázovém systému – všechny ostatní objekty, které uchovávají strukturu databáze jsou vždy přidružené ke své databázi – nacházejí se v kontextu databáze.

```
CREATE DATABASE <název databáze>
[ON [PRIMARY]
([NAME = <’logický název databáze>,]
FILENAME = <’název souboru s cestou, ve kterém je databáze uložena>
[, SIZE = <prostor vyhrazený pro databázi uvedený v KB, MB, GB, TB >]
[, MAXSIZE = maximální vyhrazený pro databázi uvedený v KB, MB, GB, TB >]
[, FILEGROWTH = <nárůst databáze při překročení vyhrazeného prostoru uvedený v % | KB,
MB, GB, TB>])]
[LOG ON
([NAME = <’název log souboru>,]
FILENAME = <’ název log souboru>
[, SIZE = < prostor vyhrazený pro databázi uvedený v KB, MB, GB, TB >]
[, MAXSIZE = maximální vyhrazený pro databázi uvedený v KB, MB, GB, TB >]
[, FILEGROWTH = < nárůst databáze při překročení vyhrazeného prostoru uvedený v % | KB,
MB, GB, TB >])]
 [;]
```

Použití příkazu pro vytvoření databáze vypadá následovně:

```
USE [master]
GO
CREATE DATABASE [NESEHNUTI] ON  PRIMARY
( NAME = N'NESEHNUTI',
FILENAME = N'c:\Program Files\Microsoft SQL Server\MSSQL.1\MSSQL\DATA\NESEHNUTI.mdf' ,
SIZE = 3072KB ,
MAXSIZE = UNLIMITED,
FILEGROWTH = 1024KB )
 LOG ON
```

```
( NAME = N'NESEHNUTI_log',
FILENAME = N'c:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\DATA\NESEHNUTI_log.ldf' ,
SIZE = 1024KB ,
MAXSIZE = 2048GB ,
FILEGROWTH = 10%)
```

CREATE TABLE – Příkaz pro vytvoření tabulky

Tabulka je základním objektem v rámci databáze. Bez tabulek by nebylo možné uchovávat data, to znamená, že by se s nimi nedaly provádět žádné operace.

```
CREATE TABLE [název databáze.[vlastník objektu].]název tabulky
(<column name> <datový typ>
[[DEFAULT <implicitní hodnota>]
|[IDENTITY [(seed, increment)]]  -- automatické id – inkrementuje se automaticky
[NULL|NOT NULL]           -- povolení | zamezení NULL hodnoty
[<column constraints>]          -- omezení pro sloupečky
|[<table_constraint>]            -- omezení pro tabulky
)
```

Následující ukázka uvádí příklad vytvoření tabulky – za zmínku stojí použití příkazu USE, ten určuje v jakém databázovém kontextu se má tabulka vytvořit.

```
USE [NESEHNUTI]
GO
CREATE TABLE [dbo].[osoba](
	[id] [int] IDENTITY(1,1) NOT NULL,
	[jmeno] [varchar](50) NOT NULL,
	[prijmeni] [varchar](50) NOT NULL,
	[titul] [varchar](20) NULL,
	[pohlavi] [varchar](10) NOT NULL,
	[ulice] [varchar](50) NULL,
	[cp] [varchar](20) NULL,
	[obec] [varchar](50) NULL,
	[psc] [int] NULL,
	[okres] [varchar](50) NULL,
	[kraj] [varchar](50) NULL,
	[pusobiste] [varchar](100) NULL,
	[telefon] [varchar](50) NULL,
	[mail] [varchar](100) NULL,
	[aktivni] [varchar](10) NOT NULL CONSTRAINT [DF_osoba_aktivni]  DEFAULT
(N'ANO'),
 CONSTRAINT [PK__osoba__7C8480AE] PRIMARY KEY CLUSTERED
(
	[id] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
```

CREATE INDEX – Příkaz pro vytvoření Indexu

Co jsou Indexy? Indexy asistují při vykonávání databázových dotazů tím, že urychlují přístup k datům, která jsou uložena v tabulkách a pohledech. Indexy umožňují uspořádaný přístup k datům na základě logického řazení řádků dat. Řádky se uspořádávají na základě hodnot uložených

v určitých sloupcích. Tyto sloupce tvoří sloupce klíčů indexu a jejich hodnoty jsou klíče indexu jednotlivých řádků. Indexy se tedy používají, aby se poskytly účinnější metody přístupu k datům. Místo toho, aby se musely pokaždé prohledávat všechny datové stránky tabulky. Index ulehčuje získávání specifických řádků, aniž by se musel číst veškerý obsah tabulky. V normální neindexované tabulce se řádky standardně neukládají v nějakém konkrétním pořadí. Takové tabulce v neuspořádaném stavu se říká halda. Je-li třeba získat řádky z haldy na základě nějaké sady vyhledávacích podmínek, musí SQL Server přečíst všechny řádky tabulky. I když vyhledávacím kritériím bude vyhovovat jen jediný řádek, který může být náhodou načten jako první řádek, který databázový stroj přečte, bude muset SQL Server přesto vyhodnotit i všechny ostatní řádky tabulky, protože nemá žádnou šanci zjisti, že v tabulce už není žádný další vyhovující řádek. Takové hledání informací se nazývá jako úplné prohledávání tabulky – full table scan. U rozsáhlých tabulek to může znamenat přečíst i miliony záznamů jen proto, aby získal požadovaný řádek. Kdyby taková tabulka byla indexována a zmíněný dotaz by hledal dle indexu, dojde k vyhledání mnohem efektivněji – efektivně znamená především rychleji. Proto je databázový objekt Index velmi důležitý z pohledu vývojáře.
Příklad jak vytvořit Index vypadá následovně:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX název_indexu
       ON {
       [název_databáze. [název_schématu] . | název_schématu. ]
název_tabulky_nebo_pohledu}
(název_sloupce [ ASC | DESC] )
```

Argumenty příkazu jsou uvedeny v tabulce 1.4

Argument	**Popis**
`UNIQUE`	Argument říká, jsou-li hodnoty v tomto Indexu jedinečné.
`CLUSTERED | NONCLUSTERED`	Specifikuje typ Indexu. Clusterovaný index může být v tabulce jen jeden, neclusterovaných indexů může být až 249.
název_indexu	Název nového indexu.
`[název_databáze. [název_schématu] .[název_schémetu. ] název_tabulky`	Tabulka (nebo pohled), který se má indexovat.

`název_sloupce`	Název sloupce (nebo sloupců), které se mají indexovat

Tabulka 2.4 – Argumenty příkazu CREATE INDEX

Ukázka vytvoření indexu:

```
CREATE NONCLUSTERED INDEX [osoba_mail_index] ON [dbo].[osoba]
(
	[mail] ASC
)
```

2.5.3.1.2. Příkaz ALTER

Příkaz ALTER provádí uložení provedených úprav nad databázovým objektem (databází, tabulkou, indexem). To znamená, máme-li již vytvořen nějaký databázový objekt v databázi a chceme ho změnit, provedeme příkaz ALTER. Na uvedených objektech, na kterých jsem ukázal příkaz CREATE, předvedu příkaz ALTER. Stejně jako pro vytváření objektů i pro úpravu objektů lze plně využít Management Studio. Příkaz ALTER tedy uvádím pouze jako přehled použití.

ALTER DATABASE – Příkaz pro úpravu existující databáze

```
ALTER DATABASE <název databáze>
ADD FILE
([NAME = <’název databáze’>,]
FILENAME = <’ název souboru s cestou, ve kterém je databáze uložena’>
[, SIZE = <prostor vyhrazený pro databázi uvedený v KB, MB, GB, TB >]
[, MAXSIZE = maximální vyhrazený pro databázi uvedený v KB, MB, GB, TB >]
[, FILEGROWTH = < nárůst databáze při překročení vyhrazeného prostoru uvedený v % | KB,
MB, GB, TB >])]
[, OFFLINE ]
|ADD LOG FILE                    -- přidání log souboru
([NAME = <’ název log souboru’>,]
FILENAME = <’ název log souboru’>
[, SIZE = < prostor vyhrazený pro databázi uvedený v KB, MB, GB, TB >]
[, MAXSIZE = maximální vyhrazený pro databázi uvedený v KB, MB, GB, TB >]
[, FILEGROWTH = <nárůst databáze při překročení vyhrazeného prostoru uvedený v % | KB,
MB, GB, TB>])]
|REMOVE FILE < název databáze > [WITH DELETE] -- odebrání log souboru
```

ALTER TABLE – Příkaz pro úpravu existující tabulky

```
ALTER TABLE název_tabulky
{[ALTER COLUMN <název_sloupce>
[NULL|NOT NULL]
|ADD
<název_sloupce> <datový_typ>
[[DEFAULT <implicitní_hodnota>]
|[IDENTITY [(<počáteční_hodnota>, <inkrementační_hodnota>) [NOT FOR REPLICATION]]]]
 [NULL|NOT NULL]
[<omezení_sloupce>]
|ADD
[CONSTRAINT <omezení_název>]
{[{PRIMARY KEY|UNIQUE}
[CLUSTERED|NONCLUSTERED]
```

ALTER INDEX – Příkaz pro úpravu existujícího Indexu

```
ALTER INDEX { <name of index> | ALL }
ON <table or view name>
{ REBUILD                                -- rebuild indexu – index bude kompletně
přepracován
 [ [ WITH (
| [[,] SORT_IN_TEMPDB = { ON | OFF } ]
| [[,] IGNORE_DUP_KEY = { ON | OFF } ]
| [[,] STATISTICS_NORECOMPUTE = { ON | OFF } ]
| [[,] ONLINE = { ON | OFF } ]
| [[,] ALLOW_ROW_LOCKS = { ON | OFF } ]
| [[,] ALLOW_PAGE_LOCKS = { ON | OFF } ]) ]
| [ PARTITION = <partition number>
[ WITH ( <partition rebuild index option> ) ] ] ]
| DISABLE                                -- zneplatnění indexu – vypnutí zamezení
funkcionality
| REORGANIZE                             -- provede optimalizaci indexu, vynechá
optimalizaci listů
[ PARTITION = <partition number> ]
[ WITH ( LOB_COMPACTION = { ON | OFF } ) ]
| SET ([ ALLOW_ROW_LOCKS= { ON | OFF } ]
| [[,] ALLOW_PAGE_LOCKS = { ON | OFF } ]
| [[,] IGNORE_DUP_KEY = { ON | OFF } ]
| [[,] STATISTICS_NORECOMPUTE = { ON | OFF } ]
)}
```

2.5.3.1.3. Příkaz DROP

Posledním DDL příkazem je příkaz DROP. Příkaz DROP je nejjednodušší DDL příkazem. Jak z názvu příkazu se dá vyčíst, co tento příkaz umí - provede smazání (lépe řečeno zrušení) objektu. Obecná syntaxe příkazu DROP:

```
DROP <typ_objektu> <název_ objektu>
```

DROP DATABASE – Příkaz pro zrušení objektu databáze

```
DROP DATABASE <název_databáze>
```

Jen pro jistotu dodám nad slunce jasnou informaci, bude-li vykonán tento příkaz dojde ke zrušení úplně všech jeho objektů.

DROP TABLE – Příkaz pro zrušení objektu tabulka

```
DROP TABLE <název_tabulky>
```

Po vykonání příkazu na zrušení tabulky dojde ke zrušení tabulky včetně všech jejích dat.

DROP INDEX – Příkaz pro zrušení objektu Index

```
DROP INDEX <název_tabulky>.<název_indexu>
```

Po vykonání příkazu na zrušení objektu Index dojde ke zrušení Indexu. Nebudou dotčena žádná data, ale dojde ke zrušení navázaných objektů (je-li index používán např.: jako primární klíč, na který je odkazován jiný, cizí klíč, dojde ke zrušení této vazby) .

2.5.3.2. DML

Příkazy skupiny DML (Data Manipulation Language – volně přeloženo jako „jazyk pro manipulaci s daty“) jsou příkazy požívané v databázích pro účely získávání dat z tabulek, vkládání dat do tabulek, mazání dat v tabulkách a změnu dat v tabulkách. Jsou to příkazy SELECT, INSERT, DELETE a UPDATE. Na rozdíl od příkazů DDL, jsou právě příkazy DML tím nejstěžejnějším pro práci vývojáře.

2.5.3.2.1. Příkaz INSERT – Příkaz pro vložení dat do tabulky

Obecná syntaxe příkazu INSERT

```
INSERT [INTO] <název_tabulky>
      [(seznam_sloupců)]
VALUES (DEFAUL | NULL, výraz)
```

Příkaz INSERT slouží pro vkládání řádků do tabulky. V té nejpřirozenější podobě vkládá příkaz INSERT pouze jeden řádek do tabulky viz uvedený příklad.

```
INSERT INTO OSOBA(
jmeno, prijmeni, titul, pohlavi, ulice, cp, obec, psc, okres, kraj, pusobiste, telefon,
mail, aktivni)
VALUES(
'Petr','Klíčník',null, 'M', 'J. Š. Baara', '49', 'Brno',37001, 'Brno', 'Brno', 'Brno',
'+420601131231','p.kl@gmail.com', 'ANO')
```

Tento příklad uvádí sloupce definované pro tabulku osoba. V přídě, že jsou vkládány data do všech sloupců může být vypuštěna část seznamu sloupců. Jisté ulehčení práce je tedy možnost napsat příkaz INSERT takto:

```
INSERT INTO OSOBA
VALUES(
'Petr','Klíčník',null, 'M', 'J. Š. Baara', '49', 'Brno',37001, 'Brno', 'Brno', 'Brno',
'+420601131231','p.kl@gmail.com', 'ANO' )
```

Seznam sloupců je třeba uvést v případě, že nejsou vkládána data do všech sloupců – je třeba uvést sloupce, kterých se příkaz INSERT týká:

```
INSERT INTO OSOBA
(jmeno, prijmeni, pohlavi, telefon, mail)
VALUES
('Petr','Klíčník','M', '+420601131231', 'p.kl@gmail.com')
```

Uvedené příkazy umí vložit pouze jeden požadavek. Je dobré vědět, že existuje způsob, jak do tabulky vkládat více než jeden řádek. Pro tento způsob vkládání řádků do tabulky existuje příkaz INSERT…SELECT – použije na místo výčtu vkládaných hodnot příkaz SELECT (bude uveden o kousek dál). Zvlášť vhodný je tento způsob vkládaný v databázových systémech, kde

jsou zavedeny tzv. archivační tabulky. Archivační tabulka slouží pro přesun dat z aktivní tabulky v případě, kdy již nejsou v aktivní tabulce potřebné. To znamená, že archivní tabulka má totožnou strukturu jako tabulka aktivní.
Syntaxe příkazu INSERT…SELECT se liší od „obyčejného“ INSERT příkazu jen v uvedení vkládaných hodnot:

```
INSERT [INTO] <název_cílové_tabulky> [seznam_sloupců]
SELECT seznam_sloupců FROM <název_zdrojové_tabulky>
```

Praktický příklad by vypadalo následovně:

```
INSERT procesy_archiv (proces_id, proces_nazev, proces_start, proces_konec)
SELECT (proces_id, proces_nazev, proces_start, proces_konec)
FROM procesy_aktiv
WHERE proces_konec is not null
```

2.5.3.2.2. Příkaz UPDATE – Příkaz pro modifikaci uložených dat

Obecná syntaxe příkazu UPDATE

```
UPDATE <název_tabulky>
SET <název_sloupce> = {výraz | DEFAULT | NULL}
WHERE <podmínka>
```

Příkaz provede změnu dat nad uvedenými sloupci, které se mají měnit pro všechny řádky v tabulce, pro něž platí podmínka v klauzuli WHERE – může dojít ke změně dat např.: nad jedním jediným řádkem, ale stejně tak mohou být změněna data v celé tabulce.
Příklad provedení příkazu UPDATE na základě klauzule WHERE – hodnota ve sloupci ‘aktivni‘ bude změněna na hodnotu ‘NE‘:

```
UPDATE osoba
SET aktivni = 'NE'
WHERE jmeno = 'Petr' AND prijmeni = 'Klíčník'
```

2.5.3.2.3. Příkaz DELETE – Příkaz pro mazání záznamu z tabulky

Obecná syntaxe příkazu DELETE

```
DELETE [FROM] <název_tabulky>
WHERE <podmínka>
```

Příkaz DELETE maže záznam z tabulky, pro něž je splněna podmínka v klauzuli WHERE. Příkazem je možné mazat záznamy jednotlivě, ale i všechny záznamy v tabulce. Avšak pro případ, že je potřeba smazat všechna data z tabulky je lepší použít příkaz TRUNCATE – ten provede vyprázdnění tabulky bez protokolování akce, to znamená mnohem rychleji.

Příklad pro provedení příkazu DELETE může vypadat takto – potřebuji smazat všechny osoby z tabulky ‘osoba‘, kteří mají ve sloupci ‘aktivni‘ hodnotu ‘NE‘:

```
DELETE osoba WHERE aktivni = 'NE'
```

2.5.3.2.4. Příkaz SELECT – Příkaz pro získání dat z tabulky

Příkaz SELECT je ten nejdůležitější příkaz ve světě SQL. Tento příkaz získává data (hodnoty ve sloupcích) z tabulek databází.
Základní syntaxe příkazu SELECT v T-SQL je následující:

```
SELECT <seznam_sloupců>
[INTO <název_tabulky>]
[FROM <zdrojová_tabulka>]
[WHERE <omezující_podmínka>]
[GROUP BY <název_sloupce_nebo výraz_založený_na_sloupci_ze_seznamu_select>]
[HAVING <omezující_podmínka_postavena_na_výsledcích_klauzule_GROUP_BY>]
[ORDER BY <seznam_sloupců>]
[FOR XML] [RAW, AUTO, EXPLICIT] [, XMLDATA] [, ELEMENTS][, BINARY base 64]]
[OPTION (<pokyn_optimalizatoru_dotazu>)]
```

Příkaz SELECT tedy získává řádky ze zdrojové tabulky. Je-li napsán příkaz:

```
SELECT * FROM osoba
```

tento databázový dotaz vybere všechny záznamy v tabulce 'osoba' bez omezení. Jsou-li třeba získat data, na základě nějaké konkrétní podmínky, bude použita klauzule WHERE.

Klauzule WHERE

Klauzule WHERE omezuje výběr řádků z tabulky. Omezení se porovnává přes operátory. Které operátory lze použít zobrazuje následující tabulka.

Operátor	Popis
`=, >, <, >=, <=,<>, !=,!>, <!`	Standardní operátory porovnání
`AND, OR, NOT`	Standardní booleovská logika; pomocí těchto operátorů může spojit několik dílčích podmínek do jedné klauzule WHERE.
`BETWEEN`	Vyznačuje uzavřený interval hodnot.
`LIKE`	Porovnání podle vzoru. Umožňuje zápis zástupných znaků - % a _. Znak procento může nahradit libovolný počet znaků, znak podtržení nahrazuje

	jeden libovolný znak
IN	Vrací TRUE, pokud se hodnota levé straně klíčového slova IN shoduje s libovolnou z hodnot, zapsaných v seznamu po pravé straně operátoru.
ALL	Když se použije s porovnávacím operátorem a poddotazem, tak se vrátí řádky, jestliže všechny získané hodnoty splňují podmínku.
ANY	Když se použije s porovnávacím operátorem a poddotazem, tak se vrátí řádky, jestliže alespoň jedna získaná hodnota splňuje podmínku.
SOME	Když se použije s porovnávacím operátorem a poddotazem, tak se vrátí řádky, jestliže alespoň jedna získaná hodnota splňuje podmínku.
EXISTS	Nabude hodnoty TRUE, pokud zadaný poddotaz vrátí alespoň jeden řádek.
CONTAINS	Slouží k vyhledání slov a frází.
IS NULL	Zjistí, jestli zadaná hodnota je hodnota NULL
ESCAPE	Bude chápat znak uvedený před nějakým zástupným symbolem tak, že se má hledat přímo daný zástupný symbol, což znamená, že daný znak nebude použit jako zástupný symbol.
FREETEXT	Vyhledává v datech znakového charakteru slova podle významu, ne jako literál.

Tabulka 2.5 – Přehled operátorů pro použití v klauzuli WHERE

Klauzule ORDER BY

Použijeme-li klauzuli ORDER BY, obdržené řádky vrácené databázovým strojem, budou seřazeny dle nastavených podmínek řazení. Využití této klauzule zní dobře především s využitím klíčového slova TOP, kdy toto klíčové slovo říká databázovému stroji, že požaduje pouze několik prvních řádků. Kolik to řádků bude záleží na zadavateli dotazu. Následující dotaz vrátí pouze prvních pět záznamů, seřazených podle sloupce ‘id_uziv‘; na výběr máme buď řazení vzestupné (ASC) nebo řazení sestupné (DESC):

```
SELECT TOP 5 id_uziv, uživatel
FROM uživatel
```

```
ORDER BY id_uziv ASC
```

Klauzule GROUP BY – seskupování dat

Pomocí klauzule GROUP BY se v dotazu SELECT určují skupiny, do nichž mají jednotlivé řádky patřit. V příkladu je uveden dotaz, který vybere kraje a zjistí, kolik mužů z tabulky ‘osoba‘ je vedeno v tomto kraji:

```
SELECT kraj as 'NÁZEV KRAJE' , count(osoba) as 'POČET MUŽŮ VEDENÝCH V KRAJI'
FROM osoba
WHERE pohlavi = 'M'
GROUP BY okres
```

V dotazu jsem použil alias pro sloupce (as 'NÁZEV KRAJE') a také agregační funkci, takže na tomto místě je ten správný okamžik, abych o agregačních funkcích podal více informací.

Agregační funkce

Agregační funkce se obvykle vyskytují v klauzuli GROUP BY – v této souvislosti pracují se skupinami dat. V předchozím příkladu se pro každou skupinu (každý existující kraj v tabulce ‘osoba‘) spočítal celkový počet mužů (viz klauzule WHERE). Agregační funkce ukazují svoji sílu zejména v klauzuli GROUP BY. To ale neznamená, že by agregační funkce byly použitelné jen v těchto funkcích. Zapíše-li se totiž agregační funkce do běžného dotazu bez klauzule GROUP BY, provede se agregační funkce nad celou vybranou sadou (vyberou se všechny řádky, které vyhovují podmínce WHERE). Ale u dotazů bez klauzule GROUP BY se určité agregace smí ve výběrovém seznamu objevit jen vedle jiných agregací. To znamená, že pokud dotaz neobsahuje klauzuli GROUP BY, nesmí se agregace s názvem nějakého sloupce zapsat do výběrového seznamu. Jestliže tedy v dotazu chybí klauzule GROUP BY, můžeme například funkci AVG spojit s funkcí SUM, nikoliv však s konkrétním sloupcem.

Funkce AVG

Tato funkce slouží k výpočtu průměru.

Funkce MIN / MAX

Význam těchto funkcí je odvoditelný u jejich názvů. Ano, jedná se o funkce na zjištění minima a maxima z každého vybraného sloupce.

Funkce COUNT

Funkce COUNT() slouží ke zjištění (spočítání) počtu řádků ve výsledcích dotazu. Nejběžnější příklad užití této funkce je pro získávání informace o velikosti tabulky, co se počtu řádku týká. Kromě původního příkladu na agregační funkci uvedu příklad na zjištění počtu oprávnění pro danou osobu:

```
SELECT os.prijmeni, count(op.id_opravneni)
FROM osoba os, uzivatel uz, opravneni op
WHERE os.id = uz.id_osoba
AND uz.id_uziv = op.id_uziv
GROUP BY os.prijmeni
```

Klauzule INTO

Klauzule INTO příkazu SELECT umožňuje vytvořit novou tabulku na základě sloupců a řádků výsledků dotazu. V ideálním případě by se měly všechny tabulky vytvářet definičním příkazem CREATE TABLE. Ale klauzule INTO poskytuje rychlý a pohotový prostředek, jak vytvořit novou tabulku, aniž by bylo třeba explicitně definovat názvy a datové typy jejích sloupců. Syntaxe INTO vypadá takto:

```
SELECT <seznam_sloupců>
INTO <cílová_tabulka>
FROM <zdrojové_tabulky>
```

Konkrétní použití klauzule INTO:

```
SELECT
kod,
nazev,
reference,
popis
INTO param_vycet_archiv
FROM param_vycet
```

Po uvedeném příkladu dojde k vytvoření nové tabulky (param_vycet_archiv), kde datové typy sloupců jsou totožné jako datové typy ve zdrojové tabulce (param_vycet). Výhoda se zdá být jasná, není třeba explicitně definovat tabulku zápisem nové struktury. Ale tento příkaz má jednu vlastnost, takovou, že dojde ke zkopírování struktury a naplnění nově vzniklé tabulky řádky, který vrátí výsledek dotazu. Nedojde však k vytvoření dalších navázaných objektů, jako je tomu u tabulky zdrojové. Takže po tomto příkaze bude ještě třeba doplnit ostatní objekty, jako jsou omezení, indexy a jiné objekty dodatečně.
Pro případ, že je třeba pouze nakopírovat struktura, ale nevkládat řádky z výsledného dotazu, jednoduše uvedený příkaz omezíme v klauzuli WHERE. Předchozí příkaz bude doplněn tou správnou WHERE klauzulí, aby splnil požadavek nevložení nových řádků:

```
SELECT
kod,
nazev,
reference,
popis
INTO param_vycet_archiv
FROM param_vycet
WHERE 1 = 0
```

Klauzule HAVING

Co umí klauzule HAVING? Doposud byly všechny podmínky formulovány pro konkrétní řádky v klauzuli WHERE. Jestliže daný sloupec neobsahoval zadanou hodnotu, případně zadanou hodnotu daného intervalu, pak byl celý řádek vynechán. To vše databázový stroj provádí ještě před tím, než začne seskupovat jednotlivé hodnoty ve sloupcích. Co tedy použít v případě, že je potřeba definovat podmínky pro podobu samostatných skupin, kdy je třeba zahrnout do skupin všechny jednotlivé řádky, ale potřebnou podmínku vykonat až po naplnění všech skupin? Právě pro tyto potřeby je vhodná klauzule HAVING. Klauzuli lze uvést do dotazu jen tehdy, že je v něm uvedena klauzule GROUP BY. Zatímco ale na všechny jednotlivé řádky se aplikuje klauzule WHERE, která rozhodne, jestli má daný jednotlivý řádek vůbec šanci se stát členem nějaké skupiny, klauze HAVING se aplikuje až na agregovanou hodnotu příslušné skupiny.
Jestliže si prohlídnete předešlý příklad u klauzule GROUP BY, zjistíte, že dotaz vracel celkový počet mužů vedených v každém kraji. Následující příklad použije onen vzor a doplní klauzuli HAVING, která omezí výstup jen na kraje, kde je vedeno více než 10 mužů:

```
SELECT kraj as 'NÁZEV KRAJE' , count(osoba) as 'POČET MUŽŮ VEDENÝCH V KRAJI'
FROM osoba
WHERE pohlavi = 'M'
GROUP BY okres
HAVING count(osoba) >10
```

Příkaz funguje v následujícím pořadí: Nejprve se provede SELECT, který podle klauzule WHERE vyhledá všechny muže (pohlavi = 'M'), poté provede agregační příkaz GROUP BY a spočítá kolik se v daném kraji nachází vedených mužů. A až na konec se provede omezení v klauzuli HAVING.

Obecné techniky pro příkaz SELECT

Primárním účelem příkazu SELECT je definovat, které sloupce se mají vrátit v sadě výsledků dotazu, ale jeho funkcionalita není omezena pouze na toto. Následující seznam uvedených bodů ukazuje techniky, které je možné použít v příkazu SELECT:

- Odstranit duplicitní hodnoty klíčovým slovem DISTINCT.
- Přejmenování sloupců pomocí aliasů.

- Zřetězení hodnot řetězců do jediného sloupce
- Vytvořit seznam hodnot oddělených čárkou
- Použití příkazu SELECT pro vytvoření skriptu

Odstranění duplicitních hodnot pomocí DISTINCT

Výchozí chování příkazu SELECT je takové, že implicitně používá klíčové slovo ALL. Znamená to, že se získají a zobrazí i duplicitní řádky (v případě, že existují). Bude-li uvedeno místo ALL klíčové slovo DISTINCT, v navrácené sadě se zobrazí pouze jedinečné řádky. Jako příklad můžeme použít dotaz do tabulky seznamu oprávnění a vybrat všechny uživatele, které jsou zde uvedeni:

```
SELECT DISTINCT os.prijmeni, os.jmeno
FROM  osoba os, uzivatel uz, opravneni op
WHERE os.id = uz.id_osoba
AND uz.id_uziv = op.id_uziv
```

Přejmenování sloupců pomocí aliasů, spojování řetězců

Pro vypočítávané sloupce, agregační funkce nebo zřetězené hodnoty se mohou specifikovat aliasy, jimiž se definují explicitní názvy těchto sloupců pro výstup dotazu. Pomocí aliasu sloupců se také dají přejmenovat sloupce, které už mají přidělený název. Alias sloupce se specifikuje pomocí klíčového AS nebo se jednoduše uvede za názvem sloupce nebo vypočítávaného sloupce. Následující příklad uvádí spojení dvou uvedených technik – použití aliasů a zároveň zřetězení výstupních hodnot do jedné výstupní hodnoty:

```
SELECT
jmeno + ' ' +
prijmeni + ' bydli na adrese: ' +
ulice + ' ' +
cp + ' v obci ' +
obec + ' jenž patří do ' +
kraj + 'ho kraje ' AS 'Komentář k bydlení'
FROM osoba
```

Zde uvedenému aliasu předchází klíčové slovo DISTINCT. Ale jestliže ho vypustím, výsledek bude stejný. Přikláním se k názoru, že je vždy lepší používat nezkrácený zápis. V tomto případě vypuštění klíčového slova DISTINCT žádný čas neušetří. Nezkrácený zápis nám udrží transparentnost kódu.

Vytvoření seznamu hodnot oddělených čárkou

Následující technika uvádí, jak se pomocí dotazu SELECT vytvoří seznam hodnot oddělených čárkami. Toto se může hodit v případě, kdy je třeba provést integraci hodnot do uživatelsky

definované funkce, která bude vracet seznam hodnot (tyto hodnoty budou odděleny čárkou). Následující příklad jak tento seznam vytvořit:

```
DECLARE @mail varchar(50)
SET @mail = ''
SELECT @mail=@mail + mail + ', '
FROM osoba
SELECT @mail
```

Nyní se do kódu vmísily novinky, o kterých bude pojednáno v kapitole „To ostatní“. Ale tento kus kódu splnil očekávání, na výstupu se objevil seznam mailových adres oddělených čárkou.

Použití příkazu SELECT pro vytvoření skriptu

Jak na databázový administrátor, tak i vývojář se mohou ocitnout v situaci, kdy potřebuje vytvořit skript T-SQL, který by se mohl spouštět nad několika objekty, nacházejícími se uvnitř nějaké databáze nebo nad několika databázemi nějaké instance SQL Serveru. Nebo existuje nějaká velmi rozsáhlá tabulka, s mnoha sloupci, jejichž platnost je třeba ověřit pomocí vyhledávacích podmínek, ale psát názvy všech sloupců ručně by mohlo být náročné.
Uvedený příklad nabízí techniku, která dokáže ušetřit čas, protože skript jazyka T-SQL za vývojáře umí vytvořit příkaz. V příkladu předpokládám, že je třeba najít řádky, které obsahují hodnotu NULL (nebo třeba NOT NULL, záleží na výběru). Tabulka má větší počet sloupců, proto bude využita právě tato technika:

```
SELECT column_name + ' IS NULL AND '
FROM information_schema.columns
WHERE table_name = 'osoba'
ORDER BY ORDINAL_POSITINO
```

Vráceným výsledkem z databázového stroje bude následující:

```
id IS NULL AND
jmeno IS NULL AND
prijmeni IS NULL AND
titul IS NULL AND
pohlavi IS NULL AND
ulice IS NULL AND
cp IS NULL AND
obec IS NULL AND
psc IS NULL AND
okres IS NULL AND
kraj IS NULL AND
pusobiste IS NULL AND
telefon IS NULL AND
mail IS NULL AND
aktivni IS NULL AND
```

což snadno můžeme zakomponovat do připraveného příkazu SELECT a klauzule WHERE. Dokážete-li si představit tabulku s větším počtem sloupců, pak oceníte tuto techniku vytváření skriptů z databázových dotazů.

Klauzule FOR XML

SQL Server také nabízí možnost výstupu výsledků ve formátu XML na místo klasické výsledné sady. Tato funkcionalita se může hodit v prostředí s různými platformami.
Podrobnosti pro použití XML zde uvedeny nebudou, ale jen okrajově uvedu příklad jak lze klauzuli FOR XML použít, aby vrátila výsledek a v jakém formátu:

```
SELECT *
FROM osoba
FOR XML RAW
```

Příkaz provede obyčejný SELECT s klauzulí FOR XML u něhož je použit argument RAW. Tento argument specifikuje výstupní formu výsledku dotazu. A jak vypadá konečný výstup? Uveden je jeden řádek z výsledku – ve formátu XML.

```
<row id="13" jmeno="Petr" prijmeni="Klíčník" titul="Ing." pohlavi="M" ulice="J. Š.
Baara" cp="49" obec="České Budějovice" psc="37001" okres="České Budějovice"
kraj="Jihočeský" pusobiste="České Budějovice" telefon="+420601131231"
mail="p.kl@gmail.com" aktivni="ANO" />
```

Klauzule OPTION – nebo-li HINT

Tato klauzule představuje možnost jak říci databázovému stroji (resp. optimalizátoru databázového stroje), aby se neřídil podle svých vlastních představ při sestavování výkonného plánu, jak realizovat databázový dotaz příkazem SELECT. Touto klauzulí lze tedy nastavit (v případě jejího uvedení) jak se má optimalizátor databázového stroje chovat při sestavování dotazů. Nutno podotknout, že v drtivé většině se optimalizátor SQL Serveru zachová nejefektivněji a zvolí tu nejrychlejší cestu pro obdržení výsledné sady. Klauzule OPTION by se měla používat jen tehdy, když selže efektivita vykonání procesu rozhodování optimalizátoru databázového stroje.

Poddotazy

Poddotaz je takový dotaz SELECT, který je vnořen do jiného příkazu SELECT, INSERT, UPDATE nebo DELETE. Poddotaz může být také vnořen do jiného poddotazu. Poddotazy lze často přepsat do dotazů s normálními spojovacími podmínkami (JOIN), někdy však mají lepší výkon, než ekvivalentní dotazy, v nichž se poddotazy nepoužívají. Uvedený příklad využívá operátor EXISTS

```
SELECT DISTINCT os.prijmeni, os.jmeno
FROM osoba os
WHERE EXISTS (
SELECT uz.id_osoba
FROM uzivatel uz, opravneni op
WHERE uz.id_uziv = op.id_uziv
AND os.id = uz.id_osoba)
```

Operátor EXISTS je využíván protože v některých konkrétních případech pracuje efektivněji než dotazy, kde se na místo operátoru EXISTS použije spojení tabulek. Jak je to možné? Je to tím, že operátor EXIST vrací booleovskou hodnotu (FALSE nebo TRUE). Booleovskou hodnotu vrací ve chvíli, kdy vyhodnotí poddotaz jako úspěšný (tzn.: nalezne alespoň jeden řádek v poddotazu) a dál již nehledá – zajímá ho jen jeden jediný výskyt na místo hledání všech výskytů při spojování tabulek nebo jako neúspěšný ve chvíli, když se pro uvedený poddotaz nenalezl ani jeden odpovídající řádek.

Spojování tabulek – dotazy na více zdrojů dat

Při práci v prostředí normalizovaných tabulek (resp.: normalizovaná databáze) nastávají situace, kdy všechny požadované informace nelze zjistit z jediné tabulky. A nebo nastávají situace, kdy k formulování podmínek jsou potřebná data v jiné tabulce. Pro takové situace se používají spojování tabulek. V SQL Serveru k tomuto spojování tabulek slouží klauzule JOIN. Toto klíčové slovo propojí informace ze dvou tabulek do jedné výsledné sady. Tyto výsledné sady se dají označit za virtuální tabulky, která má sloupce a řádky. Jak funguje klauzule JOIN při spojování dat do výsledné sady? To určí způsob použití při spojování tabulek. Celkem jsou čtyři typu JOIN klauzule:

- INNER JOIN – vnitřní spojení
- OUTER JOIN – vnější spojení (může být buď pravé nebo levé spojení – RIGHT OUTER JOIN, nebo LEFT OUTER JOIN)
- FULL JOIN – plné spojení
- CROSS JOIN – křížové spojení

Vnitřní spojení – INNER JOIN

Toto spojení mezi tabulkami je nejběžnějším spojením. Spojení INNER JOIN spojuje záznamy podle shody jednoho nebo více společných polí. Jak se spojování INNER JOIN děje?
Ve spojovaných tabulkách se najdou pole, která se shodují. Do výsledné sady jsou zahrnuty záznamy, které si podle shodných polí odpovídají. To znamená, že se do výsledku zahrnou záznamy, které mají na obou stranách JOIN odpovídající protějšky. Je to složité? Příklad to osvětlí:

```
SELECT os.jmeno, os.prijmeni, uz.uzivatel
FROM osoba os INNER JOIN uzivatel uz
ON os.id = uz.id_osoba
```

Uvedený příklad se dá také zapsat:

```
SELECT os.jmeno, os.prijmeni, uz.uzivatel
FROM osoba os JOIN uzivatel uz
ON os.id = uz.id_osoba
```

V druhém, téměř totožném dotazu, bylo pouze odstraněno klíčové slovo INNER. Na výsledek to ale nemá žádný vliv, výsledná sada bude stejná pro oba příklady. Je to tím, že klíčové slovo INNER je nepovinné, nemusí se tudíž uvádět. Samotné klíčové slovo JOIN pro přehlednost plně postačuje.
Příklad uvádí dvě tabulky spojené na základě identifikátorů. Výsledná sada bude obsahovat všechny osoby, které mají vytvořený uživatelský účet (jsou zavedeni v tabulce ‘uzivatel‘). Zde je jistá analogie s použitím klauze WHERE. Je pravda, že jde o identický zápis, výsledek je stejný, kdyby byl použit takovýto dotaz:

```
SELECT os.jmeno, os.prijmeni, uz.uzivatel
FROM osoba os, uzivatel uz
WHERE os.id = uz.id_osoba
```

Zápis INNER JOIN je použitelný pouze v databázích SQL Server, na rozdíl od použití klauzule WHERE a operátoru ‘=‘, který funguje v jiných databázích.
Jedna důležitá informace se týká zápisu sloupců, které požaduje začlenit do výsledné sady.
Ve spojeních se nedoporučuje používat zástupný znak ‘*‘ (hvězdičky). Ze tří důvodů. Prvním důvodem je skutečnost, že dochází k neprůhlednému zápisu, vrácené sloupce budou všechny sloupce ze spojovaných tabulek (opravdu jsou třeba všechny sloupce?). Druhým faktem může být kolize názvů sloupečků ze spojovaných tabulek. A tím třetím je skutečnost nutnosti zahrnutí všech sloupců do výsledné sady. V případě, že tyto sloupce nejsou potřeba, dochází ke zbytečnému zatížení databázového stroje (musí sám od sebe dohledávat názvy sloupců) a také přenosových prostředků, které přenášejí zbytečně velké množství dat, které není využito. Při zápisu požadovaných sloupců tedy bude použito tečkové notace (Název_tabulky.název sloupce) nebo použití aliasů nad tabulkami, které spojujeme (alias_tabulky.název_sloupce).

Vnější spojení – OUTER JOIN

Toto spojení není používáno tak často jako spojení typu INNER JOIN, ale je důležité ho znát – může pomoci při řešení problémů, které by mohlo být zbytečně složité v případě neznalosti právě

tohoto spojení. V neposlední řade je důležitá informace, že vnější spojení bývá často rychlejší než vnořený poddotaz.

Zatímco vnitřní spojení INNER JOIN jsou výlučné operace (vyřazují záznamy, které nemají na obou stranách spojení ekvivalentní hodnoty), vnější spojení (ale i plná spojení FULL JOIN) jsou naopak zahrnující. Každé spojení JOIN má dvě strany – levou a pravou. U vnitřních spojení je tato informace zbytečná, ale u vnějších spojení je velice důležitá (řekl bych životně), kde je levá a kde pravá strana. Z pohledu na dotaz, kde je použito vnější spojení je určení levé a pravé strany jednoduché.

Uvedená syntaxe klauzule OUTER JOIN mnohé napoví:

```
SELECT <výběrový_seznam_select>
FROM <levá_tabulka >
<LEFT | RIGHT> [OUTER] JOIN <pravá_tabulka>
       ON <spojovací_podmínka>
```

I když uvedená syntaxe opravdu napoví, která tabulka je levá a která pravá, je třeba pro jistotu vysvětlit detaily. Za levou tabulku je považována tabulka uvedená před klíčovým slovem JOIN, zatímco pravá tabulka je zapsána za klíčovým slovem JOIN.

Vnější spojení OUTER JOIN jsou spojení, která se označují za zahrnující. Co to znamená? Znamená to, že do výsledku jsou zahrnuty všechny záznamy z té tabulky, jejíž strana je uvedena v klauzuli [LEFT | RIGHT]OUTER JOIN. Je-li v klauzuli uvedeno spojení LEFT OUTER JOIN, budou do výsledné sady zahrnuty všechny záznamy z levé tabulky spojení OUTER JOIN. A to i v případě, že ve spojované tabulce chybí ekvivalentní hodnoty. Analogicky toto konstatování platí i pro spojení typu RIGHT OUTER JOIN.

Příklad pro spojení LEFT OUTER JOIN uvede vše v jasnost:

```
SELECT os.jmeno, os.prijmeni, uz.uzivatel
FROM osoba os LEFT JOIN uzivatel uz
ON os.id = uz.id_osoba
```

Výslednou sadou jsou informace, které uživatelské jméno náleží jaké osobě (tabulka ‘osoba‘ je uvedena na levé straně do výsledné sady jsou zahrnuty všechny její záznamy). Toto tvrzení platí pro případ existence uživatelského účtu pro konkrétní osobu. Ale co když osoba není zavedena v tabulce uživatelů? Pro tento případ doplní do neexistujících hodnot databázový stroj hodnoty NULL.

Příklad pro spojení RIGHT OUTER JOIN použije stejný příklad, jen bude zaměněno klíčové slovo LEFT za RIGHT:

```
SELECT os.jmeno, os.prijmeni, uz.uzivatel
FROM osoba os RIGHT JOIN uzivatel uz
```

```
ON os.id = uz.id_osoba
```

Očekávaný výsledek se dostavil. Ve výsledné sadě jsou všechny záznamy z tabulky ‘uzivatel‘. A z důvodů integritních omezení existuje pro každého uživatele z tabulky ‘uzivatel‘ odpovídající záznam v tabulce ‘osoba‘ nejsou ve výsledné sadě záznamy bez odpovídajících ekvivalentních hodnot v protější tabulce.
Praktické využití se nabízí z uvedených příkladů. V případě že je třeba zjistit všechny osoby, které nemají vytvořený účet v tabulce ‘uzivatel‘, použijeme následují databázový dotaz:

```
SELECT os.jmeno, os.prijmeni
FROM osoba os LEFT JOIN uzivatel uz
ON os.id = uz.id_osoba
WHERE uz.id_osoba is NULL
```

Oboustranné vnější spojení – FULL JOIN

Operace plného vnějšího spojení, jak lze z předchozích ukázek a názvu tohoto spojení vyčíst, vypíše záznamy se shodnými údaji na protějších stranách a navíc do výsledků zahrne veškeré záznamy z obou stran. Plné vnější spojení by nemělo být téměř vůbec v praxi používáno pro vývojářské účely. Pro co je tedy dobré? Jeho hlavním významem je podat kompletní pohled na vztahy mezi daty, aniž by byla jedna ze stran preferovaná, či ne. Jednoduše se vybere vše z tabulek a na nic se nesmí zapomenout. ¨
Příklad na spojení typu FULL JOIN vypadá následovně:

```
SELECT uz.uzivatel, op_k.nazev
FROM uzivatel uz
	FULL JOIN opravneni op
		ON uz.id_uziv = op.id_uziv
	FULL JOIN opravneni_konf op_k
		ON op.id_opravneni = op_k.id_opravneni
```

Příklad počítá s tabulkami 'uzivatel', 'opravneni' a 'opravneni_konf'. Požadavkem je získat informaci, kdo má jaké oprávnění, popřípadě která oprávnění nejsou vůbec nikomu přiřazena. Tímto jednoduchým dotazem dostane tvůrce databázového dotazu okamžité informace.

Křížová spojení – CROSS JOIN

Typ spojení CROSS JOIN se od ostatních liší v tom, že mu chybí operátor ON a dále, že spojuje každý záznam z jedné strany s každým záznamem ze strany druhé. Jedná se o kartézský součin množin záznamů z tabulek. Syntaxe je shodná s ostatními typy spojení (vyjma již zmíněného operátoru ON).

```
SELECT uz.uzivatel, op_k.nazev
FROM uzivatel uz
```

```
CROSS JOIN opravneni op
CROSS JOIN opravneni_konf op_k
```

K čemu je takový typ spojení dobrý? Nejvhodnější případ pro použití je vytváření testovacích množin dat (jde vytvořit velký objem dat během chvíle).

I pro spojení typu CROSS JOIN existuje alternativní zápis:

```
SELECT uz.uzivatel, op_k.nazev
FROM uzivatel uz, opravneni op, opravneni_konf op_k
```

Operátor UNION

Poslední zastávkou v kategorii příkazu SELECT je operátor UNION. Je to operátor, pomocí něhož můžeme ze dvou nebo více dotazů vygenerovat jednu sloučenou sadu výsledků. UNION je prostředek pro připojení dat z jednoho dotazu na konec dat druhého dotazu – do výsledku přidává další řádky.

Pro práci s operátorem UNION platí následující:

- Všechny dotazy, sloučené pomocí UNION do jedné sady, musí ve výběrovém seznamu obsahovat stejný počet sloupců. Pokud má tak například první dotaz ve svém výběrovém seznamu pět sloupců, musí být pět sloupců uvedeno i ve druhém dotazu.
- Záhlaví (názvy sloupců) výsledné sady se přebírá vždy jen z prvního dotazu.
- Datový typ každého ze sloupců v prvním dotazu musí být implicitně kompatibilní. s datovým typem sloupců, které se v ostatních dotazech nacházejí na stejné pozici. Tyto datové typy nemusí být shodné, ale musí být implicitně kompatibilní a převoditelné.
- Na rozdíl od dotazů bez operátorů UNION není výchozí variantou dotazů s operátorem UNION typ ALL, ale DISTINCT.

Příklad pro operátor UNION vypadá následovně:

```
SELECT * FROM OSOBA
UNION
SELECT * FROM OSOBA
```

V tomto případě dojde k vybrání neduplicitních řádků, jiný výsledek by se ukázal v případě použití tohoto příkazu s klíčovým slovem ALL:

```
SELECT * FROM OSOBA
UNION  ALL
SELECT * FROM OSOBA
```

2.5.3.3. To ostatní - uložené procedury a uživatelsky definované funkce

Po krátkém a mírně povrchním úvodu do T-SQL přišla na řadu ta důležitější a zajímavější část kterou T-SQL nabízí. Úvodní název je téměř nic neříkající. Do kategorie „To ostatní…“ patří

programování v SQL Serveru. Především jsou to uložené procedury (Stored procedure), uživatelsky definované funkce a systémové funkce. To vše tvoří opravdové programování uvnitř databázové platformy SQL Server 2005.
Programování na SQL Serveru pomocí jazyka T-SQL vychází z potřeb databáze a právě proto je jeho síla v programování na straně databáze (a pouze na ni) ohromná.

2.5.3.3.1. Uložené procedury

Uložená procedura je seskupení jednoho nebo více příkazů T-SQL do jedné logické jednotky, která je uložená jako objekt v databázi SQL Serveru. Někdy jsou uložené procedury přirovnávány ke skriptu nebo programové dávce. Rozdíl je ovšem v tom, že uložená procedura je uložena na databázi (kdežto skript bývá uložen v souboru) a na rozdíl od skriptu má uložená procedura vstupní a výstupní parametry.
Když se uložená procedura vykoná vůbec poprvé (od okamžiku posledního nastartování instance SQL Serveru), na rozdíl od jednotlivých dotazů nebo pohledů, určí SQL Server optimální plán přístupu k dotazu a uloží ho do cache vyhrazené pro plány. SQL Server pak může tento plán použít znovu, až se bude uložená procedura později opět vykonávat. Opětovné využívání plánů umožňuje uložené procedury vykonávat rychle a se spolehlivým výkonem ve srovnání s jednoúčelovými, nekompilovanými ekvivalentními dotazy nebo skripty. O výhodách uložených procedur je tedy jasno.
Nevýhody uložených procedur bývají prezentovány jako přílišná složitost pro vývojáře logiky na aplikační vrstvě, kteří raději používají jednoúčelové SQL právě na aplikační vrstvě. Toto je nevýhoda uměle vyvolávaná. Právě používání jednoúčelových SQL příkazů na jiné vrstvě než na té databázové přináší nejen zvýšené nároky na přenosovou soustavu a větší nároky na systém, ale také potencionálně vyšší chybovost v programovém kódu.
Výhody uložených procedur tedy jsou:

- Uložené procedury pomáhají soustředit kód T-SQL do vrstvy dat. Možnost pohodlnější modifikace v provozním prostředí (na rozdíl od webových nebo aplikačních jednoúčelových SQL). Dojde-li totiž k potřebě opravit programovou chybu, nastávají problémy s jednoúčelovým SQL – aplikace obsahující tento jednoúčelový SQL musí být celá překompilována.
- Uložené procedury podněcují opětovné využívání kódu. Je-li dobře navržená již jednou uložená procedura, která je používána na více místech, dochází k zpřehlednění kódu (struktury celé aplikace). Nemusí se opakovaně vkládat stejný kód.

- Uložené procedury umožňují skrýt metodu získávání dat. Dojde-li ke změně zdrojové tabulky, odkud jsou získávána zdrojová data, mohou se pomocí uložených procedur skrýt tyto změny před aplikací. Při této změně není třeba měnit kód v aplikační vrstvě.
- Uložené procedury mají stabilizační vliv na to, za jak dlouho přijde odpověď na dotaz. Může nastat problém při používání jednoúčelových SQL dotazů. Občas jsou na vině faktory jako uzamykání tabulky nebo potíže s prostředky (málo paměti, zatížení CPU). Na druhé straně se mohou jednoúčelové dotazy vykonávat neefektivně (pomalu) také proto, že SQL Server pravidelně vybírá méně efektivní plán vykonávání. S uloženými procedurami lze dosáhnout spolehlivějšího ukládání plánů vykonávání do cache, a tedy i lepší opětovnou využitelnost.

Je třeba ale uvést velmi důležitý dovětek k tomuto výčtu pozitiv pro uložené procedury. Dojde-li totiž k vytvoření mizerné uložené procedury, pak dochází k popření těchto pozitivních bodů! Existuje však nesporná výhoda uložených procedur a tou je ukrytí objektů databáze (samozřejmě kromě uložených procedur, které jsou vystaveny aplikační vrstvě). Budou-li se používat pouze uložené procedury, není možné zjistit z „okolního světa“ databázové objekty.

Vytvoření uložené procedury

Jak se vytvoří taková uložená procedura? Procedura se vytváří jako většina objektů v databázi – příkazem CREATE (v prostředí Management Studia). Uvedený příklad demonstruje vytvoření uložené procedury:

```
CREATE PROCEDURE <název_procedury>
       -- seznam vstupních parametrů
       <@parametr1> <datový_typ_parametr1> = <implicitní_hodnota1>,
       <@parametr2> <datový_typ_parametr2> = <implicitní hodnota2>
AS
       -- deklarační část pro globální proměnné
BEGIN
       -- výkonný kód procedury
        [sady_příkazů | volán_volání_jiných_uložených_procedur | volání_funkcí

       [RETURN hodnota]
END
GO
```

Následující příklad ukazuje konkrétnější vytvoření uložené procedury:

```
CREATE PROCEDURE procGetAllPersons
AS
BEGIN
       SELECT * FROM osoba
END
```

Tato elementární procedura udělá pouze to, že vybere všechny záznamy z tabulky osoba. Procedura neobsahuje žádné vstupní, ani výstupní parametry. Jak provedu spuštění procedury? Nejjednodušší způsob je z vývojového prostředí Management Studia:

```
EXEC procGetAllPersons
```

Po vykonání procedury bude do prostředí Management Studia vrácena výsledná sada jako kdyby byl proveden pouze samotný SQL dotaz SELECT, který je volán uvnitř těla procedury procGetAllPersons.

Změna uložené procedury

Změnu již existující uložené procedury provede příkaz ALTER. Je-li k dispozici skript, který vytvořil uloženou proceduru, bude stačit pouze přepsání příkazu CREATE na příkaz ALTER a potřebnou změnu v kódu T-SQL jako v následujícím příkladu:

```
ALTER PROCEDURE procGetAllPersons
AS
BEGIN
	SELECT jmeno, prijmeni FROM osoba
END
```

Modifikovaná uložená procedura se v databázi pozmění a je k dispozici.

Odstraňování uložené procedury

Existuje-li v databázi uložená procedura a již dále není žádaná, může být zrušena, provede-li se příkaz DROP:

```
DROP PROCEDURE <název_procedury>
```

Jako následující příklad:

```
DROP PROCEDURE procGetAllPersons
```

Parametrizace procedur

Uložená procedura již sama o sobě nabízí možnost vytvoření podprogramu a tím představuje určité zefektivnění výsledného zpracování. Síla použití uložených procedur tkví také ve využití procedur pro rozdílná data. To znamená, že za jistých podmínek se provede jistá akce. Těmi jistými podmínkami mám na mysli vstupní parametry. Vstupní data mohou proceduře říci, která data má vybrat (nebo smazat). Kromě vstupních parametrů existují i parametry výstupní. Výstupní parametry předávají informace z vykonané procedury do míst, odkud byla procedura volána. Může to být informace, jestli procedura doběhla v pořádku, kolik záznamů se upravilo atd. Při deklaraci parametrů se uvádí příklad název parametru (musí začínat znakem @ - zavináč a nesmí obsahovat mezery), jeho datový typ, výchozí hodnota (nepovinný parametr) a směr parametru (pokud není uvedeno jinak, jedná se o vstupní parametr):

```
@ název_parametru [AS]  datový_typ [ = implicitní_hodnota] [VARYING] [OUTPUT | OUT]
```

Použití procedury se vstupními a výstupními parametry lze demonstrovat na vložení řádku do tabulky:

```
CREATE PROCEDURE procInsertPerson
      @jmeno varchar(50),
      @prijmeni varchar(50),
      @pohlavi varchar(10),
      @mail  varchar(100),
      @aktivni     varchar(10),
      @return_id   int    OUTPUT
AS
BEGIN
      INSERT INTO osoba(
            jmeno,
            prijmeni,
            pohlavi,
            mail)
      values(
            @jmeno,
            @prijmeni,
            @pohlavi,
            @mail)
      SELECT   @return_id = @@IDENTITY
END
```

Procedura vloží nový záznam do tabulky 'osoba ' a vrátí id záznamu (které je vedené jako datový typ identita). Pro vykonání procedury zadáme následující:

```
DECLARE @new_id     int

EXEC procInsertPerson
      @jmeno = 'Ivan',
      @prijmeni = 'Karbas',
      @pohlavi = 'M',
      @mail = 'ikarbas@seznam.cz',
      @aktivni = 'ANO',
      @return_id = @new_id OUTPUT

select @new_id
```

2.5.3.3.2. Řízení toku programu

Tak jako každý jiný v současnosti rozšířený programovací jazyk má i jazyk T-SQL své vlastní příkazy pro rozhodování řízení toku programu. Bez těchto příkazů by byl programovací jazyk degradován na úroveň jednolitých dávek (skriptů). V jazyce T-SQL jsou to tyto příkazy:

- IF…ELSE
- CASE
- GOTO
- WHILE
- WAITFOR

Příkaz IF…ELSE

Příkaz IF…ELSE funguje stejně, jako v každém jiném jazyce (až na pár drobných rozdílů). Příklad na rozhodování příkazem IF…ELSE vypadá takto:

```
IF <booleovský_výraz>
	<příkaz_SQL> | BEGIN <posloupnost_příkazů_SQL> END
[ELSE
	<příkaz_SQL> | BEGIN <posloupnost_příkazů_SQL> END
```

Platnost rozsahu po uvedení klíčových slov IF a ELSE. Z příkladu syntaxe příkazu IF…ELSE lze vypozorovat, jaké jsou rozsahy platnosti větví po uvedených klíčových slovech. Je-li použit jediný příkaz za uvedenými klíčovými slovy, není třeba explicitně uvádět blok BEGIN…END, který vymezuje rozsah platnosti příkazů. Bude-li ale použito více příkazů ve větvích podmíněného zpracování, je třeba tento blok uvést a do něj vložit seznam příkazů, které se mají vykonat. Dále lze ze syntaxe vypozorovat, že větev ELSE nemusí být uvedena – je nepovinná.
Uvedení praktického příkladu:

```
CREATE PROCEDURE [dbo].[procUpdateOsoba]
	@id int, @jmeno varchar(50),
	@prijmeni varchar(50),
	@titul varchar(20),
	@pohlavi varchar(10),
	@ulice varchar(50),
	@cp    varchar(20),
	@obec  varchar(50),
	@psc   int,
	@okres varchar(50),
	@kraj  varchar(50),
	@pusobiste   varchar(100),
	@telefon     varchar(50),
	@mail  varchar(100),
	@aktivni     varchar(10)
AS
	DECLARE @pocet      int, @error  varchar(500)
BEGIN
	-- zjisteni poctu zaznamu
	SELECT @pocet = COUNT(id) FROM dbo.osoba WHERE id = @id

	IF @pocet > 0
	-- byl nalezen zaznam, bude proveden update
		UPDATE osoba
		SET     jmeno = @jmeno,
			prijmeni = @prijmeni,
			titul = @titul,
			pohlavi = @pohlavi,
			ulice = @ulice,
			cp = @cp,
			obec = @obec,
			psc = @psc,
			okres = @okres,
			kraj = @kraj,
			pusobiste = @pusobiste,
			telefon = @telefon,
			mail = @mail,
			aktivni = @aktivni
```

```
		WHERE id = @id
	ELSE
	-- nenalezen zadny zaznam, bude vlozen novy
		INSERT INTO osoba
		VALUES(
			@jmeno,
			@prijmeni,
			@titul,
			@pohlavi,
			@ulice,
			@cp,
			@obec,
			@psc,
			@okres,
			@kraj,
			@pusobiste,
			@telefon,
			@mail,
			@aktivni)
END
```

V příkladu uložené procedury je na první pohled zřejmé, co se provádí. Na základě vstupního parametru identifikátoru osoby (id) se jako první provede kontrola, jestli náhodou v tabulce osob již tato osoba existuje (dotaz s funkcí count(id)). V případě existence záznamu (@pocet>0) se provede modifikace záznamu (příkaz UPDATE), v opačném případě se provede vložení nového záznamu (příkaz INSERT). Blok BEGIN…END nemusí (ale může) být uveden, protože v obou větvích příkazu IF…ELSE se provádí pouze jedna operace (jeden příkaz T-SQL).

Příkaz CASE

Příkazem CASE se vrací hodnota na základě vyhodnocení nějakého výrazu. Druhou možností je vrácení hodnoty založené na výsledku booleovského výrazu. Syntaxe pro první případ vypadá takto:

```
CASE <vstupní_výraz>
	WHEN <výraz_když_1> THEN <výsledný_výraz_1>
WHEN <výraz_když_2> THEN <výsledný_výraz_2>
WHEN <výraz_když_n> THEN <výsledný_výraz_n>
[ELSE výsledný_výraz]
END
```

Syntaxe pro druhý případ pak vypadá následovně:

```
CASE <vstupní_výraz>
	WHEN <booleovský_výraz_1> THEN <výsledný_výraz_1>
WHEN < booleovský_výraz_2> THEN <výsledný_výraz_2>
WHEN < booleovský_výraz_n> THEN <výsledný_výraz_n>
[ELSE výsledný_výraz]
END
```

Příkaz GOTO

S pomocí příkazu GOTO se lze v dávkách jazyka T-SQL pohybovat pomocí skoků. Jak název příkazu napovídá, tento příkaz umí „jít na požadované místo“. Co to znamená? Na základě označeného místa v kódu (návěští) může být tímto příkazem přeskočen kód, který následuje za příkazem skoku. Syntaxe příkazu GOTO:

```
GOTO <kód_návěští>
Návěští: kód
```

Používání příkazu GOTO mi přijde v rozsáhlém kódu nepřehledné a nemůžu se zbavit pocitu, že tento příkaz je jakousi berličkou pro velmi rozsáhlé a nepřehledné zdrojové kódy. Zastávám názor, že příkaz GOTO jde nahradit jinými technikami, které v kódu vypovídají transparentněji o tom,
co má uvedený kód vykonat.

Příkaz WHILE

Příkaz WHILE vykonává příkazy ve smyčce tak dlouho, dokud se stanovená podmínka vyhodnocuje na TRUE. Syntaxe příkazu WHILE vypadá takto:

```
WHILE <booleovský_výraz>
       <příkaz_sql | blok_příkazů_sql>
       [BREAK]
       < příkaz_sql | blok_příkazů_sql >
       [CONTINUE]
```

Klíčová slova BREAK a CONTINUE jsou v příkazu WHILE prvky násilné povahy. Co to znamená? Násilně přerušují běh cyklu WHILE. Klíčové slovo BREAK provede ukončení cyklu, aniž by ho zajímala hodnota booleovského výrazu. Klíčové slovo CONTINUE provádí pravý opak. Z vnitřku cyklu WHILE předá řízení na začátek cyklu do rozhodovacího procesu pro booleovský výraz. U příkazu WHILE je potřeba dávat pozor na skutečnost, aby se cyklus ukončil. Pokud by se do cyklu WHILE dostala nějaká chyba, Může mít v těch nejčernějších možných případech za následek i zhroucení SQL Serveru (například v případě nekonečné alokace paměťového prostoru).

Příkaz WAITFOR

Příkaz WAITFOR odkládá vykonání kódu T-SQL. Syntaxe příkazu WAITFOR:

```
WAITFOR
       DELAY <čas> | TIME <kdy_se_má_spustit>
```

Příkaz dělá přesně co jeho název říká. Čeká na zadaný čas a poté provede akce, které následují za tímto příkazem. Může se hodit například pro periodické služby, které mají být spuštěné v konkrétním čase každého dne.
Parametr DELAY udává množství času, po kterém se má čekat. Do tohoto parametru nelze zadat počet dní, ale pouze čas v hodinách, minutách nebo sekundách. Povolená prodleva je 24 hodin.
Parametr TIME požaduje čekání do určitého konkrétního okamžiku v průběhu dne. Nelze zadat žádné datum. Pouze čas v běžné 24hodinové časové míře. To znamená, že i zde je limit čekání roven jednomu dni.
Uvedený příklad prezentuje možnosti příkazu WHILE a WAITFOR dohromady:

```
WHILE 1=1
	BEGIN
		WAITFOR TIME ’01:00‘
		-- vykonání příkazů
	END
```

Příklad běží v nekonečné smyčce a každý den v uvedený čas provede připravené akce, které jsou uvedeny za příkazem WAITFOR.

2.5.3.3.3. Uživatelsky definované funkce

Co je to uživatelsky definovaná funkce? Do značné míry jsou to podobné objekty jako uložené procedury, posloupnost příkazů jazyka T-SQL, které jsou předem optimalizovány a zkompilovány, a které se dají vyvolat jako ucelená jednotka. Nejmarkantnější rozdíl mezi těmito objekty je ve způsobu navrácení výsledků. Nejsou to ekvivalentní objekty, takže se nedají nahrazovat mezi sebou. Každý z objektů má své opodstatnění a hodí se pro něco jiného.
Do uložené procedury lze předávat vstupní parametry a také získávat hodnoty, které jsou výstupní parametry. V proceduře lze také nadefinovat návratovou hodnotu, ale jejím hlavním smyslem je pouze informovat, jestli daná procedura doběhla úspěšně nebo při jejím vykonávání havarovala. Z procedury je možné vracet sadu výsledků, ale v dotazech je nelze využít (napřed se musí uložit do nějaké tabulky) a až poté s nimi lze pracovat.
U uživatelsky definované funkce je možné použít vstupní parametry, ale parametry výstupní ne. Výstupní parametry jsou nahrazeny mechanismem návratových hodnot. Uživatelsky definované funkce mohou vracet skalární hodnoty. Ale tyto hodnoty nejsou omezené pouze na celočíselný datový typ jako je tomu u uložených procedur. Skalární hodnota není to jediné, co uživatelsky definovaná funkce může navrátit
Uživatelsky definované funkce jsou trojího typu (podle typu návratové hodnoty):

- Skalární hodnota
- Přímá tabulková
- Vícepříkazová tabulková

Skalární, uživatelsky definované funkce

Tento typ uživatelsky definované funkce přebírá nula nebo více vstupních parametrů a vrací jedinou hodnotu. Takovéto funkce bývají využívány pro zjištění nějaké konkrétní jednoduché informace. Syntaxe vytvoření objektu vypadá následovně:

```
CREATE FUNCTION <název_funkce>
(
	<@vstupní_parametr> <datový_typ_parametru>
)
RETURNS <datový_typ_navratové_hodnoty>
AS
BEGIN
	DECLARE <@proměnná> <datový_typ_proměnné>

	SELECT <@proměnná > = <@vstupní_parametr>
-- návratová hodnota se předá příkazem RETURN
RETURN <@proměnná >

END
```

Syntaxe je velmi podobná syntaxi uložených procedur. Skutečně jediné rozdíly jsou v typu objektu (zde je FUNCTION) a návratová hodnota musí být uvedena (u uložených procedur tomu tak není). Příklad uživatelsky definované funkce:

```
CREATE FUNCTION [dbo].[whatAction](@id int, @par_kod varchar(50), @poradi int)
RETURNS varchar(50)
AS
BEGIN
	DECLARE @pocet int
	DECLARE @retval varchar(50)
	-- dotaz na existenci parametru
	SELECT @pocet=count(*)
		FROM osoba_info
			WHERE id = @id
			AND kod = @par_kod

	IF (@pocet < 1) AND (not isnull(@poradi,-1)=-1)
		SET @retval='INSERT'
	ELSE
		SET @retval='UPDATE'

RETURN @retval
END
```

Uvedená funkce vrací hodnotu typu varchar na základě dotazu na existenci záznamu v tabulce.

Přímé tabulkové uživatelsky definované funkce

Tabulková funkce vrací tabulku. Jako návratovou hodnotu není třeba explicitně definovat tabulku. Příkaz SELECT automaticky určí, jaké řádky a sloupce definovat. Přebírá nula nebo víc vstupních

parametrů a vrací data na základě jediného příkazu SELECT. Syntaxe vytvoření objektu vypadá takto.

```
CREATE FUNCTION <název_funkce>
(
	<@název_parametru> <datový_typ_parametru>
)
RETURNS TABLE
AS
RETURN
(
	SELECT  <seznam_sloupců> [FROM <název_tabulky>]
)
```

Příklad použití může vypadat takto

```
CREATE FUNCTION [dbo].[funGetPerson]
(
	@status	varchar(100)
)
RETURNS TABLE
AS
RETURN
(
	SELECT * FROM osoba WHERE aktivni = @status
)
```

Funkce provádí dotaz do tabulky osob (‘osoba‘) a vybírá podle sloupce ‘aktivni‘ na základě vstupního parametru do funkce. Využít se dá funkce po vytvoření následovně:

```
SELECT jmeno, prijmeni, telefon, mail
FROM funcGetPerson('ANO')
```

V klauzuli FROM není uvedena žádná z existujících tabulek v databázi, ale uživatelsky definovaná funkce. Výhoda této vlastnosti pro takovýto typ uživatelsky definovaných funkcí je zřetelný.
Pro složitější složené dotazy lze vytvořit funkce, a ty poté pro potřebu zjednodušení přehlednosti kódu využívat.

Vícepříkazové uživatelsky definované funkce

Vícepříkazové funkce, které vracejí tabulku, se mohou volat z klauzule FROM stejně jako přímé tabulkové funkce, ale na rozdíl od nich nejsou ve své definici omezeny jedním jediným příkazem SELECT. V těle vícepříkazové funkce lze uvést několik příkazů T-SQL, jimiž bude určena finální sada výsledků, kterou funkce vrátí. Základní syntaxe vytvoření tohoto typu funkcí vypadá takto:

```
CREATE FUNCTION <název_funkce>
(
	<@název_parametru > <datový_typ_parametru>,
)
RETURNS
<@název_návratové_tabulky> TABLE
(
```

```
      <název_sloupce> <datový_typ_sloupce>
)
AS
BEGIN
      -- sled příkazů, které mají být vykonány v těle funkce
      -- naplnění tabulky, která má být vrácena
      RETURN
END
```

U těchto funkcí je třeba explicitně definovat návratový typ (na rozdíl od předchozího typu uživatelsky definované funkce) jako tabulku i s jejími sloupci (uvedené za klíčovým slovem RETURNS). Návratovou hodnotou tedy bude tabulka, kterou poté můžeme využít stejně jako u přímé tabulkové uživatelsky definované funkce.
Pro modifikování a rušení objektů uživatelsky definovaných akcí platí stejná pravidla jako pro uložené procedury. Příkazem DDL ALTER lze objekt modifikovat a příkazem DROP lze objekt zrušit.

2.5.3.3.4. Zpracování chyb v příkazech T-SQL

Zpracování vzniklých chyb je velice silný pomocník pro odhalení chyb a jejich následné opravení. Vznikne-li totiž chyba při vykonávání rozsáhlého kódu, aplikace je připravena tuto chybu zachytit. Chyba je pak snadno identifikovatelná a tím pádem i rychle opravitelná. V situaci, kdy není zvládnuto zpracování chyb v příkazech T-SQL a dojde k chybě, pravděpodobně nebude existovat žádná přesná informace v jaké části aplikace tato chyba vznikla. Hledání původce chyby se může stát noční můrou a ztrátou velkého množství času.
Pozitivní novinkou v SQL Serveru 2005 je existence bloků TRY…CATCH tak jako ve většině používaných programovacích jazycích. Tento mechanismus dává vývojáři k dispozici možnost reakce na vzniklou chybu při běhu programu. Je pak na vývojářovi, jak naloží s informacemi o vzniklé chybě.
Samozřejmě lze namítnout, že je lepší chybám předcházet (proaktivní přístup) a provádět kontroly tak, aby k chybám nedocházelo. Ne vždy lze chybu odhalit kontrolami a hlavně systém zpracování chyb dává vývojáři možnost vyvolat uživatelsky definované chybové zprávy.

Systémové a uživatelsky definované chybové zprávy

Systémové chybové zprávy jsou zabudované a generují se jako reakce na standardní chyby SQL Serveru (porušení integritního omezení, práce s nekompatibilními datovými typy a tak dále). Systémové chyby jsou uloženy v tabulce sys.messages každého SQL Serveru

Uživatelsky definované chybové zprávy se používají ve vyrobených aplikacích jako reakce na určité události.

Vlastnosti chybové zprávy

- message_id – identifikátor chybové zprávy
- severity – závažnost chybové zprávy
- is_event_logged – zdali se chyba zapíše do protokolu událostí Windows
- text – text chybové zprávy

Úroveň závažnosti chyby má rozpětí od 1 do 25, a jednotlivé závažnosti spadají do kategorií:

- Úroveň závažnosti od 0 do 10 označují informativní zprávy
- Úroveň závažnosti od 11 do 16 jsou chyby databázového stroje, které může napravit uživatel (chybějící databázové objekty, odmítnutí přístupu kvůli nedostatečným povolením, či syntaktické chyby)
- Úroveň závažnosti od 17 do 19, které požadují zásah administrátora systému (v případě, že SQL Server spotřeboval veškerou dostupnou paměť nebo bylo dosaženo limitů nastavených pro databázový stroj)
- Úroveň závažnosti od 20 do 25 jsou fatální chyby a systémové problémy (poškození hardwaru nebo softwaru, která mají dopad na databázi)

Hodnota text obsahuje skutečný popis chyby, který je předáván koncovému uživateli

Vytvořením uživatelsky definované chybové zprávy lze doplnit do systémové tabulky chybových hlášení (sys.messages) vlastní definované chyby. K tomuto slouží systémová uložená procedura:

```
sp_addmessage
@msgnum,
@severity,
@msgtext
[, @lang]
[, @with_log]
[, @replace]
```

Popis jednotlivých vstupních parametrů uvádí tabulka 1.6

Argument	Popis
`@msgnum`	Uživatelem dodané identifikační číslo chyby, v rozsahu od 50 001 do 2 147 483 647.
`@severity`	Definuje úroveň závažnosti (1 až 25)
`@msgtext`	Text chybové zprávy.

`@lang`	Jazyk, ve kterém je zpráva napsaná
`@with_log`	Zdali se zpráva zapíše do protokolů aplikace Windows, když dojde k chybě.
`@replace`	Když bude uveden tento argument, přepíše se existující uživatelsky definovaná chyba novými parametry předanými do systémové uložené procedury

Tabulka 2.6 – Popis vstupních parametrů systémové procedury sp_addmessage

Příklad uživatelsky definované chyby:

```
exec sp_addmessage
      50001,
      11,
      'Nastala chyba - neocekavana hodnota'
```

Vyvolání chybové zprávy

Je-li třeba vyvolat chyba, SQL Server nabízí jednoduchý příkaz:

```
RAISERROR(
<id_zpraávy>|<řetězec_zprávy>,
<závažnost_zprávy>,
<stav>
)
```

Při vyzkoušení zavolání tohoto příkazu:

```
RAISERROR(50001, 11, 1)
```

Bude navrácen chybový text, který byl uložen do tabulky systémových chyb.

Blok TRY…CATCH

Jak bylo uvedeno, SQL Server 2005 obsah novinku ve zpracování chyb, kdy je možné vložit kód do bloku, který je hlídán a při vzniku v tomto hlídacím bloku dojde k zachycení v bloku odchytávaných chyb. Syntaxe TRY…CATCH vypadá následovně:

```
BEGIN TRY
      < příkaz_sql | blok_příkazů>
END TRY
BEGIN CATCH
      <příkaz_sql | blok_příkazů>
END CATCH
```

Blok CATCH dává vývojářovi do rukou možnosti vypořádat se s vzniklými chybami v aplikaci (nastavení implicitních hodnot, zalogovat informaci do log souboru, oznámení problému uživateli aplikace, atd.).

```
CREATE FUNCTION [dbo].[isOsoba](@id int)
RETURNS bit
AS
BEGIN
	BEGIN TRY
	DECLARE @pocet int, @retval bit
	-- dotaz na existenci parametru
	SELECT @pocet=count(*)
		FROM osoba
		WHERE id = @id

	IF @pocet > 0
		SET @retval=1
	ELSE
		RAISERROR('Chyba pri kontrole osoby - vlastnosti - neexistuje nadrizeny
zaznam', 11, 1)
	END TRY

	BEGIN CATCH
		SET @retval=0
	END CATCH

RETURN @retval

END
```

3. Visual C++ .NET

3.1. Úvod

V roce 2002 vydává společnost Microsoft novou verzi vývojového prostředí Visual Studio 2002, jehož obsahem bylo obrovské množství změn, které měly významný vliv na odvětví výpočetní techniky. Zejména na vývoj aplikací serverových, webových a zejména pak pro osobní počítače (desktopové aplikace). Uvedený seznam uvádí výčet zásadních změn, které doprovázely vydání Visual Studia 2002:

- Vznik platformy .NET Framework
- Vznik nového programovacího jazyka C# (finální verze ISO normy)
- Programovací jazyk Visual Basic prošel drastickou řadou změn pro potřeby platformy .NET Framework
- Programovací jazyk C++ byl přepracován pro potřeby platformy .NET Framework (řízené C++)
- Dokumentace pro .NET Framework (Microsoft Visual Studio Documentation)

Všechny tyto změny mají za následek ulehčení (zpřístupnění, zjednodušení) vývoje softwaru především pro tyto oblasti:

- Vývoj Windows formulářů
- Vývoj webových aplikací (ASP.NET) a vývoj webových služeb (Web services)
- Vývoj mobilních technologií na přenosná zařízení jako jsou PDA (.NET Compact Framework)
- Možnost vývoje aplikací, jejichž části mohou být psány v různých jazycích (v rámci .NET Framework)

V následujícím roce 2003 společnost Microsoft vydává nové Visual Studio, jež obsahuje jen drobné změny oproti předchozímu vydání. Změny se týkají především:

- Upgrade platformy .NET Framework (verze 1.1)
- Drobná vylepšení podpory mobilních zařízení a ASP.NET
- Rozšíření podpory Visual C++; vydání Visual C++ Toolkit 2003
- Různé edice Visual Studia (Academic, Professional, Enterprise Developer, and Enterprise Architect)

V zásadě opravdu nic nového. Významnější uvolnění do světa nového Visual Studia proběhlo v říjnu roku 2005:

- Upgrade platformy .NET Framework (verze 2.0)
- Zásadní změny pro jazyk C++/CLI, jenž nahradilo řízené C++ (podpora pro vývoj Windows formulářů)
- ASP.NET 2.0

3.1.1. Co je to ten Microsoft .NET Framework?

.NET Framework je platforma pro vytváření a provozování aplikací. Jejími zásadními komponenty jsou společný běhový modul (Common Language Runtime – CLR) a knihovna tříd .NET (.NET Framework Vlase Library – FCL). CLR abstrahuje služby operačního systému a slouží jako vykonávací jádro pro řízené aplikace – aplikace, jejichž všechny akce podléhají schválení u CLR. FCL poskytuje objektově orientované rozhraní API, do něhož řízené aplikace zapisují. Je-li tedy vytvářena aplikace rámce .NET, není použit žádný z nástrojů API Windows, MFC, ATL, COM ani další nástroje. Na místo nich je pracováno pouze s FCL. Samozřejmě existuje možnost volat tyto technologie z prostředí .NET, ale to by se dít nemělo, protože je k tomu zapotřebí přechodu z řízeného kódu CLR do neřízeného. A takový přechod omezuje výkonnost (nehledě na to, že je správce systémů může zakázat).
Klíčem k pochopení prostředí .NET Framework a různých jím podporovaných modelů programování je porozumět společnému jazykovému běhovému modulu a FCL.

3.1.1.1. Společný jazykový běhový modul

Každý zapsaný bajt kódu běží buď v CLR nebo mu CLR povolí běžet mimo modul – bez akce CLR se nic nestane
CLR se nachází nad operačním systémem a poskytuje virtuální prostředí pro hostování řízených aplikací. Když je otevřen řízený vykonatelný soubor, CLR nahraje modul obsahující daný spustitelný soubor a vykoná v něm obsažený kód. Kód zacílený na CLR se označuje za řízený (managed) kód a skládá se z instrukcí zapsaných v pseudo-strojovém kódu označovanému za společný zprostředkovaný jazyk neboli CIL (Common Intermediate Language). Instrukce CIL se na požádání (just-in-time -JIT) zkompilují do nativního strojového kódu za běhu. Ve většině případů se určitá metoda zkompiluje technikou JIT jen jednou – při svém prvním volání – a následně se uloží do paměti, aby ji příště bylo možné vykonat bez zpoždění. Kód, který se nikdy nezavolá, se ani nezkompiluje. Třebaže kompilace JIT nepochybně snižuje výkonnost, její negativní efekty jsou minimalizovány skutečností, že daná metoda se kompiluje jen jednou během

celého života aplikace, a rovněž tím, že tým CLR ve společnosti Microsoft věnoval extrémní úsilí snaze o vytvoření tak rychlého a výkonného kompilátoru JIT, jak je to jen možné. Teoreticky dokáže být kód zkompilovaný JIT výkonnější než obvyklý kód, protože kompilátor JIT dokáže optimalizovat nativní kód vytvářený pro určitou verzi hostitelského procesoru, na kterém právě pracuje. Kód zkompilovaný pomocí JIT není totéž jako interpretovaný kód.

Výhod provozování kódu v řízeném prostředí CLR je velmi mnoho. Když kompilátor JIT převádí instrukce CIL na nativní kód, používá proces ověřování kódu zajišťující typovou bezpečnost daného kódu. Je téměř nemožné vykonat nějakou instrukci, která přistupuje k paměti, k níž nemá oprávnění. V řízené aplikaci nebudete mít nikdy problémy se ztracenými ukazateli, protože CLR před použitím takového ukazatele vyvolá výjimku. Typ neleze převést na něco, čím není, protože taková operace není typově bezpečná. A nelze ani zavolat metodu se špatně zformátovaným rámcem zásobníku, protože v CLR k tomu prostě nemůže dojít. Kromě eliminování těch nejobvyklejších chyb, které nepříznivě ovlivňují aplikační programy, zabezpečení ověřováním kódu výrazně znesnadňuje vytváření zlomyslného kódu, který úmyslně škodí hostitelským systémům.

Zabezpečení ověřováním kódu je rovněž zásadní technologií, která dává CLR možnost hostit více aplikací v jediném procesu – jde o rozdělení procesu na virtualizované oddíly označované za aplikační domény. Systém Windows vzájemně izoluje aplikace tím, že je hostí v oddělených procesech. Negativním vedlejším důsledkem tohoto modelu samostatného procesu pro každou aplikaci je vyšší spotřeba paměti. Výkonné využívání paměti není tak důležité na samostatných systémech sloužících jednomu uživateli, je však naprosto klíčové na serverech nastavených na zpracování tisíců uživatelů najednou.

Další výhoda provozování v řízeném prostředí vyplývá z faktu, že prostředky alokované řízeným kódem se samočinně uvolňují z paměti. V praxi to znamená, že i když nejsou volány destruktory na vzniklé objekty, které alokovaly paměť, paměť je uvolňována inteligentním nástrojem CLR, který sleduje odkazy na objekty vytvářené kódem a tyto objekty ničí, když je jimi obsazená paměť zapotřebí jinde. Tento nástroj na uvolňování paměti se nazývá garbage collector (sběrač odpatků). Díky tomuto nástroji nezpůsobují aplikace, jež se skládají výhradně z řízeného kódu, paměťové úniky. Uvolňování paměti dokonce zvyšuje výkonnost, protože algoritmus alokování paměti používaný v CLR je rychlý – mnohem rychlejší než odpovídající rutiny alokování paměti v runtimeovém modulu jazyka C. Nevýhodou je, že když dochází k uvolňování paměti, vše ostatní

se na chvíli zastaví. Naštěstí však dochází k uvolňování paměti relativně zřídkakdy, čímž se výrazně snižuje jeho vliv na výkonnost.

3.1.1.2. Programovací jazyky

Další výhodou provozování aplikací v hostitelském prostředí CLR je to, že veškerý kód se omezuje na CIL, takže je téměř jedno, jaký je zvolen programovací jazyk k psaní kódu. Označení „společné“ v názvu společného jazykového běhového systému naznačuje, že CLR nečiní mezi jazyky žádný rozdíl.
Jazyk je syntaktický prostředek pro vytváření CIL a jen s několika málo výjimkami lze všeho dosažitelného v jednom jazyku docílit i ve všech jiných jazycích. Navíc, bez ohledu na jazyk použitý k jejich vytvoření, používají všechny řízené aplikace stejné API,: a to z knihovny třídy rámce .NET.

3.1.1.3. Řízené moduly

Když je sestavován nějaký program pomocí kompilátoru, který je schopen generovat CIL, daný kompilátor vytváří řízený modul. Řízený modul je prostě vykonatelný soubor vytvořený pro běh pod CLR. Obvykle, není to však pravidlo, má příponu souboru EXE, DLL nebo NETMODULE. Uvnitř řízeného modulu se nacházejí čtyři důležité elementy:

- Hlavička přenositelného vykonatelného (Portable Executable – PE) souboru Windows.
- Hlavička CLR obsahující důležité informace o daném modulu, jako je umístění jeho CIL a metadat.
- Metadata popisující vše uvnitř modulu a jeho externí závislosti.
- Instrukce CIL vygenerované ze zdrojového kódu.

Každý řízený modul obsahuje metadata popisující jeho obsah. Metadata nejsou volitelná, každý kompilátor odpovídající specifikaci CLR je musí vytvářet. To je důležité, protože to znamená, že každý řízený modul popisuje sám sebe. Když se vezme neobvyklý soubor EXE nebo DLL, lze jej snadno „vykuchat“ a zjistit, které třídy se v něm nacházejí a které členy tyto třídy obsahují? Ani náhodou! Když se však jedná o řízený modul, není to žádný problém. Metadata jsou jako typová knihovna COM, ale se dvěma odlišnostmi:

- Knihovny typů jsou volitelné, zatímco metadata nejsou.
- Metadata úplně popisují daný modul; knihovny typů to někdy nečiní.

Metadata jsou důležitá, protože modul CLR musí být schopen určit, jaké typy se nacházejí v jednotlivých nahrávaných řízených modulech. Rovněž jsou však důležitá pro kompilátory a další nástroje, které pracují s řízenými vykonatelnými soubory. Díky metadatům dokáže Visual Studio zobrazit kontextový seznam nabízených metod a vlastností, když se napíše název instance nějaké třídy do editoru programu, což je prvek IntelliSense. A díky metadatům se může kompilátor C++/CLI podívat dovnitř knihoven DLL obsahující určitou třídu napsanou v jazyce C# a použít ji jako bázovou třídu pro nějakou odvozenou třídu napsanou v jazyce C++/CLI.

3.1.1.4. Společný typový systém

Všechny jazyky, jejichž výstupem je kód pro CLR, musí používat typy vyhovující typovému systému CLR, aby byly kompatibilní s jinými jazyky. Tento sdílený typový systém dovoluje snadnou výměnu dat mezi jednotlivými jazyky; pokud je třeba předat typ Double nebo String mezi řízeným kódem C++/CLI a Visual Basicem .NET, nebudou s tím žádné problémy, režie spojená s marshalingem mezi typy různých jazyků zde odpadá.

3.2. Přehled základních vlastností Visual C++ .NET

3.2.1. Řízené prostředí

V jazyku C++ byly dvě místa, ve kterých bylo možné alokovat objekty: run-time zásobník a halda. Lokální proměnné byly alokovány v run-time zásobníku, zatímco instance typů, vytvořené operátorem C++ new, byly alokovány na haldě. Programátor zodpovídal za odstranění dynamicky alokovaných instancí, jakmile již nejsou zapotřebí. U programování COM vrůstá složitost celého řešení, protože se musí správně počítat odkazy na objekty. Správa životnosti objektů tímto způsobem může programátorům působit obtíže a zapříčinit vznik ztrát paměti a porušení ochrany. V .NET se vše zásadně mění. Běhový modul CLR obsahuje garbage collection, který automaticky sleduje počet odkazů na objekt v rámci celé aplikace. Pokud se aplikace na nějaký objekt odkazuje, je objekt udržován v paměti. Jakmile se aplikace na objekt přestane odkazovat, bude objekt odstraněn, i když ne okamžitě. CLR kvůli tomuto mechanismu obsahuje další místo, ve kterém je možné alokovat objekty: řízenou haldou (managed heap). Řízená halda je obdobou klasické C++ haldy, ovládá ji však garbage collector. Na řízené haldě je možné vytvářet pouze instance řízených typů. V C++/CLI se vytváří instance typu pomocí klíčového slova gcnew.

Alokace na řízené haldě jsou rychlejší než na C++ haldě a téměř stejně rychlé jako alokace na run-time zásobníku. Příčinou této rychlosti je struktura a uspořádání řízené haldy, která je prostým souvislým blokem paměti se základním ukazatelem a ukazatelem na následující volnou adresu paměti, která bude použita při alokaci dalšího objektu. Při každé alokaci objektu je odpovídajícím způsobem upravena hodnota tohoto druhého ukazatele.

3.2.1.1. Garbage collector

Pokud množství volné paměti na řízené haldě již nepostačuje pro uspokojení požadavku na alokaci dalšího objektu, spustí se garbage collector. Jakýkoli odkaz na aplikace na instanci řízeného typu je označován jako kořen (root). CLR sleduje všechny kořeny pro určitý typ objektu. Během shromažďování sestaví run-time z kořenů aplikací graf všech dosažitelných objektů alokovaných na haldě. Jestliže najde objekt, který je nedosažitelný, je kandidátem pro odstranění z paměti. Je-li během fáze shromažďování nalezen dostatečný počet nedosažitelných objektů, bude proveden přesun všech dosažitelných objektů na jiné místo v řízené haldě. Jakmile je zhutnění paměti hotovo, jsou všechny kořeny aplikací aktualizovány, aby odráželi aktuální polohu objektů.
To znamená, že na data v řízené haldě se nelze odkazovat prostřednictvím klasických ukazatelů C++; adresy paměti se mění. Na místo nich je nutné použít řízené ukazatele, jejichž adresy jsou aktualizovány automaticky.

3.3. Formuláře Windows

3.3.1. Úvod do formulářů Windows

Microsoft .NET Framework je především platformou pro vytváření webových aplikací a webových služeb, podporuje však i další programovací modely. Formuláře Windows (Windows Forms) – je programový model pro vytváření GUI aplikací pro rámec .NET.
Zvnějšku vypadají aplikace formulářů Windows jako normální aplikace pro systém Windows. Obsahují okna a často používají obvyklé elementy GUI, jako jsou nabídky, ovládací prvky a dialogová okna. Uvnitř jsou však řízenými aplikacemi ve všech významech tohoto označení. Obsahují kód společného zprostředkujícího jazyka (CIL) a metadata, používají knihovnu typů rámce .NET (FCL) a běží ve vysoce řízeném prostředí společného běhového modulu (CLR).

Hlavní výhodou vytváření aplikací Windows pomocí formulářů Windows je to, že rámec homogenizuje programovací model GUI a odstraňuje mnoho chyb, chytáků a nejednotností, které zamořují API Windows.

3.3.2. Programovací model formulářů Windows

V označení formuláře Windows je termín „formulář“ synonymem okna. Hlavní okno aplikace je formulář. Má-li daná aplikace další okna nejvyšší úrovně, jsou to také formuláře. Dialogová okna jsou rovněž formuláře. Aplikace formulářů Windows se spoléhají na třídy nacházející se ve jmenném prostoru System::Windows::Forms FCL. Tento jmenný prostor zahrnuje třídy jako Form, která modeluje chování oken, Menu, jež představuje nabídky a Clipboard, která nabízí řízené rozhraní ke schránce operačního systému. Tento jmenný prostor rovněž obsahuje mnoho tříd představujících ovládací prvky, které mají názvy jako Buton, TextBox a další.Téměř každá aplikace formulářů Windows je třída odvozená ze System::Windows::Forms::Form. Instance této odvozené třídy představuje hlavní okno aplikace. Dědí z Form mnoho vlastností a metod, které vytvářejí bohaté programové rozhraní formulářů.
Dalším důležitým stavebním blokem aplikace formulářů Windows je třída jmenného prostoru System::Windows::Forms Application. Tato třída obsahuje statickou metodu Run, která pohání aplikaci formulářů tím, že jí poskytuje pumpu zpráv. Tuto pumpu zpráv samozřejmě není vidět, protože samotnou existenci zpráv .NET abstrahuje.

3.3.3. Formuláře Windows pro C++/CLI

V původním řízeném C++ (uvedené ve Visual Studiu 2002 a 2003) neexistovala podpora pro vytváření formulářů Windows. Na rozdíl od jazyků Visual Basic nebo C# nebyla možnost složit grafické rozhraní aplikací z předem připravených kusů (Toolbox). Naštěstí pro vývojáře, kteří se rozhodli programovat v jazyce C++/CLI již ve verzi Visual Studia 2005 byl tento nedostatek odstraněn a jazyk C++ se dočkal podpory Toolboxu.
Systém vytváření formulářů je tedy něčím úplně novým – nenavazuje na žádnou dosud existující technologii.
Nástroj pro vytváření GUI aplikací je téměř všemocný. Při vygenerování nového projektu se vždy nabídne tato kostra pro hlavní formulář:

```
#pragma once
namespace FormWindowsExample {
```

```
	using namespace System;
	using namespace System::ComponentModel;
	using namespace System::Collections;
	using namespace System::Windows::Forms;
	using namespace System::Data;
	using namespace System::Drawing;

	public ref class FormWindows : public System::Windows::Forms::Form
	{
	public:
		FormWindows(void)
		{
			InitializeComponent();
		}

	protected:
		~FormWindows()
		{
			if (components)
			{
				delete components;
			}
		}

	private:
		System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
		void InitializeComponent(void)
		{
			this->components = gcnew System::ComponentModel::Container();
			this->Size = System::Drawing::Size(300,300);
			this->Text = L"Form1";
			this->Padding = System::Windows::Forms::Padding(0);
			this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
		}
#pragma endregion
	};
}
```

Tato kostra je základem všech formulářů které dědí z System::Windows::Forms::Form. Veškerá inicializace se děje v metodě InitializeComponent.

Spuštění hlavního okna má na starost metoda Run:

```
#include "stdafx.h"
#include "FormWindowsExample.h"

using namespace FormWindowsExample;

[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
	Application::EnableVisualStyles();
	Application::SetCompatibleTextRenderingDefault(false);
	Application::Run(gcnew FormWindows());
	return 0;
}
```

3.4. ADO.NET

ADO.NET je snadno použitelné databázové rozhraní pro řízené aplikace. Platforma ADO.NET je vystavena jako sada tříd ve jmenném prostoru *System::Data* knihovny .NET Framework a v jejích následnících. Na rozdíl od ADO a OLE DB, což byly předchůdci ADO.NET byla platforma ADO.NET již od počátku vytvářena pro práci v propojeném světě webu. Dokonale se rovněž integruje s XML a překonává tak mezeru mezi relačními daty a daty XML, čímž je zjednodušen přístup mezi těmito principy.

3.4.1. Podpora pro připojení databáze

ADO.NET podporuje primárně dva poskytovatele (providery) dat:

- Poskytovatele .NET pro SQL Server, který se napojuje na databáze programu Microsoft SQL Server bez pomoci neřízených poskytovatelů.
- Poskytovatele .NET pro OLE DB, který se napojuje na databáze prostřednictvím neřízených poskytovatelů OLE DB.

Protože se tato práce zabývá databází SQL Server 2005 nebude poskytovateli .NET OLE DB věnována pozornost.
Všechny interakce s databázemi prostřednictvím ADO.NET zahrnují, ať už implicitně nebo explicitně, objekty připojení a příkazů. Objekty připojení představují fyzická připojení k databázi. Objekty příkazů reprezentují příkazy vykonávané na databázi.
Každé připojení z řízeného prostředí .NET Framework za asistence ADO.NET má stejný vzor:
Vytvoření objektu připojení (zapouzdřujícího nějaký řetězec)
Otevření tohoto připojení voláním metody *Open* na objektu připojení
Vytvoření objektu příkazu zapouzdřujícího jak nějaký příkaz SQL, tak i připojení, které tento příkaz použije
Zavolání určité metody pro vytvořené spojení, která zajistí vykonání daného příkazu
Uzavření připojení zavoláním metody *Close* na objektu připojení.

3.4.2. Připojení k databázi

3.4.2.1. Vytvoření databázového spojení

Než lze na databázi vykonat nějakou akci z prostředí aplikací řízeného kódu, je nutné k databázi otevřít cestu – připojení. Třída System::Data::SqlClient::SqlConnection představuje připojení k databázím SQL Serveru. Uvnitř objektu SqlConnection je řetězec připojení. Následující příklad uvádí vytvoření objektu SqlConnection. Objekt je zároveň inicializován řetězcem připojení, který se připojuje do databáze, která se jmenuje NESEHNUTI:

```
String^ connectString = L"Data Source=PLATITO;Initial Catalog=NESEHNUTI;Integrated
Security=True";
SqlConnection^ sqlConnection = gcnew SqlConnection(connectString);
```

Při vytvoření objektu dojde pouze ke kontrole, je-li přihlašovací řetězec (connectString) správně naformátován. Ale nezajišťuje kontrolu, jestli je uvedená databáze funkční – kontrola správnosti hodnot nastavených parametrů probíhá až v době, kdy se provádí pokus o otevření spojení. Parametr *Data Source* identifikuje instanci SQL Serveru, která obsahuje příslušnou databázi a počítač, na kterém se nachází. Toto však není jediný správný způsob zápisu. V uvedeném příkladu lze zaměnit parametr *Data Source* za parametr *Server* a parametr *Initial Catalog* za parametr *database*. Poslední uvedený parametr identifikuje databázi. Existují ale i jiné parametry, které se uvádějí při nastavení připojení. Mezi další často používané řetězce připojení patří *Min Pool Size* a *Max Pool Size*, které zadávají limity velikosti fondu připojení. Pokud nejsou tyto parametry uvedeny, jsou nastaveny implicitní hodnoty na 0 resp. 100. Parametr *Integrated Security* povoluje (resp. zakazuje) integrované zabezpečení. Je-li nastaveno na *True*, pak dochází k ověřování tokeny systému Windows. Parametr *Connect Timeout* nastavuje nejvyšší dobu v sekundách, kterou se má čekat při otevírání spojení.

3.4.2.2. Otevírání a zavírání spojení

Pouhá inicializace objektu připojení a zadání řetězce připojení nezajistí fyzické otevření připojení k databázi. Pro otevření spojení slouží zavolání metody *Open* objektu připojení. Připojení, které bylo otevřeno pomocí metody *Open* musí být také zavřeno – to dokáže metoda *Close*. Následující příklad zobrazuje jak otevřít a zavřít připojení k databázi:

```
String^ connectString = L"Data Source=PLATITO;Initial Catalog=NESEHNUTI;Integrated
Security=True";
```

```
SqlConnection^ sqlConnection = gcnew SqlConnection(connectString);
sqlConnection->Open();
// vykonání databázových příkazů nad otevřeném databázovém připojení
sqlConnection->Close();
```

SqlConnection->Open() vyvolává výjimku *SqlException*, pokud nedokáže ustanovit připojení k databázi. Operace vykonnané na databázi otevřeným připojením rovněž vyvolávají výjimky *SqlException*, jestliže selžou. Z důvodu možnosti selhání těchto operací a možnosti vyvolání výjimek by měl být zdrojový kód dostatečně robustní:

```
 String^ connectString = L"Data Source=PLATITO;Initial Catalog=NESEHNUTI;Integrated
Security=True";
SqlConnection^ sqlConnection = gcnew SqlConnection(connectString);
try{
sqlConnection->Open();
// vykonání databázových příkazů nad otevřeném databázovém připojení
}
catch(SqlException^ e){
	// zpracování výjimky
}
finally{
	sqlConnection->Close();
}
```

Volání metody *Close* pro připojení, které není otevřené, ničemu neškodí. Strukturování kódu přístupu k databázi uvedeným způsobem zajistí, že dojde k uzavření připojení i v případě časově nevhodných chyb. Jestliže dojde k neuzavírání otevřených připojení, znamená to zásadní negativní dopad na výkonnost i celé fungování vytvořené aplikace. Proto je dobré vždy zavírat připojení v bloku *finally*.

3.4.3. Třídy příkazů

Samotné otevření připojení není ničím přínosným, pokud není vykonán nějaký příkaz do databáze. Pro účely vykonávání dotazů do databáze SQL Server poskytuje ADO.NET třídu příkazů SqlCommand. V tomto objektu dochází k zapouzdření příkazů T-SQL vykonávaných na databázi. Z části kde jsem se věnoval T-SQL je zřejmé, že je možné do databáze volat z řízené aplikace přes platformu ADO.NET tyto příkazy:

- DDL příkazy – CREATE, ALTER, DROP
- DML příkazy – INSERT, UPDATE, DELETE, SELECT
- Programové jednotky – uložené procedury, uživatelsky definované funkce

Následující příklad uvádí možnost jednoduchého volání T-SQL příkazu do databáze:

```
String^ connectString = L"Data Source=PLATITO;Initial Catalog=NESEHNUTI;Integrated
Security=True";
SqlConnection^ sqlConnection = gcnew SqlConnection(connectString);
```

```
try{
sqlConnection->Open();
SqlCommand^ sqlProc = gcnew SqlCommand(
L"delete from osoba where prijmeni='Ptáček' ", sqlConnection);
	sqlProc->ExecuteNonQuery(); // vykonání příkazu
}
catch(SqlException^ e){
	// zpracování výjimky
}
finally{
	sqlConnection->Close();

}
```

3.4.3.1. Metody vykonání příkazů T-SQL

Metoda ExecuteNonQuery

Tato metoda je nástrojem na vykonávání příkazů INSERT, UPDATE, DELETE a dalších příkazů, které nevracejí hodnoty - sem patří i příkazy CREATE, ALTER a DROP. Při použití s INSERT, UPDATE nebo DELET vrací metoda počet řádků ovlivněných daným příkazem. U všech ostatních vrací hodnotu -1. Předchozí příklad již uvedl praktické využití metody ExecuteNonQuery.
Tato metoda vykonání příkazu také vykonává uložené procedury. Ale o tom o kousek dál.
Jestliže metoda ExecuteNonQuery selže, vyvolá výjimku doprovázenou objektem SqlException. Příklady příkazů, které mohou vyvolat výjimku jsou příkazy, ve kterých jsou použity neplatné názvy sloupců, INSERT příkazy porušující integritní omezení atd. Ale příkazy, UPDATE a DELETE zaměřené na neexistující záznamy nezpůsobují chyby – metoda prostě vrátí hodnotu 0.

Metoda ExecuteScalar

Metoda ExecuteScalar vykonává nějaký příkaz T-SQL a vrací první řádek prvního sloupce ve výsledné sadě. Jedním z nejčastějších použití je vykonávání funkcí SQL, jako jsou COUNT, AVG, MIN, MAX a SUM, které vracejí výsledkové množiny s jedním řádkem a jedním sloupcem. Příklad na metodu ExecuteScalar – SQL dotaz vrátí počet záznamů v tabulce osoba:

```
String^ connectString = L"Data Source=PLATITO;Initial Catalog=NESEHNUTI;Integrated
Security=True";
SqlConnection^ sqlConnection = gcnew SqlConnection(connectString);
try{
sqlConnection->Open();
SqlCommand^ sqlProc = gcnew SqlCommand(L"select count(*)  from osoba ", sqlConnection);
int pocet = (int)sqlProc->ExecuteScalar();
Console::WriteLine(L"Počet zaznamu v tabulce osoba je: " + pocet);
}
catch(SqlException^  e){
	// zpracování výjimky
}
```

```
finally{
	sqlConnection->Close();

}
```

Při předávání výsledku z vykonání příkazu je třeba přetypovat požadovaný typ. Metoda ExecuteScalar totiž vrací typ Object. V případě nesprávného přetypování by byla vyvolána výjimka InvalidCastException.

Metoda ExecuteReader

Metoda ExecuteReader existuje z praktického důvodu vykonávat databázové dotazy a co nejefektivněji (nejrychleji a nejvýkonněji) získávat výsledky. ExecuteReader vrací objekt typu SqlDataReader. SqlDataReader má metody a vlastnosti, které lze volat při iterování výslednou sadou. Je to rychlý mechanismus pouze dopředný a pouze pro čtení pro procházení výsledky databázových dotazů. Je extrémně výkonný pro přebírání výsledných množin ze vzdálených počítačů, protože přejímá pouze ta data, která jsou požadována. Dotaz může vytvořit milion záznamů, ale pokud z nich pomocí objektu SqlDataReader bude přečteno jen deset, bude opravdu navrácen pouze zlomek celkové výsledné sady.
Výstup z objektu SqlDataReader lze využít všelijak. Pokud bude vykonán dotaz na získání informací z databáze, může se stát, že stejné informace budou potřeba o kousek dál a bylo by zbytečné znova se dotazovat do databáze. Lze tedy vzít výslednou sadu a předat data z ní do nějaké struktury, která nám umožní ony data využít znova. V uvedeném příkladu budou uložena data z objektu SqlDataReader do kolekce ArrayList:

```
permissions = gcnew ArrayList();
SqlConnection^ sqlConnection = gcnew SqlConnection(
L"Data Source=PLATITO;Initial Catalog=NESEHNUTI;Integrated Security=True");
SqlCommand^ sqlProc = gcnew SqlCommand(L"getPermissions", sqlConnection);
sqlProc->CommandType = CommandType::StoredProcedure;

// vstupní parametry uložené procedury
sqlProc->Parameters->AddWithValue("@uzivatel", user);
sqlProc->Parameters->AddWithValue("@heslo", password);

try
	{
	// otevreni spojeni a vykonani dotazu
	sqlConnection->Open();
	SqlDataReader^ reader = sqlProc->ExecuteReader();

	while (reader->Read())
		{
			this->permissions->Add(reader["nazev"]);
		}
	}
catch(Exception^ e)
	{
```

```
		// ošetření výjimkytvorného stavu
	}
finally
	{
		sqlConnection->Close();
	}
```

V uvedeném příkladě není použit databázový dotaz, jako v těch předchozích. Na místo SQL příkazu je použit objekt z SQL Serveru – uložená procedura se dvěma vstupními parametry. Tato uložená procedura na straně databáze provádí dotaz do tabulky na základě jejích vstupních parametrů. Ale o tom později (v kapitole Uložené databázové procedury). Důležité pro tento příklad je, že se vrátí výsledná sada stejně, jako by tomu bylo při použití dotazu SELECT. Každé volání metody Read vrací jeden řádek z výsledné sady. Pro práci s celým řádkem je používán indexer vlastnosti k extrahování hodnot pole „nazev“ záznamu. Indexem se lze odkazovat buď názvem (jako je uvedeno v příkladu) nebo číselným indexem (s počátkem v nule).
Objekt SqlDataReader lze použít i při zjišťování struktury tabulky. Uvedený příklad vypíše názvy sloupců, definovaných v tabulce (do konzolového prostředí):

```
SqlConnection^ sqlConnection = gcnew SqlConnection(
L"Data Source=PLATITO;Initial Catalog=NESEHNUTI;Integrated Security=True");
SqlCommand^ sqlProc = gcnew SqlCommand(
L"SELECT * FROM osoba", sqlConnection);

try
	{
	sqlConnection->Open();
	SqlDataReader^ reader = sqlProc->ExecuteReader();

	for(int i=0;i<reader->FieldCount;i++)
			{
				Console::WriteLine(reader->GetName(i));
			}
	}
catch(Exception^ e)
	{
		// ošetření výjimkytvorného stavu
	}
finally
	{
		sqlConnection->Close();

	}
```

3.4.4. Transakční příkazy

Transakční databázové operace jsou důležitou součástí mnoha datových aplikací. Transakce přestavuje logickou jednotku, která v sobě může zahrnovat i více nezávislých jednotek činností. Pokud je třeba použít transakci, je tím řečeno, že je třeba provést operace, které musí být všechny úspěšné – podle hesla „buď všechno nebo nic“. Téměř v každé publikaci se hovoří o typickém

příkladu jako o převodu peněz z jednoho účtu na druhý – proběhne transakce odebrání finanční částky z jednoho účtu a zároveň se musí provést připsání té samé finanční částky na druhý účet. Pouze v případě, že se obě akce podaří, je akce potvrzena jako úspěšná a dojde potvrzení transakce. V případě, že jedna z těchto operací skončí chybou, celá transakce se zruší a žádná ze změn se do databáze nepromítne. Já bych to ovšem zjednodušil, na případ že nemusí jít o více databázových akcí. V případě, že dojde k jakékoliv chybě, transakci prostě celou odvolám:

```
SqlTransaction^ trans;
SqlConnection^ sqlConnection = gcnew SqlConnection(
L"Data Source=PLATITO;Initial Catalog=NESEHNUTI;Integrated Security=True");
SqlCommand^ sqlProc = gcnew SqlCommand(
L"UPDATE osoba SET aktivni = ‘NE‘ ", sqlConnection);

try
	{
	sqlConnection->Open();
trans = sqlConnection->BeginTransaction(IsolationLevel::Serializable);
	sqlProc->ExecuteNonQuery();
	// potvrzení transakce
trans->Commit();
	}

catch(Exception^ e)
	{
		// zrušení transakce
		trans->Rollback();
	}
finally
	{
		sqlConnection->Close();

	}
```

Kód volá metodu BeginTransaction na otevřeném objektu SqlConnection, čímž spouští lokální transakci. IsolationLevel::Seriazible přiřazuje této transakci nejvyšší úroveň izolace, která zamyká záznamy účastnící se transakce během jejich aktualizace, aby je nebylo možné číst ani zapisovat. Celkový rámec databázových transakcí je velmi rozsáhlý. Zde uvedený rozsah ale dává informace, o co jde a jak to funguje, protože transakce jsou velmi důležitým mechanismem v databázovém světě.

3.4.5. Parametrizované příkazy

Je obvyklé, že aplikace vykonává stejný příkaz na databázi opakovaně, přičemž se mění jen hodnota nebo hodnoty používané v tomto příkazu. Typickým příkladem je několikanásobné modifikování tabulky příkazem UPDATE. Je-li třeba v jednom otevřeném spojení vykonávat tentýž příkaz SQL, který se liší pouze v některých parametrech, pak právě pro tento případ jsou

parametrizované příkazy tím pravým řešením. Není třeba znova vypisovat celý SQL příkaz, nýbrž stačí jen změnit parametry SQL příkazu.

```
SqlConnection^ sqlConnection = gcnew SqlConnection(
L"Data Source=PLATITO;Initial Catalog=NESEHNUTI;Integrated Security=True");
try
{
SqlCommand^ sqlProc = gcnew SqlCommand(
L"UPDATE osoba SET aktivni = @statuts WHERE id = @id   ", sqlConnection);

      // připravení parametrů pro použití
sqlProc->Parameters->Add("@status", SqlDbType::varchar);
sqlProc->Parameters->Add("@id", SqlDbType::int);

      // první vykonnání příkazu
sqlProc->Parameters["@status"]->Value = ‘ANO‘>Parameters->archar);vat celý SQL příkaz,
nýbrž stačí jen změnit parametry SQL příkazu.odnota nebo hodnoty používané v tomto □;
sqlProc->Parameters["@id"]->Value = 10>Parameters->archar);vat celý SQL příkaz, nýbrž
stačí jen změnit parametry SQL příkazu.odnota nebo hodnoty používané v tomto □;
sqlProc->ExecuteNonQuery();

      // druhé vykonnání příkazu
sqlProc->Parameters["@status"]->Value = ‘NE‘>Parameters->archar);vat celý SQL příkaz,
nýbrž stačí jen změnit parametry SQL příkazu.odnota nebo hodnoty používané v tomto □;
sqlProc->Parameters["@id"]->Value = 15>Parameters->archar);vat celý SQL příkaz, nýbrž
stačí jen změnit parametry SQL příkazu.odnota nebo hodnoty používané v tomto □;
sqlProc->ExecuteNonQuery();

      // třetí vykonnání příkazu
sqlProc->Parameters["@status"]->Value = ‘ANO‘>Parameters->archar);vat celý SQL příkaz,
nýbrž stačí jen změnit parametry SQL příkazu.odnota nebo hodnoty používané v tomto □;
sqlProc->Parameters["@id"]->Value = 97>Parameters->archar);vat celý SQL příkaz, nýbrž
stačí jen změnit parametry SQL příkazu.odnota nebo hodnoty používané v tomto □;
sqlProc->ExecuteNonQuery();
}
catch(SqlException^ ex)
{
      //ošetření vzniklé výjimky
}
finally
{
      SqlConnection->Close();
}
```

Vykonání příkazů SQL se fyzicky děje v metodě vykonání (v uvedeném příkladě ExecuteNonQuery). Stačí tedy vždy pouze pozměnit parametry v objektu SqlCommand a není třeba inicializovat nový objekt, ani opětovně zadávat celý SQL příkaz. O to se postará nastavení nadefinovaných parametrů. Ze zápisu přikladu je zřejmé, že objekt SqlCommand byl vytvořen pouze jednou, stejně jako řetězec s SQL příkazem, avšak příkaz se na databázi provedl celkem třikrát. Výhoda parametrických dotazů je tedy zřejmá.

3.4.6. Uložené databázové procedury

Uložené procedury jsou nejlepším nástrojem pro vývojáře na straně SQL Serveru 2005. Uložené procedury jsou uživatelem definované objekty uvnitř databáze SQL Server. Jsou to procedurální jednotky, které umí vykonávat téměř všechny příkazy jazyka T-SQL. To znamená, že umí provádět příkazy DDL (CREATE, ALTER, DROP) , také příkazy DML (INSERT, DELETE, UPDATE a SELECT) a dále umí ještě další věci navíc (různé manipulace s daty, systémové funkce a další). Pro potřeby databáze SQL Serveru jsou plně komplexní.

3.4.6.1. Uložené procedury versus dynamické SQL příkazy

Všechny tyto vlastnosti umí i dynamické SQL příkazy, jak byly uvedeny v předchozích příkladech. Tak proč existují uložené procedury? Objekty uvnitř databáze mají ty vlastnosti, že jsou zkompilované již před použitím v aplikační vrstvě – protože jsou zkompilované, jsou efektivnější (přesnější slovo je rychlejší). Výkonnostní rozdíl se dá přirovnat k rozdílu mezi přeloženým a interpretovaným kódem.

3.4.6.2. Výhody uložených procedur

Zakódování často používaných příkazů jako uložených procedur je obecná technika zvyšování výkonnosti datových aplikací. Právě v koncových databázích se často nacházejí nejužší výkonnostní místa, takže vše, co dokáže urychlit databázové operace, bude mít přímý dopad na celkovou výkonnost. Nepopíratelnou výhodou je oddělení aplikační logiky od získávání dat z databáze.
Uložené procedury prostě zapouzdřují SQL příkazy, čímž jsou snadněji udržované (jakákoliv změna se děje na vrstvě databáze.)
ADO.NET podporuje uložené procedury. Jejich syntaxe se velmi podobá parametrizovaným příkazům.
Uvedený příklad poukazuje na zapouzdření a oddělenou aplikační logiku od logiky získávání data. Mějme uloženou proceduru, která na základě vstupů dává výstupní informaci (tabulka parametrů pro aplikaci):

```
CREATE PROCEDURE [dbo].[getParamSetting]
      @par_kod     varchar(100),
      @return            varchar(100) OUTPUT
AS
```

```
	SET NOCOUNT ON;
BEGIN
	SELECT @return=hodnota FROM sprava_params
		WHERE nazev = @par_kod;

	IF isnull(@return, '#@') = '#@'
		BEGIN
			RAISERROR('Uvedený parametr se nenachází v tabulce systémových
parametrù',11,1)
		END
END
```

Uvedená uložená procedura se dotazuje do databáze na základě vstupního parametru. Zároveň provádí kontrolu existence parametru a výslednou informaci předá o úroveň výš (buď se dohledá zadaný parametr a uložená procedura vrátí výslednou hodnotu parametru nebo vygeneruje výjimku, kterou předá na místo výsledné hodnoty), takže se aplikační vrstva dozví výsledek operace.

A zde je volání programové jednotky v jazyce z aplikační vrstvy jazyka C++/CLI – funkce má návratový typ String – při získání hodnoty z databáze je to hodnota v příkazu return

```
String^ ConnectClass::getSettingParam(String^ param)
{
	SqlConnection^ sqlConnection = gcnew SqlConnection(
L"Data Source=PLATITO;Initial Catalog=NESEHNUTI;Integrated Security=True");

/* třída SqlCommand je obsahuje místo dynamického příkazu SELECT název uložené
procedury*/
	SqlCommand^ sqlProc = gcnew SqlCommand(L"getParamSetting", sqlConnection);

	/* instance třídy SqlCommand je nadefinována jako uložená procedura*/
	sqlProc->CommandType = CommandType::StoredProcedure;

	/* vstupní parametry*/
	sqlProc->Parameters->AddWithValue("@par_kod", param);

	/* výstupní parametry*/
	SqlParameter^ retval = sqlProc->Parameters->Add("@return", SqlDbType::VarChar);
	retval->Size::set(100);
	retval->Direction = ParameterDirection::Output;

	try
	{
		sqlConnection->Open();
		/* vykonání uložené procedury*/
		sqlProc->ExecuteNonQuery();
	}
	catch(SqlException^ e)
	{
		MessageBox::Show(
			"Pri zíkávání informací z databáze došlo k chybě: " + e-
>ToString(),
			"Chyba",
			MessageBoxButtons::OK,
			MessageBoxIcon::Exclamation);
	}

	finally
	{
```

```
            sqlConnection->Close();
      }

      return retval->Value->ToString();
}
```

Základním kamenem pro volání databázové uložené procedury z aplikační vrstvy je pojmenování uložené procedury (její identifikace na SQL Server) na místo vložení dynamického SELECT příkazu v instanci SqlCommand a nastavení typu příkazu této instance na StoredProcedure. Poté se již pracuje s instancí SqlCommand stejně jako ve všech předešlých příkladech při volání výkonných metod na otevřeném připojení do databáze.

Uložené procedury nejsou omezeny pouze na jednu návratovou hodnotu. Protože jsem se zmínil, že lze v uložených procedurách také vykonávat databázové dotazy příkazem SELECT, umí uložené procedury vracet výsledné sady databázových dotazů. Viz následující příklad.

Objekt uložené procedury na straně SQL Serveru vypadá následovně:

```
CREATE PROCEDURE [dbo].[getPermissions]
      @uzivatel    varchar(100),
      @heslo varchar(100)
AS
BEGIN
      SET NOCOUNT ON;
      -- dotaz do databáze
      select opk.nazev
            from  opravneni_konf opk, opravneni op, uzivatel uz
                  where op.id_opravneni = opk.id_opravneni
                  and op.id_uziv = uz.id_uziv
                  and uz.uzivatel = @uzivatel
                  and uz.heslo = @heslo
END
```

Na základě vstupních parametrů se procedura dotazuje na seznam oprávnění nastavených pro konkrétního uživatele aplikace – přístup je zajišťován na aplikační úrovni.

Volání uložené procedury z aplikační vrstvy je opět zapouzdřeno do aplikační funkce jazyka C++/CLI:

```
ArrayList^ ConnectClass::getPermissions(String^ user, String^ password)
{
       permissions = gcnew ArrayList();
       sqlConnection = gcnew SqlConnection(getSqlConnectString());
       SqlCommand^ sqlProc = gcnew SqlCommand(L"getPermissions", sqlConnection);
       sqlProc->CommandType = CommandType::StoredProcedure;

      /* vstupní parametry*/
       sqlProc->Parameters->AddWithValue("@uzivatel", user);
       sqlProc->Parameters->AddWithValue("@heslo", password);

       try
       {
            /* otevreni spojeni a vykonani dotazu*/
            sqlConnection->Open();
            SqlDataReader^ reader = sqlProc->ExecuteReader();
```

```
                while (reader->Read())
                {
                        this->permissions->Add(reader["nazev"]);
                }
        }

        catch(Exception^ e)
        {
                MessageBox::Show(
                      "Pri prihlasovani nastala chyba: " + e->ToString(),
                      "Chyba",
                      MessageBoxButtons::OK,
                      MessageBoxIcon::Exclamation);
                permissions->Add(L"NO_PERMISSIONS");
        }

        finally
        {
                sqlConnection->Close();
        }
        return permissions;
}
```

Metoda po vykonání uložené procedury provede naplnění instance typu ArrayList daty z databázového dotazu vykonaným na SQL Serveru.

3.4.6.3. Použití uložených procedur v aplikacích

Nedovedu si představit, že by z vrstvy aplikace programovacího jazyka C++/CLI posílal občasné T-SQL příkazy nějaký administrátor a tím způsobem by se staral o chod databáze. V ostatních případech se jedná o naprogramované jednotky, jejichž volání se neustále opakuje. Touto úvahou lze dojít k závěru, že při používání databáze SQL Server 2005 je neprofesionální, velice netransparentní a v jistých případech kontraproduktivní používat dynamické příkazy psané přímo do aplikační vrstvy. Pokud požadujeme od aplikace, aby byla efektivní snadno upravitelná pak je třeba provést oddělení aplikační logiky od pohledu na získávání data a interakci s nimi. Toho lze dosáhnout uloženými procedurami na straně databáze.

3.4.7. Využití objektů DataSet a DataAdapter

Třída SqlDataReader nabízí proudový přístup k výsledkům databázových dotazů. Proudový přístup je rychlý a výkonný, je však pouze dopředný. Nelze znovu načíst předchozí záznam pomocí DataReader ani změnit výsledky a zapsat je do databáze. Právě proto ADO.NET podporuje kromě proudového přístupu k datům také množinový přístup k datům. Přístupy s využitím množin zachycují celý obraz do paměti a podporují zpětné i dopředné procházení

výsledkovou množinou. Rovněž dovolují upravovat data získaná databázovými dotazy, postupovat takové změny zpět do datového zdroje a mnoho dalšího.
Množinové přístupy k datům se točí kolem dvou tříd: DataSet, což je třída ekvivalentní pamětové databáze, definovaná ve jmenném prostoru System::Data, a DataAdapter, což je třída sloužící jako most mezi sadami DataSet a fyzickými zdroji dat. Přístup používání těchto tříd abstrahuje datový model SQL.

3.4.7.1. Třída DataSet

Třída DataSet je považována za databázi uloženou v paměti. Vlastní data jsou uložená v objektech DataTable, které jsou analogické k tabulkám v databázi (databázové tabulky jsou na ně mapovány).
Objekty DataSet jsou ideální pro zachycování výsledků databázových dotazů a jejich ukládání do paměti za účelem zkoumání a třeba také úpravy dat. Na rozdíl od třídy DataReader, která podporuje pouze dopředný přístup k datů, jež zapouzdřuje, podporuje DataSet náhodný přístup.

3.4.7.2. Třída DataAdapter

Třídy DataSet nepracují s databázemi přímo. Tuto práce přenechávají objektům DataAdapter. Smyslem objektu DataAdapter je vykonávat databázové dotazy a vytvářet tabulky DataTable obsahující výsledky dotazů. Jsou také schopny zapisovat učiněné změny v prvcích DataTable zpět do databáze. DataAdapter funguje jako prostředník zajišťující určitou vrstvu abstrakce mezi DataSet a fyzickým datovým zdrojem.
Zásadními metodami třídy DataAdapter jsou Fill a Update. První se dotazuje do databáze a inicializuje DataSet obdrženými výsledky. Druhá postupuje změny zpět do databáze.
Pro tento druh práce s databázemi poskytuje Visual Studio 2005 naprosto perfektní prostředí pro vytváření tzv. „data source“. Nejde o nějakou hubenou součást Visual Studia 2005, ale velmi intuitivní GUI prostředí, ze kterého lze navrhovat a spravovat tabulky, vztahy mezi nimi tak, aby byly vhodné pro práci s aplikací (samozřejmě musí být objekty vytvořeny na databázové straně).

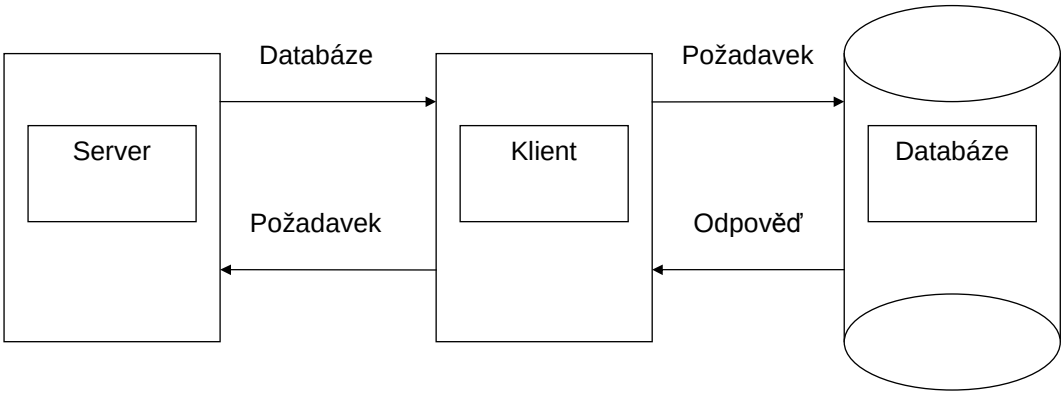
3.4.8. Výběr modelu programování

Na výběr jsou tedy dva modely pro získávání dat z databáze a pro práci s nimi. První přes proudový přístup objektu SqlCommand (DataReader) využívat uložené procedury na straně SQL Serveru, druhý je použití velké flexibility tříd poskytujících mapování fyzických tabulek za asistence tříd DataSet a DataAdapter.
Odpověď na rozhodnutí není jednoznačná, závisí na použití a práci s daty. Jestli je třeba data převážně číst, tak je vhodné využít proudového přístupu (objekt DataReader). Je-li ale potřeba pro získání dat tyto data často měnit a zpět vkládat do databáze, pak je vhodnější přístup k datům za pomoci třídy DataSet.
Další argument pro rozhodování způsobu získávání dat je uvědomit si, s jakými ovládacími prvky bude aplikace pracovat. Některé ovládací prvky podporují vázání dat k objektům DataSet. To však neznamená, že by nebylo možné na ty samé prvky výkonně vázat i objekt DataReader – naopak, vázání k objektům DataReader bývá někdy rychlejší, protože objekt DataReader neponechává výslednou množinu v databázi.

3.5. Vzdálené řízení - .NET Remoting

3.5.1. Úvod do .NET Remoting

I přesto, že společnost Microsoft velice důsledně prosazuje nový model programování pro programování tenkých klientů (aplikace využívající možnosti webového prohlížeče) – ASP.NET, nezatracuje v platformě .NET Framework vývoj aplikací založených na tlustých klientech (klientská aplikace, která funguje na hostitelském počítači v jeho operačním systému – aplikace s formuláři Windows). Právě naopak. Platforma .NET Framework i v této oblasti prošla zásadní obměnou. Totiž ne všechny aplikace se hodí pro filosofii tenkých klientů. Toho si je společnost Microsoft vědoma.
Výhoda klientských aplikací je především ve využívání přenosové kapacity sítě. Mohou se spolehnout na binární protokoly namísto http. Úzká vazba umožňuje rovněž stavová spojení mezi klienty a servery, což zjednodušuje úkol vytváření stavových aplikací. Především překonává omezení HTML.
Nespornou výhodou je skutečnost rozdělení celého aplikačního systému do více vrstev. Na obrázku je znázorněna třívrstvá architektura.



Třívrstvá architektura je již dobře známá. Já chci poukázat na možnosti při rozdělení aplikační logiky a v neposlední řadě také rozdělení výkonu mezi tyto vrstvy. V případě, že existuje aplikace, kterou využívají tisíce, desetitisíce nebo ještě větší množství uživatelů, musí tato aplikace poskytnout dostatečný výkon. Protože existují tři vrstvy v uvedeném aplikačním systému, lze rozvrhnout zátěž na tyto vrstvy. Proč má serverová aplikace dělat akce, které může vykonat klient aplikačního systému nebo databázový server?
Výhodou u použití této architektury může být distribuce změn v aplikaci. Dojde-li ke změně aplikační logiky a změnu lze provést pouze na serverové aplikaci pak jde o nespornou výhodu. Protože není třeba nutit klientské aplikace, aby provedly upgrade svých aplikací. Na místo klientských aplikací proběhne změna pouze na té serverové.
Použití vzdáleného řízení umožňuje komunikovat různým aplikacím mezi sebou, díky technologii .NET Remoting, celkem jednoduše.

3.5.2. Základy vzdáleného řízení

Vzdálené řízení začíná třídou, která se má použít jako vzdálená. Konvenční třída může být využita pouze klienty běžícími v téže aplikační doméně. Vzdáleně použitelná třída může být zpracovávána klienty v jiných aplikačních doménách, což může představovat jiné aplikační domény v daném klientském procesu, aplikační domény v jiných procesech nebo aplikační domény na jiných počítačích.
Jak lze napsat vzdálenou třídu?
Takto vypadá vytvoření místní třídy, kterou lze vytvořit pouze v aplikační doméně klienta:

```
public ref class LocalClass{…}
```

Následující třídu lze zapsat v aplikační třídě klienta nebo ve vzdálené aplikační doméně:

```
public ref class RemoteClass : MarshalByRefObject {…}
```

Když klient vytváří vzdálenou instanci RemoteClass, vytváří .NET Framework zprostředkovací objekt proxy v klientské aplikační doméně. Tento proxy se tváří jako skutečný objekt. Volání přijatá proxy se však vysílají do vzdáleného objektu pomocí kanálu spojujícího obě aplikační domény. Tento objekt, obsluhovaný proxy, byl postoupen odkazem, protože se tento objekt nekopíruje do klientské aplikační domény; klient jen drží odkaz na tento objekt. Tímto odkazem je proxy.

Druhým krokem při vzdáleném používání objektu je zajistit, aby serverový proces zaregistroval vzdáleně použitelnou třídu takovým způsobem, aby ji bylo možné aktivovat z jiné aplikační domény.

Existuje několik typů registrace vzdálených tříd:

- Registrování kanálů vestavěné ve zdrojovém kódu – programová konfigurace
- Registrování kanálů z konfiguračního souboru – deklarační konfigurace

Já použiji pouze jeden z typů. Deklarační konfiguraci. Vede mne k tomu především skutečnost, že tato konfigurace jde změnit, aniž by bylo třeba upravovat zdrojový kód a opětovně ho kompilovat.

Jak vytvořit aplikaci využívající vzdálenou třídu? Jednoduše:

- Vytvoří se třída, která s má používat jako vzdálená
- Vytvoří se serverová aplikace, která zaregistruje serverový kanál
- Vytvoří se klientská aplikace, která zaregistruje klientský kanál

Vytvoření vzdáleně použitelné třídy

```
namespace ConnectServer{

      public ref class ConnectClass : MarshalByRefObject
      {
      // definice třídy
      };
}
```

To nejjednodušší je hotovo.

Vytvoření serverové aplikace

Serverová aplikace provede registraci komunikační linky mezi objektem a vzdáleným klientem. Vzdálená třída běží totiž v aplikační doméně serverové aplikace a klientské aplikace se připojí k té serverové. Nejjednodušší serverová aplikace vypadá následovně:

```
int main(array<System::String ^> ^args)
{
```

```
	Runtime::Remoting::RemotingConfiguration::Configure(
		"ConnectDBServer.exe.config");
	Console::WriteLine(L"Server poslouchá");
	Console::ReadLine();
}
```

Serverová aplikace dělá pouze to, že zaregistruje komunikační kanál, na kterém poslouchá volání vzdálených klientských aplikací. Deklarační konfigurace je obsažena v konfiguračním souboru ConnectDBServer.exe.config:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
                   type="ConnectServer.ConnectClass, ConnectServer"
                   objectUri="temp" />
      </service>
      <channels>
        <channel ref="tcp" port="9555">
          <serverProviders>
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

Hodnoty v konfiguračním souboru lze měnit, aniž by byla nutná rekompilace zdrojového kódu.

Vytvoření klientské aplikace

```
public ref class FormMain : public System::Windows::Forms::Form
{
public:
	FormMain(void)
	{
		TcpClientChannel^ clientChannel = gcnew TcpClientChannel();
		ChannelServices::RegisterChannel(clientChannel);
		RemotingConfiguration::Configure("ConnectDBClient.exe.config");
		ConnectClass^ connectClass = gcnew ConnectServer::ConnectClass();

		/* ostatní definice třídy */
	};
}
```

Klientská aplikace při registrování vzdáleného objektu vytvoří komunikační kanál, který je nakonfigurován v konfiguračním souboru ConnectDBClient.exe.config. Vytvoření komunikačního kanálu probíhá nastavením adresy serverová aplikace, přes který se má komunikovat:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type="ConnectServer.ConnectClass, ConnectServer"
        url="tcp://192.168.0.3:9555/temp" />
```

```
      </client>
      <channel ref="tcp" port="0">
        <serverProviders>
          <formatter ref="binary" typeFilterLevel="Full" />
        </serverProviders>
      </channel>
    </application>
  </system.runtime.remoting>
</configuration>
```

To je vše, teď stačí již jen inicializovat objekt vzdálené třídy:

```
ConnectClass^ connectClass = gcnew ConnectServer::ConnectClass();
```

Při vývoji klientské aplikace lze od chvíle inicializace vzdáleného objektu využívat jeho metody a vlastnosti.

4. Ukázková aplikace

Ukázková aplikace, která je součástí této práce, je systém na udržování kontaktů neziskového sdružení. Systém udržuje osoby a informace o nich v databázi. Hlavní funkcionalita aplikace je ve sdružování těchto kontaktů podle zadaných kritérií. V GUI prostředím aplikace lze odesílat hromadnou e-mailovou korespondenci i s přílohami na základě uvedených kritérií. Doprovodná aplikace je interní informační systém občanského sdružení NESEHNUTÍ (http://www.nesehnuti.cz/).

4.1. Instalace aplikace v cizím prostředí

Ukázková aplikace, která byla vytvořena jako podklad k této práci musí být upravena pro cílové prostředí.

Připojení databáze

Instalační CD, obsahuje databázi SQL Server 2005. Tato databáze je obsažena v databázovém souboru „NESEHNUTI.mdf“ ve složce data.
Uvedený soubor je třeba připojit k databázovému stroji na cílovém počítači (v Management Studiu v nabídce „Attach…“).

Nastavení serverové aplikace

Ve složce „Server“ se nachází zdrojové kódy pro serverovou aplikaci. Ve složce „Server/debug“ jsou uloženy soubory, ve kterých je implementována serverová aplikace. Pro potřeby v cizím prostředí je třeba nakopírovat složku „/debug“ do cílového adresáře a nastavit komunikační kanál v konfiguračním souboru „ConnectDBServer.exe.config“.

Nastavené klientské aplikace

Ve složce Klient se nachází zdrojové kódy pro klientskou aplikaci. Ve složce „Klient/ ConnectDBClient“ je uložen projekt klientské aplikace. Pro potřeby v cizím prostředí je třeba definovat adresu serverové aplikace v konfiguračním souboru „ConnectDBClient.exe.config“. A nastavit zdroj dat pro Windows formuláře v souboru „NESEHNUTIDataSetNew.xsd“ – položka (tag Connection). Klientská aplikace se po těchto úpravách musí nově zkompilovat.

Nastavení vzdálené třídy

Ve složce „VzdalenaTrida“ je uložen projekt využívané vzdálené třídy. V tomto projektu je třeba nastavit statickou proměnnou „connectString“ na správnou cestu k datovému zdroji pro vzdálenou třídu. Poté celý projekt zkompilovat a nakopírovat do adresářů serverové i klientské aplikace.

5. Závěr

Cílem práce bylo poskytnout komplexní přehled pro vytváření aplikací na platformě Windows. Zejména pro zájemce programování těch aplikací, které spolupracují s databázovou platformou SQL Server 2005. Programovacím jazykem pro aplikační část byl vybrán C++/CLI.
Zejména popis SQL Server 2005 a jeho vnitřního jazyka T-SQL považuji za příjemný a přehledný (leč stručný) úvod do této problematiky.
Při popisování využití jazyka C++/CLI jsem se pak zaměřil jen na části, které přímo souvisí s databází SQL Server 2005. Zde jsem popsal některé části do mělkých základů. Myslím si , že pro zvídavé čtenáře (především tím míním lačné vývojáře) to je informativní materiál, který by mohl probudit jejich zájem o dané téma.
Bohužel musím konstatovat, že i přes obecné a neoficiální prohlášení společnosti Microsoft, jenž hlásá o rovnosti programovacích jazyků vývojové platformy .NET Framework, není tak toto tvrzení zcela korektní. Při programování ukázkové aplikace k této práci jsem nesčetněkrát narazil na problém, který mne utvrdil, že jazyk C++/CLI není tak rovný, jako jazyk C# nebo jazyk Visual Basic. Především jde-li o dokumentaci k jazyku C++/CLI. I syntaxe jazyka C++/CLI je trochu kostrbatější ve srovnání s programovacím jazykem C#.

6. Seznam použitých zdrojů

http://msdn2.microsoft.com

http://www.artofprogramming.net/development/dev_sql_programming.html

http://objekty.pef.czu.cz/2003/sbornik/Wiszczor2003.pdf

Robert Vieira: SQL Server 2000 Programujeme profesionálně, Wrox 2001

Joseph Sack: Velká kniha T-SQL SQL Server 2005; APRESS 2005

Aravind Corera a kol.: Visual C++ .NET pro programátory v C++; Wrox 2003

Heft Prosise: Programování v Microsoft. NET webové aplikace v .NET .NET Framework; Microsoft 2003