TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

PROFILOVÁNÍ APLIKACÍ PRO .NET A SQL SERVER

Bakalářská práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802R022 – Informatika a logistika

Autor práce: **Antonín Urban**
Vedoucí práce: Ing. Roman Špánek Ph.D.

Liberec 2018

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

PROFILING OF APPLICATIONS FOR .NET AND SQL SERVER

Bachelor thesis

Study programme: N2612 – Electrotechnology and informatics
Study branch: 1802R022 – Informatics and logistics

Author: **Antonín Urban**
Supervisor: Ing. Roman Špánek Ph.D.

Liberec 2018

Technická univerzita v Liberci
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2017/2018**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Antonín Urban**
Osobní číslo:	**M14000100**
Studijní program:	**B2612 Elektrotechnika a informatika**
Studijní obor:	**Informatika a logistika**
Název tématu:	**Profilování aplikací pro .NET a SQL Server**
Zadávající katedra:	**Ústav mechatroniky a technické informatiky**

Zásady pro vypracování:

1. Seznamte se se způsoby sledování a ladění výkonu desktopových aplikací v prostředí .NET využívající SQL Server k uložení dat.
2. Na vybraných příkladech demonstrujte použití různých metod pro odlaďování výkonnostních problémů aplikace a jejich řešení.
3. Pro vzorovou aplikaci zkoumejte vliv Vámi vybraných parametrů systému, jako jsou například operační paměť, procesor, pevný disk, způsoby komunikace s SQL Serverem a další, na vykonání typických úkonů.
4. Shrňte získané znalosti a diskutujte možnosti současných nástrojů pro ladění výkonu aplikace při vývoji a nasazení.

Rozsah grafických prací: **dle potřeby dokumentace**

Rozsah pracovní zprávy: **30–40 stran**

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

[1] **Jeff Prosise: Practical Performance Profiling: Improving the efficiency of .NET code, Red gate books (March 9, 2012), ISBN-10: 1906434824,ISBN-13: 978-1906434823.**

[2] **Brad M. McGehee: Mastering SQL Server Profiler, ISBN 1906434158 (ISBN13: 9781906434151), 2009.**

[3] **Tracing and Instrumenting Applications: . NET Framework (current version). 2016. MSDN [online]. [cit. 2017-10-05]. Dostupné z https://msdn.microsoft.com/cs-cz/library/zs6s4h68(v=vs.110).aspx**

Vedoucí bakalářské práce: **Ing. Roman Špánek, Ph.D.**
Ústav mechatroniky a technické informatiky

Datum zadání bakalářské práce: **10. října 2017**

Termín odevzdání bakalářské práce: **14. května 2018**

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

L.S.

doc. Ing. Milan Kolář, CSc.
vedoucí ústavu

V Liberci dne 10. října 2017

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum: 13. 5. 2018

Podpis: 

Poděkování

Rád bych zde poděkoval vedoucímu mé práce, Ing. Romanovi Špánkovi Ph.D., že se dodatečně ujal mé práce ve chvíli, kdy jsem pro ni neměl vedoucího. Děkuji také za jeho cenné rady, připomínky a vstřícnost při konzultacích. Další poděkování patří Ing. Janě Vitvarové, Ph.D. a Ing. Janu Krausovi, Ph.D. kteří mi pomohli s hledáním vedoucího. Poslední poděkování bych chtěl věnovat firmě Winfo s.r.o. za poskytnutí vzorové aplikace a databáze pro účely této práce.

Abstrakt

Tato práce se zabývá měřením a optimalizací aplikací platformy .NET a databází SQL Server. Cílem práce je seznámit s pojmem profilování a ukázat některé dostupné postupy a nástroje sloužící pro měření a optimalizaci aplikací využívajících SQL Server jako zdroj dat.

Pro měření výkonu aplikací i činnosti SQL Server je v práci popsáno použití výkonnostních čítačů, zaměřených na měření vytížení procesoru, operační paměti a disku. Z pohledu optimalizace práce seznamuje s dvěma profilery pro platformu .NET (Visual Studio profiler, dotTrace) a nástrojem Database Engine Tuning Advisor pro optimalizaci databází v SQL Server. Demonstrace nástrojů je provedena na vzorové aplikaci a databázi, kdy je detailně popsán celý průběh optimalizace společně s měřením pomocí výkonnostních čítačů.

Klíčová slova:

profilování, .NET, profiler, měření aplikací, Výkonnostní čítače systému Windows, Visual Studio Profiler, dotTrace, Microsoft SQL Server, Database Engine Tuning Advisor, SQL Server Profiler

Abstract

This thesis deals with measurement and optimalization of applications of the .NET platform and SQL Server databases. The aim of the thesis is to familiarize reader with the term profiling and show some of the available procedures and tools used for measuring and optimizing applications using SQL Server as a data source.

Use of performance counters is described for measurement of application performance and activity of SQL Server. It is focused on measuring processor, operation memory and disk usage. In terms of optimalization, there are two profilers for the .NET platform (Visual Studio profiler, dotTrace) and the Database Engine Tuning Advisor tool for SQL Server database optimalization. The use of all the tools is demonstrated on a sample application and database. The overall optimalization process is described in details alongside with the use of performance counters for measurements.

Key words:

profiling, .NET, profiler, aplication measurement, Windows Performance Counters, Visual Studio Profiler, dotTrace, Microsoft SQL Server, Database Engine Tuning Advisor, SQL Server Profiler

Obsah

Seznam snímků obrazovky

Seznam tabulek

Seznam zkratek

.NET	Microsoft .NET Framework
API	Application Programming Interface
C#	C Sharp
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Common Language Specification
CPU	Central Processing Unit
DB	Database
DBMS	Database Management System
DTA	Database Engine Tuning Advisor
ETW	Event Tracing for Windows
JIT	Just In Time
MSIL	Microsoft Intermediate Language
MSQLS	Microsoft SQL Server
RAM	Random Access Memory
SQL	Structured Query Language,
SSMS	SQL Server Management Studio
TSQL	Transact-SQL
VES	Virtual Execution System
VS	Visual Studio
WPF	Windows Presentation Foundation

Úvod

Struktura moderní počítačové aplikace, naprogramované objektově ve vyšším programovacím jazyce, bývá typicky složitá a sestává z několika celků. Pokud chceme v dnešní době takovou aplikaci programovat, velice často neprogramujeme jen za pomoci základních knihoven daného jazyka. Na velkou část dílčích částí aplikace existují již vytvořená řešení ve formě komponent a nám je stačí jen vhodně implementovat. Takový postup se v dnešní době dá označit za trend. Výhodou využívání této modulárnosti je ušetřený čas na vývoji, který by programátor musel strávit vlastním řešením jednotlivých celků. Naproti tomu je ale potřeba vybrat nejvhodnější řešení – zpravidla jich lze nalézt několik na jednu oblast aplikace. Mimo jednodušších komponent, specializujících se na dílčí problémy (např. tvorba logu) se můžeme také setkat s komplexními softwarovými balíky zaměřující se na velkou oblast problémů. Příkladem mohou být balíky pro tvorbu grafického rozhraní, kdy balík obsahuje několik desítek komponent a souvisejících funkcí. Tyto, často komerční, balíky se jejich vývojáři snaží nabízet pro co největší počet vývojářů. Tato variabilita komponent společně s modulárností jako takovou může přinášet problémy z pohledu efektivity aplikace při využívání výpočetního výkonu.

Vyvstává možná otázka, proč je potřeba v dnešní době sledovat efektivitu aplikace. Často máme kolem sebe zařízení nabízející mnohem více výkonu, než potřebujeme. V první řadě záleží na účelu aplikace – například na zpracovávání a reprezentaci velmi objemných dat bude vždy výhodné mít k dispozici efektivní aplikaci, protože takové operace mohou i dnes narážet na hardwarové limity. Silný hardware nám nemusí automaticky zaručit rychlou a efektivní aplikaci. Špatně naprogramovaná aplikace může být pomalá i na dostatečně silném hardwaru. Pokud máme navíc aplikaci určenou pro běžného spotřebitele, možná ji bude zapotřebí optimalizovat pro slabší hardware, než s jakým jsme se setkávali při vývoji. Právě v takových situacích může být vhodné uvažovat o profilování.

Pokud optimalizujeme jednodušší aplikaci, máme většinou určitý přehled o tom, jak dlouho by určité operace měly trvat. Už v průběhu programování tak víme o místech, která jsou problematická a můžeme se na ně později zaměřit. V případě optimalizace aplikace se složitou strukturou ale může být problém vůbec najít nějaké kritické místo. Jedna operace aplikace se může v kódu sestávat ze sekvence několika činností, kdy jednotlivé dílčí části mohou vykonávat různé objekty z různých částí struktury aplikace. Abychom tedy mohli nalézt kritické místo, je vhodné mít určitou

reprezentaci běhu aplikace. Většina profilovacích nástrojů nám právě takovou reprezentaci běhu aplikace nabízí a pomáhá nám tak s hledáním míst k optimalizaci. Výsledek se samozřejmě odvíjí od toho, jak dobře dokážeme výstup z profilování interpretovat. Obecně je k vyhodnocování výsledků často potřeba určité znalosti profilované aplikace a programování obecně. Podstatná může být pro úspěch také volba, jakou metodu profilování vůbec použijeme.

Dalším významným použitím profilování je měření parametrů běžících aplikací. V praxi se můžeme setkat například s požadavky na stanovení minimální velikosti operační paměti pro běh aplikace. Zde se již primárně nemusíme bavit o specializovaných profilovacích nástrojích, profilováním může být i sbírání dat za běhu aplikace nadefinované přímo v jejím kódu. Profilování se dá také využít už při samotném vývoji aplikace. V praxi se často problémy s výkonem aplikací řeší až v průběhu testování, nebo ještě hůře až přímo při nasazení. Pokud budeme profilovací nástroje využívat už ve fázi vývoje, může dosáhnout větší efektivity celého vývojového cyklu.

Tato práce má za úkol seznámit s profilováním a nástroji pro měření a ladění výkonu pro platformu .NET a databázový systém SQL server. Rešeršní část práce by měla krátce tyto platformy představit a následně popsat pro ně dostupné možnosti profilování. Praktická část by pak měla detailněji popsat vybrané nástroje pro měření a optimalizaci a jejich použití následně demonstrovat na vzorové aplikaci.

1 Pojem profilování

Pokud pojem profilování (profiling) zadáme do anglického slovníku, setkáme se většinou s vysvětlením pojmu ve vztahu k psychologii. Například Cambridge Dictionary uvádí definici přibližně takto: "aktivita sbírání informací o nějaké osobě za účelem popsání takové osoby"[16]. Tato definice pojmu samozřejmě nemá přímou souvislost s profilováním softwaru, s určitým nadhledem se v ní dá pozorovat vazba na pojem profilování spojený s vývojem aplikací. Profilování, jakým se zabývá tato práce, má také podobu sběru informací za účelem popsání nějaké entity (zde aplikace); výsledky pak mohou vést k odhalení výkonnostního problému.

1.1 Profilování aplikací

Pojem profilování aplikací (dále jen profilování) není striktně ukotven a ani nelze najít velké množství definic, je tedy poměrně složité ho přesně definovat. V části případů se s pojmem profilování nebo profilace setkáváme ve spojení se specializovanými nástroji pro odhalování výkonnostních problémů aplikací – takzvanými profilery.[3] [1] [10] Samotný pojem profilování ale není v těchto případech obecně definován, vystupuje zde spíše ve vztahu k činnosti profileru (profiler profiluje, profilování = používání profileru).

Dalším výskyt pojmu je v textech zabývajících se měřením výkonu běžících aplikací a jejich optimalizací. Zde bývá profilování popisováno jako používání různých nástrojů k odhalení výkonnostních problémů. Nástroji zde nejsou myšleny pouze profilery, jedná se prakticky o cokoliv, co nám umožní změřit výkon určité aplikace. Stejně tak odhalení výkonnostních problémů nemusí být v tomto významu spojeno pouze s problémy špatně napsaného kódu (kam většinou směřuje použití profilerů). Problém můžeme hledat například v použitém hardwaru nebo pomalé databázi. [29] [30]

1.2 Význam profilování

Z výše uvedeného by se tedy profilování dalo zobecnit na měření běžících aplikací za účelem optimalizace. Stejně jako pojem profilování není ukotveno ani použití profilování. Primárně se setkáme s použitím k optimalizaci. Profilování, respektive měření aplikací obecně, může mít ale i jiné využití.

Příkladem takového využití profilování je například scénář, kdy máme neznámou a složitou aplikaci a potřebujeme zjistit, jak funguje její určitá část. Použitím profilovacího nástroje v kódu zjistíme metody a funkce využité danou operací, což nám může pomoci se v aplikaci zorientovat. Stejně tak měření aplikace z pohledu dopadu na výkon hardwaru nemusí vždy sloužit primárně k optimalizaci. Můžeme chtít zjistit minimální konfiguraci potřebnou pro běh aplikace nebo pouze srovnávat dvě verze aplikace.

2 Úvod do platformy .NET a SQL Server

Tato kapitola se věnuje stručnému popisu platformy .NET a databázového systému SQL Server. Součástí kapitoly je také popis struktury platformy .NET sloužící k pochopení základních pojmů spojených s touto platformou. Závěr kapitoly se pak věnuje pojmu profilovací API.

2.1 SQL Server

Microsoft SQL Server (někdy zkracováno na SQL Server nebo MSQLS) je relační databázový systém společnosti Microsoft. Patří mezi takzvané systémy řízení báze dat (anglicky database management system – DBMS). Hlavní charakteristikou těchto systémů je poskytovat softwarové rozhraní, umožňující uživatelům nebo aplikacím operovat s databázovou strukturou a daty. SQL Server hraje jednu z klíčových rolí mezi DBMS, především v industriální sféře.

Softwarové rozhraní je u SQL Server zastoupeno především aplikací SQL Server Management Studio (SSMS). SSMS poskytuje funkce a nástroje pro konfiguraci, monitorování a administraci serverů a databází v SQL Server. Mezi další nástroje SQL Server patří SQL Server profiler a Database Engine Tuning Advisor (DTA). První nástroj slouží k sledování a zaznamenávání událostí běžícího serveru, druhý k optimalizaci databází.

Rozšíření jazyka SQL pro SQL Server se nazývá Transact-SQL a bývá běžně zkracováno na TSQL. [37] [22]

2.2 Platforma .NET

.NET je platforma společnosti Microsoft určená pro systémy Windows. Platforma zprostředkovává prostředí pro vývoj softwaru v některém z podporovaných jazyků. Mezi základní zastřešované jazyky patří C#, Managed C++ (C++ implementované do platformy .NET) a jazyk F#.

Cílem frameworku .NET je poskytnout jednotné prostředí s minimální vazbou na konfiguraci nebo verzi cílové platformy. Framework .NET se tedy dá chápat jako

určitá softwarová vrstva nad cílovou platformou umožňující běh všech aplikací v podporovaných jazycích. Výhodou je přenositelnost aplikací – v tomto případě mezi verzemi systému Windows. Pokud je aplikace napsaná v jazyce platformy .NET, máme zaručeno, že tato aplikace poběží na jakékoliv konfiguraci a verzi systému Windows podporující danou verzi .NET frameworku. Zároveň běhové prostředí .NET poskytuje například automatické uvolňování paměti, základní datové typy, optimalizaci běhu aplikace, správu referencí a podobně. Velkou výhodou frameworku je také rozsáhlá sada základních knihoven, které jsou přímo jeho součástí. [25] [13] [8] [35] [19] [38]

2.2.1 Struktura platformy .NET

Kód vytvořený v jazyce platformy .NET je po kompilaci kompilátorem převeden do intermediárního („přechodného“) jazyka nazývaného Common Intermediate Language (CIL, lze se setkat i se starším názvem Microsoft Intermediate Language – MSIL). Kód tohoto jazyka se někdy označuje jako řízený (managed code), protože k jeho výkonu je potřeba služba, která tento kód řídí. U platformy .NET se tato služba nazývá Common Language Runtime (CLR). [25] [8] [19] [38] [26] [5]

2.2.2 Standard Common Language Infrastructure

Standard Common Language Infrastructure (CLI) popisuje základní aspekty, definující intermediární jazyk CIL: Common Type System, metadata, Common Language Specification a Virtual Execution System. CLI obecně zaručuje dva důležité aspekty platformy .NET – jazykovou a hardwarovou nezávislost.

Common Type System

Jednotný systém datových typů. Smyslem je definovat datové struktury na platformě .NET jako jsou třídy, struktury, enumerátory a podobně. CTS také definuje datové typy atributů objektů (např. property, field, method, constructor) a přístupové modifikátory (např. private, protected, abstract). [6] [11]

Metada

Metada jsou binární informace popisující kód. Tyto informace popisují strukturu kódu aplikace především detailním popisem tříd – metadata obsahují informace jako například název třídy, návratový typ třídy, název implementovaného interface, přístupový modifikátor třídy a podobně. Dále metadata popisují sestavení aplikace – název, verzi, jazykovou kulturu a podobně. Smyslem metadat je vytvořit sebepopisující komponenty a popsat tak kód – a to neutrálním jednotným způsobem. Tím lze umožnit snadné referencování částí kódu i pro ostatní jazyky, než je jazyk daného kódu (například spolupráce mezi jazykem C# a formuláři WPF). Metadata mají dále klíčovou roli při používání reflexe, která umožňuje psát kód pracující s objekty dynamicky. Příkladem může být volání funkce GetType() v jazyce C# umožňující

programátorovi získat informaci o typu libovolného objektu za běhu aplikace. [6] [17] [21] [40] [7]

Common Language Specification

Soubor pravidel, který umožňuje definovat jazykově nezávislý kód pro platformu .NET. Pokud jazyk splňuje CLS, je možné používat atributy a třídy napříč všemi jazyky podporující platformu .NET. Podmnožinou těchto pravidel je i CTS. [15]

Virtual Execution System

Prostředí pro vykonání intermediární kódu. Podstatou tohoto prostředí je převod CIL do nativního strojového jazyka dané konfigurace. VES se tedy stará o to, aby CIL bylo přeloženo do správné sady instrukcí pro danou konfiguraci. [28]

2.2.3 Běhové prostředí (CLR)

Běhové prostředí CLR se dá označit za základní stavební jednotku frameworku .NET. Obecně toto běhové prostředí řídí intermediární kód. Pod pojmem řídí se skrývá několik činností, základní je překlad kódu z CIL do nativního strojového jazyka počítače, k čemuž je využíváno prostředí VES definované výše. Mimo to se CLR stará o uvolňování nepoužívaných zdrojů, rozděluje paměť přidělenou aplikaci, řídí vlákna přidělená aplikaci, obsluhuje výjimky a zprostředkovává debuggovací služby. Díky CTS a CLS ze standardu CLI, jenž musí implementovat každý vstupní kód pro CLR, je zaručena typová bezpečnost a spolupráce mezi jazyky. [35] [19] [5]

2.2.4 Just in Time kompilace

Princip překladu aplikace pomocí CLR při každém spuštění aplikace sebou nese problém větší náročnosti na výkon. Aby se tento problém alespoň částečně eliminoval a běh aplikace zefektivnil, místo překladu celé aplikace je použita takzvaná just in time kompilace (JIT). Tato kompilace ve své základní podobě provádí překlad pouze té části kódu, která je zrovna potřeba. Přeložený kód pak přidá do paměti a při opětovném volání již není tento kód potřeba překládat. [5] [42] [41]

Ne vždy je výhodný takový postup, proto má v .NET vývojář možnost u některých částí kódu zapnout takzvanou pre-JIT kompilaci. Ta přeloží danou část kódu přesně v moment, kdy vývojář potřebuje. Příkladem, kdy by bylo vhodné použít pre-JIT kompilaci, může být aplikace využívající velké množství náročných ovládacích prvků (například ve WPF). Pomocí pre-JIT kompilace můžeme jejich překlad přesunout na vedlejší vlákno a spustit ho na pozadí při přihlašování uživatele do aplikace. [36]

2.2.5 Profilovací API

.NET nabízí vlastní profilovací API určené pro vývojáře profilovacích nástrojů. Vývojář profileru vytvoří knihovnu, která je načtena pomocí CLR za běhu aplikace

a přidána k procesu profilované aplikace. Není však nikdy součástí této aplikace a je nežádoucí, aby tato profilovací knihovna ovlivňovala běh aplikace, což by ani profilovací API jako takové nemělo umožňovat. Stejně tak je nežádoucí scénář, kdy by aplikace profilovala sebe sama nebo na profilování závisela. Profilovací knihovna připojená k profilovanému procesu by také měla obsahovat pouze metody pro sbírání dat – zpracování dat by mělo probíhat ve vlastním procesu, aby byl dopad měření na profilovanou aplikaci minimální. Tato omezení vyplývají z faktu, že profilovací API nemá sloužit pouze k měření aplikací, ale i k monitorování .NET aplikací a například i k jejich speciálnímu debuggování. Základní funkcí tohoto API je tedy poskytnout přístup k aplikacím mimo řízený kód platformy .NET za účelem měření nebo debuggování aplikací. Z toho také vyplývá, že profilovací API nelze implementovat v řízeném kódu platformy .NET (například jazyk C# tedy použít nejde). Pro implementování je možné využít mimo jiné jazyk C++ v jeho základní podobě, implementovaný jazyk managed C++ pro .NET už použít nelze.

Profilovací knihovna implementuje rozhraní, pomocí kterého komunikuje CLR s knihovnou volání funkcí upozorňující na události v profilovaném procesu – například vstup a výstup do funkce. Rozhraní pak ještě obsahuje funkce, jimiž profilovací knihovna může naopak požádat běhové prostředí o určité doplňující informace o měřeném procesu (příkladem může být název třídy obsahující právě prováděnou funkci). Profilovací API také umožňuje přistupovat ke kódu v intermediárním jazyce (CIL) ještě před jeho kompilací, čehož lze využít pro vkládání vlastních instrumentačních funkcí pro hlubší analýzu. [20] [39] [27]

3 Způsoby měření aplikací

Tato kapitola obecně seznamuje s různými způsoby, jak měřit běžící aplikace v systému Windows.

3.1 Microbenchmarking

Za základní způsob měření aplikace by se dalo označit prosté analyzování kódu (příkladem může být i zkoumání struktury aplikace na papíře) a případné vkládání vlastních měřících bloků do kódu. Toto se někdy označuje jako benchmarking, případně microbenchmarking. Předpokladem pro využití takovéhoto měření je detailní znalost struktury aplikace z pohledu vývojáře a při správném použití na správném místě může toto měření přinášet velice rychlé a jasně vypovídající výsledky. Do určité míry je pro takové efektivní a rychlé použití předpokladem i fakt, že toto měření bude použito na malé oblasti kódu (proto microbenchmarking). Pro komplexnější měření je potřeba stejně komplexní struktura použití takovýchto měřících funkcí. Stejně tak relativní je i to, jak dobře a rychle čitelný bude výstup měření – vše záleží na tom, jak dobře si programátor výstup přizpůsobí. Obecně se tedy dá říct, že velkou výhodou takovéto metody měření je její přirozená přizpůsobitelnost a nevýhodou zvětšující se náročnost použití pro detailní výsledky. Velkou nevýhodou je také malá flexibilita u komplexních měření – byť se opět dá diskutovat o jisté relativnosti. Pokud má programátor dost zkušeností a času si vytvořit dobrou měřící strukturu, může i tento problém eliminovat. S tím kromě problému časové náročnosti vystává i otázka, zda nebudeme vlastně aplikaci zbytečně zatěžovat něčím, co není potřeba k její činnosti. Za předpokladu, že by aplikace měla reagovat na výsledky měření, pak by se o takovém použití dalo uvažovat jako o vhodném. Pokud se ale snažíme o měření za účelem odhalení míst k optimalizaci, vhodnější by mělo být použití jiných metod. [29]

3.2 Vestavěné nástroje systému Windows

Za určitý mezikrok mezi benchmarkingem a specializovanými profilery se dá považovat používání struktur přímo v systému Windows. Patří sem tzv. výkonnostní čítače (anglicky performance counters) a speciální logovací framework ETW. Způsob nastavování a používání má poměrně blízko k benchmarkingu. Blíže k profilovacím nástrojům je naopak posouvá sbírání dat probíhající mimo aplikaci. Zpracování a

zobrazení výstupu je taktéž obstaráváno vlastní aplikací a není tedy třeba myslet na formát výstupu jako u benchmarkingu. [29]

3.2.1 Výkonnostní čítače

Systém Windows obsahuje integrované výkonnostní čítače sledující celou škálu různých parametrů. Základním použitím výkonnostních čítačů je prosté čtení zvolených hodnot přes systémovou aplikaci *Sledování výkonu* (anglicky *Performance monitor*). Data se po nastavení čítače začnou v reálném čase automaticky vykreslovat do grafu v hlavním okně. Čítače jsou rozděleny do kategorií podle různých kritérií, několik vlastních kategorii má v seznamu například i .NET nebo SQL server (pokud jsou tyto komponenty v systému přítomny).

Každý čítač nabízí k výběru instance, pro které bude data sbírat. Například čítač *% času procesoru* může čítat data jen pro některé z dostupných jader procesoru, pokud je vybrán z kategorie *Procesor*. Pokud je vybrána kategorie *Proces*, je možné stejný čítač nastavit pro některý běžící proces – dá se tedy využít i k měření jedné konkrétní běžící aplikace. Kromě základního zobrazení dat do real-time grafu v hlavním okně se hodnoty dají také ukládat rovnou do souboru.

Čítače mají možnost nastavování přímo v kódu aplikace přes třídu *System.Diagnostics.PerformanceCounter*. Mimo systémem definovaných čítačů lze vytvořit i svůj vlastní čítač, buď přímo přes rozhraní Visual Studia nebo v kódu. Do vlastního čítače se dají ukládat libovolná numerická data a následně je možné s nimi v aplikaci Sledování výkonu pracovat stejně jako u systémových čítačů. [29]

3.2.2 Event Tracing for Windows

Pokud aplikace vytváří log, znamená to, že nějakým způsobem zaznamenává svůj průběh. Co aplikace loguje záleží na nastavení logování v dané aplikaci. Zpravidla se jedná o důležité body běhu aplikace a především chybové hlášky. Soubory, kam se tyto záznamy z běhu ukládají, se nazývají logy.

Z principu logování by se mohlo zdát, že tato činnost je vždy užitečná a proto má smysl jí pokaždé implementovat do aplikace. Častým problémem je dopad logování na výkon aplikace – pokud má aplikace detailní log a tedy zapisuje velké množství informací do takového logu, může tato činnost aplikaci zatěžovat. Proto se v praxi často logování ve finální verzi omezuje jen na nejdůležitější místa. Detailní log se pak například zapíná na vyžádání, případně existují speciální debugovací verze aplikace, kde je detailní log ponechán.

Fakt, že samotný zápis do logu zpomaluje aplikaci, nenahrává použití klasického logování pro měření aplikací. Místo běžných frameworků pro logování se dá použít Event Tracing for Windows (ETW). Dalo by se definovat jako speciální logování implementované přímo v systému Windows a vykazující velmi malý dopad na výkon počítače nebo aplikace.

Na rozdíl od klasického logování ETW nezapisuje přímo do souboru ale podává informace o tom, zda nastala určitá událost. O tom, jakou událost má ETW sledovat rozhoduje takzvaný poskytovatel (provider). Systém Windows i běhové prostředí platformy .NET obsahují již předdefinované poskytovatele umožňující sledovat například čtení z disku nebo informace o JIT kompileru. Ovládání ETW pomocí stanovování poskytovatelů a následný sběr s případnou reprezentací dat musí obstarat speciální nástroj k tomu určený (například PerfView nebo Windows Performance Toolkit). Na platformě .NET si lze přímo kódu nadefinovat i vlastního poskytovatele implementováním rozhraní EventSource. [29] [33] [31]

3.3 Profilery

Pro profilování lze použít speciální software určený k této činnosti – profiler. Profiler je vždy vázaný na konkrétní platformu nebo jazyk. Výhodou u platformy .NET je, že profiler je platný pro celou platformu nezávisle na jazyku, což vychází z právě z architektury této platformy. Nespornou výhodou platformy .NET pro profilování je také vlastní profilovací API, usnadňující vývoj profilerů pro tuto platformu.

Profilery jsou ve většině případů nabízeny jako samostatné aplikace, výjimkou je pouze profiler integrovaný přímo do Visual Studia. Setkat se lze s nekomerčními i komerčními profilery. Nabízené metody měření jsou většinou dvě základní – samplování a instrumentace (často pod jiným názvem). Případně bývají metody doplněné o speciální profilování alokované paměti nebo sbírání dat o činnosti více vláken aplikace (pokud je vícevláknová). Způsoby měření ani získanými výsledky se profilery pro platformu .NET příliš neliší, což je dáno implementací stejného profilovacího API a využívaní stejných prvků architektury. Rozdíly tak jsou spíše v interpretaci výsledků a přehlednosti uživatelského prostředí.

Za základní profilovací nástroj platformy .NET se dá považovat zdarma dostupný profiler ve Visual Studiu. Další významné profilery jsou už komerční, patří sem například dotTrace, Stackify, ANTS Profiler, YourKit .NET Profiler. Významným profilerem byl ještě profiler JustTrace, jeho tvůrci ho ovšem stáhli z důvodu, že vedle profileru ve Visual Studiu nemá význam. [14]

3.3.1 Základní metody sbírání dat

Ačkoliv existuje velké množství profilerů nabízejících různě pojmenované metody měření, jedná se pokaždé o jeden ze dvou základních způsobů sběru dat – samplování a instrumentace.

Samplování je metoda využívající systémových přerušení pro sbírání vzorků. Profiler periodicky vyvolává systémová přerušení a odečítá data ze zásobníku volaných funkcí platných pro měřený proces. Pro funkci na vrcholu zásobníku, tedy právě prováděnou, profiler inkrementuje počet exklusivních vzorků. U funkcí volající tuto

prováděnou funkci pak dochází k inkrementaci inklusivních vzorků – tedy vzorků platných pro funkci a volané potomky. Samplování má velkou výhodu v minimálním dopadu na běh aplikace, naopak nevýhodou může být menší přesnost měření. Volání některých funkcí nemusí být vůbec zaznamenáno. Stane se tak, pokud je funkce volána jen zřídka a její průběh se stačí vejít mezi interval systémového přerušení vyvolaného profilerem. V některých případech může být nevýhodou samplování také nezaznamenávání práce aplikace se vstupně výstupními zařízeními. Pokud funkce dlouho čekala na odpověď disku, tento čas se do měření pomocí samplování nepromítne.

Instrumentace práci se vstupně výstupními zařízeními zaznamenat dokáže. Měření probíhá sbíráním informací o vstupování a vystupování aplikace do funkcí. Instrumentace tedy měří počet volání funkcí, dopočítává i dobu jejich průběhu (rozdíl mezi časem vstupu a výstupu [30]). Z principu měření vychází, že jsou zaznamenány všechny funkce včetně těch rychlých a zřídka kdy volaných. Časy vykonávání funkcí také dokáží ukázat na trvání vstupně výstupních operací. Velkou nevýhodou instrumentace je dopad měření na aplikaci. Sbírání dat o vstupu a výstupu aplikace z funkce přidává dodatečné zatížení. Instrumentace pak tyto časy nedokáže oddělit od celkového času, proto jsou výsledné časy strávené ve funkci zkreslené oproti běžnému průběhu. V knize Practical Perfomance Profiling [30] je na straně 199 citován Oleg Stepanov ze společnosti JetBrains (komerční profiler dotTrace [12]) vysvětlující rozdíly mezi samplováním a instrumentací. V textu tvrdí, že problém se zpomalením při instrumentaci je způsoben tím, jak se profiler musí přepínat mezi kódem aplikace a kódem profileru. U samplování tento problém samozřejmě nevzniká, profiler nemusí do kódu aplikace zasahovat vůbec, data pocházejí pouze ze systémového zásobníku volaní.

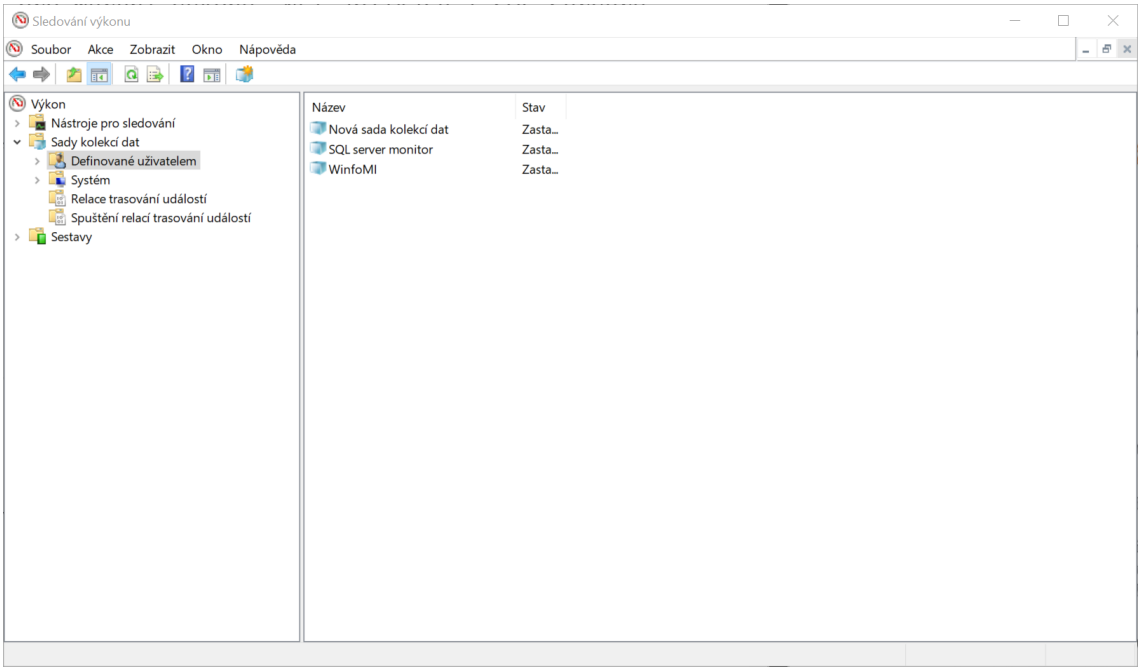
Pro běžné použití je tedy vhodné nejprve začínat samplovací metodou a v případě nedostatečnosti této metody použít instrumentaci. [29] [30] [24]

4 Popis použitých nástrojů

Tato kapitola obsahuje popis konkrétních nástrojů použitých v této práci. Kapitola se věnuje výkonnostním čítačům systému Windows, použitým pro měření výkonu .NET aplikace i SQL Serveru. Déle jsou v kapitole popsány dva profilery – zdarma dostupný profiler ve Visual Studiu a komerční profiler dotTrace. Poslední část kapitoly je určena nástroji pro optimalizaci SQL Server – Database Engine Tuning Advisor, dostupnému zdarma v rámci balíku nástrojů pro SQL Server. Popis nástroje se zaměřuje na jeho použití ve spolupráci s nástrojem SQL Server Profiler, sloužícím ke sběru záznamů o běhu SQL Serveru.

4.0.1 Použití výkonnostních čítačů

K ovládání čítačů slouží systémový nástroj Sledování výkonu. Někdy se nástroj nazývá také Perfmon podle příkazu spouštějící tento nástroj (perfmon.exe). Okno nástroje je vidět na snímku č. 1.



Snímek 1: Hlavní okno nástroje Sledování výkonu.

Pro vytvoření měření s výstupem slouží položka *Sady kolekcí dat*. Po jejím rozkliknutí se zobrazí několik dalších položek, nás bude zajímat

položka *Definované uživatelem*. Pravým kliknutím na položku a výběrem možnosti *Nová položka → Nová sada kolekcí dat* spustíme průvodce nastavení nového měření. V otevřeném okně zadáme název měření a zvolíme možnost *Vytvořit ručně*. Na další stránce průvodce necháme vybranou možnost *Protokoly vytváření dat* a zaškrtneme možnost *Čítač výkonu*. Přejdeme na další stránku, která se věnuje nastavení čítačů pro měření. Tlačítkem *Přidat* se otevře další okno, jehož levá strana obsahuje seznam dostupných čítačů. Čítače jsou zobrazeny v kategoriích, rozkliknutím se zobrazí relevantní čítače pro danou kategorii. Nakliknutím čítače pak vlevo dole v podokně můžeme vybírat instance pro daný čítač. Pokud by bylo instancí více, dole pod nimi lze použít vyhledávání. Vyhledávat čítače nelze, je vždy potřeba znát správnou kategorii čítače a najít ho ručně. Tlačítkem *Přidat* zvolený čítač pro vybranou instanci přidáme do výsledného seznamu. Ve spodním rohu se nachází volba *Zobrazit popis* zobrazující textový popis k právě přidávanému čítači. Potvrzením tlačítkem *Ok* se vrátíme zpět do průvodce. V seznamu *Čítače výkonu* se objeví vybrané čítače, pod ním lze ještě vybrat interval měření – nejmenší dostupný je 1 sekunda. V dalším okně průvodce vybereme adresář pro vytvořený soubor s výsledky měření, po potvrzení se dostaneme do poslední části průvodce. Zde vybereme případně uživatele, pod kterým chceme spustit měření a zvolíme možnost *Otevřít vlastnosti pro tuto sadu kolekcí*. Tato volba nás rovnou přesune do vlastností právě vytvořené sestavy dat (stejnou nabídku je možné vyvolat i později pravým klikem na sadu kolekce dat a výběrem volby *Vlastnosti*). Na dostupných záložkách můžeme zkontrolovat, případně změnit různé parametry sady nastavené v průvodci; lze ale také doplnit některé další.

Po vytvoření sady se v pravém podokně hlavního okna nástroje zobrazí nová položka kolekce dat (nejčastěji pojmenovaná jako *DataCollector01*). Pravým kliknutím na tuto položku a výběrem volby *Vlastnosti* zobrazíme poslední důležitá nastavení – nastavení výstupního souboru. Pod záložkou *Čítače výkonu* můžeme ještě jednou upravit vybrané čítače a především zvolit formát dat výběrem ze seznamu *Formát protokolu*. Na výběr jsou možnosti *Oddělené čárkou* a *Oddělené středníkem* pro export do csv nebo tsv souboru a *SQL* pro export do databázové tabulky. Poslední volba *Binární* umožňuje vytvořit výstup, jenž se dá otevřít přímo v nástroji Sledování výkonu. Pro zpracování v tabulkovém procesoru je nejlepší možnost souboru odděleného tabulátorem. Záložka soubor ještě obsahuje možnost nastavení jména výstupního souboru. Spuštění měření probíhá pravým kliknutí na sadu kolekce dat v levém okně (nikoli kolekci dat v pravém) a volbou *Spustit*. Po spuštění se dá ze stejného menu měření také zastavit. Pozastavit měření možné není. Spuštěnou sadu kolekce dat poznáme podle zelené šipky u ikonky spuštěné sady. Do jedné sady kolekce dat můžeme také pravým kliknutím do levého podokna kolekcí dat vytvořit další kolekce volbou *Nová položka → Kolekce dat*. Spuštěním sady pak pouštíme všechny kolekce najednou, přičemž každá bude disponovat vlastním výstupním souborem.

Příkladem zpracování tabulátorem oddělených dat může být použití programu Microsoft Excel – po změně koncovky z tsv na csv lze soubor v programu otevřít.

Následně stačí vybrat první sloupec obsahující naměřená data a v záložce *Data* volbou *Text do sloupců* převést data na další sloupce. Ve spuštěném průvodci převodu není v jeho výchozím nastavení potřeba nic měnit. V případě, že výstupní soubor je ve formátu odděleném čárkou, je je potřeba ho v průvodci nastavit jako oddělovač. Problém může být ještě s oddělením desetinných míst. V českém jazykovém prostředí se oddělovač z dat měření (vždy tečka, pravděpodobně kvůli csv formátu) neshoduje s českým oddělovačem desetinných míst (čárka) a je potřeba ho změnit v nastavení aplikace Excel.

Popis použitých čítačů

Sada čítačů dostupných pro sestavu se může lišit. Jejich seznam lze získat zadáním příkazu *typeperf -q -o counters.txt* do konzole systému Windows – v adresáři, kde se zrovna nacházíme, se vytvoří textový soubor *counters.txt* obsahující všechny dostupné čítače. Formát řádku souboru je ve tvaru *\název kategorie \ název čítače*. Ve výstupním souboru z měření jsou pak názvy použitých čítačů pojmenovány odpovídající sloupce. Tvar názvu sloupce je ve formátu *\\název počítače\název kategorie(název instance)\název čítače* (např. *\\MAC-BC-WIN10\Proces(WinfoMI)\% času procesoru*). V práci jsou použity čítače pokrývající tři základní hardwarové komponenty počítače – procesor, operační paměť a disk. Níže je jejich seznam s komentářem:

Procesor(_Total)\% času procesoru – celkové vytížení procesoru pro všechny jádra v procentech. Pro správnou interpretaci musí být jeho hodnota vydělena počtem jader, jedná se totiž o součet % vytížení všech jader (např. pro čtyřjádrový procesory tedy bude dosahovat až 400%). Při hodnotách 100% (po vydělení počtem jader) je potřeba zvážit, zda měření není ovlivněno nedostatečným výkonem procesoru.

Proces(název procesu)\% času procesoru – vytížení procesoru v procentech pro zvolenou instanci (proces). Jeho hodnotu není potřeba dělit počtem jader, jedná se již o upravenou hodnotu dosahující maximama 100%.

SQLServer:Memory Manager\Total Server Memory (KB) – počet KB v operační paměti obsazených procesy SQL Server.

SQLServer:Memory Manager\SQL Cache Memory (KB) – počet KB paměti v cache obsazených procesy SQL Server.

Paměť\% využití potvrzených bajtů – procento celkového obsazení operační paměti. Při hodnotách 100% je potřeba zvážit, zda měření není ovlivněno nedostatkem paměti.

Paměť\Potvrzené bajty – počet bajtů obsazené operační paměti pro všechny procesy.

Proces(název procesu)\Nesdílené bajty – počet bajtů operační paměti obsazených daným procesem.

Proces(název procesu)\Virtuální bajty – počet bajtů obsazených procesem v prostoru virtuálních adres. Obsahuje tedy počet bajtů operační paměti pro daný proces, počet paměti vyhrazený pro knihovny volané procesem a také bajty odložené stránkováním.

Fyzický disk(_Total) \ % času disku – jedná se průměrnou délku diskové fronty převedenou na procenta. Ta je vypočtenou podle Littlova zákona z teorie front a počítá se jako střední doba disku za přenos děleno počtem přenosů za s. Tento čítač se používá ve spolupráci s čítačem *Aktuální délka fronty disku*. Pokud *% času disku* vykazuje více než 100% a zároveň je aktuální délka fronty disku vyšší než dva, jedná se pravděpodobně o výkonnostní problém spojený s diskem.

Fyzický disk(_Total) \ Aktuální délka fronty disku – počet požadavků čekajících ve frontě ke zpracování diskem. Jedná se aktuální hodnotu naměřenou v danou chvíli.

Fyzický disk(_Total) \ Střední doba disku/čtení – průměrný čas, jaký disku trvá čtení z v danou chvíli. Hodnota je v sekundách (někde bývá udáváno v milisekundách, je potřeba se podívat do popisu čítače při jeho přidávání). Práh se může pro každou konfiguraci lišit, obecně by neměly hodnoty pro obyčejný pevný disk přesahovat 30 až 50 ms. Pro určení vhodného prahu na dané konfiguraci je možné vytvořit měření s tímto čítačem a následně nějaký časový úsek provádět běžné operace čtení disku (například procházet složky a soubory). Z vývoje hodnot se potom dá přibližně určit, jaký je práh pro daný disk.

Fyzický disk(_Total) \ Střední doba disku/zápis – průměrný čas, jaký disku trvá zapisování v danou chvíli. Stejně jako u předchozího čítače je prahovou hodnotu potřeba určit pro danou diskovou konfiguraci, obecně by se ale hodnoty neměly pohybovat mezi více jak 15 - 25 ms.

SQLServer:Buffer Manager \ Page reads/sec – počet stránek čtení disku procesy SQL Server. Hodnota slouží v návaznosti k předchozím diskovým čítačům – pokud zjistíme výkonnostní problém, můžeme se na tento čítač podívat, zda v danou chvíli probíhalo čtení procesy SQL Server.

SQLServer:Buffer Manager \ Page writes/sec – počet stránek zapisování na disk procesy SQL Server. Hodnotu opět slouží v návaznosti k předchozím diskovým čítačům.

[18] [9] [34] [2] [23] [32]

4.0.2 Visual Studio Profiler

Microsoft již několik let nabízí svůj vlastní profiler implementovaný přímo ve Visual Studiu. Jedná se o volitelnou součást instalace Visual Studia (u VS 2017 pod názvem

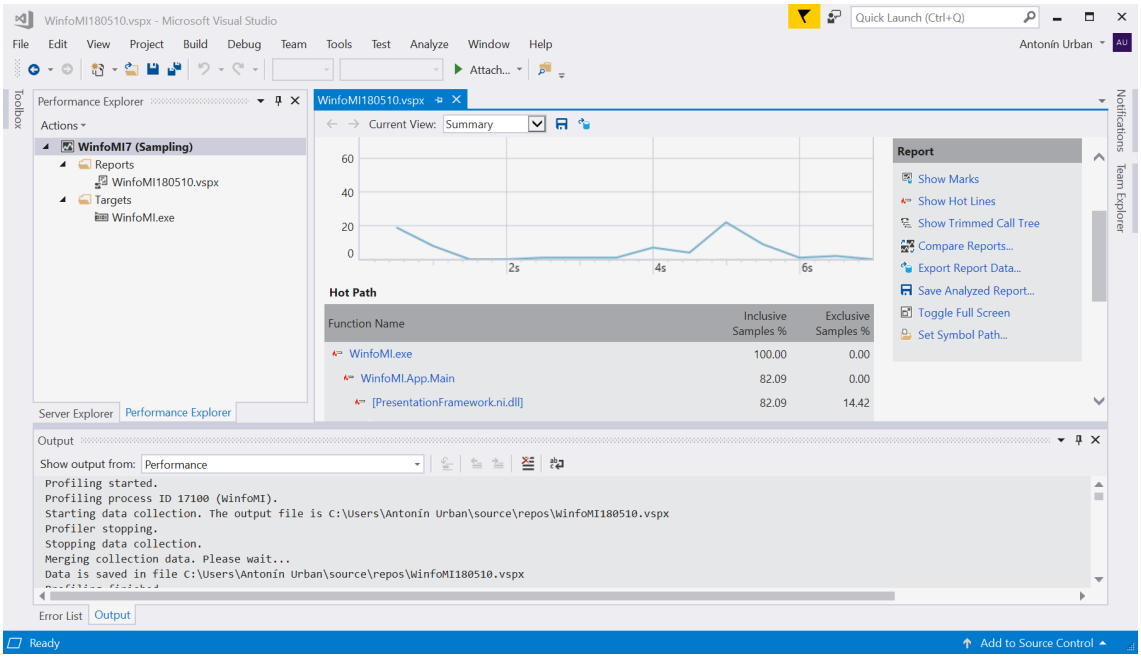
Nástroje pro profilaci .NET). Pokud je tato součást nainstalována, v menu pod záložkou *Debug* lze nalézt položku *Performance Profiler*, jenž spustí hlavní okno pro profilování. Profiler lze spustit bez nutnosti vytvářet nebo otevírat projekt – lze se připojit na již běžící .Net proces nebo definovat cestu ke spustitelnému souboru aplikace, kterou si profiler spustí. V případě, že je profiler spuštěn z otevřeného projektu, můžeme profilovat přímo otevřený projekt. Toto nastavení se volí hned po otevření okna profileru v menu vybrat cíl (Chose Target).

Po zvolení cíle měření se zobrazí podmenu s dostupnými nástroji (Available Tools), kde je ve výchozím stavu zaškrtnutá položka průvodce nastavením profileru (Performance Wizard). Tlačítkem *Start* volbu potvrdíme a průvodce spustíme. Zobrazí se dostupné možnosti profilování – samplování, instumentace, alokace paměti a profilování vícejádrových aplikací. Po vybrání metody se ještě v případě zvolení cíle jako spustitelného souboru objeví dvě okna s výběrem tohoto souboru. V poslední části pak můžem zvolit, zda se má profilování rovnou spustit a tlačítkem *Finish* ukončíme průvodce. Vlevo se otevře okno *Performance Explorer*, kde se vytvoří nová instance profileru, kterou můžeme pravým kliknutím a volbou vlastnosti (Properties) přenastavit nebo spustit (Start profiling). Ve vlastnostech se dají kromě parametrů z průvodce nastavit ještě další parametry, většina je dostupná pouze pro instrumentaci. Profilování skončí buď ukončením profilované aplikace, nebo ukončením profilování. Výsledek profilování je označován jako report a je uložen v samostatném souboru s příponou vspx.

Report má několik pohledů. Základní pohled *Summary* nabízí především graf s průběhem běhu aplikace z pohledu využití procent výkonu procesoru. U grafu je možné vybrat pouze určitou část průběhu aplikace a celý report vyfiltrovat pouze na tuto část. To se může hodit v případě, že na grafu vidíme velký skok ve využití procesoru měřenou aplikací – zobrazíme si pouze oblast s tímto skokem.

Dalším dostupným pohledem je pohled *Call Tree*, tedy strom volání funkcí. Funkce jsou zde seřazeny dle vytíženosti, jenž je reprezentována podle metody profilování – u samplování se jedná o počty naměřených vzorků v dané funkci, u instrumentace pak o čas aplikací strávený ve funkci. V obou případech jsou hodnoty vytíženosti rozděleny na inklusivní a exklusivní – tedy na hodnoty platící pro funkci včetně potomků a na hodnoty platné pouze pro danou funkci. Funkce lze rozkliknutím otevírat a odhalovat tak volané potomky. Ve stromu jsou také ikonkou plamene vyznačeny takzvané „horké“ cesty (Hot Paths) – tedy cesty, o kterých si profiler myslí, že jsou velmi vytížené a mohlo by být dobré je zkontrolovat.

Významný je také pohled *Functions*, zobrazující všechny změřené funkce. Pomocí řazení v tomto pohledu získat přehled o funkcích s nejvyšším počtem exklusivních a inklusivních vzorků. Podobný je pohled s moduly (Modules), místo funkcí zobrazuje DLL knihovny (včetně základních platformy .NET). Pohled s detaily funkcí (Function Details) zobrazuje informace o vybrané funkci – graficky je znázorněn



Snímek 2: Okno Visual Studio Profileru s otevřeným reportem.

rodič a potomci funkce. V případě, že se jedná o funkci z vlastního kódu aplikace (tedy ne funkci DLL knihovny), ve spodním okně se zobrazí implementace kódu funkce. Do pohledu se dá dostat z jiných pohledů pravým kliknutím na vybranou funkci a volbou *Show Functions Details*. Podobný je také pohled *Caller/Callee* zobrazující také rodiče a potomky, oproti pohledu detailu funkcí ale zobrazuje všechny rodiče a všechny potomky.

V tabulkových pohledech (jako je například pohled s funkcemi nebo strom volání) se dají kliknutím do hlaviček sloupců přidávat nebo odebírat sloupce a zobrazovat tak další informace o řádcích. Data z tabulkových pohledů se také dají exportovat do csv nebo xml souboru kliknutím na příslušnou ikonku nacházející se vpravo od volby pohledu. Reporty se také dají porovnávat, kdy profiler vytvoří seznam funkcí nebo modulů společně s rozdíly mezi vzorky.

Hlavní sledovanou hodnotou při analýze by měl být počet vzorků u samplování, respektive strávený čas v metodě pro instrumentaci. U instrumace může být ještě vhodné sledovat počet volání funkce (Number of Calls). Trochu matoucí může občas být počet exklusivních vzorků (nebo exklusivního času). Profiler se totiž u funkcí používajících nějakou knihovnu třetích stran „zanoří“ do takové knihovny a exklusivní vzorky načte pro její vnitřní funkce. Kvůli tomu při seřazení podle počtu exklusivních vzorků dostáváme funkce, jenž nejsou součástí vlastního kódu aplikace. Proto je lepší analýzu provádět přes inklusivní vzorky spolu s analýzou potomků a rodičů. Tím dokážeme stanovit, jaké funkce jsou konečné pro kód aplikace – jejich počet exklusivních vzorků se dá pak považovat za inklusivní.

4.0.3 DotTrace

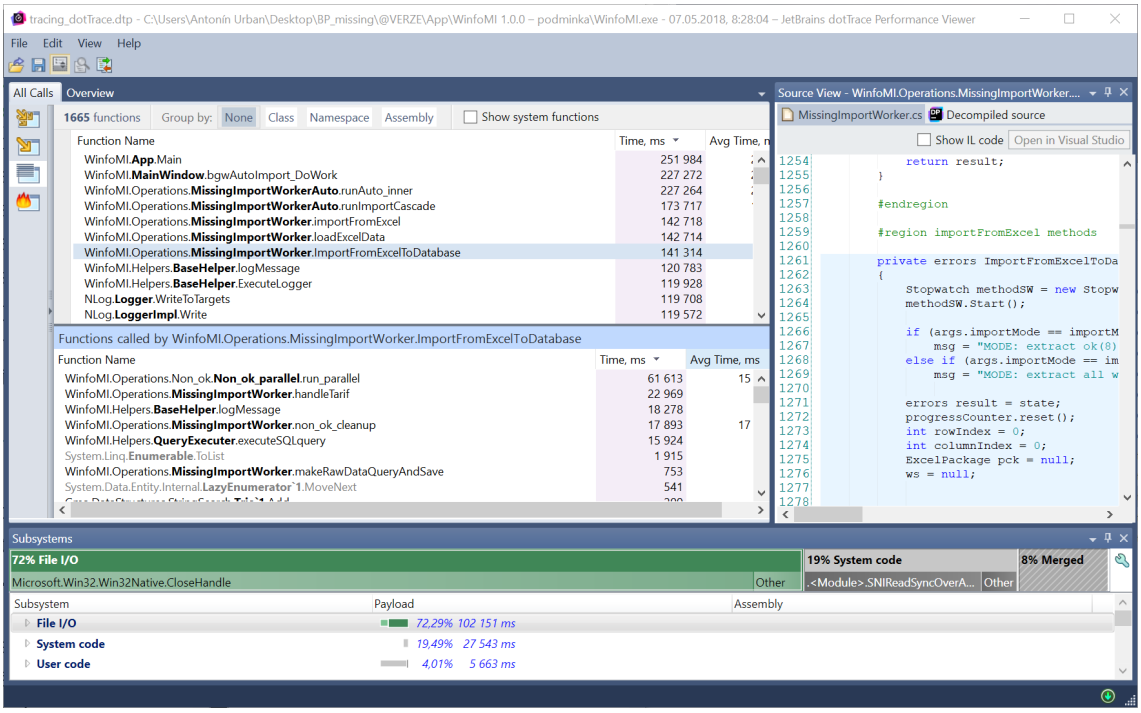
DotTrace je komerční profilovací nástroj od společnosti JetBrains. Pro akademické účely je studentům nabízen zdarma.

Po spuštění profileru vybíráme v hlavním okně vlevo cíle měření – na výběr je běžící proces (Attach to Running App) nebo lokální aplikace (Profile local app). Případně je také možnost se připojit ke vzdálené aplikaci přes síť (Profile Remote App). Volba lokální aplikace nabízí několik možností, jak aplikaci vybrat, základní je výběr spustitelného souboru z disku (Standalone). Vpravo pak vybíráme pofilovací metodu. Tři odpovídají metodám z Visual Studia – samplování (Sampling), instrumentaci (Tracing) a profilování vícevláknové aplikace (Timeline). Metoda řádek po řádku (Line-by-line) je rozšíření instrumentace, kdy profiler dokáže určit čas strávený ve funkci, počet volání funkce a navíc ještě počet volání jednotlivých řádků. To může být užitečné v případě, že chceme detailně profilovat implementaci určité funkce obsahující například smyčky – můžeme tak odhalit, že nějaká část algoritmu se vykonávala vícekrát, než bychom čekali. Po zaškrtnutí políčka *Advanced* v pravém horním rohu se zobrazí další možnosti nastavení profilování. Zajímavá je hlavně volba režimu, jak má profiler zjišťovat čas strávený ve funkci (dotTrace i u samplování používá místo vzorků čas strávený ve funkci). Můžeme si zvolit, zda má být čas měřen instrukcemi procesoru, nebo výkonnostními čítači a zda chceme z měření vyřadit čas, po který vlákno spalo nebo čekalo. Pro instrumentační metody je možnost zapnout preciznější mód, kdy se profiler snaží při měření nezapočítávat časy potřebné pro přepínání mezi kódem aplikace a profilerem – výsledky jsou přesnější, projevuje se ale ještě větší zpomalením aplikace, než u klasické instumentace. [30] [4]

Po spuštění měření se zobrazí malý dok s možností kdykoliv pozastavit aplikaci a udělat snímek (Snapshot) – tedy vytvořit report z dosud naměřených dat. Toto je výhoda oproti Visual Studio Profileru, jelikož můžeme už v průběhu měření zaznamenat různé části/činnosti měření aplikace. Profiler nabízí k ovládání i vlastní API a je možné ho ovládat i z kódu aplikace (spustit měření, vytvořit snímek apod.).

Report z měření nabízí záložky přehled (Overview) a všechna volání (All calls). První záložka obsahuje graf vytížení procesoru a informace o konfiguraci, měřené aplikaci a nastavení měření. Druhá záložka nabízí pohledy podobné těm ve Visual Studio Profileru. Jedná se o strom volání vláken (Threads Tree), strom volání funkcí (Call Tree), list funkcí (Plain List) a vytížená místa doporučená k analýze (Hot spots).

Stejně jako Visual Studio, umožňuje dotTrace porovnávat reporty, naopak neumožňuje data z pohledů exportovat do csv souboru. Obecně dotTrace zobrazuje méně informací, než dokáže zobrazit profiler ve Visual Studiu, nechybí mu však důležité informace. Uživatelské prostředí se dá považovat za přívětivější.



Snímek 3: Okno profileru dotTrace s otevřeným reportem.

4.1 Měření a optimalizace SQL server

Tato kapitola se zabývá možnostmi sledování výkonu SQL Server databází a oficiálním optimalizačním nástrojem pro tuto platformu – Database Engine Tuning Advisor.

4.1.1 Měření výkonu SQL Server

Pokud nám stačí monitorovat SQL Server v reálném čase, užitečným nástrojem pro sledování výkonu SQL Server může být nástroj Activity Monitor, dostupný přes SSMS (pod zkratkou Ctrl+Alt+A). Nástroj nabízí několik pohledů:

Přehled (Overview) – okno obsahuje 4 grafy zobrazující čítače související s SQL Serverem (% vytížení procesoru, počet čekajících úkolů, počet MB/s vstupně výstupních operací a počet dávek TSQL příkazů za sekundu).

Procesy (Processes) – informace o procesech patřících k činnosti SQL Server.

Zdroje čekání (Resource Waits) – přehled všech možných zdrojů čekání SQL Server.

I/O souborů dat (Data file I/O) – přehled o vstupně výstupních operacích jednolivých databází (MB čtení za sekundu, MB zapisování za sekundu, doba odezvy v ms).

Nedávné pomalé příkazy (Recent Expensive Queries) – zobrazuje nedávno doběhlé SQL příkazy. Pravým kliknutím na příkaz ho lze upravovat, nebo zobrazit jeho exekuční plán.

Aktivní pomalé příkazy (Recent Expensive Queries) – zobrazuje právě běžící SQL příkazy.

Nevýhodou nástroje je nemožnost ukládat data do souboru. Data zobrazená v seznamech se dají řadit nebo filtrovat podle hodnoty vybraného sloupce. Nástroj také nenabízí téměř žádné možnosti nastavení, jedinou volbou je čas měřícího intervalu (pravý klik na jeden z grafů a volba *Refresh interval*).

Podobné monitorovací nástroje lze najít jako komerční – například SQL Monitor od společnosti Redgate Software (součást balíku nástrojů SQL Toolbelt), ApexSQL Monitor, dbForge Studio a podobně. Speciálním nástrojem, použitelným pro některá měření je také SQL Server Profiler měřící na základě událostí. Poslední možností měření výkonu SQL Server jsou stejně jako v případě platformy .NET výkonnostní čítače – SQL Server má mezi čítači vlastní kategorie obsahující dohromady několik set různých čítačů.

4.1.2 Database Engine Tuning Advisor

Microsoft SQL Server k optimalizaci nabízí nástroj, jenž dokáže navrhnout vhodné zavedení indexů, statistik a rozdělení tabulek v databázi – Database Engine Tuning Advisor (DTA). Nástroj je k dispozici s grafickým uživatelským prostředím, případně se dá používat i z příkazového řádku systému Windows.

Po spuštění programu je automaticky otevřené okno k vytvoření nového ladění (v programu označené jako *session*). Na hlavní záložce je možnost upravit jméno ladící instance a vybrat pro analýzu relevantní databáze. Lze případně zaškrtnout jen určité tabulky. Podstatným krokem ke spuštění analýzy je stanovit zdroj zatížení, pro které chceme databázi optimalizovat (Workload). Na výběr je několik možností:

Soubor (File) – slouží k otevření souborů se stopou činnosti serveru. Takový soubor je možné nechat vygenerovat pomocí programu SQL Server profiler.

Tabulka (Table) – umožňuje připojit se na databázovou tabulku se stopami činnosti serveru. K získání dat pro tabulku je možné opět použít SQL server profiler. Při průběhu ladění není možné tabulku plnit daty.

Plan Cashe – umožňuje spustit optimalizaci bez předem vygenerované zátěže – při použití této volby DTA použije prvních 1000 událostí činnosti SQL Serveru a poté spustí analýzu.

Query store – *query store* je funkce shromažďující automaticky historii dotazů, statistik a plánů z činnosti SQL serveru. Pokud je tato funkce pro cílovou databázi zapnuta, je možné výslednou historii použít jako zátěž pro DTA.

Další záložka obsahuje nastavení ladění (Tuning Options). První možnost umožňuje zastavit analýzu v určitý den a čas, což může být výhodné u velkých databází nebo pro velký zdroj zátěžových dat. Analýza se sice dá zastavit manuálně, ale samotné zastavení a především pak vyhodnocení a určení výsledků může několik dalších minut trvat. Pokud tedy očekáváme výsledek do určitého termínu a zároveň předpokládáme dlouhou analýzu, je možné si naplánovat ukončení před termínem touto volbou.

Zbylé možnosti nám určují, jaká doporučení na konci od DTA dostaneme. Základní volbou jsou indexy, na výběr jsou i indexované pohledy, případně pouze neclusterované indexy. Jednou z možností je i optimalizace již vytvořených indexů a indexovaných pohledů za předpokladu, že nehodláme další nové přidávat. Dvě další přidružená zaškrtávací pole nám umožňují do doporučení zahrnout také filtrované indexy a sloupcové indexy.

Další nastavení zahrnuje doporučení rozdělení velkých tabulek na menší a poslední položkou nastavení pro DTA definujeme, jaké existující typy optimalizačních struktur si přejeme v rámci doporučení zachovat. Pokud máme podezření, že cílová databáze obsahuje nevhodné nebo špatně nastavené optimalizační struktury, můžeme je tímto nastavením označit k analýze. DTA následně doporučí smazání struktur vyhodnocených jako neužitečné.

Ladění nabízí ještě několik pokročilých nastavení. Uživatel může nadefinovat maximální velikost paměti, jakou je ochoten obětovat pro optimalizaci databáze – vytvoření indexů a indexovaných pohledů zvětší výslednou velikost databáze. V případě nedostatku místa může být vhodné tuto skutečnost zahrnout do ladění. Pokud není zadána tato hodnota uživatelem, DTA vybere menší z těchto dvou hodnot:

- trojnásobek velikosti dat databáze
- volné místo na všech připojených discích společně s velikostí databáze.

Další pokročilá volba upravuje maximální počet sloupců na jeden index. Pomocí poslední volby může uživatel stanovit, jestli výsledná doporučení musí být aplikovatelná za běhu databáze (i za cenu potencionálně horších výsledků optimalizace). Takové nastavení dává smysl v případech, kdy je nežádoucí nebo složité databázi vypnout.Při výběru očekávaných doporučení je potřeba myslet na fakt, že každé dodatečné nastavení může prodloužit čas potřebný k vyhodnocení analýzy. Pokud tedy nechceme na vyhodnocení zbytečně čekat, je vhodné nastavit pouze takové hodnoty, jaké mají pro cílovou databázi smysl.

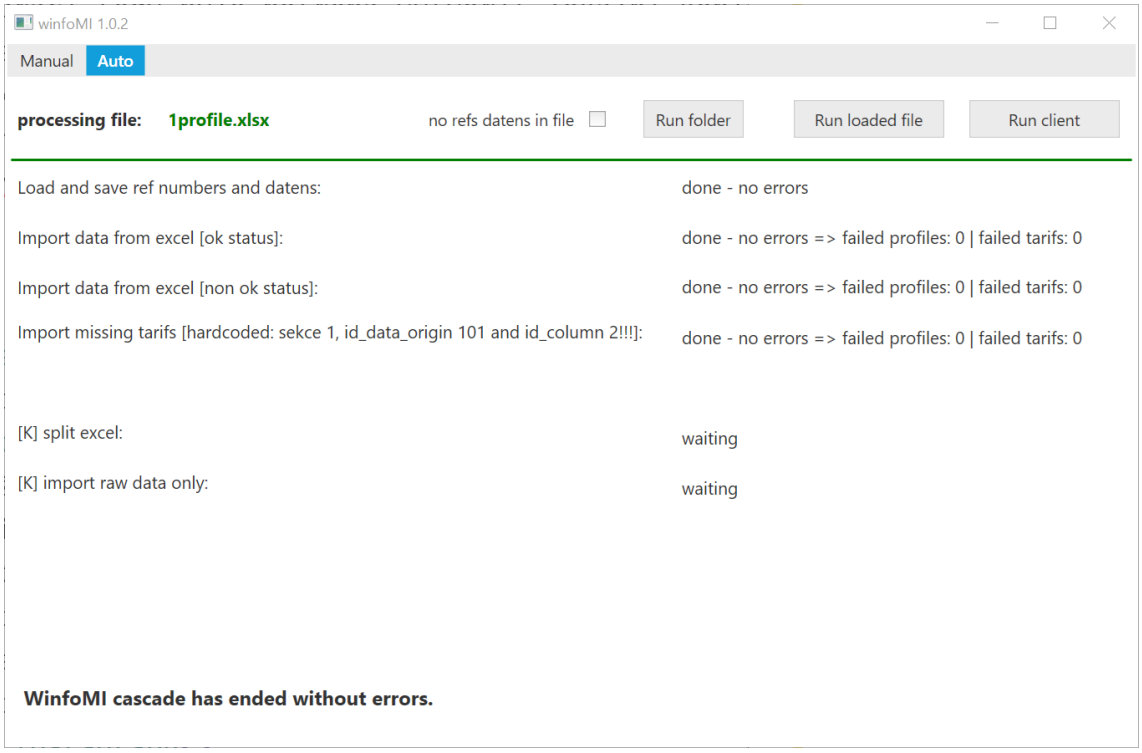
Po spuštění analýzy se zobrazí další záložka s průběhem všech fází ladění. Po skončení se pak přidají dvě poslední záložky – doporučení (Recommendations) a zpráva z měření (Reports). Záložka doporučení obsahuje seznam všech vygenerovaných doporučení a umožňuje je analyzovat, nebo rovnou použít. Případně je možné si doporučení pouze uložit, prohlídnout vygenerovaný SQL skript a podobně. Doporučení lze aplikovat okamžitě, nebo naplánovat na určitý termín.

5 Měření a optimalizace vzorové aplikace a databáze

V této části práce je popsáno měření a optimalizace na vzorové aplikaci a vzorové databázi. Kapitola se nejdříve věnuje měření a optimalizaci SQL Server databáze. V druhé části kapitoly je pak využito profilerů k optimalizaci aplikace. Naměřená data, výpočty a grafy jsou na přiloženém CD ve složce *Data z měření*.

5.1 Popis vzorové aplikace

Aplikace vybraná k měření se jmenuje WinfoMI a byla poskytnuta pro tuto práci firmou Winfo s.r.o. Jedná se o aplikaci k internímu použití. Základní funkcí aplikace je ověřování a doplňování dat v databázi ze zdrojových Excel souborů.



Snímek 4: Hlavní okno aplikace WinfoMI získané k demonstraci měření.

Ve praxi se WinfoMI používá v kombinaci s databází MySQL, před použitím

pro tuto práci tedy bylo potřeba vytvořit lokální kopii části originální databáze v SQL Server a následně také upravit vlastní aplikaci kvůli funkčním a syntaktickým odlišnostem mezi oběma DB systémy. Aplikace je napsaná jazyce C# pro verzi .NET 4.5 a pro komunikaci s databází využívá Entity Framework. Grafické rozhraní aplikace je vytvořeno ve WPF. Pro čtení a zápis Excel souborů aplikace využívá open source knihovnu EPPlus.

5.1.1 Struktura dat zpracovávaných aplikací

Základní stavební jednotky dat pro aplikaci jsou objekty *profil* a *tarif* s vazbou 1:N. Řádově má jeden profil několik stovek tarifů, nejběžněji v rozpětí 300 – 500. Samotný tarif i profil nesou pouze několik vlastností včetně identifikačního čísla. Ostaní parametry těchto objektů jsou pak zaneseny v tabulkách s vlastnostmi profilů a tarifů. Obě tyto pomocné N:M vazební tabulky mají přiřazený číselník s identifikátorem a popisem vlastnosti. Zjednodušeně se dá říct, že aplikace ověřuje a doplňuje data právě v těchto vazebních tabulkách, především pak pro objekty typu tarif.

Vstupní Excel soubory mají běžně dva sešity, kdy první obsahuje profily, druhý pak tarify. Vždy platí, že jeden řádek v Excelu je roven jednomu profilu nebo tarifu podle typu sešitu. Problémem vstupních dat je, že používají jiné hodnoty identifikátorů, než mají data v databance. Kvůli tomu je potřeba data správně identifikovat a spojit s již existujícími v databázi na základě určitých pravidel. Databázová struktura po převedení do SQL Server čítala 23 tabulek a dva pohledy. Pro představu o databázovém modelu je jeho diagram, vygenerovaný ve Visual Studiu, vidět na snímku č. 9 v přílohách.

Ověřování a import dat se dá rozdělit na 4 samostatné, na sebe navazující, operace. Každá operace se dá ručně spustit bez nutnosti zůstat v jedné instanci aplikace, veškerá návaznost dat je realizována v databance namísto vnitřní paměti aplikace. Je tedy možné otevřít 4 nezávislé instance WinfoMI a postupně spouštět jednotlivé operace. Jedinou podmínkou správného zpracování dat je dodržení návaznosti operací – tedy spustit další operaci nad jedním souborem až po úspěšném dokončení operace předcházející. Aplikace také nabízí automatický režim, který se postará o spuštění všech operací ve správném pořadí. První operace nemá z pohledu měření velký význam, protože není náročná na výkon a pracuje s výrazně menší množinou dat než ostatní operace. Úkolem je spojit data profilů s existujícími profily v databázi pomocí jednoduché vazby. Propojení dat je pak realizováno zanesením vlastnosti, obsahující identifikátor dat z Excelu, do databáze. Tento identifikátor se využívá pro další operace zpracovávající tarify, což jsou druhá a třetí operace. Tyto dvě operace zpracovávají tarify podle vlastnosti *status* – každá z těchto dvou operací zpracovává tarify pro jednu sadu hodnot statusu, zpracování dat se liší. Poslední, čtvrtá operace, klonuje a doplňuje tarify společně s vlastnostmi podle dat zpracovaných v třetí operací. V průběhu importování a ověřování dat aplikace zapisuje do logu informace o právě prováděných operacích kvůli případné kontrole algoritmu. Z druhé operace aplikace vytváří výstup ve formátu xlsx.

5.2 Optimalizace SQL server pro vzorovou aplikaci

V této kapitole je popsán postup a výsledky optimalizace SQL server pro vzorovou databázi missing_db pro použití s aplikací WinfoMI.

5.2.1 Nahrazení funkce INSERT OR UPDATE

Podmět k optimalizačnímu kroku databáze vznikl při její konverzi z původní MySQL databáze na lokální SQL Server databázi. Po převedení struktury a naimportování části dat bylo potřeba také přepsat některé SQL příkazy přímo v aplikaci kvůli syntaktickým rozdílům mezi SQL Server a MySQL. Mezi nekompatibilní zápisy patřil poměrně často volaný příkaz využívající funkci INSERT OR UPDATE. Cílem zápisu bylo vkládat pouze neexistující záznamy, případně existující aktualizovat.

Pro přepis do SQL server nebyla nalezena přímá alternativu funkce INSERT OR UPDATE (viz zdrojový kód č. 1). Nabízela se dvě řešení – použít podmínku přímo v příkazu INSERT nebo použít trigger nad cílovou tabulkou [1]. Pro změření vhodnější varianty byl použit SQL Server Profiler s nastavením sledování dokončení jedné dávky TSQL příkazů (SQL:BatchCompleted). Po doběhnutí sledování byla data vyexportována do tabulky v databázi (jiné než sledované). SQL Server Profiler totiž neumožňuje jednou získaná data filtrovat ani exportovat do csv souboru, exportem do tabulky se dá tento nedostatek obejít, kdy můžeme data filtrovat pomocí příkazu SELECT. Stačilo tedy jen vyfiltrovat všechny stejné dávky obsahující požadovaný příkaz (hledané dávky vždy začínaly specifickým textovým řetězcem) a zaznamenat výsledné časy. Měření proběhlo pro každý případ 3x a výsledné hodnoty jsou aritmetickým průměrem hodnot z těchto tří měření.

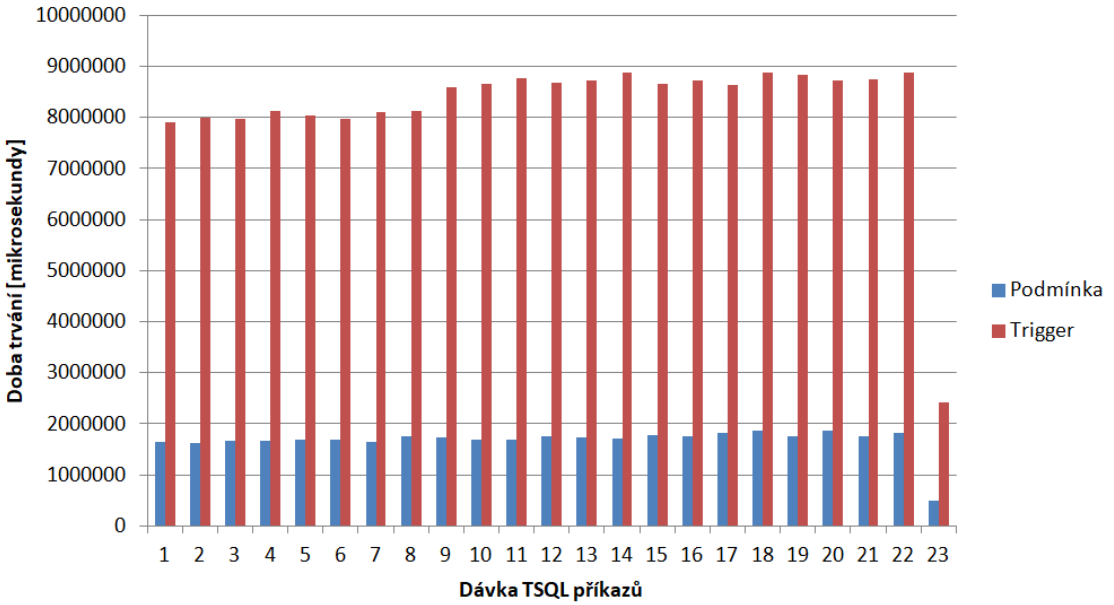
```
1 INSERT
2 INTO [tarif_ma_vlastnosti]
3 ([id_tarif], [id_tarif_vlastnosti], [hodnota])
4 VALUES
5 (id_tarif, id_tarif_vlastnosti,hodnota)
6 ON DUPLICATE KEY
7 UPDATE [hodnota] = VALUES (hodnota);
```

Zdrojový kód 1: Schématický zápis původní INSERT OR UPDATE funkce v MySQL

Graf na snímku číslo 5 porovnává obě varianty a ukazuje, že verze s podmínkou byla rychlejší. Velký rozdíl mezi časy odpovídal i rozdílu v rychlosti celé aplikace – verze s triggerem byla výrazněji pomalejší než verze s podmínkou. Porovnání aritmetického průměru z 10 importů jednoho profilu je vidět v tabulce číslo 1. Hodnoty byly získány přímo z aplikace, při importu kód zaznamenává časy jednotlivých částí a po skončení pak aplikace zobrazuje celkový uplynutý čas od spuštění importování.

[1] Viz zdrojové kódy č. 4 a 5 v přílohách.

Pravděpodobným důvodem rozporu mezi časy byl fakt, že trigger plošně ovlivnil veškeré INSERT příkazy nad tabulkou tarif_ma_vlastnosti, zatímco verze s podmínkou upravila pouze onu jednu funkci v kódu. Podmínka tedy platila pro příkazy INSERT vytvořené skrze tuto funkci, zároveň ale ne všechny tyto příkazy z tohoto místa pocházely. Pro některá data, vkládaná do této tabulky, neplatil totiž předpoklad, že záznam už může existovat a nebylo tedy funkci INSERT OR UPDATE potřeba používat.



Snímek 5: Graf porovnávající časy dávek TSQL příkazů pro případ s použitím triggeru a případ s použitím podmínky – použití podmínky je výrazně rychlejší.

Tabulka 1: Aritmetický průměr z 10 nezávislých importů pro každou z verzí společně s maximální a minimální hodnotou. Verze s podmínkou je výrazně rychlejší.

	s triggerem [ms]	**s podmínkou [ms]**
aritm. průměr	443 348	184 615
maximum	461 651	193 143
minimum	424 780	178 789

5.2.2 Optimalizace SQL databáze

Po naměření časů pro obě varianty nad neoptimalizovanou databází bylo potřeba ji optimalizovat a zjistit, jak se situace mezi verzemi změní. Pro optimalizaci byl použit nástroj Database Engine Tuning Advisor, který je součástí balíku nástrojů pro práci s databázemi SQL Server.

Samotné optimalizaci přes DTA ještě předcházel důležitý krok zazálohování výchozího stavu databáze přes SSMS. Díky tomu bylo možné databázi po optimalizaci jedné verze vrátit zpět do výchozího stavu a následně jí optimalizovat pro druhý případ.

Aplikování DTA na testovací databázi

Optimalizace probíhala pro oba případy stejně. Prvním krokem bylo vytvoření zátěže v SQL Server Profileru. Nastavení profileru je v přílohách na snímku číslo 10. Jako šablona je vybráno ladění (Tuning), soubor se stopou je rovnou i uložen na disk. Limit velikosti souboru je nastaven na 100 MB, což je hodnota, kterou profiler v tomto případě nemohl překročit. Pokud by limit zůstal na původní hodnotě, byla by stopa rozdělena na několik souborů (pro toto měření měla stopa přibližně 12MB). V DTA by pak bylo potřeba vytvořit ladění pro každý soubor.

S vytvořenou zátěží mohla proběhnout vlastní optimalizace v DTA. Jako očekávané struktury k doporučení byly nastaveny indexy s možností filtrovaných indexů. Původním záměrem bylo nastavit možnost indexů i indexovaných pohledů, nicméně při této možnosti optimalizace nikdy nedoběhla. Nástroj vždy po několika desítkách minut činnosti spadl bez udání důvodu. Výsledné nastavení DTA je zobrazeno v přílohách na snímku č. 11.

V tabulce č. 2 je vidět porovnání několika hodnot z výsledku optimalizace pro oba případy. Doba optimalizace je relativně podobná, rozdíl je především v počtu navržených struktur a odhadovaném zrychlení. U triggeru DTA odhaduje větší zrychlení než u podmínky. O výsledném zrychlení tato hodnota mnoho neprozrazuje, protože trigger byl několikanásobně pomalejší na neoptimalizované databázi. V příloze na snímcích č. 12 a 13 je vidět okno s návrhy optimalizace pro oba případy.

Tabulka 2: Několik statistik z výsledku ladění v DTA.

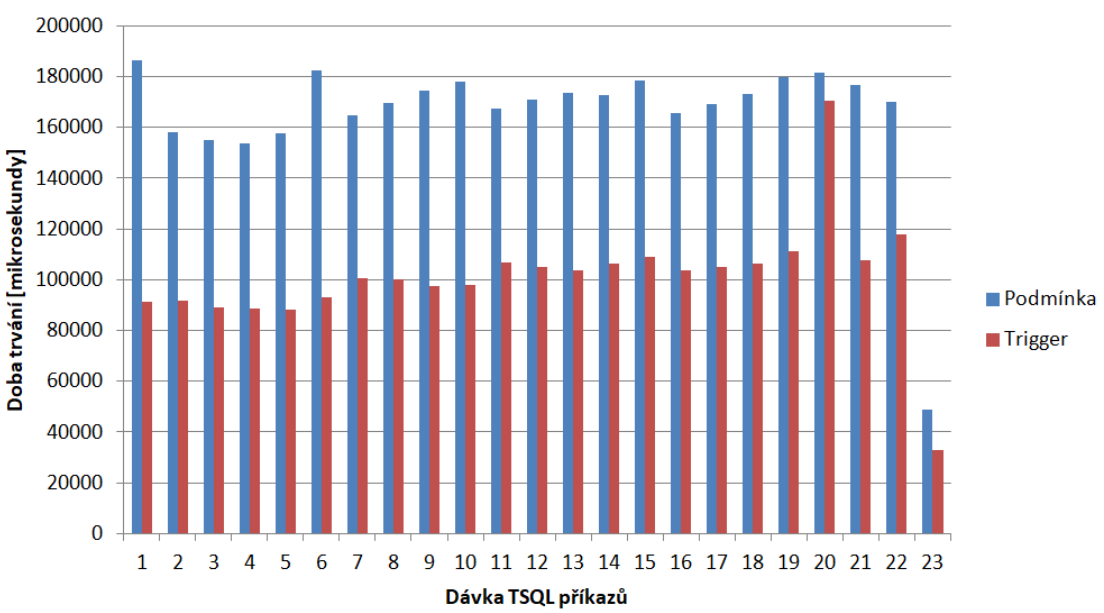
	trigger	**podmínka**
doba ladění [hh:mm]	02:23	02:02
předpokládané zrychlení [%]	64.95	23.00
počet navržených indexů	3	1
počet navržených statistik	15	7

5.2.3 Měření SQL Server a databáze

Díky zazálohování všech tří stavů databáze (před optimalizací, optimalizace pro trigger, optimalizace pro podmínku) bylo možné libovolně přepínat mezi stavy databáze a provádět měření v libovolném pořadí.

Dávky TSQL příkazů

Prvním měřením bylo doplnění časů pro dávky TSQL příkazů (viz kapitola 5.2.1) a probíhalo stejně jako ve zmiňované kapitole. Graf na snímku č. 6 porovnává oba případy na optimalizovaných databázích. Grafy v přílohách na snímcích č. 14 a 15 ukazují zrychlení triggeru a podmínky před a po optimalizaci.



Snímek 6: Graf se srovnáním časů dávek TSQL příkazů po optimalizaci databáze pro každý z případů. Podmínka je navzdory výchozímu stavu pomalejší než trigger. Před optimalizací tomu bylo naopak.

Tabulka č. 3 ještě srovnává zrychlení odhadované pomocí DTA a dopočítané zrychlení dávek měřených příkazů. Měřené dávky dopadly výrazně lépe než odhadovaný čas. Důvodem je to, že DTA počítá zlepšení přes všechny (jednotlivé) příkazy obsažené v zátěži, zatímco vypočtené průměrné zlepšení platí pouze pro měřených 23 dávek.

Tabulka 3: Porovnání odhadovaného zrychlení z DTA pro celou databázi (všechny TSQL příkazy v zátěži) s vypočteným průměrným zrychlením u měřených dávek.

[%]	**trigger**	**podmínka**
odhadované celkové zrychlení	23.00	64.95
vypočt. průměr zrychlení měřených dávek	98.77	90.14

Z porovnání obou měřených verzí vyplývá, že původně pomalejší trigger (viz snímek č. 5) je po optimalizaci z pohledu sledovaných dávek rychlejší. Co se týče časů importování naměřených přímo v aplikaci, trigger je opět pomalejší, jak je vidět v tabulce č. 4. Rozdíl již není tak dramatický, jako před optimalizaci – časy se dají považovat za velmi podobné.

Tabulka 4: Aritmetický průměr z 10 importů puštěných po optimalizaci databáze, společně s maximální a minimální hodnotou. Podmínka je pořád rychlejší, jako před optimalizací, rozdíl je ale nyní poměrně malý.

	s triggerem [ms]	**s podmínkou [ms]**
aritm. průměr	38 803	36 645
maximum	43 6719	41 647
minimum	34 175	31 265

Z pohledu srovnání verzí před a po optimalizaci databáze je vidět řádové zlepšení. Zatímco u staré verze se časy dávek pohybovaly v řádu 10^6 mikrosekund, po optimalizaci tyto časy spadly na hranu řádů 10^5 a 10^4.

Měření výkonu SQL Server a aplikace

Druhé měření bylo zaměřeno na výkon SQL Serveru a aplikace – cílem bylo zjistit, jak se optimalizace podepsala na výkonu testovací soustavy a zda z pohledu výkonu byly mezi oběma případy výrazné rozdíly.

K měření se v obou případech jako nejlepší varianta jevilo použít čítače systému Windows, především proto, že umožňují ukládat výstup ve vhodném tvaru. Níže je seznam vybraných čítačů pro SQL Server:

1. Procesor(_Total)\% času procesoru
2. Proces(sqlservr)\% času procesoru
3. Proces(SQLAGENT)\% času procesoru

4. Proces(sqlceip)\% času procesoru
5. Proces(sqlceip#1)\% času procesoru
6. Proces(sqlwriter)\% času procesoru
7. SQLServer:Memory Manager\Total Server Memory (KB)
8. SQLServer:Memory Manager\SQL Cache Memory (KB)
9. Paměť\% využití potvrzených bajtů
10. Fyzický disk(_Total)\% času disku
11. Fyzický disk(_Total)\Aktuální délka fronty disku
12. Fyzický disk(_Total)\Střední doba disku/čtení
13. Fyzický disk(_Total)\Střední doba disku/zápis
14. SQLServer:Buffer Manager\Page reads/sec
15. SQLServer:Buffer Manager\Page writes/sec

K sledování procesoru sloužili čítače 1 až 6. Čítač 1 sledoval celkové procento vytížení procesoru, zbylých 5 čítačů pak patřilo jednotlivým procesům SQL Serveru. Jejich součtem bylo získáno celkové vytížení procesoru činností SQL Serveru. Čítač jedna pak sloužil především jako ukazatel případného přetížení procesoru. Pokud by k němu došlo, měření by jím bylo negativně ovlivněno. Čítače 7 a 8 sledovali v KB obsazení paměti SQL Serverem, čítač 9 pak procento celkového obsazení operační paměti. Poslední sada čítačů se věnovala pevnému disku. Dvě dvojice čítačů 10,11 a 12,13 sloužili k odhalení případného problému s pomalým diskem. Poslední dva čítače měly v případě takového problému odhalit, zda za ním mohl stát SQL Server.

K měření výkonu aplikace již nebylo potřeba měřit vytížení disku, bylo pokryté v měření výkonu SQL Serveru. Sada čítačů se změnila na:

1. Proces(WinfoMI)\% času procesoru
2. Procesor(_Total)\% času procesoru
3. Proces(WinfoMI) \Nesdílené bajty
4. Proces(WinfoMI)\Virtuální bajty
5. Paměť \Potvrzenébajty
6. Paměť\ % využitípotvrzených bajtů

V seznamu je vidět, že bylo potřeba použít více čítačů z kategorie proces. Čítač 1 a 2 opět měří procesor, zbylé čítače se starají o operační paměť.

Pro vlastní měření byl použit systémový nástroj Sledování výkonu, kdy byla vytvořena nová sada kolekcí dat. Hlavními body nastavení (kromě navolení čítačů) byla především volba výstupu – pro zpracování v tabulkovém procesoru bylo nejlepší nastavit výstup oddělený tabulátorem. Interval měl být nejmenší možný, tedy jedna sekunda. Měření probíhalo jako třikrát po sobě jdoucí import jednoho profilu. Po skončení měření nástroj Sledování výkonu vytvořil ve zvolené složce soubor s koncovkou tsv. Po změně koncovky na csv šel soubor otevřít v programu Excel. Následovalo převedení textových dat do sloupců a úprava hodnot pomocí vzorců (vydělení čítače *Procesor(_Total) \ % času procesoru* počtem jader, sečtení procent čítačů všech procesů SQL Server a pod.). Nakonec byly z upravených dat vytvořeny grafy.

Výsledky

Níže je srovnání výsledků měření pro tři měřené hardwarové komponenty – tedy procesor, operační paměť a pevný disk.

Procesor před optimalizací dosahoval pro trigger i podmínku podobných okamžitých hodnot, jak je vidět na snímcích č. 16 a 17 v přílohách. Nejčastěji se vytížení procesy SQL Serveru drželo kolem 25% výkonu, několikrát se na jednotky sekund vyšplhalo až k 90%. V součtu bylo vytížení u triggeru horší vzhledem k vyššímu času importování. Doby vyššího vytížení trvaly v obou případech stejně, tedy přibližně kolem 20s. Celkové vytížení procesoru všemi procesy bylo větší v případě triggeru, což by mohlo znamenat vyšší zátěž z aplikace. Měření aplikace pro trigger i podmínku (snímky č. 29 a 30 v přílohách) vykazovalo z pohledu procesoru velmi podobné hodnoty, vyšší celkové vytížení je tedy v případě triggeru pravděpodobně způsobeno nestejnými podmínkami měření. Po optimalizaci databáze vytížení procesoru SQL Serverem (snímky č. 18 a 19 v přílohách) v obou případech výrazně kleslo – drželo se kolem nuly s několika skoky do 25%. Vývoj grafů se ale po optimalizaci pro oba případy dá považovat za velmi podobný, z pohledu vytížení procesoru nelze jednoznačně určit, zda byla lepší podmínka nebo trigger. Dopad na výkon CPU z hlediska aplikace se po optimalizaci databáze téměř nezměnil (snímky č. 31 a 32 v přílohách).

Průměrné hodnoty obsazení operační paměti jsou vidět v tabulce č. 5. Z pohledu aplikace se obsazená paměť RAM nepatrně snížila, mezi verzemi velký rozdíl nebyl. V případě databáze se optimalizací zvýšilo obsazení operační paměti pro trigger a snížilo pro podmínku. Závěrem tedy je, že v případě paměti RAM je lepší použití podmínky.

Poslední část měření se týkala pevného disku a proběhla jen ve fázi měření SQL Server, protože se jednalo o obecné měření. Cílem bylo určit, zda disk nezpůsobuje zpomalení celé aplikace. K vyhodnocení sloužili 4 grafy, jenž jsou pro všechna měření

Tabulka 5: Průměrné hodnoty obsazení operační paměti pro všech 8 měření.

[MB]	před optimalizací		po optimalizaci	
	trigger	**podmínka**	**trigger**	**podmínka**
průměr RAM MSQLS	621,3	1061,3	1201,0	807,1
průměr RAM aplikace	168,9	168,3	141,7	147,8

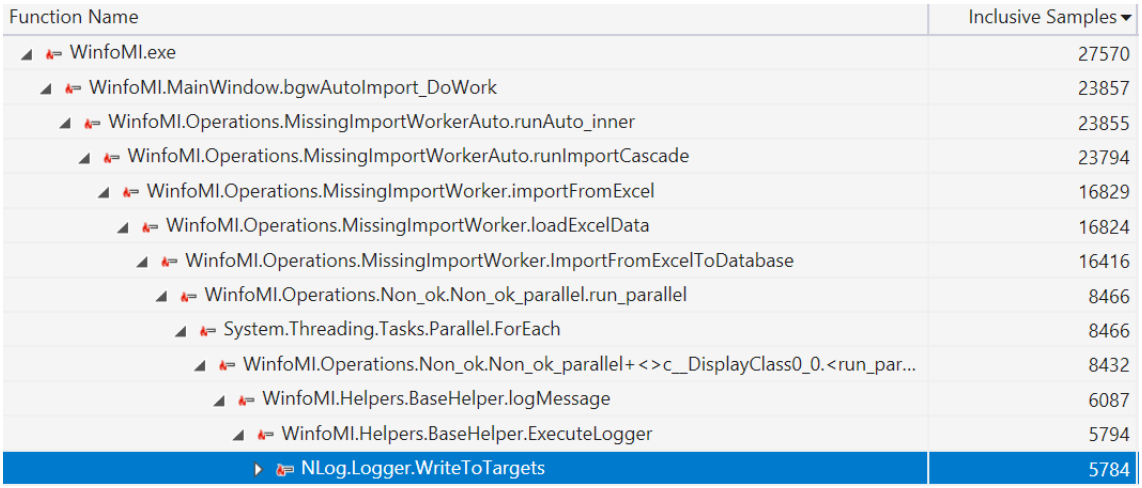
v přílohách na snímcích č. 20, 21, 22 a 23. První graf na každém snímku (nahoře vlevo) zaznamenával aktuální délku fronty disku, červeně je vyznačen práh, jaký by tato hodnota neměl přesahovat. Samotné překonání prahu nemuselo znamenat výkonnostní problém, při vyhodnocení bylo potřeba vzít v potaz hodnotu z čítače s % času procesoru (viz kapitola 4.0.1). Třetí graf obsahuje údaje o střední době čtení a zápisu disku. Práh je zde stanoven na 40 ms pro čtení a 10 ms pro zápis. Prahové hodnoty byly odhadnuty na základě měření při vykonávání běžných operací s diskem (viz. kapitola 4.0.1, výsledek těchto měření je vidět na snímku č. 28 v přílohách). Poslední, čtvrtý graf, pak ukazuje, zda probíhalo čtení nebo zápis na disk procesy SQL Server.

Před optimalizací vykazovalo vytížení disku mírně větší hodnoty, než po ní. Grafy s % času disku a délkou fronty v některých případech vykázaly společné překročení prahových hodnot ve stejnou chvíli, jednalo se však o sekundové výkyvy. Graf se střední dobou čtení a zápisu disku ve všech případech překračoval několikrát stanovené prahy, vzhledem ke kolísavosti tohoto ukazatele lze výchylky považovat za nevýznamné – v průměru se hodnoty držely pod prahem, což bylo pravděpodobně způsobeno i tím, že disk nebyl celkově příliš vytěžován v průběhu měření. Především pak SQL Server s diskem téměř neoperoval, jak je vidět na posledním ze čtveřice grafů. Celkově se tedy dá shrnout, že disk nevykazoval velké zatížení a až na pár nevýznamných odchylek nebyl přetěžován.

5.3 Profilování vzorové aplikace

Pro profilování vlastní aplikace byla vybrána verze s podmínkou. K analýze byly vytvořeny 2 profily – samplováním ve Visual Studiu a samplováním v dotTrace. K profilování bylo použito importování 4 profilů, aby profilery nasbíraly více vzorků.

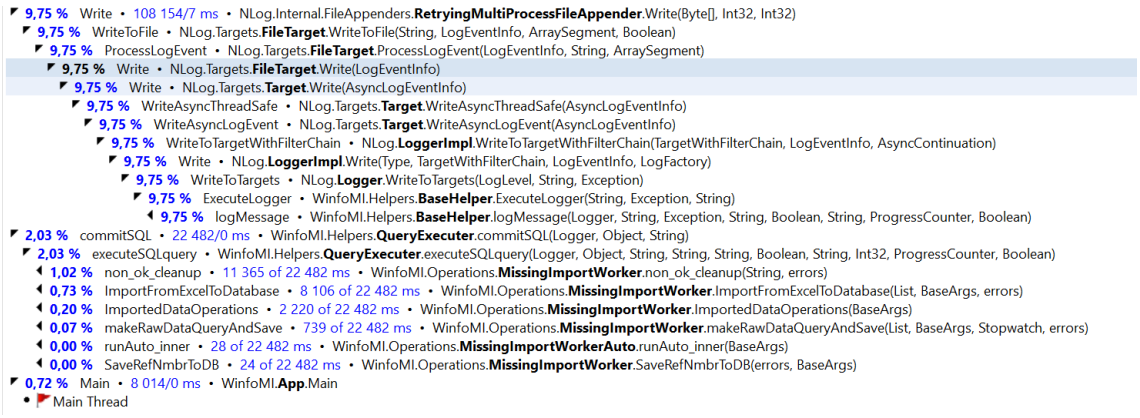
Oba profilery se snažili poukázat na kritické místo určitým zvýrazněním. Visual Studio takto zvýraznilo vytíženou cestu, dotTrace zase zobrazil několik nejvytíženějších funkcí. Cesta v případě Visual Studia vedla do vnitřku knihovny Nlog, starající se o logování z kódu. Postupem hierarchií odspodu výše se objevila funkce *logMessage*, jenž se stará právě o zapisování do logu. Většina vzorků této funkce pocházela z knihovny NLog (metoda *NLog.Logger.WriteToTargets*). Na funkci *logMessage* bylo tedy pomalé vlastní zapisování do logu.



Function Name	Inclusive Samples
WinfoMI.exe	27570
WinfoMI.MainWindow.bgwAutoImport_DoWork	23857
WinfoMI.Operations.MissingImportWorkerAuto.runAuto_inner	23855
WinfoMI.Operations.MissingImportWorkerAuto.runImportCascade	23794
WinfoMI.Operations.MissingImportWorker.importFromExcel	16829
WinfoMI.Operations.MissingImportWorker.loadExcelData	16824
WinfoMI.Operations.MissingImportWorker.ImportFromExcelToDatabase	16416
WinfoMI.Operations.Non_ok.Non_ok_parallel.run_parallel	8466
System.Threading.Tasks.Parallel.ForEach	8466
WinfoMI.Operations.Non_ok.Non_ok_parallel+<>c__DisplayClass0_0.<run_par...	8432
WinfoMI.Helpers.BaseHelper.logMessage	6087
WinfoMI.Helpers.BaseHelper.ExecuteLogger	5794
NLog.Logger.WriteToTargets	5784

Snímek 7: Vyznačená „horká“ cesta ve VS profileru.

Zvýrazněná cesta se týkala pouze jedné oblasti kódu. Ještě výše v hirerarchii bylo vidět, pro jakou oblast kódu cesta platila – v tomto případě se jednalo o průběh paralelní smyčky foreach ve funkci *Non_ok_parallel.run_parallel*. První doporučení pro optimalizaci se tedy mohlo týkat omezení logování v této funkci. Otázkou ale zůstávalo, zda logování nezpomalovalo také další části kódu. Funkce *logMessage* měla ve vyznačené cestě 6087 inklusivních vzorků, v pohledu funkcí byla hodnota inklusivních vzorku vyšší – 15 056. To znamenalo, že metoda byla vytížená i z jiných částí kódu. Seřazením pohledu funkcí sestupně podle počtu vzorků bylo zjištěno, že funkce byla jednou z nejvytíženějších. Dokonce se jednalo o nejvytíženější funkci nevolající další funkce z kódu aplikace.



Snímek 8: Doporučená místa k analýze v dotTrace.

DotTrace ve svém doporučení obsahoval také funkci *BaseHelper.logMessage*, plus několik dalších. Z jejich analýzy nevyplynulo žádné další místo k optimalizaci – jednalo se o opodstatněně zatížené funkce.

5.3.1 Optimalizace problematického místa

Možností, jak optimalizovat problematický log, bylo více. Za předpokladu, že bychom log nepotřebovali, bylo jednou z možností vytvořit přepínač na vypínání logu a nechat ho ve výchozím stavu vypnutý. Uživatel by ho pak zapínal jen v případě potřeby. Dalším řešením byla možnost log omezit, což by bylo poměrně náročné, protože bychom museli projít celý kód a rozhodnout, jaké informace jsou v logu opravdu potřeba. Navíc by takový zásah nemusel vést k zásadní optimalizaci. Řešením také mohlo být použití jiné logovací knihovny, z principu problému by ale ani takové řešení nejspíš nepřinášelo výsledky – logování trvá především z důvodu okamžitého zápisu do souboru, což je podstata činnosti většiny logovacích knihoven. Posledním řešení bylo zapisovat logovací záznamy do kolekce a po skončení importu je vypsat na jednom místě. Z logování by se tak v podstatě stalo obyčejné vypsání záznamů do souboru. Vzhledem k účelu logu při importování (v podstatě záznam činností importu) se toto řešení zdálo ideální.

Prvním krokem v realizaci vybraného řešení bylo zajistit, aby se nový způsob logování týkal pouze importování, jiné části kódu aplikace nebylo cílem ovlivnit. K tomuto účelu byla vytvořena proměnná nastavení projektu pojmenovaná jako *importIsRunning*. Principem bylo nastavit tuto proměnnou při začátku na hodnotu *true* a po skončení na hodnotu *false*. Pro ukládání dat logu bylo dále potřeba vytvořit kolekci s řádky logu. Abychom mohli tuto kolekci, (respektive list) volat z různých míst, k realizaci bylo potřeba použít objekt typu singleton. V kódu již jeden singleton byl, stačilo ho tedy jen rozšířit. Předposledním krokem řešení bylo vygenerovat řádek logu odpovídající řádku původního logu. Celá implementace uvnitř logovací metody je vidět ve zdrojovém kódu č. 2.

```
bool importIsRunning =
  Properties.Settings.Default.ImportIsRunning;

if (importIsRunning == true)
{
  var progressCounterInner = progressCounter;

  if (progressCounterInner == null)
    progressCounterInner = ProgressCounter.Instance;

  string timetag =
    DateTime.Now.ToString("dd MM yyyy - HH:mm:ss:ffff");

  string logString = timetag + " " + level + " - " + message;

   if (ex != null) //if there
    logString += Environment.NewLine + " Exception: " + ex;

   progressCounterInner.logList.Add(logString);
}
else
{
  ExecuteLogger(message, ex, level);
}
```

Zdrojový kód 2: Úprava části logovací metody, aby v případě běžícího importu zapisovala do kolekce.

Nakonec stačilo na vhodném místě po doběhnutí importu vypsat data z logovací kolekce do souboru. Použití původní logovací knihovny nemělo smysl, protože by bylo nutné zapisovat řádek po řádku a o ušetřený čas při importování bychom opět přišli. Implementace je vidět ve zdrojovém kódu č. 3 – řádky logu jsou spojeny přes funkci *string.Join* do jedné textové proměnné a ta je následně zapsána do souboru.

```
string customLogText = string.Join(
  Environment.NewLine, progressCounter.logList);

string timeTag = DateTime.Now.ToString("yyyyMMdd_hhmm_ss");

string path = logPath + timeTag + "_customImportLog.txt";

System.IO.File.WriteAllText(path,customLogText);
```

Zdrojový kód 3: Zápis řádků logu do souboru po dokončení importu. Díky použití funkcí string.Join a File.WriteAllText je operace i pro velký počet řádků dobře optimalizovaná.

5.3.2 Srovnání optimalizované verze s původní verzí

V tabulce č. 6 je vidět porovnání časů importů pro neoptimalizovanou i optimalizovanou verzi. Hodnoty pro neoptimalizovanou verzi jsou převzaty z předchozích měření. Z časů je vidět, že optimalizace logu byla z pohledu časů úspěšná. U vytížení procesoru a operační paměti k velkému zlepšení nedošlo (viz snímek č. 33 v přílohách).

Tabulka 6: Srovnání průměrů z 10 měření importu pro neoptimalizovanou aplikaci a optimalizovanou.

	před optimalizací [ms]	**po optimalizaci [ms]**
aritm. průměr	36 645	7 692
maximum	41 647	10 861
minimum	31 265	5 760

Závěr

Cílem práce bylo představit pojem profilování a demonstrovat použití nástrojů pro měření a optimalizaci. Teoretická část práce obsahuje vysvětlení pojmu profilování. Následuje stručný popis platformy .NET a databázového systému SQL Server. Součástí teoretické práce je také rozbor struktury platformy .NET, kdy tato kapitola je ukončena vysvětlením pojmu profilovací API.

Pro měření dopadu běhu aplikací na systémové prostředky práce využívá výkonnostní čítače systému Windows, pro optimalizaci je v ní pak obsažena dvojice profilerů a optimalizační nástroj pro databáze z balíku nástrojů SQL Server – Database Engine Tuning Advisor. Tyto nástroje jsou v práci nejprve popsány z pohledu ovládání a dostupných možností nastavení a použití – tato část práce může čtenáři posloužit jako výchozí informační základ pro vlastní používání nástrojů, případně ho může vhodně nasměrovat. Následně práce demonstruje použití vybraných nástrojů při měření a optimalizaci vzorové databáze a aplikace. Měření aplikace se zaměřuje na srovnání výkonu aplikace před a po optimalizačních krocích. Sledované parametry jsou vytížení procesoru, operační paměti a disku. Na konci optimalizace se podařilo dosáhnout zásadního zlepšení výkonu vzorové aplikace i databáze.

Výsledné zaměření práce je poměrně úzké, ale detailní. Téma práce by se dalo přenést na jiné vývojové platformy (příkladem může být jazyk Java a databáze MSQL). Nicméně i pro zvolenou platformu .NET a databázový systém SQL Server je v práci prostor pro rozšíření, například větším zaměřením se na používání nástrojů při vývoji aplikací společně s rozšířením tématu práce na debuggovací nástroje.

Největším problémem ještě před realizací práce bylo najít vhodnou aplikaci k demonstraci použitých nástrojů. Problematický byl především požadavek najít aplikaci využívající SQL Server jako zdroj dat. V původním zadání práce pod jiným vedoucím tento problém nebyl, protože práce byla zadána s přidělenou aplikací pro demonstraci. Tu již nešlo využít. Nakonec byla k demonstraci použita interní aplikace z autorovo praxe. Jelikož nevyužívala SQL Server databázi, musela pro ni být taková databáze vytvořena, samozřejmě s využitím struktury a dat původní databáze. Tím vznikla neoptimalizovaná výchozí databáze, vhodná k řešení této práce.

Literatura

[1] About ANTS Performance Profiler. [online] [cit. 2018-04-30].
Dostupné z: https://documentation.red-gate.com/app9/about-ants-performance-profiler

[2] Common Performance Monitor counter thresholds. 1995–2018.
Dostupné z: https://support.symantec.com/en_US/article.HOWTO9722.html

[3] Get Started with Performance Profiling. 2000–2018, [online] [cit. 2018-04-30].
Dostupné z: https://www.jetbrains.com/help/profiler/Get_Started_with_Performance_Viewer.html

[4] Profiler Options. 2000–2018.
Dostupné z: https://www.jetbrains.com/help/profiler/Profiler_Options.html?Wave=9\#time_measurement

[5] Běhové prostředí. 2001-2018, [online] [cit. 2018-04-30].
Dostupné z: https://cs.wikipedia.org/wiki/B\%C4\%9Bhov\%C3\%A9_prost\%C5\%99ed\%C3\%AD

[6] Common Language Infrastructure. 2001-2018, [online] [cit. 2018-04-30].
Dostupné z: https://en.wikipedia.org/wiki/Common_Language_Infrastructure

[7] Metadata (CLI). 2001-2018, [online] [cit. 2018-04-30].
Dostupné z: https://en.wikipedia.org/wiki/Metadata_(CLI)

[8] programovací jazyk C#. 2007, [online] [cit. 2018-04-30].
Dostupné z: http://www.cs.vsb.cz/behalek/vyuka/pcsharp/text/index.html

[9] Windows Performance Counters Explained. 2016.
Dostupné z: http://www.appadmintools.com/documents/windows-performance-counters-explained/

[10] Beginners Guide to Performance Profiling. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://msdn.microsoft.com/en-us/library/ms182372.aspx

[11] Common Type System. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/base-types/common-type-system

[12] DotTrace. 2018, [program].
Dostupné z: https://www.jetbrains.com/profiler/

[13] Get started with the .NET Framework. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/get-started/index

[14] JustTrace is Retired. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://www.telerik.com/products/memory-performance-profiler-sunsetting

[15] Language independence and language-independent components. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/language-independence

[16] Meaning of "profiling" in the English Dictionary. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://dictionary.cambridge.org/dictionary/english/profiling

[17] Metadata and Self-Describing Components. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/metadata-and-self-describing-components

[18] Monitoring Disk Usage. 2018.
Dostupné z: https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms175903(v=sql.105)

[19] Overview of the .NET Framework. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview

[20] Profiling Overview. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/profiling-overview

[21] Reflection (C# [program]ming Guide). 2018, [online] [cit. 2018-04-30].
Dostupné z: https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2008/ms173183(v=vs.90)

[22] SQL Server Documentation. 2018.
Dostupné z: https://docs.microsoft.com/en-us/sql/sql-server/sql-server-technical-documentation?view=sql-server-2017

[23] Troubleshooting Slow Disk I/O in SQL Server. 2018.
Dostupné z: https://blogs.msdn.microsoft.com/askjay/2011/07/08/troubleshooting-slow-disk-io-in-sql-server/

[24] Understanding Performance Collection Methods. 2018.
Dostupné z: https://msdn.microsoft.com/en-us/library/dd264994.aspx

[25] What Is Managed Code? 2018, [online] [cit. 2018-04-30].
Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/bb318664(v=vs.85).aspx

[26] What is "managed code"? 2018, [online] [cit. 2018-04-30].
Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/managed-code

[27] Dorodnicov, S.: .NET outside the box. 2018, [online] [cit. 2018-04-30].
Dostupné z: http://www.dorodnic.com/blog/2014/05/17/net-outside-the-box/

[28] Fouad, A.: Virtual Execution System. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://social.technet.microsoft.com/wiki/contents/articles/3326.virtual-execution-system.aspx

[29] Goldshtein, S.; Zurbalev, D.; Flatow, I.: *Pro .NET performance*. Berkeley, Calif.: Apress, [2012], ISBN 978-1430244585.

[30] Gouigoux, J.-P.: *Practical Performance Profiling*. USA: Redgate [book]s, první vydání, 2012, ISBN 9781906434823.

[31] Holan, J.: Tracing v .net pomocí event tracing for windows (etw). *DotNET-portal.cz*, 2018, [online] [cit. 2018-04-30].
Dostupné z: https://www.dotnetportal.cz/blogy/15/Null-Reference-Exception/8459/Tracing

[32] HUSSAIN, Q. M. S.: Key Performance Counters and their thresholds for Windows Server.
Dostupné z: https://sites.google.com/site/saifsqlserverrecipes/memory-performance-counters/key-performance-counters-and-their-thresholds-for-windows-server

[33] Högström, M.: Application Analysis with Event Tracing for Windows (ETW). 1999-2018, [online] [cit. 2018-04-30].
Dostupné z: https://www.codeproject.com/Articles/570690/Application-Analysis-with-Event-Tracing-for-Window

[34] Jeff, H.: Windows Performance Monitor Disk Counters Explained. 2018.
Dostupné z: https://blogs.technet.microsoft.com/askcore/2012/03/16/windows-performance-monitor-disk-counters-explained/

[35] JVP, J.: Components of .Net Framework. 2016, [online] [cit. 2018-04-30].
Dostupné z: http://www.developerin.net/a/39-Intro-to-.Net-FrameWork/23-Components-of-.Net-Framework

[36] Liptchinsky, V.: Pre-compile (pre-JIT) your assembly on the fly, or trigger JIT compilation ahead-of-time. 1999-2018, [online] [cit. 2018-04-30].
Dostupné z: https://www.codeproject.com/Articles/31316/Pre-compile-pre-JIT-your-assembly-on-the-fly-or-tr

[37] Margaret, R.: Microsoft SQL Server. 2005-2018.
Dostupné z: https://searchsqlserver.techtarget.com/definition/SQL-Server

[38] Mayo, J.: Introducing the .NET Platform. 2018, [online] [cit. 2018-04-30].
Dostupné z: http://www.informit.com/articles/article.aspx?p=1236098

[39] Pietrek, M.: Under the hood. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://msdn.microsoft.com/en-us/library/bb985756.aspx

[40] Scott, A. K.: Metadata and Reflection in .NET. 2004 - 2018, [online] [cit. 2018-04-30].
Dostupné z: https://odetocode.com/articles/288.aspx

[41] Todorov, T.: Understanding .NET Just-In-Time Compilation. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://www.telerik.com/blogs/understanding-net-just-in-time-compilation

[42] Tomar, N.: JIT (Just-In-Time) Compiler. 2018, [online] [cit. 2018-04-30].
Dostupné z: https://www.c-sharpcorner.com/uploadfile/nipuntomar/jit-just-in-time-compiler/

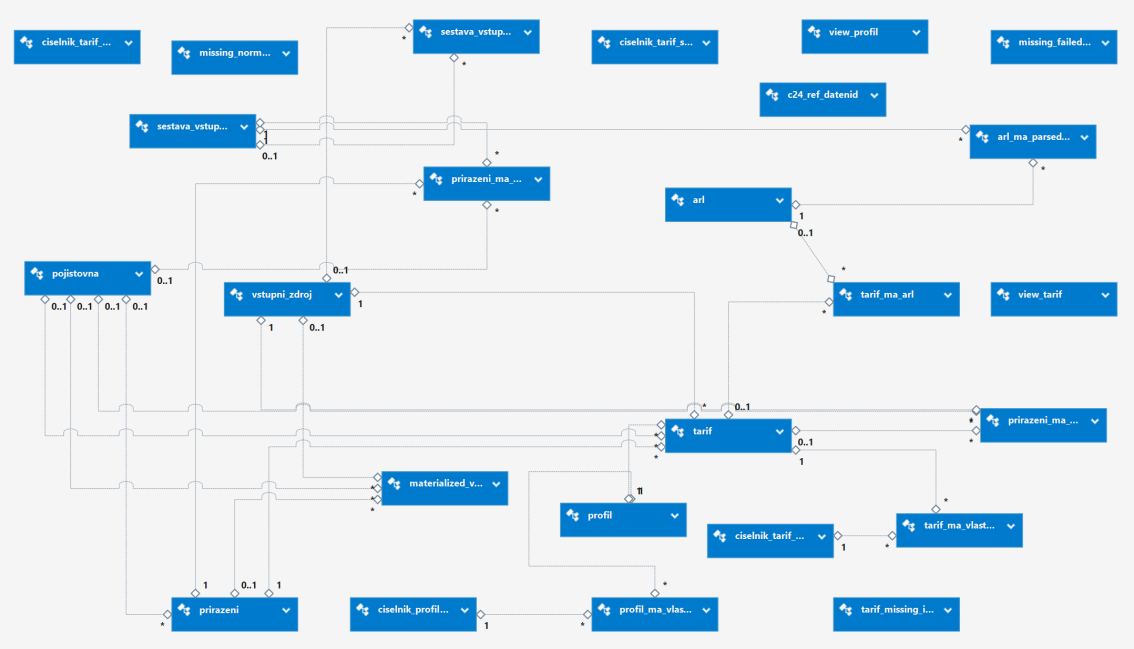
A Přílohy

```
IF EXISTS
(SELECT 1 as record FROM [tarif_ma_vlastnosti]
WHERE [id_tarif] = id_tarif
AND [id_tarif_vlastnosti] = id_tarif_vlastosti)
BEGIN
UPDATE [tarif_ma_vlastnosti] SET [hodnota] = hodnota
WHERE [id_tarif] = id_tarif
AND [id_tarif_vlastnosti] = id_tarif_vlastnosti
END ELSE BEGIN
INSERT INTO [tarif_ma_vlastnosti]
([id_tarif], [id_tarif_vlastnosti], [hodnota])
VALUES (id_tarif, id_tarif_vlastnosti, hodnota)
END;
```

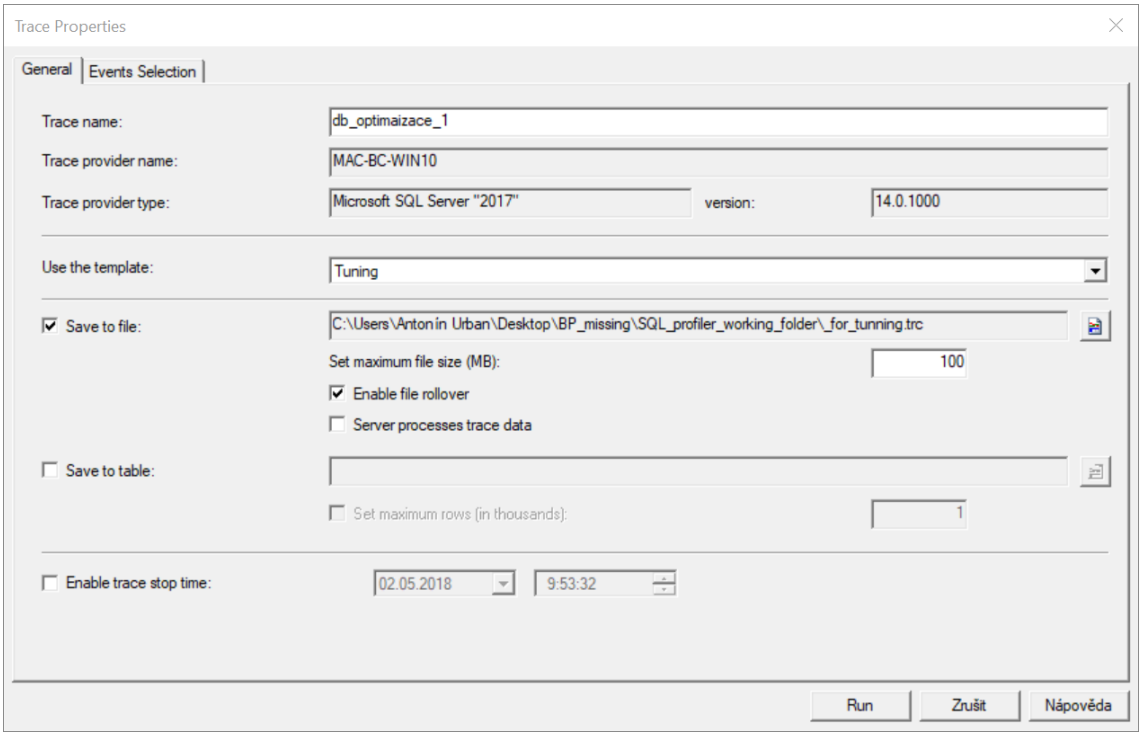
Zdrojový kód 4: Schématický TSQL zápis podmínky nahrazující MySQL funkci INSERT OR UPDATE

```
CREATE TRIGGER [dbo].[insert_on_duplicate_update]
ON [dbo].[tarif_ma_vlastnosti]
INSTEAD OF INSERT AS
 SELECT * FROM inserted as I;
 DECLARE @id_tarif_vlastnosti BIGINT,
 @id_tarif BIGINT,
 @hodnota varchar(MAX);
   SELECT
   @id_tarif_vlastnosti = inserted.id_tarif_vlastnosti,
   @id_tarif = inserted.id_tarif,
   @hodnota = inserted.hodnota
   FROM inserted;
IF EXISTS
(SELECT * FROM tarif_ma_vlastnosti
WHERE id_tarif = @id_tarif AND id_tarif_vlastnosti =
@id_tarif_vlastnosti )
BEGIN
    UPDATE tarif_ma_vlastnosti SET hodnota = @hodnota
    WHERE id_tarif_vlastnosti = @id_tarif_vlastnosti
    AND id_tarif = @id_tarif;
END
ELSE
BEGIN
   INSERT into tarif_ma_vlastnosti
    (id_tarif,id_tarif_vlastnosti,hodnota)
    values (@id_tarif,@id_tarif_vlastnosti,@hodnota)
END
```

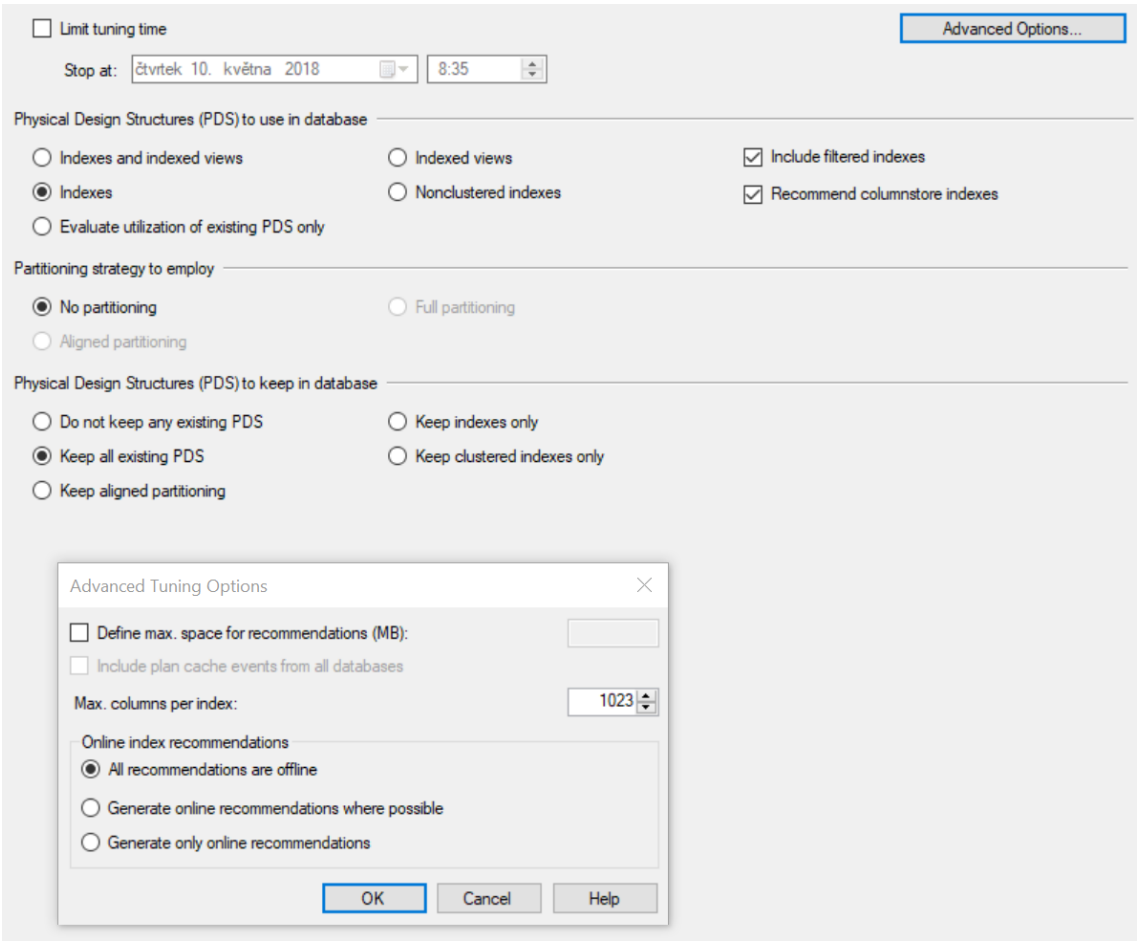
Zdrojový kód 5: Schématický TSQL zápis triggeru nad tabulkou tarif_ma_vlastnosti v databázi.



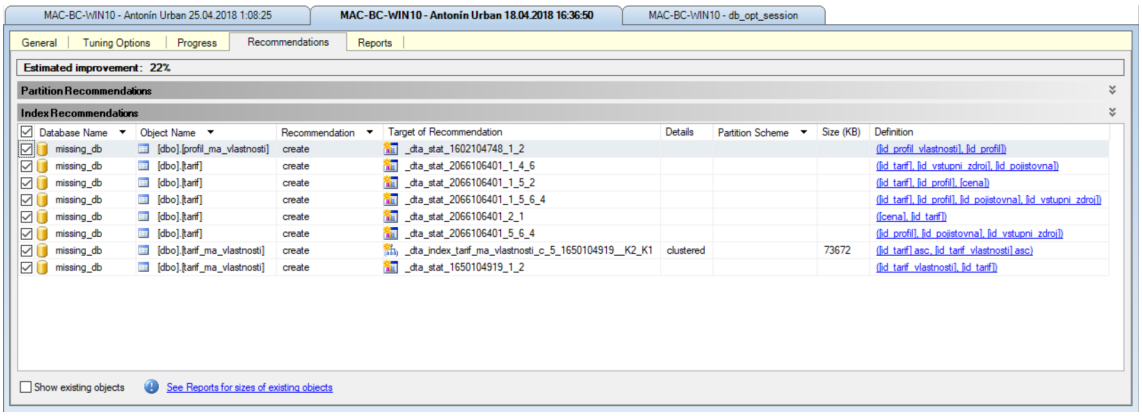
Snímek 9: Na snímku je vidět databázový model vzorové databáze – obsahuje 23 tabulek a dva pohledy.



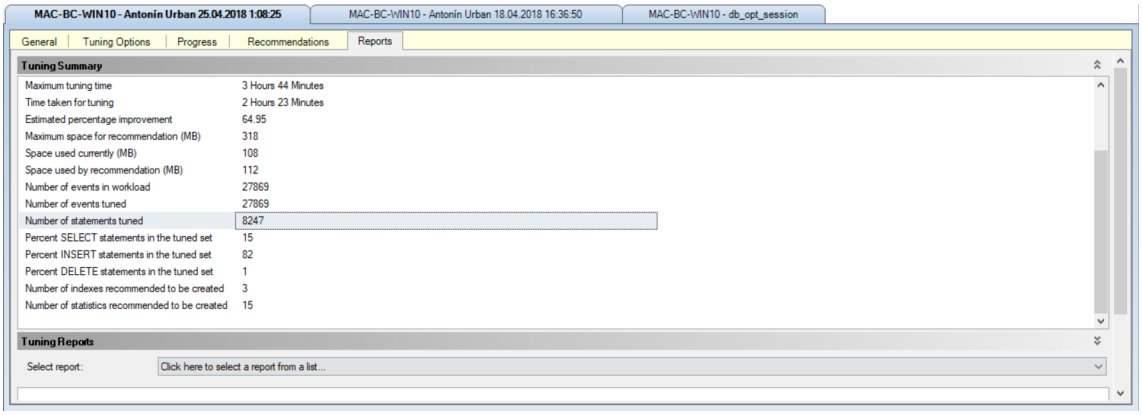
Snímek 10: Nastavení SQL Server Profiler pro získání zátěže k použití v DTA (pro testovanou databázi).



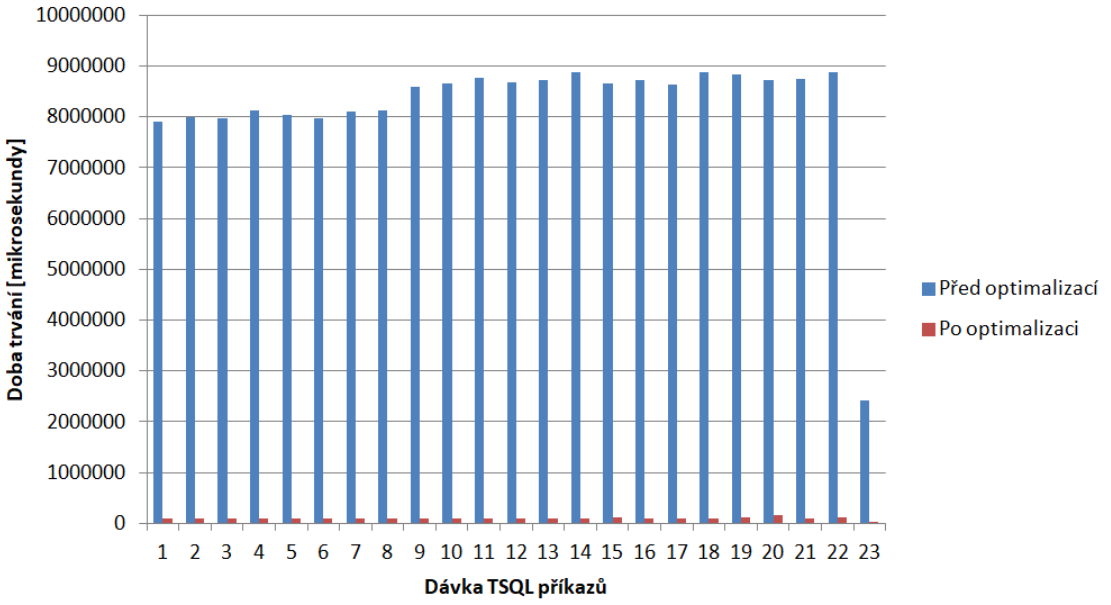
Snímek 11: Nastavení ladění v DTA pro testovanou databázi.



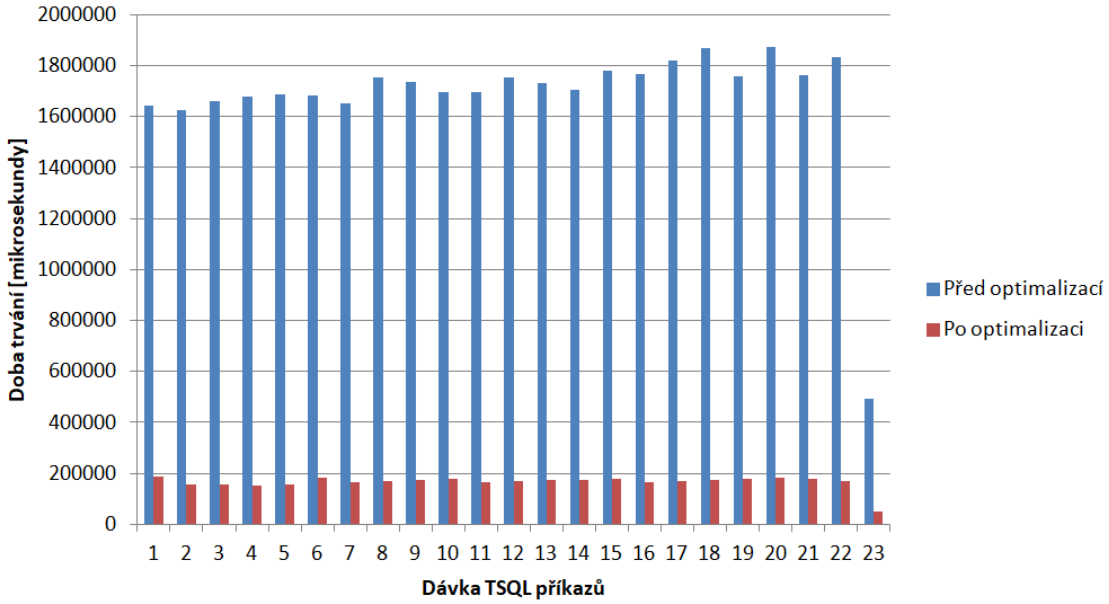
Snímek 12: Výsledná optimalizační doporučení z DTA pro podmínku: 1 indexy a 7 statistik.



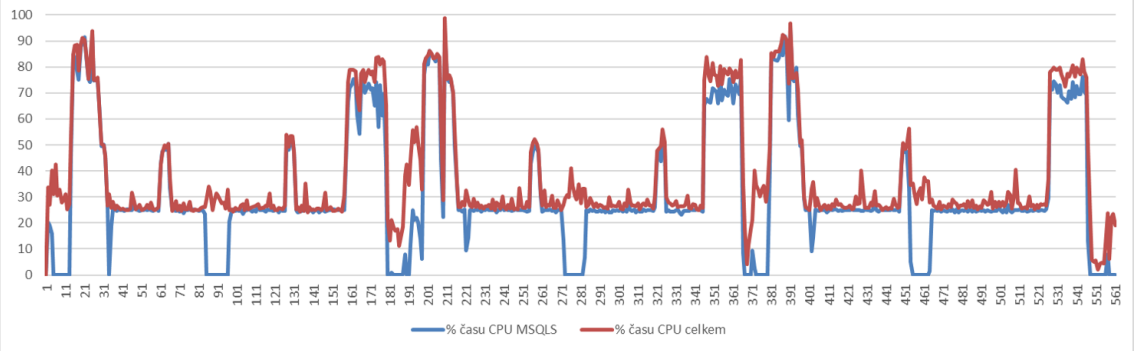
Snímek 13: Výsledná optimalizační doporučení z DTA pro trigger: 3 indexy a 15 statistik.



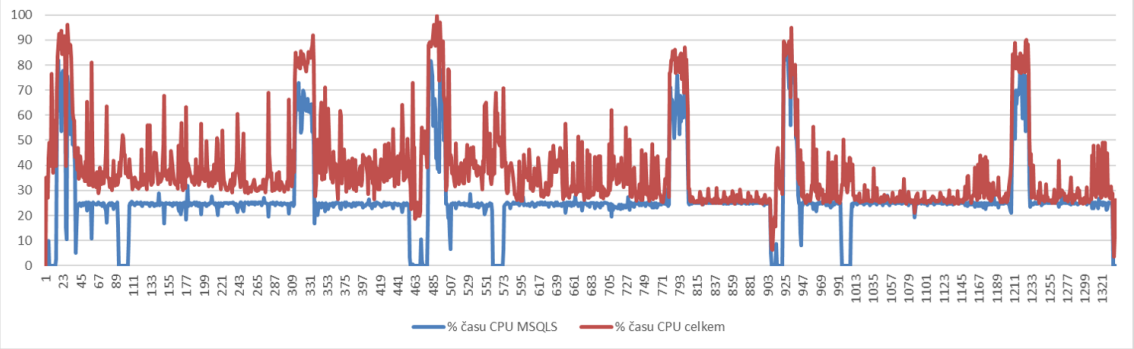
Snímek 14: Graf se srovnáním časů dávek TSQL příkazů před a po optimalizaci pro případ s triggerem.



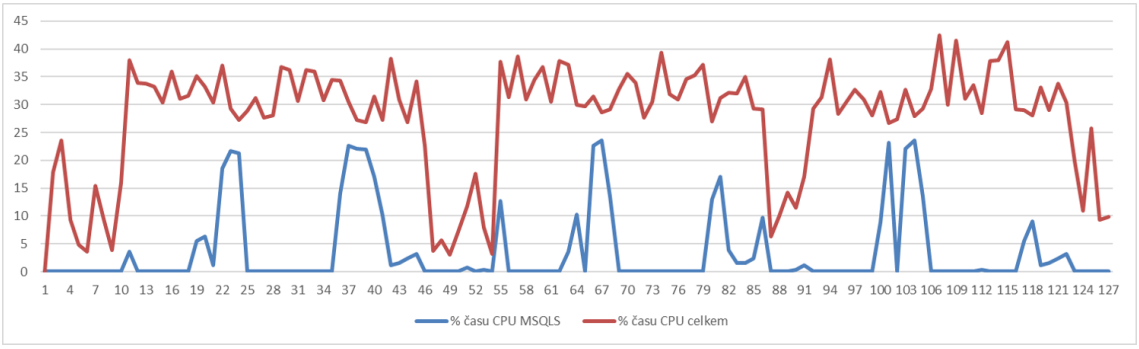
Snímek 15: Graf se srovnáním časů dávek TSQL příkazů před a po optimalizaci pro případ s podmínkou.



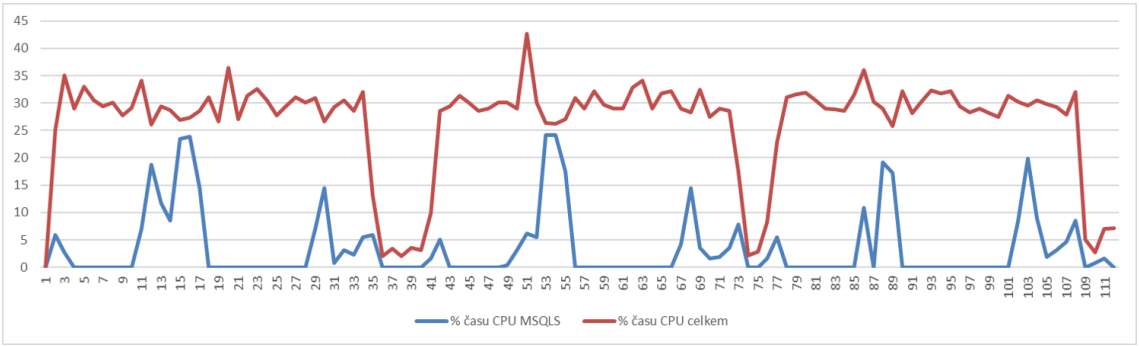
Snímek 16: Graf z měření vytížení CPU databází, verze s podmínkou před optimalizací DB.



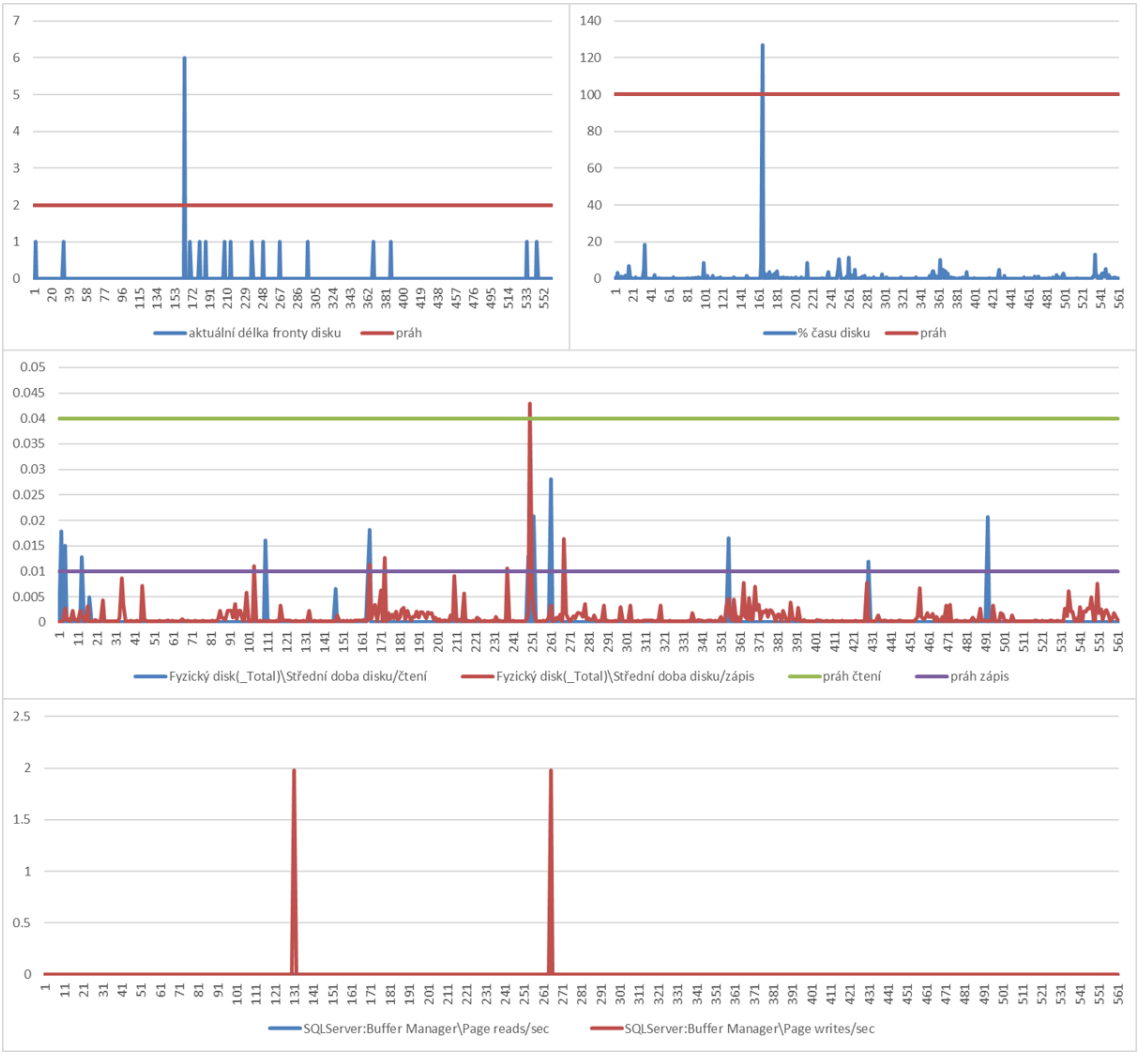
Snímek 17: Graf z měření vytížení CPU databází, verze s triggerem před optimalizací DB.



Snímek 18: Graf z měření vytížení CPU databází, verze s podmínkou po optimalizaci DB.



Snímek 19: Graf z měření vytížení CPU databází, verze s triggerem po optimalizaci DB.

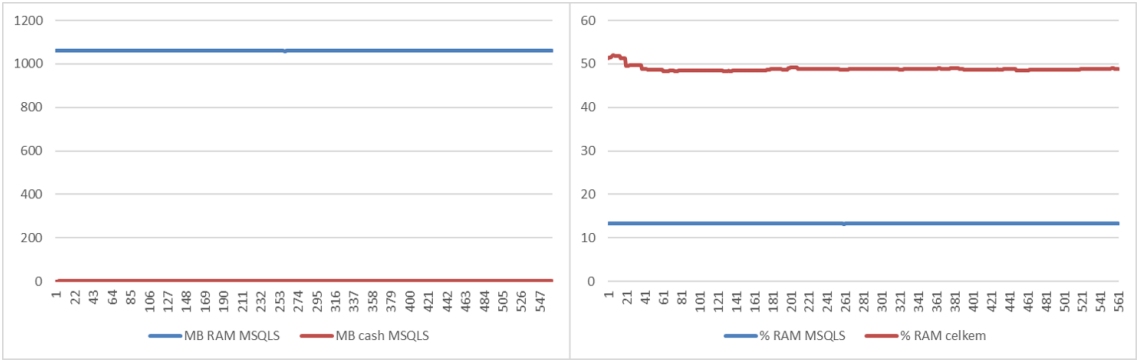


Snímek 20: Grafy z měření disku, verze s podmínkou před optimalizací DB.

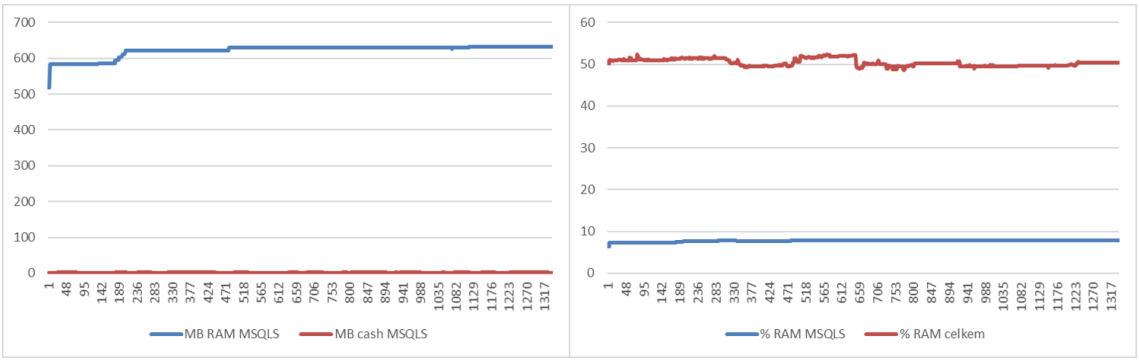
Snímek 21: Grafy z měření disku, verze s triggerem před optimalizací DB.

Snímek 22: Grafy z měření disku, verze s podmínkou po optimalizaci DB.

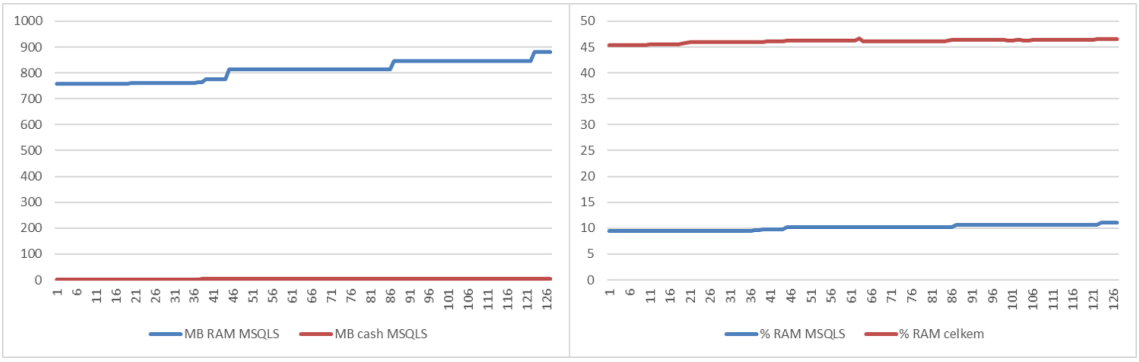
Snímek 23: Grafy z měření disku, verze s triggerem po optimalizaci DB.



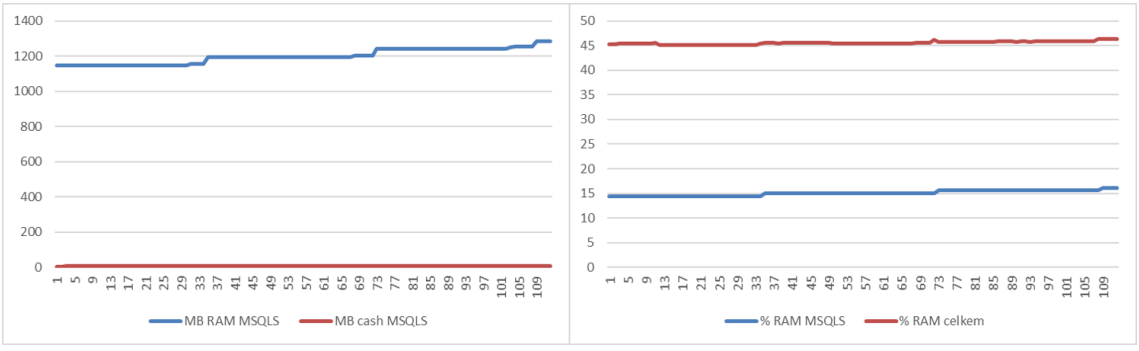
Snímek 24: Grafy z měření vytížení RAM činností MSQLS, verze s podmínkou před optimalizací DB.



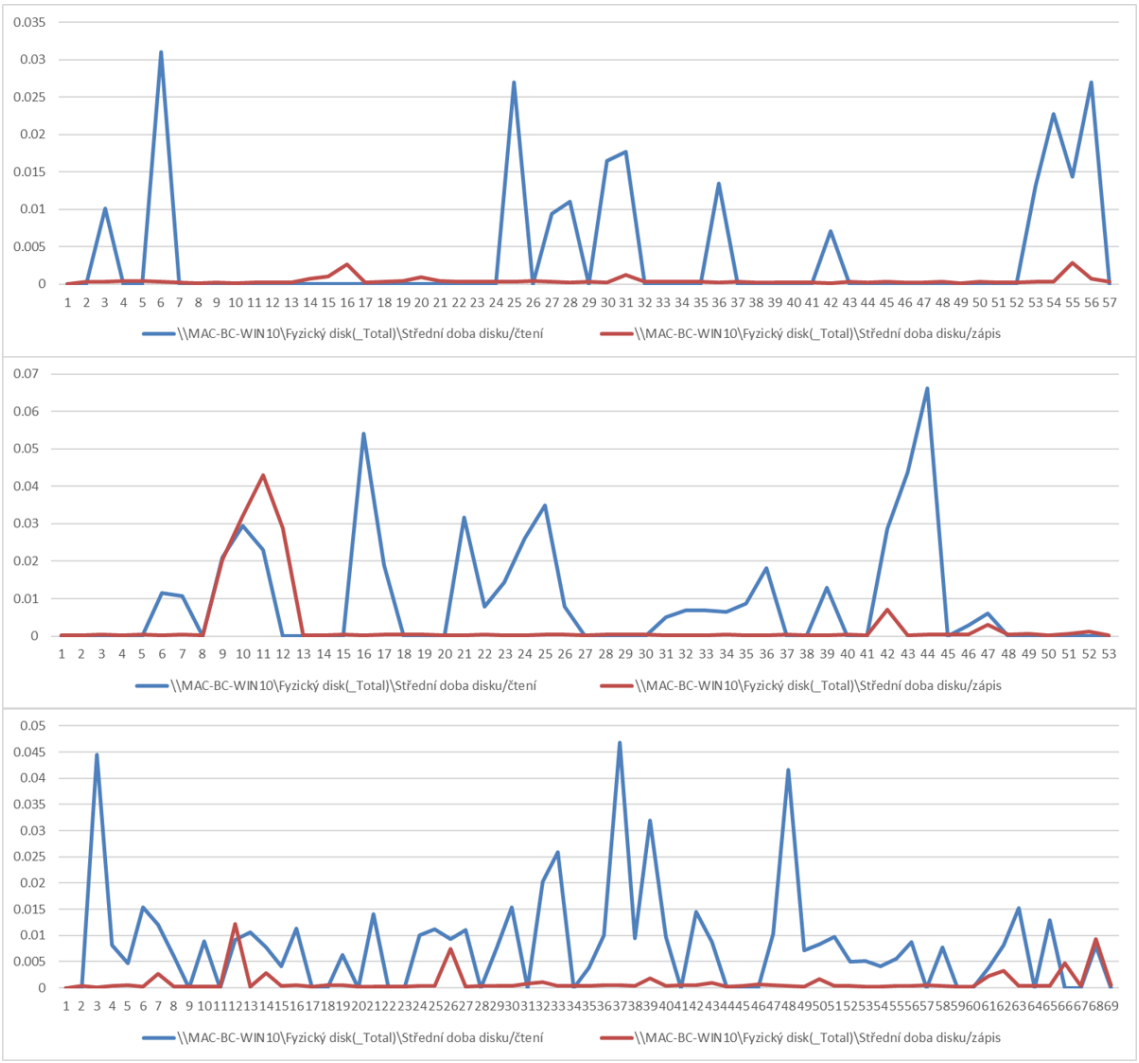
Snímek 25: Grafy z měření vytížení RAM činností MSQLS, verze s triggerem před optimalizací DB.



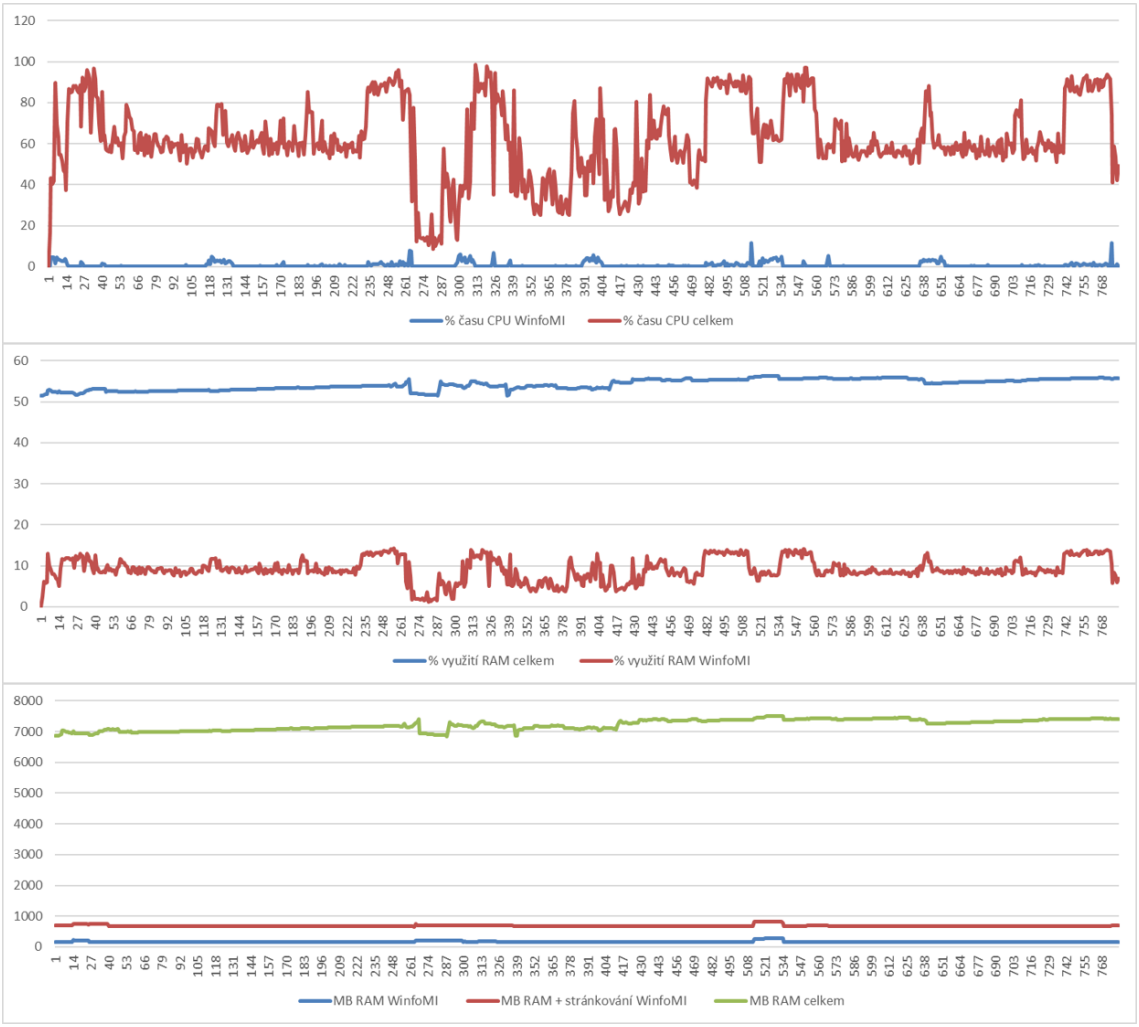
Snímek 26: Grafy z měření vytížení RAM činností MSQLS, verze s podmínkou po optimalizaci DB.



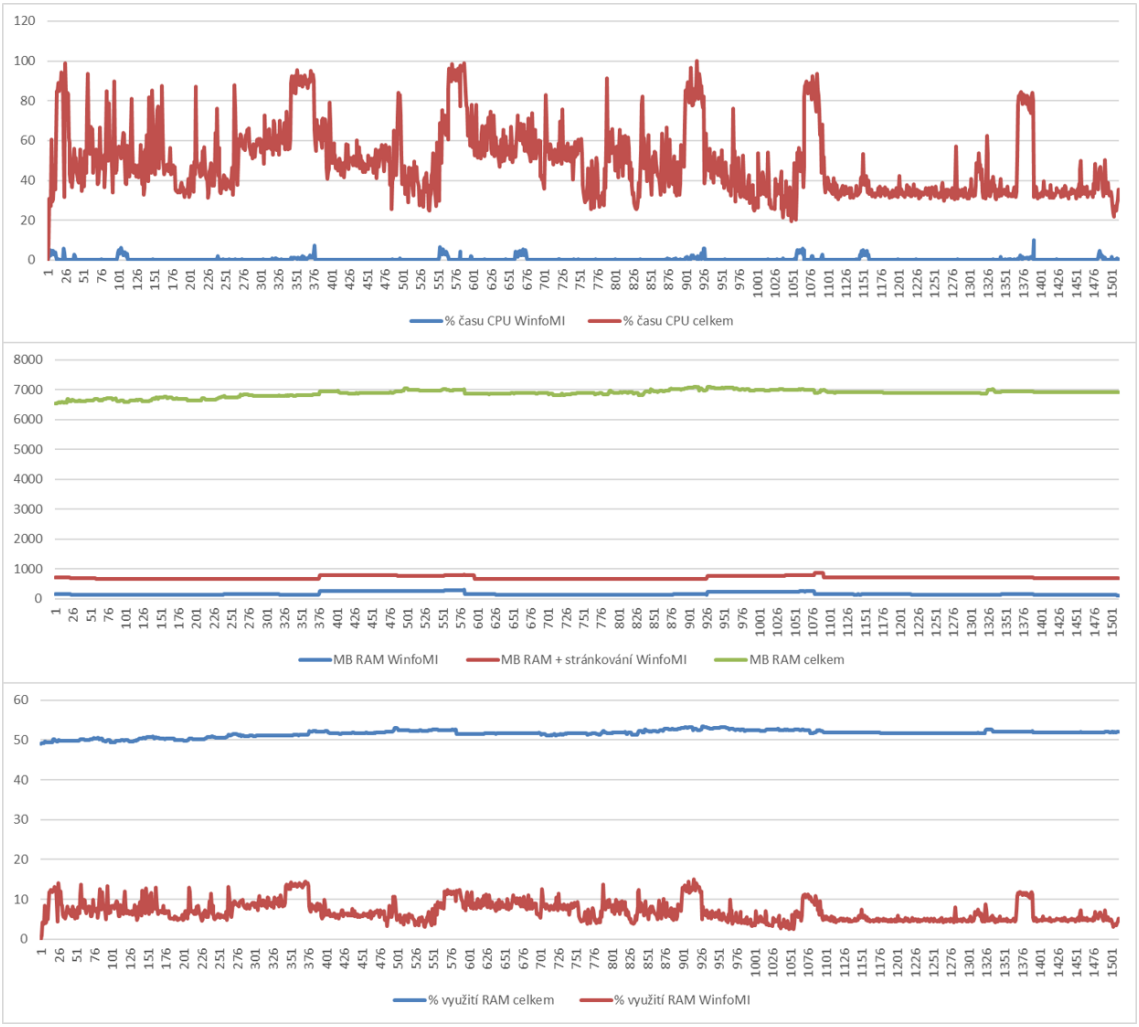
Snímek 27: Grafy z měření vytížení RAM činností MSQLS, verze s triggerem po optimalizaci DB.



Snímek 28: Grafy z měření za účelem stanovení výkonnostních prahů měřeného disku.

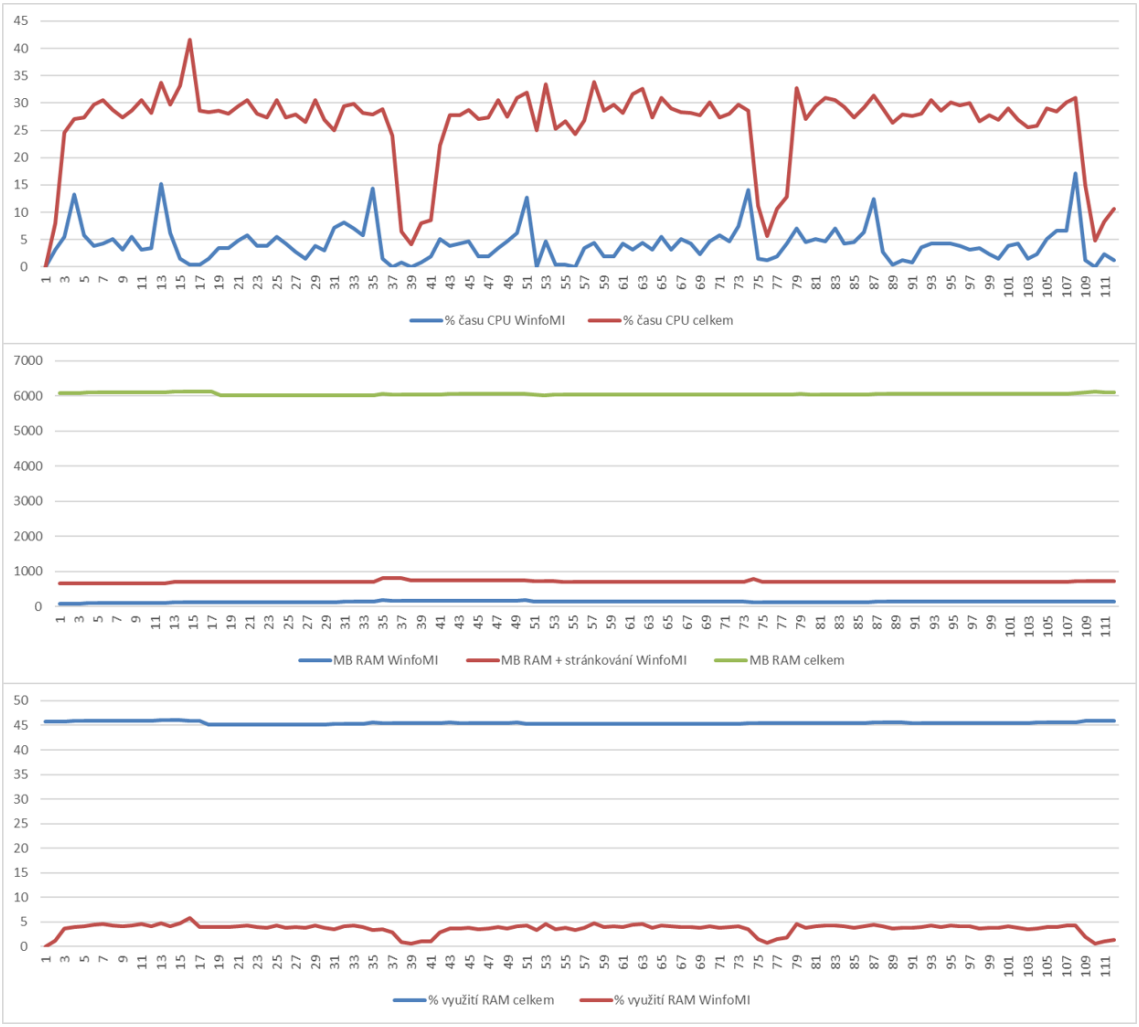


Snímek 29: Grafy z měření vytížení CPU a RAM činností aplikace, verze s podmínkou před optimalizací DB.

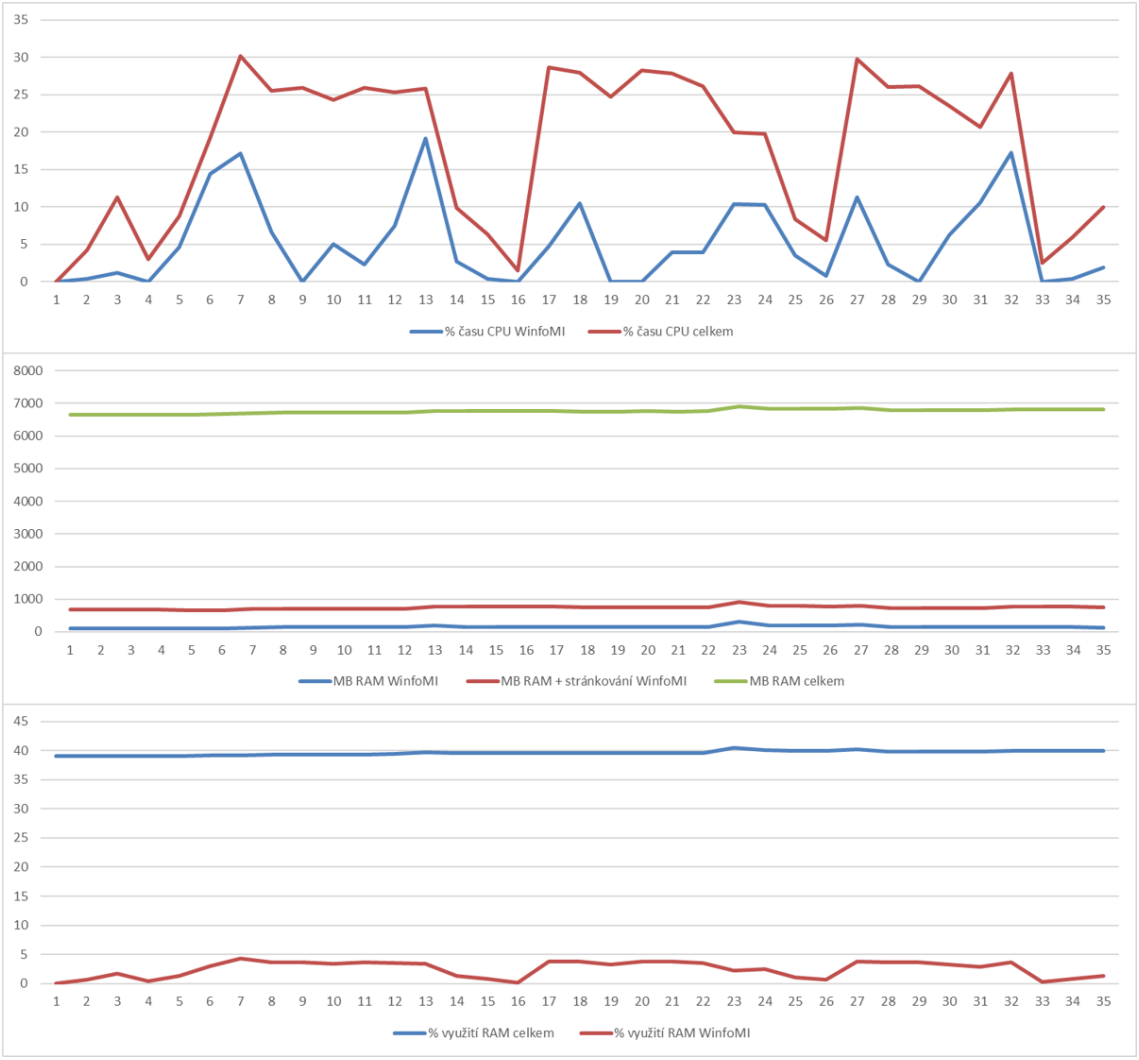


Snímek 30: Grafy z měření vytížení CPU a RAM činností aplikace, verze s triggerem před optimalizací DB.

Snímek 31: Grafy z měření vytížení CPU a RAM činností aplikace, verze s podmínku po optimalizaci DB.



Snímek 32: Grafy z měření vytížení CPU a RAM činností aplikace, verze s triggerem po optimalizaci DB.



Snímek 33: Grafy z měření vytížení CPU a RAM činností aplikace, verze po optimalizaci logu.

B Obsah přiloženého CD

- Text bakalářské práce
  - `bakalarska_prace_2018_Antonin_Urban.pdf`
- Snímky obrazovky
  - `snímky obrazovky z měření`
- Data z měření
  - `naměřené zpracované hodnoty a grafy`