Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Bakalářská práce

Přístupy k práci s relačními databázemi na platformě .NET

Josef Wudy

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Josef Wudy

Informatika

Název práce

Přístupy k práci s relačními databázemi na platformě .NET

Název anglicky

Approaches of working with relational databases in .NET platform

---

Cíle práce

Hlavním cílem práce je komparace jednotlivých přístupů k práci s relačními databázemi v aplikacích vytvořených na platformě .NET.
Mezi dílčí cíle práce patří:
- vymezení platformy .NET a jazyka C#
- interpretace pojmu relační databáze
- charakteristika jednotlivých přístupů pro práce s databázemi
- praktická ukázka práce s databázemi v aplikacích a analýza získaných poznatků

Metodika

V teoretické části práce budou specifikovány základní pojmy vztahující se k tématu. Přístupy k práci s relačními databázemi budou definovány zejména na základě poznatků získaných studiem oficiální specifikace platformy .NET, odborné literatury a článků, které se zabývají vývojem databázových aplikací využívajících přístupů připojené či odpojené databáze, nebo objektově relačního mapování.

V praktické části bude vytvořena vlastní aplikace, která bude sloužit ke komparaci jednotlivých metod přístupu k datům a to zejména z hlediska výkonu, čistoty kódu a dostupných možností. Na závěr bude provedeno srovnání výsledků pomocí vícekriteriální analýzy variant, která poslouží ke shrnutí získaných poznatků.

Doporučený rozsah práce

30 – 40 stran

Klíčová slova

.NET, relační databáze, ORM, LINQ, Entity Framework

Doporučené zdroje informací

BOEHM, Anne. a Ged. MEAD. Murach's ADO.NET 4 database programming with C# 2010. 4th ed. Fresno, CA: Mike Murach & Associates, c2011. ISBN 978-1-890774-63-9.

CALVERT, Charles. a Dinesh KULKARNI. Essential LINQ. Upper Saddle River, NJ: Addison-Wesley, c2009. Microsoft .NET development series. ISBN 978-0321564160.

LERMAN, Julia. Programming entity framework. Sebastopol, CA: O'Reilly, c2009. ISBN 978-0596520281.

PRICE, Jason. Mastering C# database programming. San Francisco: Sybex, c2003. ISBN 978-0782141832.

Předběžný termín obhajoby

2017/18 LS – PEF

Vedoucí práce

Ing. Václav Lohr, Ph.D.

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 31. 10. 2017

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 1. 11. 2017

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 20. 11. 2017

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Přístupy k práci s relačními databázemi na platformě .NET" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne _____ _____

Poděkování

Rád bych touto cestou poděkoval Ing. Václavu Lohrovi Ph.D. za odborné vedení při tvorbě této bakalářské práce.

Přístupy k práci s relačními databázemi na platformě .NET

Abstrakt

Tato bakalářská práce pojednává o jednotlivých přístupech k práci s relačními databázemi a to zejména se zaměřením na .NET framework. Teoretická část práce se orientuje na charakteristiku základních pojmů v rámci .NET frameworku, principů fungování a implementace relačních databází a vzájemné propojení těchto dvou oblastí. Na začátku jsou definovány jednotlivé složky frameworku .NET a jednotlivé programovací jazyky, se kterými se v jeho rámci dá pracovat. Následně je objasněn koncept relačních databází, jejich historie a vývoj. Dále jsou rozebrány jednotlivé způsoby práce s relačními databázemi ve spojení s aplikacemi na platformě .NET. V praktické části je následně provedena komparace jednotlivých přístupů.

Cílem této práce je provedení komparace pomocí vícekriteriální analýzy variant na základě zadání aplikačních požadavků a v poslední řadě vyhodnocení a analýza získaných výsledků.

Klíčová slova: .NET, SQL, Relační databáze, C#, Entity framework, LINQ, ADO.NET, ORM

Approaches of working with relational databases in .NET platform

Abstract

This bachelor thesis focuses on individual approaches of working with relational databases specifically in .NET framework. Theoretical part of this thesis is oriented on characteristics of basic concepts of .NET framework, relational databases principles and mutual connection of these two parts. In the beginning individual components of .NET framework and programming languages, which are possible to work with, are defined. Then the concept, history and evolution of relational databases are clarified. After that every particular approach of working with relational databases in connection with applications on .NET platform is resolved. The practical part of this thesis composes of the comparation of these approaches.

The purpose of this thesis is application of multiple-criteria decision analysis based on application requirements and lastly the evaluation and analysis of its results.

Keywords: .NET, SQL, Relational databases, C#, Entity framework, LINQ, LINQ To SQL, ADO.NET, ORM

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Žijeme v době informací a převážná většina moderních aplikací využívá ke svému fungování data uložená v databázích různého typu, mezi kterými však výrazně dominují databáze relační. I přes svou popularitu přináší relační databáze některé problémy při vývoji aplikací, které souvisejí zejména s nezanedbatelnými rozdíly mezi relačním paradigmatem na straně databázových systémů a aktuálně hojně využívaným objektově orientovaným přístupem na straně aplikací. Jako řešení tohoto problému vznikají různé mezivrstvy mezi těmito technologiemi. Například v rámci frameworku .NET, který je jedním z hlavních témat této práce, existuje hned několik řešení přímo od firmy Microsoft, mezi které se řadí funkce poskytované přímo knihovnami ADO.NET, které umožňují využití principů připojené a odpojené databáze, a dále nástroje pro objektově relační mapování LINQ to SQL a Entity Framework. I přes existenci těchto nástrojů, které byly vytvořeny přímo pro .NET framework, existuje dále velká řada nastrojů od jiných společností, které slouží k řešení těchto problémů a jsou se zmíněným frameworkem plně kompatibilní.

Nepřeberné množství těchto nástrojů a principů naopak vede k problémům při rozhodování, který z nich má být pro vývoj daných aplikací zvolen. Jako programátor se zaměřením na .NET framework a relační databáze jsem se s touto situací již několikrát osobně setkal a zároveň jsem byl svědkem negativních dopadů v případě špatně zvoleného principu pro práci s relační databází, což bylo mou hlavní motivací pro zpracování tohoto tématu v rámci mé bakalářské práce.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem této bakalářské práce je komparace jednotlivých přístupů pro práci s relačními databázemi na frameworku .NET. Práce dále obsahuje několik dílčích cílů z nichž prvním je vymezení samotného .NET frameworku, jeho komponent a základních programovacích jazyků, s nimiž je v rámci něj možné vyvíjet aplikace. Dalším z dílčích cílů je interpretace pojmu relační databáze a základních prvků, které s ním souvisejí. Posledním dílčím cílem teoretické části práce je charakteristika jednotlivých přístupů pro práci s relačními databázemi. Cílem praktické části je názorná ukázka práce s jednotlivými přístupy pomocí jazyka C# a SQL, na jejímž základě bude provedna analýza získaných poznatků.

2.2 Metodika

První část této bakalářské práce je zpracována na základě informací získaných ze zdrojů uvedených v použité literatuře. Jde zejména o knižní publikace, internetové články a oficiální dokumentaci od firmy Microsoft.

Praktická část této práce je orientována na samotnou implementaci základních funkcionalit těchto přístupů a následnou komparaci jejich výkonů. Poté bude provedena vícekriteriální analýza variant, která bude sloužit k výběru optimální varianty pro teoretický aplikační požadavek.

3 Teoretická východiska

3.1 .NET framework

Jedná se o softwarový framework vytvořený firmou Microsoft za účelem umožnění vývoje a exekuce desktopových aplikací pro OS Windows, webových aplikací, mobilních aplikací pro Windows Phone, nebo cloudových aplikací na Microsoft Azure, který přišel na trh v roce 2002.

.NET framework je obvykle dodáván s operačními systémy Windows, nebo je ke stažení na webu společnosti Microsoft. Běžným uživatelům operačního systému Microsoft Windows umožňuje .NET framework spouštět aplikace, které byly vyvinuty právě na tomto frameworku a je pro ně obvykle transparentní. [1]

Hlavní komponenty .NET frameworku tvoří virtuální stroj Common Language Runtime (CLR) a knihovna Framework Class Library (FCL).

3.1.1 CLR

CLR je běhové (runtime) prostředí poskytované .NET frameworkem zajišťující kompilaci, spuštění kódu a poskytuje služby usnadňující vývoj. Umožňuje též kompilaci typu just-in-time (JIT), která je vykonávána až při exekuci programu a nikoliv před ní.

Při kompilaci jsou zdrojové kódy programů napsané ve vyšších programovacích jazycích (C#, Visual Basic atd.) překládány kompilátorem do jazyka CIL (Common Intermediate Language), což je objektově orientovaný jazyk nižší úrovně podobný strojovému kódu, který je založený na specifikaci CLI (Common Language Infrastructure). Mezi hlavní složky CLI specifikace se řadí typový systém CTS (Common Type System) určující povolené typy (hodnotové a referenční) a specifikace CLS (Common Language Specification), která zajišťuje kompatibilitu použitých programovacích jazyků. Kód přeložený do CIL se nazývá řízený (managed code). Ten je při exekuci překládán virtuálním strojem (CLR) do strojového kódu specifického pro cílovou platformu, který již následně může být zpracován procesorem.

CLR taktéž zajišťuje alokaci operační paměti pomocí tzv. Garbage collectoru, který automaticky zahazuje nepoužívané objekty a uvolňuje tak paměť, kterou využívaly. Tato

technologie výrazně usnadňuje práci vývojářům, protože se o správu paměti nemusejí vůbec starat. [2]

3.1.2 FCL

FCL je knihovna základních tříd platformy .NET obsahující velké množství tříd řízeného kódu, které je možné využívat v jakémkoliv z programovacích jazyků využívajících tuto knihovnu, protože většina z komponent knihovny splňuje specifikaci CLS, čímž zároveň zajišťuje možnost interoperability mezi jednotlivými jazyky.

Jedná se o klíčovou složku .NET frameworku, která poskytuje třídy, rozhraní a hodnotové typy usnadňující proces vývoje a zajišťující přístup k systémovým funkcionalitám. [3]

3.1.3 Vývojové prostředí

Vývojové prostředí neboli IDE (Integrated Development Environment) je software sloužící primárně k vývoji, testování a ladění aplikací. Obvykle obsahuje velké množství nástrojů, jejichž účelem je zvýšení efektivity práce vývojářů. Různá vývojová prostředí se mohou skládat z různých součástí, avšak mezi ty základní se nejčastěji řadí tyto:

- editor zdrojového kódu – jedná se v podstatě o textový editor, který slouží k psaní a formátování kódu. Ve většině případů zároveň podporuje zvýrazňování syntaxe programovacích jazyků
- kompilátor a/nebo interpret – nástroj používaný k překladu zdrojového kódu napsaného ve vyšším programovacím jazyce do strojového kódu nebo IL (Intermediate Language)
- debugger – software pro ladění a hledání chyb v programu, který většinou umožňuje pozastavování programu za běhu a lokalizaci případných chyb ve zdrojovém kódu

Nativním vývojovým prostředím pro .NET framework je Visual Studio, jehož vydavatelem je firma Microsoft. Jedná se o robustní sadu nástrojů sloužící k usnadnění vývoje. Obsahuje všechny z výše zmíněných komponent a mnoho dalších. Mezi zajímavější ze složek se řadí například IntelliSense, CodeLens, Profiler, návrhové zobrazení pro aplikace s grafickým rozhraním, nebo možnost verzování aplikací pomocí

verzovacího nástroje Git. Uživatel si může navíc doinstalovat mnoho dalších pluginů a doplňků pomocí integrovaného nástroje Nugget. [4]

3.1.4 **Historie .NET frameworku**

.NET framework, resp. jeho první verze .NET 1.0, byl publikován v roce 2002 a od té doby se postupně rozvíjel až do dnešní podoby. V době psaní této bakalářské práce je nejaktuálnější verzí .NET 4.7. S každou další verzí se samozřejmě rozšiřoval obsah frameworku a jeho možnosti. Hlavní změny a novinky, které přinesly jednotlivé verze jsou zobrazeny v tabulce 1.

Tabulka 1 - Verze .NET frameworku

Verze	Rok vydání	Novinky a změny
1.0	2002	První verze .NET frameworku
1.1	2003	Podpora protokolu IPv6 Změny v zabezpečení Změny v ADO.NET (podpora ODBC a Data Provider for Oracle) ASP.NET Mobile Controls Side-by-Side exekuce
2.0	2006	Podpora 64-Bitových aplikací Změny v ASP.NET (nové kontrolky, možnost předkompilace) Podpora FTP Genericita a generické kolekce Serializace Podpora SMTP
3.0	2006	Windows Communication Foundaction (WCF) Windows Presentation Foundation (WPF) Windows Workflow Foundation (WF) Windows CardSpace
3.5	2007	Language-Integrated Query (LINQ) Entity Framework Změny v ASP.NET (podpora AJAX)
4.0	2010	Portable Class Library Rozšíření Base Class Library Dynamic Language Runtime (DLR) Managed Extensibility Framework (MEF)
4.5	2012	Podpora Windows Store aplikací Změny ve WPF, WCF, WF a ASP.NET
4.5.1	2013	Podpora Windows Phone Store aplikací Vylepšení v oblasti výkonu a ladění
4.5.2	2014	Nové API pro ASP.NET aplikace Nové možnosti profilingu
4.6	2015	ASP.NET 5 (cloudově zaměřené aplikace) 64-bitová verze JIT kompilátoru (RyuJIT) Zvýšení zabezpeční databáze pomocí technologie Always Encrypted
4.6.1	2015	Podpora certifikátů X509 obsahujících ECDSA Podpora HW certifikátů pro Always Encrypted
4.6.2	2016	Možnost konverze WPF a Windows Forms aplikací na UWP Podpora certifikátů X509 obsaujících FIPS 186-3 DSA, výpočet podpisů pomocí SHA-2 algoritmů
4.7	2017	Vylepšení kryptografických funkcí pomocí ECC Podpora výchozí verze protokolu TLS v závislosti na operačním systému

Zdroj: [5]

3.1.5 Programovací jazyky

Jak již bylo zmíněno, .NET framework umožňuje pracovat s velkým množstvím programovacích jazyků, které splňují specifikaci CLI. Mezi nejpoužívanější jazyky patří následující:

- C# (C sharp) je objektově orientovaný jazyk, který byl navržen firmou Microsoft přímo pro spolupráci s .NET frameworkem, se kterým je plně integrovaný. Jeho C-like syntaxe zajišťuje velkou podobnost s ostatními jazyky této rodiny, mezi které se řadí například C, Java, nebo C++. V praktické části této práce bude použit právě tento programovací jazyk. [6] Aktuálně je C# podle průzkůmů řazen mezi nejpopulárnější a nejúžívanější programovací jazyky. [7]
- VB.NET (Visual Basic .NET) je nástupcem jazyka Visual Basic, který byl obohacen o mnohé funkcionality. Patří mezi ně zejména dědičnost, výjimky, overloading, overriding, možnost tvorby konstruktorů a destruktorů, delegáty, nebo vícevláknové operace. Zároveň zachovává syntaxi svého předchůdce a je dalším z primárních jazykem frameworku .NET. [8]
- F# (F sharp) byl stejně jako C# vytvořen přímo pro platformu .NET, ale na rozdíl od něj se jedná o silně typovaný multiparadigmatický jazyk kombinující funkcionální, imperativní a objektově orientovaný přístup k programování, přičemž se zaměřuje zejména na funkcionální paradigma. [9]

3.2 Databázové systémy

Relační databáze jsou aktuálně nedílnou součástí života každého člověka. Uchovávají nezměrný objem dat, ať už se jedná například o data bankovních institucí obsahující jednotlivé finanční transakce, klienty, či úvěry, nebo záznamy skladové evidence podniku poskytující data o množství zboží na skladě a jeho pohybech. Všechna tato data je potřeba někde uchovávat a s postupnou digitalizací civilizace se data přenesla z papíru a kartoték právě do databází.

Databáze by se obecně dala označit jako organizovaná kolekce dat. Databáze zároveň poskytuje přístup k těmto datům, vyhledávání v nich a jejich modifikaci. S pojmem databáze je často spojen tzv. Database-Management System (DBMS), což je softwarový nástroj sloužící jako rozhraní ke komunikaci s databází a umožňuje uživateli

manipulovat s daty. Databáze v kombinaci s DBMS tvoří jako celek tzv. databázový systém.

Jednotlivé databáze lze rozlišovat podle použitého datového modelu, který definuje jejich logiku uspořádání dat. Mezi tyto modely patří například:

- Hierarchický model
- Síťový model
- Relační model
- Objektově orientovaný model

Téma této práce je přímo zaměřeno na relační datový model, který je v současnosti nejrozšířenější. [10]

3.2.1 **Relační databáze**

O vznik relačních databází se zapříčinil Edgar Codd svou prací „A Relational Model of Data for Large Shared Data Banks“ napsanou v roce 1970.

V relačních databázích jsou data reprezentována jako záznamy v jednotlivých tabulkách (relacích). Každá tabulka má svůj název a atributy, což jsou v podstatě názvy jednotlivých sloupců. Řádek pak představuje jednu určitou instanci dané tabulky. Jednotlivé atributy se vždy vyznačují definovaným datovým typem, jenž určuje, jaký druh informace lze do daného sloupce uložit. Tabulka 2 obsahuje výčet základních datových typů pro DBMS Microsoft SQL Server. [11]

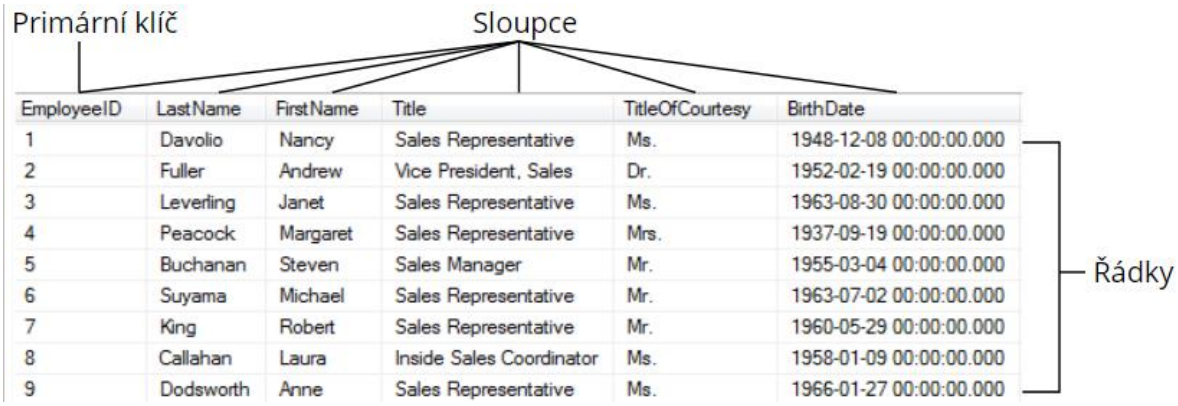
Tabulka 2 - Datové typy MSSQL

Datový typ	Popis
char(n)	Textový řetězec o délce n
varchar(n)	Textový řetězec o maximální délce n
text	Text o maximální velikosti 2GB
binary(n)	Binární řetězec o délce n
bit	Číslo nabývající hodnoty 0, 1, nebo NULL
int	Celé číslo mezi -2^{31} a 2^{31} -1
decimal(p,s)	Číslo s pevnou desetinnou čárkou mezi -10^{38} +1 a 10^{38} +1 Parametr p značí celkový počet číslic Parametr s vyjadřuje maximální počet číslic za desetinnou čárkou
float(n)	Číslo s pohyblivou desetinnou čárkou mezi -17,9E + 308 a 1,79E + 308
money	Peněžní hodnota
datetime	Datum a čas s přesností na milisekundy
date	Datum
xml	Data ve formátu XML o maximální velikosti 2GB

Zdroj: [12]

Každý záznam (řádek) v tabulce musí být jednoznačně identifikovatelný, což zajišťuje tzv. primární klíč (Primary Key – PK). Jedná se o sloupec tabulky, který pro každý řádek obsahuje unikátní hodnotu v rámci dané tabulky, která nikdy nenabývá prázdné hodnoty (NULL). Pokud má záznam jedinečnou vlastnost (atribut) v rámci tabulky, označuje se tento primární klíč za přirozený. Za přirozené primární klíče se dá považovat například rodné číslo v tabulce zaměstnanců, poznávací značka v relaci reprezentující jednotlivé automobily, nebo PSČ v tabulce obcí. Ne každá tabulka však obsahuje sloupec, podle kterého lze záznam jednoznačně odlišit, a tak je u těchto relací potřeba vytvořit umělý primární klíč, což je obvykle atribut číselného datového typu, jenž je nejčastěji označován jako ID. Hodnota umělého primárního klíče číselného typu pak při vložení nového záznamu do relace obvykle není specifikována uživatelem, ale samotným databázovým systémem, jenž zajišťuje jeho jedinečnost a tedy i tzv. entitní integritu pomocí automatické inkrementace. [11] Jednotlivé složky databázové relace jsou znázorněny na obrázku 1.

Obrázek 1 - Komponenty databázové relace



EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate
1	Davolio	Nancy	Sales Representative	Ms.	1948-12-08 00:00:00.000
2	Fuller	Andrew	Vice President, Sales	Dr.	1952-02-19 00:00:00.000
3	Leverling	Janet	Sales Representative	Ms.	1963-08-30 00:00:00.000
4	Peacock	Margaret	Sales Representative	Mrs.	1937-09-19 00:00:00.000
5	Buchanan	Steven	Sales Manager	Mr.	1955-03-04 00:00:00.000
6	Suyama	Michael	Sales Representative	Mr.	1963-07-02 00:00:00.000
7	King	Robert	Sales Representative	Mr.	1960-05-29 00:00:00.000
8	Callahan	Laura	Inside Sales Coordinator	Ms.	1958-01-09 00:00:00.000
9	Dodsworth	Anne	Sales Representative	Ms.	1966-01-27 00:00:00.000

Zdroj: autor

3.2.2 **Vazby tabulek**

Většina tabulek relačních databází je mezi sebou navzájem propojena na základě hodnot určitých atributů. Sloupce umožňující takovéto vazby se nazývají cizí klíče (Foreign Key - FK) a v podstatě se jedná o sloupce, jejichž hodnoty odkazují na primární klíč jiné tabulky. Mezi relacemi mohou existovat tři různé typy vazeb dle jejich kardinality: [13]

1) Vazba 1:1

   Pokud jsou dvě tabulky propojeny vazbou 1:1, pak pro každý záznam v tabulce A existuje právě jeden záznam v tabulce B a naopak. Jejich data by tedy mohla být uložena v jediné Relaci. Často je však užitečné ukládat data o větší velikosti (např. obrázky) do oddělené tabulky a pracovat s nimi pouze pokud je jich opravdu zapotřebí.

2) Vazba 1:M

   Tento typ vazby se v databázích vyskytuje nejčastěji a jeho typickou ukázkou je například vazba mezi matkou a potomky – jedna matka může mít více potomků, avšak každé dítě má pouze jednu biologickou matku.

3) Vazba M:N

   Pro tento vztah mezi tabulkami platí, že každý záznam v tabulce A může být asociován s několika záznamy v tabulce B a každý záznam v tabulce B může naopak odkazovat na několik řádků v tabulce A. Typickým případem využití tohoto typu vazby je například vztah mezi vysokoškolskými studenty a jejich zapsanými

předměty. V databázi je tato vazba obvykle implementována pomocí vazební tabulky, která je ve vztahu 1:M k oběma tabulkám, mezi kterými se vyskytuje vazba M:N.

3.2.3 **Normalizace databáze**

Tvorba správného návrhu databáze je základem pro její korektní fungování a umožnění následné modifikace její struktury. V praxi se v mnoha případech stává, že z důvodu nevhodně zvoleného návrhu je následně obtížné s databází pracovat a implementovat do ní změny. Ke správnému návrhu schématu databáze se zpravidla využívá procesu tzv. „normalizace“, což je technika, kterou navrhl Edgar Codd v roce 1972. V základu obsahovala tři normální formy (NF), avšak později se začaly používat i další pravidla.

Normalizace zajišťuje prevenci proti potencionálním anomáliím při změnách dat. Tyto anomálnie definoval taktéž Edgar Codd a dělí se na tři typy podle SQL příkazu, při kterém vznikají:

1) INSERT anomálie – vzniká v případě, kdy nelze vkládat záznamy do tabulky z důvodu závislosti na jiné entitě. Tato chyba vzniká z důvodu sloučení dvou různých entit do jedné relace.
2) DELETE anomálie – jedná se o opak INSERT anomálie a představuje situaci, kdy smazání záznamu jedné entity vede k nežádoucímu smazání dat odlišné entity.
3) UPDATE anomálie – dochází k ní, pokud úprava jedné konkrétní informace v datech vyžaduje modifikaci více záznamů.

Proces normalizace probíhá individuálně na každé relaci a začíná určením jednoznačného identifikátoru jejích záznamů (primárního klíče). Poté následuje systematická série kroků vedoucích k postupné normalizaci tabulky. Mezi tři základní normální formy se řadí: [14]

1) 1NF – všechny záznamy v tabulce musejí být jednoznačně identifikovatelné primárním klíčem a všechny atributy obsahují atomickou hodnotu
2) 2NF – druhá normální forma vyžaduje, aby tabulka splňovala 1NF. Cílem této NF je eliminovat sady hodnot, které se týkají více záznamů současně. V praxi je

tedy zapotřebí nahradit duplicitní hodnoty cizími klíči odkazujícími na záznamy nové tabulky obsahující původní (duplicitní) hodnoty pouze jednou.

3) 3NF – pro zajištění třetí normální formy relace je opět vyžadováno splnění předchozí, tedy druhé, normální formy. Cílem 3NF je eliminace polí, která nejsou přímo závislá na primárním klíči.

Normalizace slouží k optimalizaci struktury databáze a většinou je jejím hlavním účelem snížit počet redundantních dat a duplicitních úprav. Je to doporučený postup pro návrh databáze, avšak vždy mohou existovat výjimky, při kterých je žádoucí tato pravidla mírně „ohýbat“. [15]

3.2.4 **Pohledy**

Kromě relací se v databázových systémech vyskytují i další objekty, mezi které se řadí i pohledy (views). Ty obsahují pole z jedné nebo více databázových tabulek. Pohledy jsou také někdy označovány jako tzv. virtuální tabulky, protože narozdíl od běžných relací pouze zobrazují data ze zdrojových tabulek a sami o sobě žádná neuchovávají. Jediná informace, kterou databázový systém o pohledech uchovává, je jejich struktura. Všechny pohledy jsou v podstatě uložené databázové dotazy (query) v jazyce SQL. [16]

3.2.5 **Indexy**

V průběhu svého životního cyklu se databáze v ohledu na množství uložených dat zpravidla vždy rozrůstají a zvyšující se objem dat v jednotlivých tabulkách se často podepisuje na výkonu při vyhledávání záznamů. Při takovýchto výkonnostních problémech je vhodné vytvořit v rámci jednotlivých tabulek indexy na sloupcích, podle kterých se často filtruje. Indexy si lze zjednodušeně představit jako tabulky obsahující jeden sloupec s klíčovou hodnotou (hodnoty indexovaného sloupce) a dále sloupec s ukazatelem na řádek tabulky, kterému hodnota fyzicky náleží (v podstatě ID řádku). Pro sloupce, které neobsahují pouze unikátní hodnoty, se do druhého sloupce ukládá seznam odpovídajících ukazatelů.

Rychlost vyhledávání ovlivňují indexy ze dvou důvodů. Prvním z nich je skutečnost, že záznamy indexů jsou znatelně kratší než řádky zdrojové tabulky, což zvyšuje rychlost čtení. Dalším důvodem zvýšení výkonnosti je fakt, že záznamy

v indexech jsou vždy seřazeny, což databázovému systému umožňuje využít binární vyhledávání, které značně redukuje dobu vyhledávání.

Indexy však mají i své nevýhody, mezi které lze například počítat místo, které fyzicky zabírají. Dalším problémem je jejich údržba. Při každém přidání (INSERT) a odebrání (DELETE) záznamu, nebo změny hodnoty indexovaného sloupce musí zároveň dojít k aktualizaci indexu, což snižuje efektivitu těchto operací. Indexy jsou databázovým systémem automaticky vytvořeny na atributech, které jsou primárními klíči, nebo obsahují unikátní hodnoty. [14]

3.2.6 **SQL**

Hlavním nástrojem pro interakci s relačními databázemi je jazyk SQL (Structured Query Language), který vznikl v 70. letech ve výzkumných laboratočích firmy IBM ve městě San Jose pod názvem SEQUEL. Následně organizace ANSI a ISO přijaly SQL jako standardní jazyk pro relační databáze a v roce 1986 publikovaly jeho první oficiální specifikace. Původní standard byl nazýván SQL1 a od svého vzniku byl několikrát modifikován až do dnešní podoby s označením SQL3. Podoba jazyka SQL se v různých DBMS může značně lišit, protože velká část poskytovatelů databázových systémů přidává ve svých implementacích dodatečné funkcionality, nebo rozšíření jazyka, čímž vznikají tzv. SQL dialekty, mezi které se řadí například MySQL, PL-SQL (Oracle Database) a T-SQL (MSSQL). Všechny implementace jazyka SQL však musejí splňovat definované standardy. [17]

Práci s jazykem SQL umožňují základní příkazy, které se dají dělit na dvě skupiny podle účelu jejich využití:

1) DML (Data Modification Language)

   Tyto příkazy jsou používané zejména programátory a slouží k manipulaci s daty. Jejich názvy a účel jsou zobrazeny v tabulce 3.

Tabulka 3 - DML příkazy

Příkaz	Popis
SELECT	Výběr dat z jedné nebo více tabulek
INSERT	Přidání jednoho nebo více řádků do tabulky
UPDATE	Změna jednoho nebo více řádků tabulky
DELETE	Smazání jednoho nebo více řádků tabulky

Zdroj: [13]

2) DDL (Data Definition Language)

   Tato skupina SQL příkazů umožňuje práci s databázovými objekty a ve většině případů je využívají databázoví administrátoři. Jednotlivé příkazy jsou popsány v tabulce 4.

Tabulka 4 - DDL příkazy

Příkaz	Popis
CREATE DATABASE/TABLE/INDEX	Vytvoření databáze/tabulky/indexu
ALTER TABLE/INDEX	Změna struktury tabulky/indexu
DROP DATABASE/TABLE/INDEX	Smazání databáze/tabulky/indexu

Zdroj: [13]

3.2.7 Transakce

V databázových systémech, které jsou využívány v jeden okamžik více uživateli, může nastávat problém tzv. „konkurence“. Při neošetření konkurenčního přístupu k datům by mohlo například vést k současné modifikaci dat více uživateli najednou. K zajištění korektnosti dat se proto využívají databázové transakce, což jsou v podstatě sady jedné, či více databázových operací (INSERT, UPDATE, DELETE nebo SELECT). Ty mohou při své exekuci například zamknout upravované databázové relace, čímž znepřístupní ostatním uživatelům jejich čtení a případnou modifikaci. Dalším charakteristickým prvkem transakcí je možnost provést příkaz ROLLBACK, což je návrat dat do stavu před začátkem exekuce transakce. Tato funkce je obvykle využívána při vzniku chyby v průběhu transakce, nebo modifikaci nějaké z upravovaných relací jiným uživatelem. Naopak při úspěšném provedení transakce jsou všechny změny provedeny hromadně příkazem COMMIT.

Každý databázový systém by měl zajišťovat, že jeho transakce splňují základní čtyři vlastnosti, které shrnuje zkratka ACID: [18]

1) Atomicity (nedělitelnost) – transakce je atomickou jednotkou a je buď provedena jako celek, nebo není provedena vůbec
2) Consistency preservation (zachování konzistence) – databáze před i po provedení transakce zachovává konzistentní stav
3) Isolation (izolace) – exekuce transakce je izolována od exekuce současně běžících konkurenčních transakcí a není jimi ovlivňována
4) Durability (trvalost) – změny dat provedené potvrzenou transakcí (tedy ukončenou příkazem COMMIT) musejí být v rámci dané databáze trvalé a nesmějí být ztraceny

3.2.8 **MSSQL**

Jak již bylo zmíněno, existuje mnoho implementací databázových systémů od různých společností. Mezi jeden z nejpopulárnějších se řadí Microsoft SQL Server (MSSQL), který byl, jak už název napovídá, vytvořen firmou Microsoft. Do roku 2017 byl tento databázový systém určen pouze pro operační systémy vydávané výše zmíněnou firmou (Windows, Windows Server...), poté byla na trh uvedena verze MSSQL 2017, která již funguje i na linuxových operačních systémech. [19]

SQL Server je kompletní sada nástrojů pro práci s relačními databázemi a skládá se z několika komponent. Mezi základní z těchto částí se řadí: [20]

1) Database engine - Jedná se o klíčovou službu umožňující ukládání, zpracování a zabezpečení dat. Tato komponenta slouží k základní práci s databázemi, tedy k vytváření samotných databází a jejich objektů (tabulek, pohledů, indexů, atd.), jejich správě a manipulaci s daty
2) Analysis Services - Tyto služby zahrnují nástroje sloužící pro Online Analytical Processing (OLAP) a Data Mining.
3) Integration Services - SQL Server Integration Services (SSIS) je sada grafických nástrojů a programovatelných objektů, které slouží k vytváření tzv. balíčků sloužících k extrakci, transformaci a načítání dat.
4) Reporting Services - Tato komponenta nabízí rozsáhlé možnosti v rámci reportingu na základě dat z relačních databází na serveru.

Jako v případě většiny databázových systémů pracuje MSSQL s vlastní implementací (dialektem) jazyka SQL, která se nazývá Transact-SQL (T-SQL). Všechny aplikace, které komunikují s instancí SQL serveru, tak činí zasíláním příkazů v této variantě dotazovacího jazyka SQL a to nezávisle na platformě, na které byla aplikace vyvíjena. [21]

3.3 Propojení databáze a aplikace

Kapitola 3.1 obsahovala základní charakteristiku .NET frameworku, zatímco náplň kapitoly 3.2 se týkala zejména relačních databází. V obou přecházejících kapitolách této práce bylo na obě zmíněná témata nahlíženo víceméně jako na oddělené části, avšak v praxi se ve většině případů jedná o neoddělitelné prvky, které spolu s dalšími nástroji tvoří kompletní aplikaci.

K zajištění fungování aplikace využívající databázového systému je nejprve nutné umožnit jejich vzájemnou komunikaci. V případě .NET frameworku se pro veškerý přístup k datům a práci s nimi používá knihovna tříd ADO.NET.

3.3.1 ADO.NET

.NET framework poskytuje mnoho jmenných prostorů, které umožňují práci s relačními databázemi. Ty se dohromady označují jako ADO.NET (ActiveX Data Object.NET). Tento soubor jmenných prostorů a knihoven poskytuje konzistentní přístup ke zdrojům dat, mezi které se řadí například instance MSSQL Serveru, XML soubory, nebo datové zdroje, umožňující přístup pomocí rozhraní OLE DB nebo ODBC. Aplikace vytvořené na platformě .NET využívající knihovny ADO.NET se tak mohou k těmto zdrojům dat připojit a následně s jejich daty manipulovat. [22]

Jednotlivé třídy ADO.NET se nacházejí v knihovně System.Data.dll a jsou integrované s třídami pro práci s XML, které lze najít pod System.Xml.dll. Knihovna ADO.NET sestává z několika jmenných prostorů (namespace), z nichž základní jsou rozděleny podle typu zdroje dat: [23]

1) System.Data.SqlClient – jmenný prostor pro práci s MSSQL
2) System.Data.Oracle – jmenný prostor pro práci s databázemi Oracle

3) System.Data.Odbc – jmenný prostor pro data přístupná pomocí poskytovatele ODBC
4) System.Data.OleDb – jmenný prostor pro data přístupná pomocí poskytovatele OLE DB

Pro manipulaci s daty lze v knihovně ADO.NET nalézt mnoho užitečných tříd, které mohou vývojářům značně usnadnit práci. Nejpoužívanější z nich jsou uvedeny v tabulce 5. Protože je tato práce zaměřena na relační databáze MSSQL, obsahuje tabulka pouze třídy sdílené, které jsou společné pro všechny zdroje dat nezávisle na poskytovateli, a dále třídy specifické pro práci s daty na SQL Serveru. Třídy pro ostatní poskytovatele jsou podobné jako třídy pro práci s MSSQL a jsou vždy odvozené od stejné třídy ze jmenného prostoru System.Data.Common implementující určité rozhraní, což v praxi znamená, že práce s nimi probíhá stejným způsobem. Například třída SqlCommand je stejně jako OracleCommand odvozena od třídy DbCommand, která implementuje rozhraní IDbCommand. [24]

Tabulka 5 - Hlavní třídy ADO.NET

Třída	Namespace	Popis
DataSet	System.Data	Třída sloužící k uchování dat z databáze v paměti i po odpojení od zdroje dat. Obsahuje kolekci objektů DataTables a jejich vazeb.
DataTable	System.Data	Třída reprezentující jednu databázovou tabulku načtenou do paměti počítače.
DataRow	System.Data	Třída reprezentující jeden řádek objektu DataTable.
DataColumn	System.Data	Tato třída obsahuje definici sloupce objektu DataTable (jeho datový typ a další vlastnosti).
SqlConnection	System.Data.SqlClient	Třída představující připojení k MSSQL databázi.
SqlCommand	System.Data.SqlClient	Tato třída reprezentuje T-SQL příkaz nebo proceduru, která má být spuštěna na databázi.
SqlDataAdapter	System.Data.SqlClient	Třída obsahující sadu příkazů a databázové připojení, která se využívá k naplnění DataSetu a propsání změn zpět do databáze.
SqlDataReader	System.Data.SqlClient	Třída sloužící pro připojené čtení dat. Čtení probíhá řádek po řádku a pouze směrem vpřed.
SqlTransaction	System.Data.SqlClient	Třída reprezentující T-SQL transakci.
SqlParameter	System.Data.SqlClient	Tato třída představuje parametr pro volání uložených procedur a její instance jsou obvykle předávány objektu SqlCommand.

Zdroje: [25] [26]

ADO.NET a jeho třídy umožňují pracovat s relačními databázemi třemi základními způsoby, přičemž každý z těchto principů má své výhody i nevýhody. Komparace jejich vlastností a přínosů je hlavní náplní praktické části této práce.

3.3.2 **Připojená databáze**

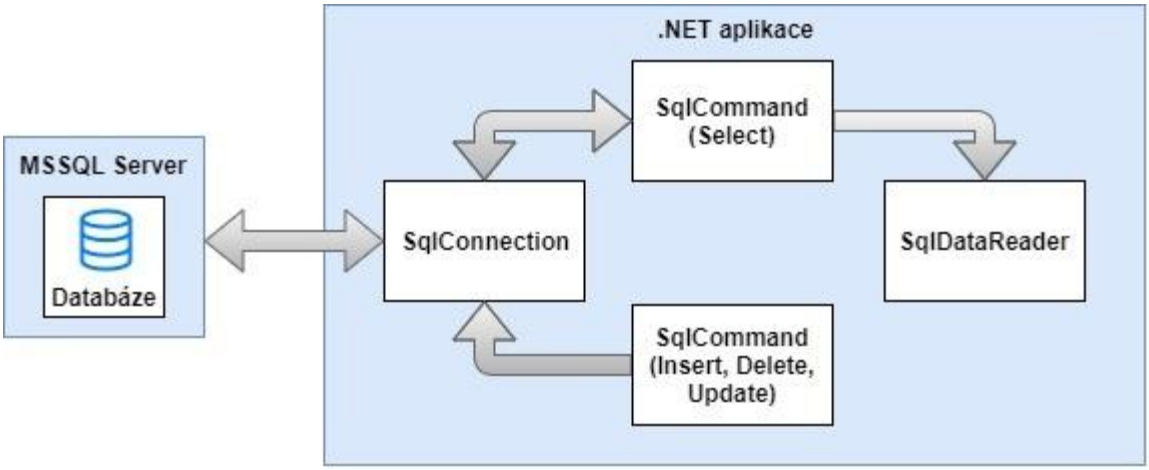
Prvním ze zmíněných způsobů je princip připojené databáze, který je založen zejména na práci s třídami SqlCommand a SqlDataReader. Komunikace s databází je v tomto případě prováděna „přímo“. Fungování připojené databáze lze znázornit pomocí diagramu zobrazeného na obrázku 2. Pro připojení k databázi se používá instance třídy SqlConnection, která obsahuje tzv. connection string (připojovací řetězec), což je textový řetězec skládající se například z názvu SQL serveru, názvu databáze, přihlašovacích údajů k serveru a případně dalších parametrů. Toto připojení je následně

předáno objektu SqlCommand, který ho využívá k přímému přístupu k databázi. SqlCommand dále obsahuje SQL příkaz a jeho případné parametry.

V závislosti na typu SQL příkazu se na objektu SqlCommand využívají tři základní metody pro jeho exekuci: [27]

1) Při provedení příkazů typu INSERT, DELETE, nebo UPDATE je SqlCommand pomocí SqlConnection odeslán na SQL server, který vrací celočíselnou hodnotu představující počet záznamů ovlivněných operací zpracovanou databázovým systémem. Tyto operace jsou zpravidla prováděny metodou ExecuteNonQuery na objektu SqlCommand.
2) Metoda ExecuteScalar se využívá zejména v případě dotazů obsahujících agregační funkce, tedy takových, které vracejí pouze jednu skalární hodnotu. Mezi tyto funkce se řadí například SUM pro součet hodnot, nebo COUNT pro počet vybraných záznamů. Tuto metodu lze využít i pro jiné typy příkazů, avšak v případě, že SQL server vrátí více záznamů o více sloupcích, je vždy vybrána pouze hodnota v prvním sloupci prvního řádku výběru.
3) Pokud je přes SqlCommand volán příkaz, který má z databáze vybrat určité záznamy (SELECT), je příkaz stejně jako v předchozím případě, předán pomocí instance třídy SqlCommand, která využívá SqlConnection, na SQL Server. Pro získání hodnot z databáze se pro tento typ příkazů využívá objektu třídy SqlDataReader, kterému se přiřadí výstup metody ExecuteReader spuštěné na instanci příkazu. Obsah objektu SqlDataReader, který se skládá z výsledku provedeného příkazu, lze následně číst pomocí metody Read. Ta vždy načte do paměti pouze jeden řádek výsledku SQL dotazu, jehož atributy slouží pouze ke čtení a není možné je modifikovat. Čtení řádků probíhá obvykle v iteracích a vždy lze číst pouze ve směru vpřed. [28]

Obrázek 2 - ADO.NET připojená databáze



Zdroj: autor na základě [27]

3.3.3 **Odpojená databáze**

Dalším způsobem práce s relačními databázemi je princip odpojené databáze, který se, jak již název napovídá, zakládá na práci s odpojenými daty, které jsou dočasně ukládána přímo do paměti počítače, kde jsou zpracována a nakonec opět synchronizována s databází na serveru. Hlavním prvkem zajišťující komunikaci mezi databází a .NET aplikací je v tomto případě objekt typu SqlDataAdapter. Sadu dat načtenou z databáze a uloženou do paměti, představuje v tomto kontextu třída DataSet. Jedná se v podstatě o kontejner obsahující kolekce objektů DataTable a DataRelation, který reflektuje skutečné databázové tabulky a vztahy mezi nimi. V případech interakce s jedinou relací lze využít pouze samostatný objekt DataTable.

Princip práce s odpojenou databází probíhá v závislosti na potřebě v jedné až ve dvou fázích. První z nich obnáší přenos dat z databáze do aplikace, tedy do objektu DataSet, popřípadě pouze DataTable. Pokud je účelem operace modifikace vybrané sady záznamů, přichází na řadu druhá fáze, při které dojde k úpravě dat na straně aplikace a následné odeslání změn zpět na databázi. Jednotlivé fáze zahrnují tyto kroky: [29]

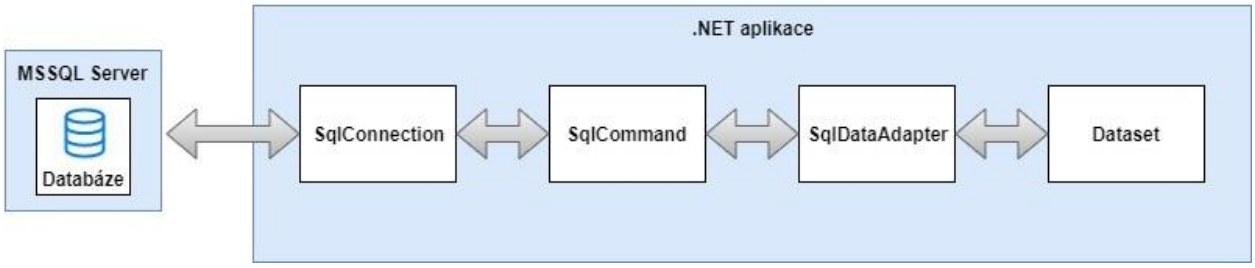
1) Přesun dat z databáze do paměti – stejně jako u principu připojené databáze je nejprve nutné vytvořit objekt SqlConnection obsahující connection string na cílovou databázi. Poté je tento objekt třeba přiřadit vlastnosti Connection na objektu SqlCommand. Tímto však podobnost těchto přístupů končí. Namísto SqlDataReaderu je nyní použita třída SqlDataAdapter, které je předán objekt SqlCommand jako vlastnost SelectCommand. Následně již může být objekt

DataSet naplněn lokální kopií selekce dat z databáze pomocí metody Fill na objektu SqlDataAdapter.

2) Přesun upravených dat zpět do databáze – po modifikaci/přidání/smazání dat v aplikaci, tedy v objektu DataSet (resp. DataTable), je přenos změn zpět na databázi opět řízen objektem SqlDataAdapter. Před odesláním změn zpět na databázi je zapotřebí specifikovat jednotlivé příkazy pro změnu dat. Objekt SqlDataAdapter obsahuje čtyři vlastnosti typu SqlCommand, pro každý typ operace (SELECT, INSERT, UPDATE a DELETE). Jak bylo zmíněno u popisu první fáze, vlastnost SelectCommand se využívá pouze pro získání dat z databáze. Ostatním vlastnostem je přiřazen odpovídající parametrizovaný SqlCommand, který je v závislosti na dané operaci, proveden pro každý upravený řádek. K následné aktualizaci databázových dat slouží metoda Update na objektu SqlDataAdapter. [28]

Celé fungování principu odpojené databáze zobrazuje ve zjednodušené podobě obrázek 3.

Obrázek 3 - ADO.NET odpojená databáze



Zdroj: autor na základě [29]

3.3.4 Objektově relační mapování (ORM)

Další technikou sloužící k práci s daty v relačních databázích je tzv. objektově relační mapování. Nástroje zajišťující tuto funkcionalitu vytváří v podstatě mezivrstvu mezi objektově orientovanou aplikací a relační databází, která zajišťuje reprezentaci dat uložených v databázových tabulkách jako objekty nativní pro daný programovací jazyk aplikace (např. C#). Zároveň úplně odstinují vývojáře od práce s jazykem SQL a zajišťují mapování jednotlivých databázových sloupců a jejich typů na vlastnosti daného objektu.

Toho ORM nástroje dosahují generováním vlastního SQL kódu na pozadí. Další výhodou těchto nástrojů je jejich schopnost zachovávat perzistenci dat v databázi, což znamená, že po úpravě entity v aplikaci se tato změna projeví i ve zdrojové relaci na databázovém serveru.

Automatické mapování tabulek na objekty a naopak zajišťuje zároveň rychlejší vývoj aplikací a méně napsaného kódu. Po napojení na databázi je obvykle vygenerováno schéma, dle kterého lze mapování pomocí grafického rozhraní modifikovat. Práce s jednotlivými instancemi objektů, tedy řádky relací, probíhá obvykle pomocí nástroje LINQ. [30]

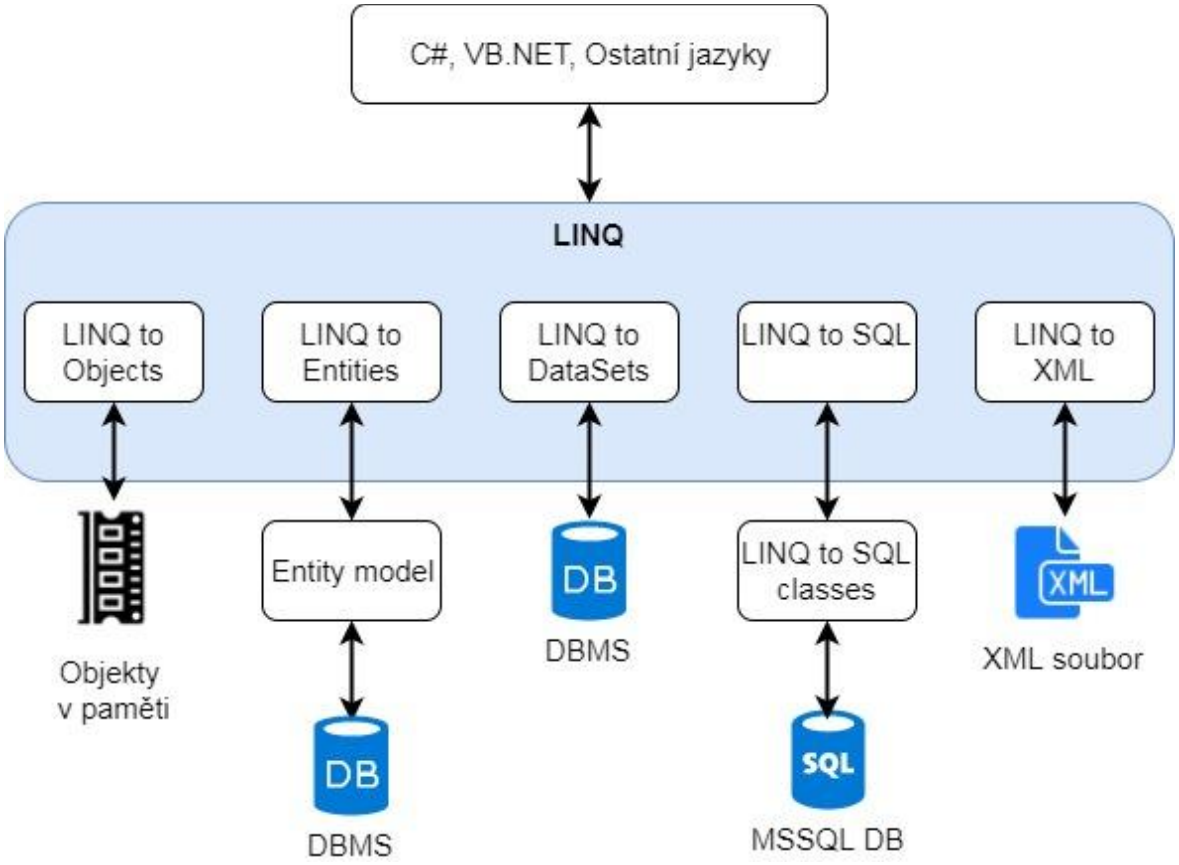
3.3.5 **LINQ**

.NET framework poskytuje pro komunikaci mezi aplikací a daty unifikované aplikačně programovací rozhraní (API) nazvané Language INtegrated Query (LINQ), které umožňuje pomocí jednotné syntaxe provádět dotazy nad různými zdroji dat přímo v jazycích podporovaných .NET frameworkem, tedy například C#, nebo VB.NET. LINQ byl představen spolu s .NET frameworkem verze 3.5. [31]

LINQ poskytuje sadu rozšiřujících (extension) metod, které jsou definované nad třídami Enumerable a Queryable. Zdroj dat, nad kterým jsou prováděny dotazy pomocí LINQ syntaxe musejí tedy implementovat rozhraní IEnumerable nebo IQueryable. [32]

LINQ podporuje několik poskytovatelů dat v závislosti na objektu, nad kterým jsou prováděny dotazy. Jednotliví poskytovatelé jsou zobrazeni na obrázku 4.

Obrázek 4 - Poskytovatelé LINQ



Zdroj: autor

Dotazy nad databází lze pomocí technologie LINQ psát dvěma různými způsoby s rozdílnou syntaxí. Pro vývojáře seznámené s jazykem SQL je obvykle snažší používat tzv. dotazovou syntaxi, protože je zmíněnému dotazovacímu jazyku v mnoha ohledech podobná. Naproti tomu poskytuje LINQ i možnost vyjadřovat dotazy pomocí metod a lambda výrazů. Funkčnost obou způsobů je identická, avšak některé dotazy, zejména ty, které využívají agregační funkce, lze formulovat pouze pomocí syntaxe využívající metod objektů. [33]

3.3.6 **LINQ to SQL**

LINQ pro SQL není pouze další implementací jazyka LINQ pro relační databáze, ale zahrnuje také nástroj pro ORM, se kterým umožňuje snadnou práci pomocí grafického rozhraní. V návrháři je zapotřebí pouze zvolit, které databázové objekty mají být namapovány, a následně se již ORM postará o vygenerování tzv. modelu entitních tříd, který reprezentuje jednotlivé databázové objekty a vazby mezi nimi. Vazby jsou vytvořeny

na základě primárních a cizích klíčů a díky této funkci již při práci s dotazy LINQ není třeba specifikovat vazby mezi tabulkami pomocí příkazu JOIN. LINQ to SQL byl prvním ORM nástrojem vytvořeným firmou Microsoft a je navržen pouze pro práci s relačními databázemi na SQL Serveru. [34]

Po propojení aplikace s databází a vygenerování entitního modelu lze již k datům přistupovat přímo v kódu jazyka C# a to pomocí parciální třídy DataContext, která je také vytvořena automaticky spolu s entitním modelem. Tato třída mimojiné obsahuje atributy, kterými jsou objekty reprezentující databázové relace. Pro potvrzení provedených změn nad třídou DataContext (mazání, přidávání, modifikace záznamů) se využívá její metoda SubmitChanges, která na základě provedených změn dynamicky vygeneruje SQL příkazy a odešle je zpět na SQL server, kde dojde k jejich exekuci. [35]

3.3.7 **Entity framework**

Dalším nástrojem pro objektově relační mapování v rámci frameworku .NET, který byl představen ve verzi .NET 3.5, je Entity framework. Tento ORM nástroj byl stejně jako LINQ to SQL vytvořen společností Microsoft, avšak od verze 6 byl publikován jako open source. [36]

Stejně jako v případě LINQ to SQL zajišťuje Entity framework možnost interakce s relačními databázemi za použití objektového modelu, který mapuje entity databáze přímo na objekty v .NET aplikaci. Zároveň, jak lze vidět na obrázku 4, Entity framework podporuje dotazování nad kolekcemi namapovaných objektů pomocí syntaxe LINQ, přičemž využívá jmenovitě poskytovatele LINQ to Entities, který tyto dotazy automaticky překládá do vygenerovaného SQL kódu, který je následně odeslán na SQL server. Entity framework také poskytuje grafické rozhraní pro úpravy entitního diagramu vygenerovaného na základě schématu databáze. [22]

Jednou z hlavních novinek oproti LINQ to SQL je podpora přístupu Code first, který umožňuje vygenerovat databázi na základě struktury aplikace, přičemž jednotlivé třídy představují vzor pro vygenerované databázové relace a atributy tříd reprezentují jednotlivé atributy a jejich datové typy. [37] Dalším podporovaným přístupem je Model first, který se používá k vytvoření modelu, na základě kterého budou vygenerovány jak třídy aplikace, tak i databáze. Posledním přístupem, při kterém se třídy aplikace mapují na základě již existující databáze, se nazývá Database first.

Entity framework navíc není limitován pouze komunikací s SQL Serverem, ale může pracovat i s relačními databázemi od jiných poskytovatelů jako například MySQL, Oracle nebo PostgreSQL.

4 Vlastní práce

4.1 Komparace výkonu

První fáze praktické části této bakalářské práce zahrnuje porovnání výkonu jednotlivých přístupů k práci s relačními databázemi v rámci frameworku .NET. Výkon bude měřen na základě rychlosti provedení základních databázových operací a to od inicializace připojení na databázi, až po dokončení exekuce příkazu, nebo případně načtení dat z databáze do paměti klienta. Měření bude probíhat nad základními tzv. CRUD operacemi, mezi které se řadí:

1) Create – vytváření nových záznamů, které je v rámci SQL prováděno příkazem INSERT
2) Retrieve – získávání záznamů, tedy selekce dat vybraných na základě zadaných kritérií pomocí SQL příkazu SELECT
3) Update – modifikace záznamů, kterou umožňuje SQL příkaz UPDATE
4) Delete – mazání vybraných řádků relací, které v rámci SQL zajišťuje příkaz DELETE

4.1.1 Příprava testovacího prostředí

Pro zajištění věrohodnosti výsledků měření bude vytvořen virtuální obraz 64-bitového operačního systému Windows 10, ve kterém budou testy prováděny. Virtuální obraz s čistou instalací systému bude použit z toho důvodu, aby výsledky měření nemohly být ovlivněny ostatními běžícími programy v počítači. Zároveň systém nebude mít přístup k internetu, takže bude zcela izolován od vnější sítě. Virtualizaci operačního systému zajistí program VirtualBox verze 5.2.6 od firmy Oracle. Již vytvořený obraz virtuálního systému byl stažen z oficiálních stránek Microsoftu, jenž umožňuje jeho oficiální využití na 90 dní zdarma. [38]

Jako databázový server byl použit Microsoft SQL Server 2017 v 64-bitové verzi Express, který je opět volně dostupný na webu společnosti Microsoft. [39] Při jeho instalaci na virtuální stroj byl vytvořen lokální SQL Server, na kterém bude testovací databáze umístěna. Jako vývojové prostředí pro vytvoření databáze bude využito SQL Server Management Studio 2017 verze 17.4. [40] Jako relační databázi, která bude sloužit

ke komparaci výkonnosti jednotlivých přístupů byla použita známá testovací databáze AdventureWorks, jejíž záloha je volně dostupná na GitHubu společnosti Microsoft [41]. Tato záloha byla následně obnovena pomocí funkce restore v rámci Management studia na lokální SQL server.

Souhrnná specifikace virtuálního systému a alokovaných HW zdrojů:

- Operační systém: Microsoft Windows 10 Enterprise (64-bit)
- Procesor: 2 jádra Intel Core i5-4690 3.50 GHz
- Operační paměť: 8192 MB
- Video paměť: 64 MB
- Nainstalovaný software: Microsoft SQL Server 2017, Microsoft Management Studio 2017, Microsoft Edge

4.1.2 **Vývoj aplikace**

Pro vývoj samotné testovací aplikace v .NET frameworku bylo využito vývojové prostředí Visual Studio 2017 od firmy Microsoft ve verzi Community. Celá aplikace je napsána za použití objektově orientovaného jazyka C# verze 7.0 a dotazovacího jazyka SQL, přesněji jeho dialektu T-SQL.

Aplikace slouží výhradně k testování rychlostí jednotlivých přístupů k práci s relačními databázemi a není vytvářena pro potřeby uživatelů, takže není nutné řešit pravidla user experience a vytvářet grafické rozhraní. Z tohoto důvodu byla vytvořena aplikace s jednoduchým konzolovým rozhraním, která jako uživatelské vstupy bere pouze typ zvoleného přístupu a typ operace. Následně provede daný příkaz pomocí zvoleného principu a vrátí celé číslo označující čas trvání jeho exekuce v milisekundách. Měření času zajišťuje instance třídy Stopwatch, která se nachází ve jmenném prostoru System.Diagnostics.

Aplikace se skládá z pěti základních tříd, z nichž čtyři třídy obsahují logiku jednotlivých databázových operací pro každý z přístupů. Jmenovitě se jedná o třídy:

1) PerformanceTestingConnDB – připojená databáze
2) PerformanceTestingDiscDB – odpojená databáze
3) PerformanceTestingEF – Entity Framework
4) PerformanceTestingLINQ – LINQ to SQL

Každá z těchto čtyř tříd implementuje rozhraní IPerformanceTesting, které se skládá pouze ze čtyř základních metod pro exekuci jednotlivých CRUD operací nad databází. Tyto metody jsou v každé ze tříd implementovány odlišně. Třída Program obsahuje pouze metodu Main, která je volána při spuštění aplikace a zajišťuje základní interakci uživatele s aplikací. Umožňuje tedy zvolit typ přístupu k práci a vytvoření instance dané třídy. Následně uživatel specifikuje typ databázové operace, která je provedena nad vytvořenou instancí třídy a aplikace zobrazí naměřenou délku trvání operace. Při startu aplikace je navíc vždy restartována SQL služba pro zajištění homogenních podmínek při následném měření výkonu. K tomuto účelo byla využita třída ServiceController, která se nachází ve jménném prostoru System.ServiceProcess.

V aplikaci byly zároveň vytvořeny dva modely entitních tříd. AWDataClasses.dbml slouží jako ORM vrstva pro práci s LINQ to SQL a AWEntityModel.edmx zase pro práci s Entity Frameworkem. Oba entitní modely obsahují pouze relace, které budou použity v databázových operacích.

V rámci inicializace objektů pro porovnávání výkonnosti je implementace mírně odlišná pro třídu PerformanceTestingEF, která slouží k měření rychlosti Entity Frameworku. V konstruktoru objektu, tedy ještě před začátkem měření, je zavolán tzv. „cold query“ dotaz nad databází. Ten je volán z toho důvodu, že Entity Framework provádí při prvním provedením databázového dotazu nad daným entitním modelem, několik operací, které mohou značně ovlivnit měření. Řadí se mezi ně například načítání metadat modelu, nebo jeho validace. Jedná se tedy o operace, které přímo nesouvisí s prováděním CRUD příkazů nad databází. Tento jednoduchý dotaz je prováděn nad relací (resp. objektem), která není zahrnuta v operacích, jejichž rychlosti mají být měřeny a to hlavně z toho důvodu, aby nedošlo k ukládání některých dat do cache paměti. Při dalších databázových příkazech se již jedná o tzv. „warm query“, jejichž rychlost již není ovlivněna vedlejšími procesy Entity Frameworku. [42]

Implementaci jednotlivých tříd, rozhraní a diagramy entitních modelů lze nalézt v přílohách této práce.

4.1.3 **Měření**

Měření rychlostí jednotlivých příkazů v rámci daných principů pro práci s relačními databázemi probíhalo na již specifikovaném virtuálním systému za použití zkompilované release verze testovací aplikace. Měření každé exekuce příkazu pro jednotlivé přístupy probíhalo vždy v deseti iteracích, mezi kterými byl virtuální systém restartován. Při spuštění aplikace navíc vždy došlo k restartu služby SQL serveru, což zajistilo smazání metadat databáze.

Jednotlivé výsledky měření jsou zachyceny v tabulce 6 a uváděny jsou v jednotkách milisekund. Měření nezachycuje pouze samotnou dobu exekuce SQL operace na SQL serveru, ale celkovou dobu provedení příkazu **včetně** souvisejících operací v rámci .NET aplikace. Tyto operace se pro každý princip liší a mohou zahrnovat například inicializaci a otevření databázového připojení, vytvoření LINQ dotazu, nebo vytvoření příkazu pro UPDATE operaci na instanci třídy SQLDataAdapter.

Z naměřených výsledků lze vypozorovat, že nejrychleji lze s relační databází pracovat pomocí principu připojené databáze, což je způsobeno tím, že na SQL server je v podstatě odeslán pouze textový řetězec daného příkazu a exekuce je následně vykonána samotným databázovým serverem. Jediným typem příkazu, pro který není připojená databáze nejrychlejším z principů, je příkaz SELECT. Ten je rychleji zpracován pomocí principu odpojené databáze, a to z toho důvodu, že do měření doby exekuce tohoto příkazu byla zahrnuta i doba načtení dat do paměti počítače, přičemž zde si vede lépa třída SQLDataAdapter než SQLDataReader, která slouží primárně k sekvenčnímu čtení dat.

Na druhé příčce se v rámci celkové průměrné rychlosti operací umístil přístup odpojené databáze, který za připojenou databází ve většině případů zaostává pouze o jednotky milisekund. Oba ORM nástroje se ukázaly být znatelně pomalejší, avšak lépe z nich si vedl LINQ to SQL, který byl ze všech přístupů třetí nejrychlejší, zatímco Entity Framework se ukázal jako vůbec nejpomalejší.

Tabulka 6- Výsledky měření

Pořadí měření	LINQ to SQL				Entity Framework				Připojená databáze				Odpojená databáze			
	UPD	INS	DEL	SEL	UPD	INS	DEL	SEL	UPD	INS	DEL	SEL	UPD	INS	DEL	SEL
1	398	194	223	182	371	244	287	269	125	129	152	158	181	134	145	144
2	407	156	201	175	344	214	274	215	123	138	148	169	190	132	161	147
3	354	170	194	183	356	212	272	218	111	140	137	155	201	147	151	146
4	349	171	240	177	359	217	294	221	146	132	145	152	183	138	162	141
5	382	215	188	194	365	219	289	223	128	129	141	151	195	144	171	121
6	370	157	211	181	360	238	283	217	110	124	150	131	200	149	149	150
7	386	174	193	170	361	223	283	219	138	122	144	169	178	130	161	157
8	375	200	222	179	354	216	286	221	115	136	141	164	195	146	149	153
9	387	155	202	197	346	225	289	230	131	116	143	154	191	137	150	153
10	366	147	212	191	376	183	300	245	114	128	149	145	197	128	154	140
Průměrná doba exekuce	**377,4**	**173,9**	**208,6**	**182,9**	**359,2**	**219,1**	**285,7**	**227,8**	**124,1**	**129,4**	**145,0**	**154,8**	**191,1**	**138,5**	**155,3**	**145,2**
Celkový průměr	**235,70**				**272,95**				**138,33**				**157,53**			

4.1.4 Komparace vlastností

Dalšími kritérii pro porovnání jednotlivých přístupů jsou jejich individuální vlastnosti a možnosti, které nabízejí. Všechny vlastnosti kromě výkonu byly ohodnoceny na stupnici od 1 až 10, kde 10 představuje nejlepší hodnocení. Tyto případy hodnocení jsou subjektivní a snažil jsem se vyjádřit svůj pohled na danou implementaci, na základě praktických zkušeností získaných při práci jednotlivými přístupy. Mezi parametry hodnocení byly zařazeny:

1) Výkon – hodnoceny výsledky předchozích měření
2) Čistota kódu – tato vlastnost je hodnocena zejména z hlediska čitelnosti kódu a možnosti odchycení případných chyb již v rámci kompilace
3) Potřebná úroveň znalostí – míra znalostí nutná ke správnému pochopení a implementaci daného přístupu. Nižší hodnota značí vyšší úroveň, tedy vyší náročnost a tedy horší hodnocení.
4) Důraz na OOP – v jaké míře klade daný princip důraz na objektově orientované paradigma

Ohodnocené vlastnosti jsou zachyceny v tabulce 7.

Tabulka 7 - Ohodnocené vlastnosti přístupů

	Připojená databáze	Odpojená databáze	LINQ to SQL	Entity Framework
Výkon	138,33	157,53	235,70	272,95
Čistota kódu	3	5	9	9
Úroveň znalostí	7	6	4	3
Důraz na OOP	4	5	8	9

4.2 Vícekriteriální analýza variant

Na základě získaných výsledků hodnocení jednotlivých principů práce s databázemi bude vybrán ten nejvhodnější z nich pro implementaci aplikačního požadavku a to za využití vícekriteriální analýzy variant, přesněji metody AHP. Požadavek na aplikaci je pouze teoretický a velmi zjednodušený. Obsahuje pouze informace nutné pro

volbu principu práce s databází a nevyskytují se v něm veškeré detaily umožňující kompletní návrh aplikace.

V rámci teoretického aplikačního požadavku hledá klient nové řešení skladové evidence svého maloobchodu s oblečením. Stávající systém je implementován pomocí zastaralých technologií a je třeba ho kompletně nahradit. Klient aktuálně využívá databázový server MSSQL 2017, který bude zachován. V budoucnu je plánováno rozšíření aplikace o pokladní systém s vlastní implementací elektronické evidence tržeb. Bude se jednat o desktopovou aplikaci postavenou na .NET frameworku pomocí technologie WPF (Windows Presentation Foundation).

4.2.1 **Kritéria**

Jako kritéria slouží výsledky předchozích hodnocení, tedy výkon (V), čistota kódu (ČK), úroveň znalostí (ÚZ) a důraz na OOP (DOOP). Ve všech případech se jedná o kritéria maximalizační. Ke stanovení vah kritérií jsem použil Saatyho metodu pro kardinální srovnání kritérií. Kritéria byla porovnávána na základě zadání aplikace. To znamená, že v rámci maloobchodního skladu nejsou například očekávány velké datové toky a proto výkonnost není nejdůležitějším faktorem a naproti tomu třeba důraz na OOP je z důvodu budoucího rozvoje aplikace znatelně důležitější. Výpočet normovaných vah kritérií je zobrazen v tabulce 8.

Tabulka 8 - Saatyho matice pro váhy kritérií

Kritérium	V	PDBS	ČK	ÚZ	DOOP	Geomean	Norm. váha
V	1	7	5	1/3	1/5	1,1847	0,1618
ČK	1/5	5	1	1/5	1/7	0,4911	0,0671
ÚZ	3	5	5	1	1/3	1,9037	0,2600
DOOP	5	7	7	3	1	3,7433	0,5112
Suma						7,3228	1

4.2.2 Výpočet

Pro výpočet byla zvolena metoda AHP (Analytický Hierarchický Proces) navržená profesorem Saatym. Po stanovení vah kritérií se porovnávají jejich hodnoty navzájem mezi sebou v hierarchické struktuře. Komparace hodnot jednotlivých kritérií je znázorněna v tabulkách 9 až 12.

Tabulka 9 - Saatyho matice pro kritérium výkon

Výkon	Připojená DB	Odpojená DB	LINQ to SQL	Entity Framework	Geomean	Norm. váha
Připojená DB	1	3	7	9	3,7078	0,5824
Odpojená DB	1/3	1	5	7	1,8481	0,2903
LINQ to SQL	1/7	1/5	1	3	0,5411	0,0850
Entity Framework	1/9	1/7	1/3	1	0,2697	0,0424
Suma					6,3667	1

Tabulka 10 - Saatyho matice pro kritérium čistota kódu

Čistota kódu	Připojená DB	Odpojená DB	LINQ to SQL	Entity Framework	Geomean	Norm. váha
Připojená DB	1	1/3	1/7	1/7	0,2872	0,0500
Odpojená DB	3	1	1/5	1/5	0,5886	0,1025
LINQ to SQL	7	5	1	1	2,4323	0,4237
Entity Framework	7	5	1	1	2,4323	0,4237
Suma					5,7404	1

Tabulka 11 - Saatyho matice pro kritérium úroveň znalostí

Úroveň znalostí	Připojená DB	Odpojená DB	LINQ to SQL	Entity Framework	Geomean	Norm. váha
Připojená DB	1	3	5	6	3,0801	0,5594
Odpojená DB	1/3	1	3	4	1,4142	0,2568
LINQ to SQL	1/5	1/3	1	3	0,6687	0,1214
Entity Framework	1/6	1/4	1/3	1	0,3433	0,0623
Suma					5,5063	1

Tabulka 12 - Saatyho matice pro kritérium důraz na OOP

Důraz na OOP	Připojená DB	Odpojená DB	LINQ to SQL	Entity Framework	Geomean	Norm. váha
Připojená DB	1	1/3	1/7	1/9	0,2697	0,0424
Odpojená DB	3	1	1/5	1/7	0,5411	0,0850
LINQ to SQL	7	5	1	1/3	1,8481	0,2903
Entity Framework	9	7	3	1	3,7078	0,5824
Suma					6,3667	1

Na základě porovnání jednotlivých kritérií a jejich vah lze určit pořadí priorit variant. Toho je dosaženo součtem jednotlivých hodnocení kritérií vynásobených jejich váhou. Vypočítané hodnoty následně vyjadřují priority jednotlivých variant, přičemž čím větší je priorita, tím je varianta optimálnější. Výsledky jsou zachyceny v tabulce 13.

Tabulka 13 - Pořadí variant podle AHP

	Výkon	Čistota kódu	Úroveň znalostí	Důraz na OOP	Součet hodnocení	Pořadí
Připojená DB	0,5824	0,0500	0,5594	0,0424	0,2647062	2
Odpojená DB	0,2903	0,1025	0,2568	0,0850	0,1640683	4
LINQ to SQL	0,0850	0,4237	0,1214	0,2903	0,2221486	3
Entity Framework	0,0424	0,4237	0,0623	0,5824	0,3492115	1
Váhy kritérií	0,1618	0,0671	0,2600	0,5112		

5 Zhodnocení

Z výsledků vyobrazených v tabulce 13 lze vyvodit, že v rámci daného aplikačního požadavku se jako jednoznačně nejvhodnější (tedy optimální) varianta přístupu k práci s relační databází jeví Entity Framework, kterému byla pomocí vícekriteriální analýzy vypočtena priorita 0,349211.

Je třeba zdůraznit, že výsledky jsou z velké části ovlivněny mým vlastním subjektivním pohledem při srovnávání jednotlivých přístupů a kritérií. Při určování hodnot kritérií (kromě výkonu) jsem se snažil pomocí bodové stupnice vyjádřit svůj pohled na implementaci dané vlastnosti, který jsem získal při zpracovávání rešeršní a praktické části této práce a jiný vývojář by jejich hodnoty mohl stanovit odlišně. Při párovém srovnání jednotlivých kritérií, které pomohlo určit jejich individuální váhy, jsem se snažil co nejlépe reflektovat faktory konkrétního aplikačního požadavku. V případě odlišného zadání by váhy mohly být opět stanoveny jiným způsobem. Například v případě aplikace, jejímž účelem by bylo zpracování velkého množství dat, by byl kladen značně vyšší důraz na výkonnost zvoleného přístupu. Získané výsledky jsou tedy aplikovatelné pouze v případě tohoto specifického zadání požadavku a v žádném případě neznačí, že by ostatní přístupy v praxi neměly být používány.

6 Závěr

Hlavním cílem této bakalářské práce byla komparace jednotlivých přístupů pro práci s relačními databázemi v .NET frameworku. V rámci teoretické části práce byl největší důraz přikládán sběru informací a následnou charakteristiku jednotlivých pojmů a technologií vztahujících se k tématu práce. Podkapitola 3.1 je zaměřena na samotný frameworku .NET, jeho funkcionality, historii a komponenty, ze kterých se skládá. Obsah podkapitoly 3.2 se věnuje databázovým systémům. Dále jsou v této části interpretovány základní principy fungování relačních databází, jazyk SQL a objekty, se kterými se v rámci relačních databází pracuje.

Propojením těchto dvou oblastí se práce zabývá v kapitole 3.3, ve které jsou charakterizovány jednotlivé přístupy k práci s relačními databázemi pomocí tříd ADO.NET. Charakterizovány byly principy odpojené a připojené databáze a objektově relační mapování pomocí LINQ to SQL a Entity Framework. U každého z přístupů jsou zároveň uvedeny základní třídy, které jsou v jeho kontextu využívány a také způsob, jakým se s nimi pracuje.

V praktické části je již práce orientována na samotnou komparaci těchto přístupů. Nejprve bylo připraveno testovací prostředí, které představoval virtuální systém Microsoft Windows 10, na který byl nainstalován SQL Server 2017. Zde byla následně vytvořena lokální instance databáze AdventureWorks. Poté byla naprogramována jednoduchá konzolová aplikace v jazyce C#, jejímž účelem bylo měření rychlosti provádění jednotlivých CRUD operací nad databází a to individuálně pro každý přístup. V měření si nejlépe vedl princip připojené databáze. Dále byla vybrána další tři kritéria pro hodnocení přístupů, přičemž každé z nich bylo ohodnoceno body na stupnici 1 až 10. Hodnoty byly stanoveny na základě poznatků získaných rešerší zdrojů a zkušeností dosažených při samotném vývoji aplikace.

Následně bylo definováno teoretické zadání aplikačního požadavku, pro nějž měl být vybrán nejvhodnější přístup k práci s relační databází. Poté byly vypočteny váhy jednotlivých kritérií pomocí Saatyho matice pro párové porovnání, které byly následně použity při výběru nejvhodnější varianty pomocí metody AHP. Při daném stanovení vah kritérií a komparaci jejich hodnot se optimální variantou pro aplikaci ukázal být Entity framework.

Na závěr je třeba ještě jednou zdůraznit, že mnou dosažené výsledky komparace pomocí vícekriteriální analýzy variant se vztahují pouze k danému zadání aplikačního požadavku a každý z daných přístupů má v praxi své reálné využití. Při výběru vhodného přístupu tedy vždy záleží zejména na kontextu aplikace a požadavcích na její funkcionality a v praktické části této práce jsem chtěl zejména poukázat na případný proces jeho výběru.

7 Seznam použitých zdrojů

[1] Get started with the .NET Framework. Microsoft Developer Network [online]. [cit. 2017-09-14]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/get-started/index

[2] Common Language Runtime (CLR). Microsoft Docs [online]. [cit. 2017-09-20]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/clr

[3] .NET Class Library Overview. Microsoft Docs [online]. [cit. 2017-09-25]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/class-library-overview

[4] Visual Studio – sada IDE. Microsoft Developer Network [online]. [cit. 2017-09-28]. Dostupné z: https://msdn.microsoft.com/library/dn762121(v=vs.140).aspx

[5] .NET Framework Versions and Dependencies. Microsoft Docs [online]. [cit. 2017-11-08]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/migration-guide/versions-and-dependencies

[6] Introduction to the C# Language and the .NET Framework. Microsoft Docs [online]. [cit. 2018-02-27]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

[7] Developer Survey Results 2017. Stack Overflow [online]. [cit. 2017-11-28]. Dostupné z: https://insights.stackoverflow.com/survey/2017

[8] What's New in the Visual Basic Language for Visual Basic 6.0 Users. Microsoft Developer Network [online]. [cit. 2017-10-20]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms172618.aspx

[9] F# Guide. Microsoft Docs [online]. [cit. 2017-10-02]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/fsharp/

[10] DBMS Database Models. Www.studytonight.com [online]. [cit. 2017-10-02]. Dostupné z: https://www.studytonight.com/dbms/database-model.php

[11] LAKE, Peter a Paul CROWTHER. Concise Guide to Databases [online]. London: Springer London, 2013 [cit. 2017-10-15]. Undergraduate Topics in Computer Science. ISBN 978-1-4471-5600-0.

[12] Data Types (Transact-SQL). Microsoft TechNet [online]. [cit. 2017-11-08]. Dostupné z: https://technet.microsoft.com/en-us/library/ms187752(v=sql.105).aspx

[13] SYVERSON, Bryan a Joel MURACH. Murach's SQL server 2016 for developers: training and reference. Fresno, CA: Mike Murach & Associates, 2016. ISBN 978-1-890774-96-7.

[14] OPPEL, Andrew J. Databases demystified. 2nd ed. New York: McGraw-Hill, c2011. ISBN 978-0071747998.

[15] POWELL, Gavin. Beginning database design. Indianapolis, IN: Wiley, c2006. ISBN 978-0764574900.

[16] HERNANDEZ, Michael J. a John L. VIESCAS. SQL queries for mere mortals: a hands-on guide to data manipulation in SQL. Third edition. Upper Saddle River, NJ: Addison-Wesley, 2014. ISBN 978-0321992475.

[17] RICARDO, Catherine M. Databases illuminated. 2nd ed. Sudbury, MA: Jones & Bartlett Learning, c2012. ISBN 978-1449606008.

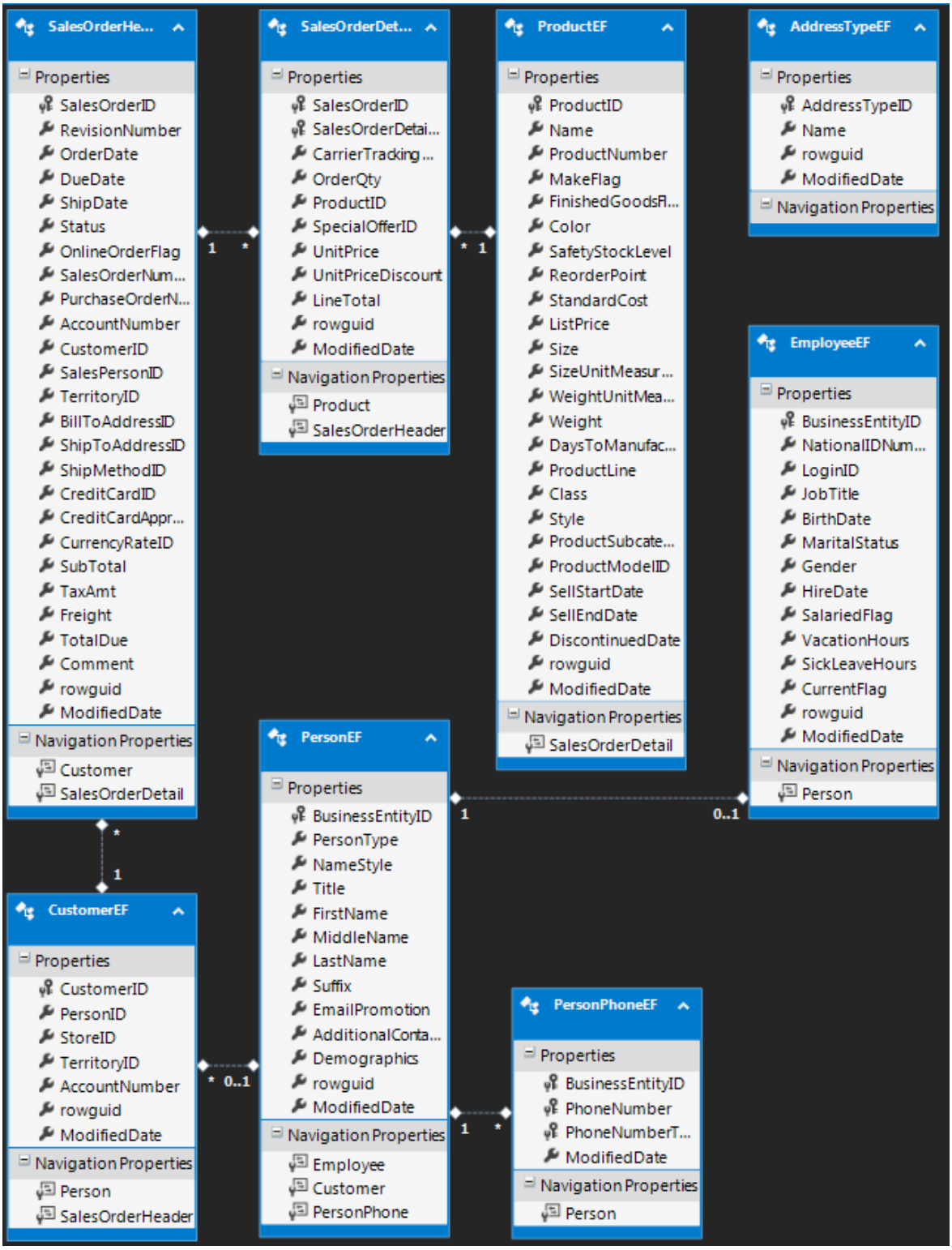
[18] ELMASRI, Ramez a Sham NAVATHE. Fundamentals of database systems. Seventh edition. Hoboken, NJ: Pearson, 2016. ISBN 978-0133970777.

[19] SQL Server 2017 available on Linux. Microsoft [online]. [cit. 2017-11-08]. Dostupné z: https://www.microsoft.com/en-us/sql-server/sql-server-2017
[20] SQL Server Overview. Microsoft TechNet [online]. [cit. 2017-11-08]. Dostupné z: https://technet.microsoft.com/en-us/library/ms166352(v=sql.90).aspx
[21] Transact-SQL Reference (Transact-SQL). Microsoft TechNet [online]. [cit. 2017-11-08]. Dostupné z: https://technet.microsoft.com/en-us/library/ms189826(v=sql.90).aspx
[22] TROELSEN, Andrew a Philip JAPIKSE. C# 6.0 and the .NET 4.6 Framework. Seventh Edition. New York: apress, 2015. ISBN 978-1484213339.
[23] NAGEL, Christian. C# 2008: programujeme profesionálně. Brno: Computer Press, 2009. Programujeme profesionálně. ISBN 978-80-251-2401-7.
[24] ADO.NET Overview. Microsoft Docs [online]. [cit. 2018-01-02]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview
[25] System.Data.SqlClient Namespace. Microsoft Developer Network [online]. [cit. 2017-11-05]. Dostupné z: https://msdn.microsoft.com/en-us/library/system.data.sqlclient(v=vs.110).aspx
[26] System.Data Namespace. Microsoft Developer Network [online]. [cit. 2017-11-20]. Dostupné z: https://msdn.microsoft.com/en-us/library/system.data(v=vs.110).aspx
[27] BOEHM, Anne. a Ged. MEAD. Murach's ADO.NET 4 database programming with C# 2010. 4th ed. Fresno, CA: Mike Murach & Associates, c2011. ISBN 978-1-890774-63-9.
[28] SKINNER Julian, Bipin JOSHI, Donny MACK, et al. Professional ADO.NET Programming. Apress, 2001. ISBN 978-1861005274.
[29] PATRICK, Tim. Microsoft ADO.NET 4 step by step. Sebastopol, Calif.: O'Reilly Media, c2010. ISBN 978-0735638884.
[30] BARSKIY, Sergery. Code-First Development with Entity Framework. BIRMINGHAM: Packt Publishing, 2015. ISBN 978-1784396275.
[31] Introduction to LINQ (C#). Microsoft Docs [online]. [cit. 2017-12-07]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq
[32] BOEHM, Anne. Murach's ADO.NET 3.5: LINQ and the Entity Framework with C# 2008. Fresno, California: Mike Murach & Associates, 2009. ISBN 978-1890774530.
[33] Query Syntax and Method Syntax in LINQ (C#). Microsoft Docs [online]. [cit. 2017-12-20]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/query-syntax-and-method-syntax-in-linq
[34] JENNINGS, Roger. Professional ADO.NET 3.5 with LINQ and the entity framework. Indianapolis, IN: Wiley Pub., c2009. Wrox professional guides. ISBN 978-0-470-18261-1.
[35] MEHTA, Vijay P. Pro LINQ object relational mapping in C# 2008. Berkeley, CA: Apress, c2008. ISBN 978-1-59059-965-5.
[36] Microsoft open-sources Entity Framework. InfoWorld [online]. [cit. 2017-12-28]. Dostupné z: https://www.infoworld.com/article/2617690/microsoft-net/microsoft-open-sources-entity-framework.html
[37] Entity Framework Code First Conventions. Microsoft Developer Network [online]. [cit. 2017-12-28]. Dostupné z: https://msdn.microsoft.com/en-us/library/jj679962(v=vs.113).aspx
[38] Download virtual machines. Microsoft Developer [online]. [cit. 2018-01-09]. Dostupné z: https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/

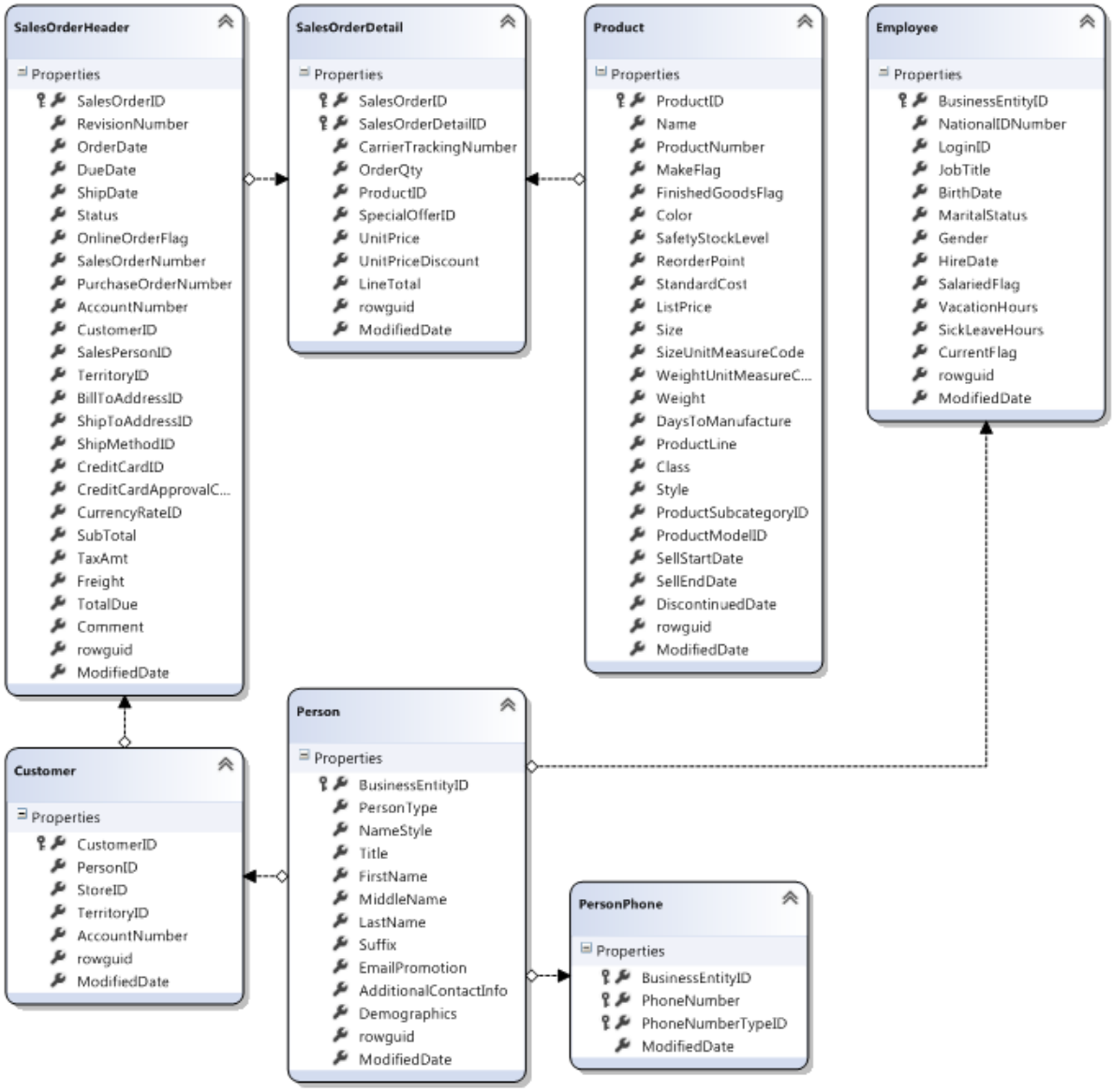
[39] Microsoft SQL Server 2017 Express. Microsoft [online]. [cit. 2018-01-15]. Dostupné z: https://www.microsoft.com/en-us/download/details.aspx?id=55994
[40] Download SQL Server Management Studio (SSMS). Microsoft Docs [online]. [cit. 2018-01-18]. Dostupné z: https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms
[41] AdventureWorks sample databases. GitHub [online]. [cit. 2018-01-28]. Dostupné z: https://github.com/Microsoft/sql-server-samples/releases/tag/adventureworks
[42] Performance Considerations for EF 4, 5, and 6. Microsoft Developer Network [online]. [cit. 2018-02-08]. Dostupné z: https://msdn.microsoft.com/en-us/library/hh949853(v=vs.113).aspx

8 Přílohy

8.1 Entity framework diagram



8.2 LINQ to SQL diagram



8.3 Třída Program.cs

8.3.1 Metoda Main

```
public static void Main(string[] args)
{
    bool validEntry = false;
    int operation = 0;
    long elapsedTime = 0;
    IPerformanceTesting perfTest = null;
    RestartWindowsService("MSSQL$SQLEXPRESS");

    while (!validEntry) {
        Console.WriteLine("Typ přístupu: \n1 - DiscDB\n2 - ConnDB\n3 - EF\n4 - LTS");
        validEntry = Int32.TryParse(Console.ReadLine(),out operation);

        switch (operation)
        {
            case 1: perfTest = new PerformanceTestingDiscDB();
                break;
            case 2: perfTest = new PerformanceTestingConnDB();
                break;
            case 3: perfTest = new PerformanceTestingEF();
                break;
            case 4: perfTest = new PerformanceTestingLINQ();
                break;
            default: validEntry = false;
                break;
        }
    }
    validEntry = false;

    while (!validEntry) {
        Console.WriteLine("Typ operace: \n1 - Select\n2 - Insert\n3 - Delete\n4 -
Update");
        validEntry = Int32.TryParse(Console.ReadLine(), out operation);
        try {
            switch (operation) {
                case 1: elapsedTime = perfTest.ExecSelectQuery();
                    break;
                case 2: elapsedTime = perfTest.ExecInsertQuery();
                    break;
                case 3: elapsedTime = perfTest.ExecDeleteQuery();
                    break;
                case 4: elapsedTime = perfTest.ExecUpdateQuery();
                    break;
                default: validEntry = false;
                    break;
            }
        }
        catch(Exception e) {
            Console.WriteLine(e.Message);
        }
        Console.WriteLine(elapsedTime.ToString());
    }
    Console.ReadKey();
}
```

8.3.2 Metoda RestartWindowsService

```
private static void RestartWindowsService(string serviceName)
{
    ServiceController serviceController = new ServiceController(serviceName);
    try
    {
        if ((serviceController.Status.Equals(ServiceControllerStatus.Running))
            ||
(serviceController.Status.Equals(ServiceControllerStatus.StartPending)))
        {
            serviceController.Stop();
        }
        serviceController.WaitForStatus(ServiceControllerStatus.Stopped);
        serviceController.Start();
        serviceController.WaitForStatus(ServiceControllerStatus.Running);
        Console.WriteLine("SQL service restarted");
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

8.4 Třída PerformanceTestingConnDB

8.4.1 Proměnné a konstruktor

```
private string connStr;
private Stopwatch sw = new Stopwatch();

public PerformanceTestingConnDB()
{
    connStr = @"Data Source=MSEDGEWIN10;Initial Catalog=AdventureWorks2017;Persist
                Security Info=True;User ID=JWU;Password=Heslo123";
}
```

8.4.2 **Metoda ExecSelectQuery**

```
public long ExecSelectQuery()
{
    sw.Start();

    using (SqlConnection conn = new SqlConnection(connStr))
    {
        SqlCommand comm = new SqlCommand();
        comm.Connection = conn;
        comm.CommandText = @"select p.Title, p.FirstName, p.MiddleName, p.LastName
                             from Person.Person p
                             join Sales.Customer c on p.BusinessEntityID = c.PersonID
                             join Sales.SalesOrderHeader soh on c.CustomerID =
                                  soh.CustomerID
                             join Sales.SalesOrderDetail sod on soh.SalesOrderID =
                                  sod.SalesOrderID
                             join Production.Product pr on sod.ProductID =
                                  pr.ProductID
                             where pr.Name = 'Front Brakes'
                             order by p.LastName";
        conn.Open();
        SqlDataReader dr = comm.ExecuteReader();
        DataTable dt = new DataTable();
        dt.Load(dr);
        dr.Close();
    }

    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.4.3 **Metoda ExecUpdateQuery**

```
public long ExecUpdateQuery()
{
    sw.Start();

    using (SqlConnection conn = new SqlConnection(connStr))
    {
        SqlCommand comm = new SqlCommand();
        comm.Connection = conn;
        comm.CommandText = @"update HumanResources.Employee
                             set ModifiedDate = GETDATE(), VacationHours =
                                  VacationHours + 1
                             where HireDate > '2009-01-01'
                             or (JobTitle like 'Marketing%'
                             and YEAR(BirthDate) between 1977 and 1984)";
        conn.Open();
        comm.ExecuteNonQuery();
    }

    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.4.4 **Metoda ExecDeleteQuery**

```
public long ExecDeleteQuery()
{
    sw.Start();

    using (SqlConnection conn = new SqlConnection(connStr))
    {
        SqlCommand comm = new SqlCommand();
        comm.Connection = conn;
        comm.CommandText = @"delete from Person.PersonPhone
                             where PhoneNumber like '%-222-%'
                             or (PhoneNumber like '111-%-333'
                             and PhoneNumberTypeID = 1)
                             or BusinessEntityID = 1399";
        conn.Open();
        comm.ExecuteNonQuery();
    }

    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.4.5 **Metoda ExecInsertQuery**

```
public long ExecInsertQuery()
{
    sw.Start();

    using (SqlConnection conn = new SqlConnection(connStr))
    {
        SqlCommand comm = new SqlCommand();
        comm.Connection = conn;
        comm.CommandText = @"insert into Person.PersonPhone
                             (BusinessEntityID, PhoneNumber, PhoneNumberTypeID,
                             ModifiedDate)
                             values
                                (1704, '963-222-223', 2, GETDATE()),
                                (1399, '725-012-638', 1, GETDATE()),
                                (1399, '89-993-181', 2, GETDATE()),
                                (7225, '111-984-333', 1, GETDATE()),
                                (8911, '111-412-333', 1, GETDATE()),
                                (17734, '96-222-814', 1, GETDATE()),
                                (17733, '689-222-13', 2, GETDATE()),
                                (20771, '111-95-333', 1, GETDATE()),
                                (19624, '321-222-891', 2, GETDATE()),
                                (9576, '111-222-333', 1, GETDATE())";
        conn.Open();
        comm.ExecuteNonQuery();
    }

    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.5 Třída PerformanceTestingDiscDB

8.5.1 Proměnné a konstruktor

```
private string connStr;
private Stopwatch sw = new Stopwatch();

public PerformanceTestingConnDB()
{
    connStr = @"Data Source=MSEDGEWIN10;Initial Catalog=AdventureWorks2017;Persist
                Security Info=True;User ID=JWU;Password=Heslo123";
}
```

8.5.2 Metoda ExecSelectQuery

```
public long ExecSelectQuery()
{
    sw.Start();

    using (SqlConnection conn = new SqlConnection(connStr))
    {
        conn.Open();
        string sqlStatement = @"select p.Title, p.FirstName, p.MiddleName, p.LastName
                                from Person.Person p
                                join Sales.Customer c on p.BusinessEntityID =
                                       c.PersonID
                                join Sales.SalesOrderHeader soh on c.CustomerID =
                                       soh.CustomerID
                                join Sales.SalesOrderDetail sod on soh.SalesOrderID =
                                       sod.SalesOrderID
                                join Production.Product pr on sod.ProductID =
                                       pr.ProductID
                                where pr.Name = 'Front Brakes'
                                order by p.LastName";
        SqlDataAdapter da = new SqlDataAdapter(new SqlCommand(sqlStatement, conn));
        DataTable dt = new DataTable();
        da.Fill(dt);
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.5.3 **Metoda ExecUpdateQuery**

```
public long ExecUpdateQuery()
{
    sw.Start();

    using (SqlConnection conn = new SqlConnection(connStr))
    {
        conn.Open();
        string sqlStatement = @"select *
                                from HumanResources.Employee
                                where HireDate > '2009-01-01'
                                or (JobTitle like 'Marketing%'
                                and YEAR(BirthDate) between 1977 and 1984)";

        SqlDataAdapter da = new SqlDataAdapter(new SqlCommand(sqlStatement,conn));

        sqlStatement = @"update HumanResources.Employee
                           set ModifiedDate = @ModifiedDate, VacationHours =
                             @VacationHours
                           where BusinessEntityID = @BusinessEntityID";
        SqlCommand updateCommand = new SqlCommand(sqlStatement, conn);
        updateCommand.Parameters.Add(new SqlParameter("@BusinessEntityID",
SqlDbType.Int, 0, "BusinessEntityID"));
        updateCommand.Parameters.Add(new SqlParameter("@ModifiedDate",
SqlDbType.DateTime, 0, "ModifiedDate"));
        updateCommand.Parameters.Add(new SqlParameter("@VacationHours",
SqlDbType.SmallInt, 0, "VacationHours"));

        da.UpdateCommand = updateCommand;
        DataTable dt = new DataTable();
        da.Fill(dt);

        foreach (DataRow row in dt.Rows)
        {
            row["ModifiedDate"] = DateTime.Now;
            row["VacationHours"] = ((Int16)row["VacationHours"]) + 1;
        }

        da.Update(dt);
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.5.4 **Metoda ExecDeleteQuery**

```
public long ExecDeleteQuery()
{
    sw.Start();
    using (SqlConnection conn = new SqlConnection(connStr))
    {
        conn.Open();
        string sqlStatement = @"select * from Person.PersonPhone
                                where PhoneNumber like '%-222-%'
                                or (PhoneNumber like '111-%-333'
                                and PhoneNumberTypeID = 1)
                                or BusinessEntityID = 1399";
        SqlDataAdapter da = new SqlDataAdapter(new SqlCommand(sqlStatement, conn));

        sqlStatement = @"delete from Person.PersonPhone
                         where BusinessEntityID = @BusinessEntityID
                         and PhoneNumber = @PhoneNumber
                         and PhoneNumberTypeID = @PhoneNumberTypeID";
        SqlCommand deleteCommand = new SqlCommand(sqlStatement, conn);
        deleteCommand.Parameters.Add(new SqlParameter("@BusinessEntityID",
SqlDbType.Int, 0, "BusinessEntityID"));
        deleteCommand.Parameters.Add(new SqlParameter("@PhoneNumber",
SqlDbType.NVarChar, 25, "PhoneNumber"));
        deleteCommand.Parameters.Add(new SqlParameter("@PhoneNumberTypeID",
SqlDbType.Int, 0, "PhoneNumberTypeID"));

        da.DeleteCommand = deleteCommand;
        DataTable dt = new DataTable();
        da.Fill(dt);

        foreach (DataRow row in dt.Rows)
        {
            row.Delete();
        }
        da.Update(dt);
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.5.5 **Metoda ExecInsertQuery**

```
public long ExecInsertQuery()
{
    sw.Start();
    using (SqlConnection conn = new SqlConnection(connStr))
    {
        DataTable dt = new DataTable();
        DateTime currentTime = DateTime.Now;
        conn.Open();
        string sqlStatement = "select * from Person.PersonPhone";
        SqlDataAdapter da = new SqlDataAdapter(new SqlCommand(sqlStatement, conn));

        sqlStatement = @"insert into Person.PersonPhone
                        (BusinessEntityID, PhoneNumber, PhoneNumberTypeID,
                          ModifiedDate)
                        values (@BusinessEntityID, @PhoneNumber, @PhoneNumberTypeID,
                          @ModifiedDate)";
        SqlCommand insertCommand = new SqlCommand(sqlStatement, conn);
        insertCommand.Parameters.Add(new SqlParameter("@BusinessEntityID",
              SqlDbType.Int, 0, "BusinessEntityID"));
        insertCommand.Parameters.Add(new SqlParameter("@PhoneNumber",
              SqlDbType.NVarChar, 25, "PhoneNumber"));
        insertCommand.Parameters.Add(new SqlParameter("@PhoneNumberTypeID",
              SqlDbType.Int, 0, "PhoneNumberTypeID"));
        insertCommand.Parameters.Add(new SqlParameter("@ModifiedDate",
              SqlDbType.DateTime, 0, "ModifiedDate"));

        da.InsertCommand = insertCommand;
        da.FillSchema(dt,SchemaType.Source);

        dt.Rows.Add(1704, "963-222-223", 2, currentTime);
        dt.Rows.Add(1399, "725-012-638", 1, currentTime);
        dt.Rows.Add(1399, "89-993-181", 2, currentTime);
        dt.Rows.Add(7225, "111-984-333", 1, currentTime);
        dt.Rows.Add(8911, "111-412-333", 1, currentTime);
        dt.Rows.Add(17734, "96-222-814", 1, currentTime);
        dt.Rows.Add(17733, "689-222-13", 2, currentTime);
        dt.Rows.Add(20771, "111-95-333", 1, currentTime);
        dt.Rows.Add(19624, "321-222-891", 2, currentTime);
        dt.Rows.Add(9576, "111-222-333", 1, currentTime);

        da.Update(dt);
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.6 Třída PerformanceTestingEF

8.6.1 Proměnné a konstruktor

```
private Stopwatch sw = new Stopwatch();

public PerformanceTestingEF()
{
    using (AdventureWorksEntities db = new AdventureWorksEntities())
    {
        var q = from at in db.AddressType
                select at;
        var b = q.First();
    }
}
```

8.6.2 Metoda ExecSelectQuery

```
public long ExecSelectQuery()
{
    sw.Start();

    using (AdventureWorksEntities db = new AdventureWorksEntities())
    {
        var q = from p in db.Person
                join c in db.Customer on p.BusinessEntityID equals c.PersonID
                join soh in db.SalesOrderHeader on c.CustomerID equals soh.CustomerID
                join sod in db.SalesOrderDetail on soh.SalesOrderID equals
                    sod.SalesOrderID
                join pr in db.Product on sod.ProductID equals pr.ProductID
                where pr.Name == "Front Brakes"
                orderby p.LastName
                select new { p.Title, p.FirstName, p.MiddleName, p.LastName }
                ;
        q.ToList();
    }

    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.6.3 **Metoda ExecUpdateQuery**

```
public long ExecUpdateQuery()
{
    sw.Start();

    using (AdventureWorksEntities db = new AdventureWorksEntities())
    {
        var q = from e in db.Employee
                where DateTime.Compare(e.HireDate, new DateTime(2009, 1, 1, 0, 0, 0)) > 0
                || (e.JobTitle.StartsWith("Marketing")
                    && e.HireDate.Year >= 1977 && e.HireDate.Year <= 1984)
                select e;

        foreach (EmployeeEF e in q)
        {
            e.ModifiedDate = DateTime.Now;
            e.VacationHours = e.VacationHours++;
        }
        db.SaveChanges();
    }

    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.6.4 **Metoda ExecDeleteQuery**

```
public long ExecDeleteQuery()
{
    sw.Start();
    using (AdventureWorksEntities db = new AdventureWorksEntities())
    {
        var q = from pp in db.PersonPhone
                where pp.PhoneNumber.Contains("-222-")
                || ((pp.PhoneNumber.StartsWith("111-")
                        && pp.PhoneNumber.EndsWith("-333"))
                    && pp.PhoneNumberTypeID == 1)
                || pp.BusinessEntityID == 1399
                select pp;

        foreach (PersonPhoneEF pp in q)
        {
            db.PersonPhone.Remove(pp);
        }
        db.SaveChanges();
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.6.5 **ExecInsertQuery**

```
public long ExecInsertQuery()
{
    sw.Start();
    using (AdventureWorksEntities db = new AdventureWorksEntities())
    {
        DateTime currentTime = DateTime.Now;
        db.PersonPhone.Add(new PersonPhoneEF {
            BusinessEntityID = 1704, PhoneNumber = "963-222-223",
            PhoneNumberTypeID = 2, ModifiedDate = currentTime });
        db.PersonPhone.Add(new PersonPhoneEF {
            BusinessEntityID = 1399, PhoneNumber = "725-012-638",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });
        db.PersonPhone.Add(new PersonPhoneEF {
            BusinessEntityID = 1399, PhoneNumber = "89-993-181",
            PhoneNumberTypeID = 2, ModifiedDate = currentTime });
        db.PersonPhone.Add(new PersonPhoneEF {
            BusinessEntityID = 7225, PhoneNumber = "111-984-333",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });
        db.PersonPhone.Add(new PersonPhoneEF {
            BusinessEntityID = 8911, PhoneNumber = "111-412-333",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });
        db.PersonPhone.Add(new PersonPhoneEF {
            BusinessEntityID = 17734, PhoneNumber = "96-222-814",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });
        db.PersonPhone.Add(new PersonPhoneEF {
            BusinessEntityID = 17733, PhoneNumber = "689-222-13",
            PhoneNumberTypeID = 2, ModifiedDate = currentTime });
        db.PersonPhone.Add(new PersonPhoneEF {
            BusinessEntityID = 20771, PhoneNumber = "111-95-333",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });
        db.PersonPhone.Add(new PersonPhoneEF {
            BusinessEntityID = 19624, PhoneNumber = "321-222-891",
            PhoneNumberTypeID = 2, ModifiedDate = currentTime });
        db.PersonPhone.Add(new PersonPhoneEF {
            BusinessEntityID = 9576, PhoneNumber = "111-222-333",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });
        db.SaveChanges();
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.7 Třída PerformanceTestingLINQ

8.7.1 Proměnné a metoda ExecSelectQuery

```
private Stopwatch sw = new Stopwatch();

public long ExecSelectQuery()
{
    sw.Start();
    using (AWDataClassesDataContext db = new AWDataClassesDataContext())
    {
        var q = from p in db.Persons
                join c in db.Customers on p.BusinessEntityID equals c.PersonID
                join soh in db.SalesOrderHeaders on c.CustomerID equals soh.CustomerID
                join sod in db.SalesOrderDetails on soh.SalesOrderID equals
                    sod.SalesOrderID
                join pr in db.Products on sod.ProductID equals pr.ProductID
                where pr.Name == "Front Brakes"
                orderby p.LastName
                select new { p.Title, p.FirstName, p.MiddleName, p.LastName }
                ;
        q.ToList();
    }

    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.7.2 Metoda ExecUpdateQuery

```
public long ExecUpdateQuery()
{
    sw.Start();

    using (AWDataClassesDataContext db = new AWDataClassesDataContext())
    {
        var q = from e in db.Employees
             where DateTime.Compare(e.HireDate, new DateTime(2009, 1, 1, 0, 0, 0)) > 0
             || (e.JobTitle.StartsWith("Marketing")
                 && e.HireDate.Year >= 1977 && e.HireDate.Year <= 1984)
             select e;

        foreach (Employee e in q)
        {
            e.ModifiedDate = DateTime.Now;
            e.VacationHours = e.VacationHours++;
        }
        db.SubmitChanges();
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.7.3 **Metoda ExecDeleteQuery**

```
public long ExecDeleteQuery()
{
    sw.Start();

    using (AWDataClassesDataContext db = new AWDataClassesDataContext())
    {
        var q = from pp in db.PersonPhones
                where pp.PhoneNumber.Contains("-222-")
                || ((pp.PhoneNumber.StartsWith("111-") && pp.PhoneNumber.EndsWith("-
333"))
                    && pp.PhoneNumberTypeID == 1)
                || pp.BusinessEntityID == 1399
                select pp;

        foreach (PersonPhone pp in q)
        {
            db.PersonPhones.DeleteOnSubmit(pp);
        }

        db.SubmitChanges();
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

8.7.4 **Metoda ExecInsertQuery**

```
public long ExecInsertQuery()
{
    sw.Start();

    using (AWDataClassesDataContext db = new AWDataClassesDataContext())
    {
        DateTime currentTime = DateTime.Now;

        db.PersonPhones.InsertOnSubmit(new PersonPhone {
            BusinessEntityID = 1704, PhoneNumber = "963-222-223",
            PhoneNumberTypeID = 2, ModifiedDate = currentTime });
        db.PersonPhones.InsertOnSubmit(new PersonPhone {
            BusinessEntityID = 1399, PhoneNumber = "725-012-638",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });
        db.PersonPhones.InsertOnSubmit(new PersonPhone {
            BusinessEntityID = 1399, PhoneNumber = "89-993-181",
            PhoneNumberTypeID = 2, ModifiedDate = currentTime });
        db.PersonPhones.InsertOnSubmit(new PersonPhone {
            BusinessEntityID = 7225, PhoneNumber = "111-984-333",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });
        db.PersonPhones.InsertOnSubmit(new PersonPhone {
            BusinessEntityID = 8911, PhoneNumber = "111-412-333",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });
        db.PersonPhones.InsertOnSubmit(new PersonPhone {
            BusinessEntityID = 17734, PhoneNumber = "96-222-814",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });
        db.PersonPhones.InsertOnSubmit(new PersonPhone {
            BusinessEntityID = 17733, PhoneNumber = "689-222-13",
            PhoneNumberTypeID = 2, ModifiedDate = currentTime });
        db.PersonPhones.InsertOnSubmit(new PersonPhone {
            BusinessEntityID = 20771, PhoneNumber = "111-95-333",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });
        db.PersonPhones.InsertOnSubmit(new PersonPhone {
            BusinessEntityID = 19624, PhoneNumber = "321-222-891",
            PhoneNumberTypeID = 2, ModifiedDate = currentTime });
        db.PersonPhones.InsertOnSubmit(new PersonPhone {
            BusinessEntityID = 9576, PhoneNumber = "111-222-333",
            PhoneNumberTypeID = 1, ModifiedDate = currentTime });

        db.SubmitChanges();
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```