Optimalizace SQL vrstvy aplikace GetmoreSystem

GetmoreSystem application SQL layer optimalization

Vít Kymla

Diplomová práce
2009

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav aplikované informatiky
akademický rok: 2008/2009

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Vít KYMLA**
Studijní program: **N 3902 Inženýrská informatika**
Studijní obor: **Informační technologie**

Téma práce: **Optimalizace SQL vrstvy aplikace GetmoreSystem**

Zásady pro vypracování:

1. **Analýza problematiky a popis klient-server aplikace GetmoreSystem.**
2. **Rešerše problematiky zvýšení výkonu SQL vrstvy.**
3. **Vytvoření úprav pomocí optimalizace struktur, optimalizace SQL kódů, cachování dat v rámci aplikace (ASP.NET) apod. se zaměřením na maximální snížení systémových nároků.**
4. **Návrh a vytvoření aplikace, která bude produkovat statistiky vyhodnocující výkon SQL vrstvy před a po optimalizaci, fragmenty databáze a důležité části SQL kódů.**

Rozsah práce:

Rozsah příloh:

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

Seznam odborné literatury:

1. **Brust, Andrew J. a Forte, Stephen. 2007. Mistrovství v programování SQL Serveru 2005. Brno : Computer Press, 2007. 978-80-251-1607-4.**
2. **Evjen, Bill, Hanselman, Scott a Muhamad, Farhan. 2006. ASP.NET 2.0 Programujeme profesionálně. [překl.] Karel Voráček. 1. vyd. Brno : Computer Press, a.s., 2006. str. 1224. ISBN 80-251-1286-1.**
3. **2008. MSDN library. [Online] Microsoft, 2008. http://msdn.microsoft.com/library/.**
4. **Nagel, Christian, a další. 2006. C 2005 Programujeme profesionálně. [překl.] Petr Dokoupil Jakub Mikulaštík. 1. vyd. Brno : Computer Press, a.s., 2006. str. 1398. ISBN 80-251-1181-4.**
5. **Prosise, Jeff. 2003. Programování v Microsoft .NET Webové aplikace. [překl.] Karel Voráček. 1. vydání, Brno : Computer Press, 2003. str. 712. ISBN-80-7226-879-1.**
6. **Sack, Joseph. 2007. Velká kniha T-SQL & SQL Server 2005. [překl.] Jan Pokorný. Brno : Zoner Press,, 2007. str. 864. 978-80-806815-57-2.**
7. **Sharp, John a Jagger, Jon. 2002. Microsoft Visual C .NET - krok za krokem. Brno : Mobil Media, a.s., 2002. str. 654. ISBN 80-86593-27-4.**
8. **Stanek, William R. 2006. SQL Server 2005 Kapesní rádce administrátora. [překl.] Luděk Horčička. Brno : Computer Press, a.s., 2006. str. 542. ISBN 80-251-1211-X.**
9. **Whalen, Edward, a další. 2008. Microsoft SQL Server 2005 - Velký průvodce administrátora. Brno : Computer Press, 2008. str. 1080. 9788025119495.**

Vedoucí diplomové práce: **doc. Ing. Zdenka Prokopová, CSc.**
Ústav aplikované informatiky

Datum zadání diplomové práce: **20. února 2009**

Termín odevzdání diplomové práce: **27. května 2009**

Ve Zlíně dne 13. února 2009

L.S.

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

ABSTRAKT

Teoretická část práce se zabývá podstatou výkonnostních problémů v databázových vrstvách aplikací. Velká část je věnována výkonnostním problémům v SQL kódech. Jsou rozebrány nejčastější chyby, které mohou způsobovat ztráty výkonu. Ve většině případů je uveden plán vykonávání dotazu s rozborem. Jsou popsány také jednotlivé fyzické operace, kterými jsou realizovány dílčí části plánu vykonávání. Jsou uvedeny teoretické podklady pro možnosti měření výkonu databázové vrstvy. Teoretická část rozebírá také vlastnosti jazyka SQL, jejichž znalost je podstatná pro provádění optimalizačních zásahů do SQL kódů. Také jsou podrobně popsány databázové struktury, které přímo souvisí s výkonem. Praktická část je věnována popisu testovací aplikace Getmore.Permon, která byla vytvořena v rámci této práce. Dále je uveden popis optimalizačních prací, které byly provedeny nad aplikací GetmoreSystem včetně jejich výsledků.

Klíčová slova: SQL, optimalizace, databáze, výkon, měření výkonu, plán vykonávání, cost based systém, index, statistika, SQL server hint

ABSTRACT

The theoretical part of this master thesis speaks about substance of application databases layer performance losses. A big part is devoted to performance troubles in SQL codes. The thesis speaks about the most common mistakes, which can cause loss of performance. In most cases is given also execution plan of the SQL code with analysis. Also some of the execution plan physical operations are described. The thesis discusses also performance measurement possibilities. It describes SQL language properties, theirs knowledge is important for SQL codes optimization. Thesis also speaks in detail about database structures, which are important for the performance. The practical part of this thesis is devoted to Getmore.Permon test application description. This application was programmed in range of this work. The thesis contains also description and results of the optimization work, which was done over the GetmoreSystem application.

Keywords: SQL, optimization, database, performance, performance measurement, execution plan, cost based system, index, statistics, SQL server hint

Poděkování

Předně děkuji vedoucí této diplomové práce – doc. Ing. Zdence Prokopové, CSc.. Děkuji také vedení společnosti Getmore s.r.o. za umožnění realizace diplomové práce v rámci pracovního úvazku.

Motto:

Za normálních podmínek jsme výkonní, teprve když nás tlačí čas nebo se ocitneme v tísni, jsme efektivní.

John C. Maxwell

The best performance improvement is the transition from the nonworking state to the working state

John Ousterhout

Věnování:

Babičce Julianně

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.

Ve Zlíně ……..…….

Podpis diplomanta

OBSAH

ÚVOD

Téměř každý softwarový produkt se jednou dostane do fáze, kdy se jeho použitelnost a uživatelská obliba začne snižovat díky zvyšování časové náročnosti na jednotlivé operace. Výrazně se tento jev projevuje u „client – server“ aplikací, kde s rostoucí dobou provozu dochází ke zvětšování velikosti databází a často také ke zvyšování počtu současně pracujících uživatelů. Prodlužující se doba odezvy začne vyvolávat potřebu po urychlení chodu systému. Analýza výkonu aplikace v těchto případech většinou odhalí slabá místa v databázové vrstvě. Tento stav je zcela přirozený – vývoj software běžně probíhá na testovacích databázích, které neobsahují dostatečně velký objem dat. Mnoho typů „výkonnostních chyb“ se proto při vývoji neprojeví a nevzniká okamžitá potřeba je řešit. Přitom je databázová vrstva velmi náchylná na vznik neoptimálních fragmentů kódu nebo neoptimálních datových struktur. Často se z těchto fragmentů kódů a struktur stávají dlouhodobě časované bomby, které udeří většinou až po uvedení do ostrého provozu.

Motivací pro tuto práci se stalo úspěšné řešení některých výkonnostních problémů v modulech aplikace GetmoreSystem (dále jen GMS). Tyto výkonnostní problémy byly lokalizovány v databázové vrstvě a podařilo se je výrazně redukovat jednoduchými zásahy do zdrojových kódů SQL dotazů. Tak vznikla myšlenka na celkovou optimalizaci a revitalizaci databázové vrstvy aplikace. Ukázalo se, že práce na optimalizaci je nejen velmi užitečná, ale také nesmírně zajímavá a obohacující. Programátor, který začne podrobně zkoumat výkonnostní problémy v existujících kódech, se snadno podobným chybám v budoucnu vyhne. Při vývoji softwarového produktu je běžné, že programátorům není dán potřebný čas, aby mohli své kódy dodatečně analyzovat - je kladen důraz na funkčnost a minimální časové nároky na vývoj. Často také chybí nástroje, které by takovou analýzu umožnily.

Cílem diplomové práce je optimalizace databázové vrstvy client-server aplikace GMS pomocí úprav SQL kódů, úprav DB struktur a také pomocí kešování dat na databázové nebo aplikační úrovni. Neskromným cílem je zvýšit výkon aplikace o cca 30% v celkovém měřítku. Dalším podstatným cílem je vytvoření pomocné aplikace, která umožní měřit výkon databázové vrstvy, a která poskytne možnosti pro rychlou analýzu SQL fragmentů (SQL kódů, procedur, funkcí…) přímo při vývoji.

I. TEORETICKÁ ČÁST

1 PROBLEMATIKA VÝKONU DATABÁZOVÉ VRSTVY

1.1 Výkon databázové vrstvy

Výkon databázové vrstvy aplikace a jeho vylaďování je značně relativní pojem. Mnohdy záleží na mnoha okolnostech, které ovlivňují samotné chápání tohoto pojmu. Vždy je nutno zohlednit, v jakých případech se daný SQL kód používá. Například skript, jehož vykonání trvá 10 minut, můžeme klidně považovat za optimální, pokud je použit například pro kešování[1] dat a jeho vykonání je odloženo na dobu, kdy je databázový server málo vytížený. Těžko bychom však akceptovali tento čas pro skript, který se spouští mnohokrát v rámci provozu aplikace. Následuje výčet klíčových faktorů, které ovlivňují výkon databázové vrstvy aplikace.

1.1.1 Návrh struktury databáze

Tento faktor je považován za jeden z nejpodstatnějších. Už jenom proto, že dodatečné zásahy do struktury databáze (resp. do struktury tabulek v databázi) jsou velmi komplikované a náchylné na chyby. Obecně existují dva směry, kterými je možno se ubírat při návrhu designu databáze. Prvním je tzv. **OLTP - Online Transaction Processing**. Tento způsob uložení dat je zaměřen na vysoký výkon a bezpečnost pro modifikaci dat v mnohouživatelském prostředí – tj. v prostředí, kde dochází k neustálé aktualizaci a vkládání dat [1]. Tento model obvykle dodržuje třetí normální formu (3NF)[2]. Zaměřuje se na integritu, odstraňování duplicit a zřizování relací mezi entitami. Zjednodušeně řešeno – model OLTP je optimalizován na rychlost aktualizací v transakcích. Druhým směrem je **OLAP – Online Analytical Processing**. Tento způsob uložení dat je určen pro uživatele, kteří potřebují analyzovat velké objemy existujících dat, vytvářet z nich sestavy, trendy apod. OLAP je zaměřen na výkon při získávání dat (data mining). OLAP struktury většinou nedodržují normální formy a mnohdy obsahují duplicity. Základními OLAP strukturami jsou star (hvězda) a snowflake (sněhová vločka). Pro tyto struktury je typické, že obsahují jednu centrální „velkoobjemovou“ tabulku (tzv. fact table), na kterou jsou vázány objemově menší, tzv. dimenzionální tabulky.

[1] Kešování = ukládání do paměti cache (česká literatura běžně používá tento počeštěný termín)
[2] Výklad normálních forem pro datové struktury je nad rámec této práce.

1.1.2 Fyzický hardware

Výkon hardware pro databázový server je bezesporu velmi podstatným faktorem pro výkon databázové vrstvy. V rámci optimalizace se můžeme dostat do fáze, kdy veškeré postupy na úrovni software selhávají a je nutno zasáhnout do hardwarové konfigurace. Zlepšení hardwarové konfigurace by však mělo být vždy tím posledním krokem v rámci optimalizačního procesu. Pro hardware databázového serveru je nejkritičtějším prostředkem **operační paměť**. Dostatečná paměť umožňuje rychlé kešování dat a plánů vykonávání. Velký vliv má také výkon **CPU** a v neposlední řadě subsystém pro fyzické uložení dat na disku (**RAID** disková pole apod.) – diskové operace jsou vždy nejvíce časově náročné úkony.

1.1.3 Výkon síťového propojení

Pokud je databázový server fyzicky oddělen od aplikačního serveru, pak je nutné uvažovat i výkon síťového propojení mezi nimi. Je nutné se zaměřit na přenosovou rychlost kanálu a také na kvalitu (stabilitu) spojení. Je pochopitelné, že význam tohoto faktoru roste s velikostí přenášených dat.

1.1.4 Politika indexů v databázi a fragmentace indexů

O indexech a jejich významu je psáno dále v kapitole 3.2 (Indexy). Indexy jsou jedny z nejpodstatnějších elementů pro výkon databázové vrstvy. Vždy je nutno zvolit vhodnou indexovou „politiku“ s ohledem na základní fakt, že indexy většinou výrazně urychlují dotahování a naopak mírně zpomalují modifikaci dat. Každý index také zabírá určité místo na disku. Také může nastat situace, kdy nevhodně zvolený index dokonce zpomalí dotažení dat, protože je pro daný dotaz málo efektivní (pokud jeho použití vynutíme pokynem pro překladač). S rostoucím časem provozu aplikace může také docházet k fragmentaci indexů – tj. k jejich rozptylování na stále větší počet datových stránek. Indexy pak přestávají plnit svoji významnou funkci pro zvýšení výkonu dotazů a je nutno tento stav napravit[1].

[1] V elektronické příloze práce je možno nalézt užitečný SQL skript pro automatickou defragmentaci indexů

1.1.5 Stav statistik

O statistikách je psáno dále v kapitole 3.3 (Statistiky). Pokud z nějakého důvodu dojde k zastarávání informací ve statistikách, nebude SQL server schopen volit optimální plány pro vykonávání SQL dotazů, což logicky může vést ke snížení výkonu databázové vrstvy.

1.1.6 Kvalita SQL kódů

Tento faktor je velmi podstatný a často se stává, že neoptimálně formulovaný SQL dotaz dokáže nabourat celou výkonnostní koncepci databázové vrstvy. Proto je na tomto poli obrovský prostor pro optimalizaci formou refaktoringu SQL kódů. Refaktoring je proces, kdy se programátor zpětně vrací k již napsaným kódům a provádí jejich údržbu za účelem zvýšení výkonu nebo přehlednosti. Kvalita kódu je také relativní pojem a její význam se zvyšuje s rostoucím objemem dat. Dotazy, které na malém objemu dat fungují bez známek ztráty výkonu, mohou na velkých objemech dat způsobit zásadní výkonnostní problémy.

1.2 Definice základních pojmů

Při analýze výkonu můžeme narazit na tzv. **bottleneck** neboli **úzké hrdlo**. Úzké hrdlo je stav, při kterém se výkon nebo kapacita celého systému výrazně sníží kvůli dílčí komponentě v systému [2]. Záleží, v jakém kontextu tento výraz chápeme. Úzkým hrdlem může být například:

- problematický JOIN v rámci SQL dotazu
- SELECT dotaz v rámci SQL procedury
- SQL procedura v rámci celé databázové vrstvy

Při identifikaci úzkého hrdla vždy postupujeme od nejobecnějšího kontextu k nejkonkrétnějšímu. Úzké hrdlo se typicky projevuje až po nárůstu objemu dat v databázi. Do té doby obvykle zůstává skryto.

Timeout je stav, kdy nedošlo k dokončení vykonávání SQL dotazu v očekávaném čase. Pochopitelně se jedná o problematickou situaci, které je dobré se vyhnout. Naopak nastavení správné hodnoty pro timeout působí jako určitý „bezpečnostní“ prvek, který zabrání problematickému SQL dotazu v nepřetržitém vytěžování systémových prostředků. Timeout je možno obvykle omezit obecně přímo na SQL serveru nebo pak konkrétně pro dílčí SQL dotazy na straně aplikace.

Plán vykonávání (execution plan)[1] – předpis, pomocí kterého se vykonává daný SQL dotaz na fyzicko-logické úrovni. Je složen z „logických" operací, které jsou následně realizovány pomocí „fyzických" algoritmů. Každý dotaz je možno realizovat více způsoby. Je hledán vždy takový plán vykonávání, který má nejmenší „náklady".

SQL optimalizér[2] – subrutina v rámci SQL serveru, která zajišťuje sestavování plánů vykonávání pro SQL dotazy. Zajišťuje sestavení různých variant pro vykonání dotazu a jejich následné ohodnocení „náklady". Následně vybírá nejlepší variantu pro vykonání (tj. tu s nejmenšími „náklady") a tuto variantu kešuje pro další použití při volání totožného dotazu.

1.3 Nejčastější výkonnostní problémy v SQL kódech

Při optimalizaci SQL kódů je nutné si upřesnit, kde může docházet k úbytkům výkonu. Následuje **výčet problematických SQL konstruktů**. V některých případech jsou uvedeny i plány vykonávání, které demonstrují, jak může SQL optimalizér interpretovat naše dotazy v závislosti na různých okolnostech. Jsou uvedeny také popisy některých fyzických operací, které se vyskytují v plánech vykonávání, a které jsou podstatné pro danou operaci z hlediska výkonu.

1.3.1 Agregační dotazy

Agregační dotazy mají za úkol rozdělit data do skupin a z těchto skupin vypočítat požadované hodnoty – nejčastěji sumy, počty, průměry apod. Tyto dotazy mohou způsobovat ztráty výkonu v závislosti na mnoha aspektech. Velkou roli hrají indexy a statistiky na sloupcích, které jsou uvedeny v GROUP BY klauzuli, ale také na sloupcích, které používáme v agregačních funkcích (SUM, COUNT, AVG…) v SELECT klauzuli.

Pro agregační dotazy se nejčastěji používají fyzické operace STREAM AGGREGATE a HASH AGGREGATE.

STREAM AGGREGATE

Tato operace požaduje vstupní řádky seřazené podle sloupců uvedených v GROUP BY klauzuli. Pokud se agreguje podle více sloupců, pak není rozhodující pořadí sloupců, podle kterých je vstupní množina seřazena. Řazení je možno zajistit pomocí explicitně uvedené-

[1] V této práci bude používán výraz „plán vykonávání" místo „execution plan"
[2] V této práci je uváděn je stručně jako „optimalizér"

ho řadícího operátoru (ORDER BY) nebo například pomocí indexu. Řazení zajistí, že řádky se stejnou hodnotou v GROUP BY sloupcích budou následovat v sekvenci za sebou.

V případě, že je použita agregační funkce[1] bez klauzule GROUP BY, není nutné záznamy řadit - výsledkem je vždy jen jedna skalární hodnota. Proto se agregaci bez GROUP BY klauzule říká „**skalární agregace**“.

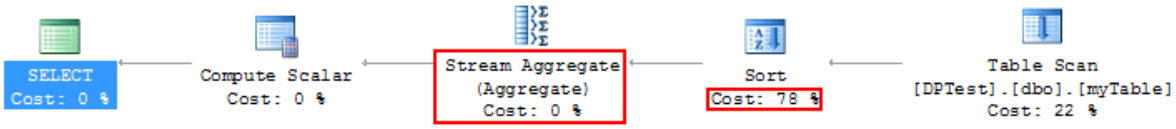
Zjednodušený pseudokód algoritmu pro operaci STREAM AGGREGARE je následující [3]:

```
Smaž aktuální agregační výsledky
Smaž aktuální hodnotu GROUP BY sloupců
PRO KAŽDÝ vstupní řádek
BEGIN
       IF (GROUP BY sloupce vstupního řádku) <> (aktuální GROUP BY sloupce)
       BEGIN
             Agregační výsledky → výstup
             Smaž aktuální agregační výsledky
             Nastav aktuální GROUP BY sloupce podle vstupního řádku
       END
       Aktualizuj agregační výsledky podle vstupního řádku
END
```

Je vidět, že tento algoritmus počítá vždy pouze jednu skupinu dat najednou. Jakmile je jedna skupina zpracována, jsou vráceny její výsledky (hodnoty agregačních funkcí) a začíná se zpracovávat následující skupina dat. Název STREAM AGGREAGTE (proudové seskupování) vychází právě z faktu, že se jednotlivé skupiny zpracovávají za sebou.

STREAM AGGREAGTE můžeme vidět v plánu vykonávání následujícího jednoduchého příkladu:

```
CREATE TABLE myTable (a INT, b INT)
SELECT SUM(a) FROM myTable GROUP BY b
DROP TABLE myTable
```



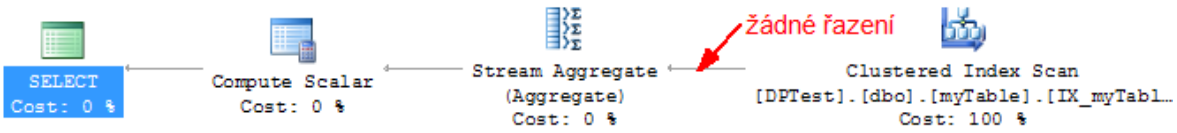
Obrázek 1 – STREAM AGGREGATE v plánu vykonávání

Je vidět, že před samotnou operací STREAM AGGREGATE je nutné provést řazení záznamů (SORT – v tomto případě podle sloupce „b“). Tato operace trvá 78% celkového

[1] Agregační funkce – SUM, AVG, MIN, MAX, COUNT apod.

času SELECT dotazu. Pokud by na tabulce existoval například **klastrovaný index**[1] pro sloupec „b“ bylo by řazení z plánu vypuštěno:

```
CREATE TABLE myTable (a INT, b INT)
CREATE CLUSTERED INDEX IX myTable b ON myTable(b)
SELECT SUM(a) FROM myTable GROUP BY b
DROP TABLE myTable
```



Obrázek 2 – STREAM AGGREGATE bez řazení v plánu vykonávání

Operace STREAM AGGREGATE bude poskytovat dobré výsledky pro skalární agregaci (není nutné řazení), pro tabulky, kde existuje index na GROUP BY sloupcích, nebo pro případy, kdy je tak jako tak potřeba data seřadit (je explicitně uveden příkaz ORDER BY).

HASH AGGREGATE

Další fyzickou operací pro realizaci agregačních funkcí je HASH AGGREGATE. Tato operace na rozdíl od STREAM AGGREGATE nevyžaduje řazení záznamů. Naopak má větší paměťové nároky a vytváří „blokování“. Toto blokování je způsobeno tím, že všechny skupiny dat jsou počítány najednou a během vykonávání nedochází k vracení dílčích výsledků. Výsledné hodnoty agregačních funkcí jsou vráceny až po zpracování všech vstupních řádků.

Zjednodušený pseudokód pro operaci HASH AGGREGATE je následující:

```
PRO KAŽDÝ vstupní řádek
BEGIN
	Spočítej HASH hodnotu z GROUP BY sloupců vstupního řádku
	IF existuje HASH hodnota v HASH tabulce
		Aktualizuj hodnoty v HASH tabulce podle vstupního řádku
	ELSE
		Vlož vstupní řádek do HASH tabulky
END
Vrať všechny řádky z HASH tabulky
```

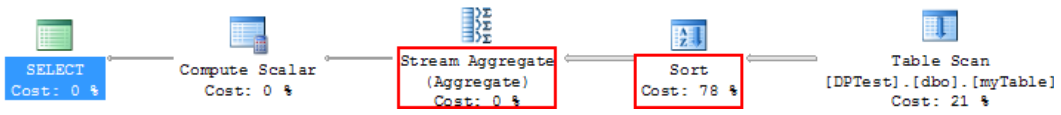
Operace HASH AGGREGATE pracuje s datovou strukturou typu HASH TABLE, kde klíčem je HASH kód z GROUP BY sloupců. Pro každou skupinu dat existuje jeden záznam v tabulce – z tohoto tvrzení lze snadno vypozorovat zvýšené paměťové nároky oproti operaci STREAM AGGREGATE. Paměťové nároky jsou přímo úměrné počtu skupin.

[1] Klastrovaný index – pojem je vysvětlen v kapitole 3.2

K operaci HASH AGGREGATE přistoupí optimalizér pro tabulku s mnoha řádky, které se budou agregovat do malého množství skupin. Většinou hraje roli velikost paměti, která bude pro operaci potřeba. Pro STREAM AGGREGATE potřebujeme data nejprve seřadit. Pro řazení musíme všechny vstupní řádky uložit do paměti. Při operaci HASH AGGREGATE však do paměti ukládáme pouze jeden řádek pro jednu skupinu.

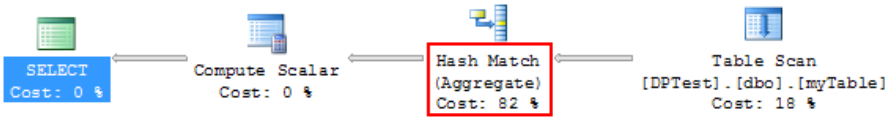
Pokud vložíme do tabulky 100 řádků, které budou rozděleny do 10 skupin, použije optimalizér operaci STREAM AGGREGATE (příklad 1). Při vložení 1000 řádků, které se rozdělí na 100 skupin, použije optimalizér operaci HASH AGGREGATE (příklad 2).

```
-- Příklad 1:
CREATE TABLE myTable (a INT, b INT)
DECLARE @i INT
SET @i = 0
WHILE @i < 100
BEGIN
        INSERT INTO myTable VALUES (@i % 10, @i)
        SET @i = @i + 1
END
SELECT SUM(b) FROM myTable GROUP BY a
```



Obrázek 3 – STREAM AGGREGATE a řazení (SORT) pro 100 řádků a 10 skupin

```
-- Příklad 2:
CREATE TABLE myTable (a INT, b INT)
DECLARE @i INT
SET @i = 0
WHILE @i < 1000
BEGIN
        INSERT INTO myTable VALUES (@i % 100, @i)
        SET @i = @i + 1
END
SELECT SUM(b) FROM myTable GROUP BY a
```



Obrázek 4 – HASH AGGREGATE pro 1000 řádků a 100 skupin

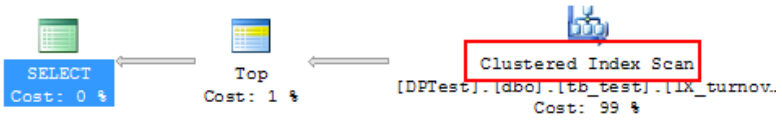
Operace HASH AGGREGATE poskytuje lepší výsledky než STREAM AGGREGATE pro datové zdroje, které obsahují velké množství záznamů. [4]

1.3.2 Řazení

Řazení je obvykle jednou z nejnáročnějších operací, se kterou se můžeme v plánu vykonávání setkat. Řazení je použito v případě explicitního uvedení klauzule ORDER BY nebo také „skrytě“ při používání klauzulí GROUP BY, DISTINCT, UNION apod.

Výkon řazení je závislý na existenci indexů na sloupcích v ORDER BY klauzuli. Pokud optimalizér nenalezne index, provádí řazení záznamů v pomocné tabulce za použití klasických řadících algoritmů. Pokud je použit klastrovaný index pokrývající všechny ORDER BY sloupce, pak je vždy logická operace SORT vyloučena a je nahrazena operací INDEX SCAN. Následující příklad předpokládá klastrovaný index na sloupci [turnover]:

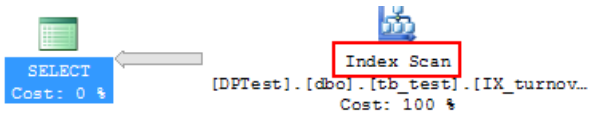
```
SELECT * FROM tb_test ORDER BY turnover
```



Obrázek 5 – CLUSTERED INDEX SCAN pro ORDER BY

Pokud je však použit **neklastrovaný** index, pak výsledek závisí na tom, zda index pokrývá všechny použité sloupce (v GROUP BY i SELECT klauzuli). Pokud jsou veškeré sloupce, které se vyskytují v dotazu, obsaženy v indexu, pak je operace SORT vypuštěna a je nahrazena operací INDEX SCAN. Následující příklad předpokládá neklastrovaný index na sloupci [turnover]:

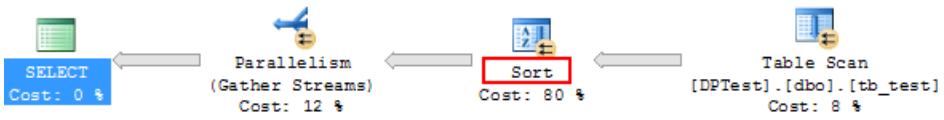
```
SELECT turnover FROM tb_test ORDER BY turnover -- pracujeme pouze s hodnotou turnover
```



Obrázek 6 - Operace INDEX SCAN pro ORDER BY

Pokud je předchozí příklad obměněn a do SELECT klauzule je přidán další sloupec, který není obsažen v indexu, pak je zvolen plán s operací TABLE SCAN a SORT, byť existuje neklastrovaný index na GROUP BY sloupcích.

```
SELECT turnover, date FROM tb_test ORDER BY turnover -- používáme i hodnotu date
```

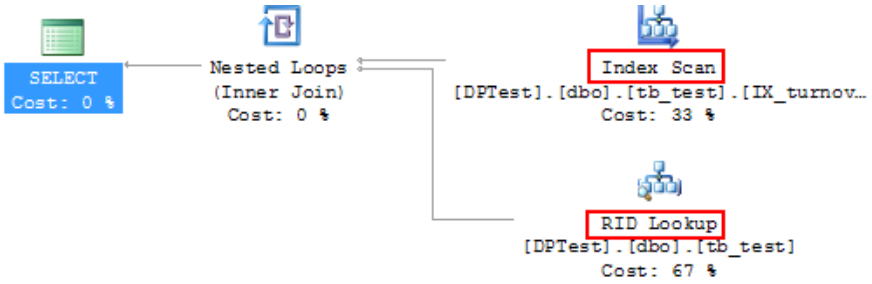


Obrázek 7 - Sort pro ORDER BY

Proč se optimalizér nerozhodl použít index, který je sám o sobě seřazený? Pokud index zcela nepokrývá používané hodnoty, musí být pro každý řádek prováděna operace typu RID LOOKUP pro dotažení dalších hodnot z tabulky. SQL optimalizér se snaží vždy najít plán, který zajistí vykonání dotazu v nejkratším možném čase. V tomto případě optimalizér rozhodl, že provádění operace RID LOOKUP pro každý řádek bude časově náročnější než

provést jednorázové seřazení. Pro vynucení použití indexu pro tento příklad je možno použít například FASTFIRSTROW hint. Tento hint (pokyn) umožňuje vrácení prvního řádku v nejkratším možném čase. To je právě realizováno využitím operace INDEX SCAN v kombinaci s RID LOOKUP. Je třeba si uvědomit, že operace SORT neumožňuje průběžné vracení řádků – vždy se musí počkat na dokončení řazení celé množiny dat a teprve potom může začít vracení řádků na výstup.

```
SELECT turnover, date FROM tb_test WITH (FASTFIRSTROW) ORDER BY turnover
```



Obrázek 8 – INDEX SCAN a RID LOOKUP pro ORDER BY

1.3.3 Filtrační dotazy

Význam klauzule WHERE (kvalifikovaný výběr) pro optimalizaci je zřejmý. Správné omezování množiny dat pomocí klauzule WHERE umožní ušetřit výkon, protože se sníží počet řádků, nad kterými se provádí další náročnější operace (agregace…) a sníží se objem dat proudících mezi serverem a klientem. V podstatě je žádoucí uvést klauzuli WHERE vždy, kdy je to možné.

Výkon operace WHERE je podmíněn logickým výrazem, který následuje za WHERE klauzulí. Jedná se o tzv. predikát, který může být vyhodnocen na hodnotu TRUE, FALSE nebo UNKNOWN. O tom, jak SQL optimalizér vyhodnocuje predikáty, je psáno v kapitole 2.1 (SQL jako „COST BASED“ systém). Kvalifikovaný výběr je obvykle prováděn pomocí operací TABLE SCAN, INDEX SCAN nebo INDEX SEEK.

TABLE SCAN:

Jedná se o fyzicko-logickou operaci, při které se prochází **všechny** řádky tabulky (tj. i všechny datové stránky) a dochází k vyhodnocení predikátu pro **každý řádek**. K této operaci se přistupuje, pokud není k dispozici žádný užitečný index. Výkon je závislý na **celkovém** počtu řádků v tabulce. Operace TABLE SCAN bude velmi neoptimální pro tabulky s velkým objemem dat, ze kterých je vybíráno pomocí predikátu jen velmi malé množství záznamů – optimalizační práce pak budou nasměrovány na zavedení indexu nad sloupce, které se vyskytují v predikátech. Tím docílíme například vynucení operace INDEX SEEK.

Operace TABLE SCAN se použije také v případě, kdy nejsou uvedeny žádné predikáty ve WHERE klauzuli a zároveň vracíme všechny sloupce z tabulky (resp. vracíme i takové sloupce, které nejsou obsaženy v žádných existujících v indexech).

INDEX SCAN:

INDEX SCAN je v podstatě totožný s operací TABLE SCAN. Výchozím prvkem pro procházení záznamů však není tabulka, ale index. Opět se prochází **všechny řádky** indexu. I zde závisí výkon na celkovém počtu řádků v indexu. V případě, že je použitý index klastrovaný, pak je operace INDEX SCAN zcela totožná s operací TABLE SCAN. Operace INDEX SCAN může být optimální strategií pro získávání řádků, pokud je tabulka malá nebo pokud většina řádků vyhovuje predikátu [5]. INDEX SCAN může být použit například tehdy, kdy vracíme **pouze** sloupce, které jsou obsaženy v indexu a zároveň nejsou uvedeny žádné predikáty ve WHERE klauzuli.

INDEX SEEK:

INDEX SEEK je operace, při které dochází k **vyhledávání** dat v závislosti na hodnotách sloupců zahrnutých v indexu. Není nutno načítat všechny datové stránky tabulky, ale jen ty, které vyhovují predikátu ve WHERE klauzuli[1]. Výkon operace INDEX SEEK tedy závisí na počtu řádků, které vyhovují predikátu. Pokud SQL dotaz vrací velké množství záznamů (jako hranice je uváděno 50% nebo 90% dat všech záznamů v tabulce), pak optimalizér přistoupí raději k použití operace INDEX SCAN. Obecně má INDEX SEEK dobrý výkon pro vysoce selektivní dotazy – tj. dotazy, které vrací méně než 10% (nebo podle některých zdrojů méně než 15%) řádků z tabulky [5]. Následující SQL dotaz bude zcela jistě realizován pomocí operace INDEX SEEK:

```
SELECT id FROM tb_test WHERE id = 1          -- tabulka tb_test má index nad sloupcem id
```

1.3.4 Kurzory

SQL server je postaven tak, aby podával dobrý výkon pro operace, které jsou založeny na **zpracovávání množin dat**. SQL server kurzor je prostředkem, který naopak umožňuje procesovat data **řádek po řádku**. V drtivé většině případů je použití kurzoru výrazně méně výkonné než použití alternativního SQL dotazu založeného na zpracovávání množin dat. Kurzory jsou tedy z optimalizačního hlediska téměř nepřípustné, protože:

[1] Poznámka – pokud je však možné vyhodnotit predikát jen z indexovaných sloupců

- mohou způsobit vyčerpání všech paměťových prostředků SQL serveru
- mohou způsobovat nadměrné uzamykání prostředků a následné deadlocky
- velmi často existují výkonnější alternativy založené na zpracování množin dat

Občas se může stát, že je použití kurzoru nevyhnutelné. V takovém případě je nutno velmi pečlivě specifikovat vlastnosti kurzoru. Následuje výčet optimalizačních typů pro používání kurzorů [6]:

- Pokud je to možné, nepoužívat kurzory.
- Uzavírat kurzory pomocí klauzule „CLOSE [cursor_name]“ nejdříve, kdy je to možné – operace CLOSE zajistí uvolnění všech sad výsledků a všech zámků provedených kurzorem.
- Dealokovat kurzor nejdříve, kdy je to možné pomocí klauzule „DEALLOCATE [cursor_name]“ – operace DEALLOCATE uvolní datové struktury kurzoru z paměti.
- Maximálně omezit počet řádků zdrojového recordsetu pomocí klauzule WHERE – nikdy neprovádět filtraci záznamů vlastním kódem až při procházení kurzorem.
- Maximálně omezit počet sloupců vracených kurzorem.
- Používat READ ONLY kurzory všude, kde je to možné. Při použití „updateable“ kurzorů dochází k uzamykání a snižuje se schopnost konkurenčního zpracování.
- Nepoužívat INSENSITIVE, STATIC a KEYSET kurzory, pokud je to možné. V těchto případech jsou v tempdb vytvářeny dočasné tabulky s kopií dat.
- Používat FAST_FORWARD kurzory, pokud je to možné. V drtivé většině případů je FAST_FORWARD kurzor dostačující. Kurzor je pouze pro čtení a umožňuje pouze dopředné procházení záznamů.
- Používat FORWARD_ONLY kurzory pro případy, kdy potřebujeme provádět UPDATE pomocí kurzoru a předpokládáme pouze dopředné procházení.
- Správně implementovat ošetření případných chybových stavů a ukončit cyklus procházení kurzoru v případě nečekaného stavu. S tím souvisí správná formulace podmínky pro ukončení cyklu, kterým kurzor procházíme.
- Pokud není možné jiné řešení, zpracovávat kurzor asynchronně jako operaci na pozadí.

Následující příklad demonstruje výkonnostní problémy kurzorů na primitivním[1] příkladě, kdy je suma hodnot počítána nejprve pomocí různých typů kurzorů a pak pomocí klasické agregační funkce SUM.

```
-- Šablona kurzoru pro počítání sumy "primitivním" způsobem
DECLARE c1 CURSOR FOR                -- posléze s obměnou na FAST FORWARD a STATIC
SELECT cisloA FROM tb_test2

DECLARE @cisloA INT, @suma BIGINT

OPEN c1

FETCH NEXT FROM c1 INTO @cisloA

SET @suma = 0

WHILE @@FETCH STATUS = 0
BEGIN
       SET @suma = @suma + @cisloA
       FETCH NEXT FROM c1 INTO @cisloA
END

CLOSE c1
DEALLOCATE c1
```

Následující tabulka orientačně srovnává doby provádění výše uvedeného SQL kódu (v různých variacích) nad tabulkou obsahující přes 2,5 milionu záznamů:

Tabulka 1 – Porovnání výkonu kurzorů vůči funkci SUM

	CURSOR (DYNAMIC)	FAST_FORWARD CURSOR	STATIC CURSOR	SUM(cisloA) bez kurzoru
Čas výpočtu [s]:	200	112	99	1

1.3.5 Sjednocení

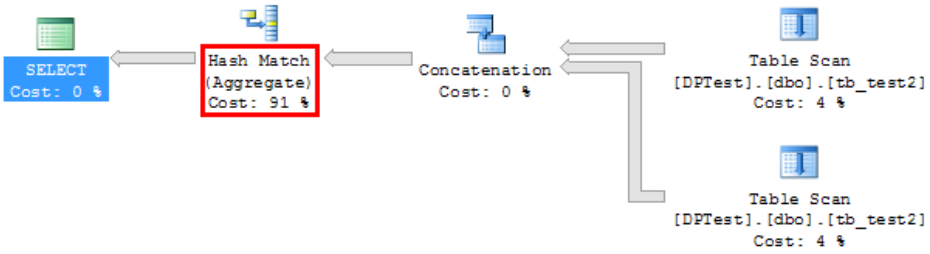
Klauzule UNION se používá ke sjednocení dvou recordsetů se stejnou strukturou. Tato poměrně „nevinná“ operace však může skrývat velmi zásadní výkonnostní problémy. Operace UNION musí recordsety spojit a zároveň vyloučit duplicitní řádky (což je zásadní rozdíl proti operaci UNION ALL). Vyloučení duplicitních řádků může být časově náročná operace a konkrétní způsob realizace závisí na okolnostech. Následující příklady ukazují různé způsoby sjednocení dvou recordsetů.

Příklad 1 ukazuje použití operace UNION na sjednocení dvou recordsetů, které vycházejí z tabulek, nad kterými není index a zdrojová data pro sjednocované množiny byla získána operací TABLE SCAN (není zajištěno pořadí záznamů). Je vidět, že samotné spojení (CONCATENATION) má zanedbatelné režie v rámci dotazu. Naopak vyloučení duplicit-

[1] Je zřejmé, že nikdo by tento příklad pomocí kurzoru neřešil.

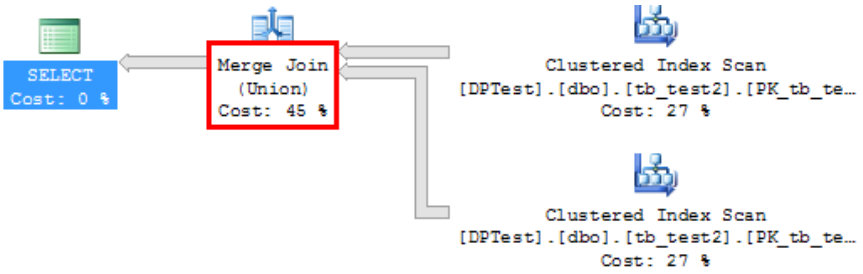
ních řádků (realizované operací HASH AGGREGATE – viz kapitola 1.3.1) vezme 91% času.

```
-- UNION - Příklad 1:
SELECT * FROM tb test2
UNION
SELECT * FROM tb_test2
```



Obrázek 9 – Plán vykonávání pro operaci UNION nad nesetříděnými daty

Příklad 2 ukazuje stejnou situaci (je použit tentýž SQL kód jako u příkladu 1). Na tabulce však v tomto případě existuje klastrovaný **unikátní** index pro jeden ze sloupců. Index zajistí, že data jsou seřazena. Pokud jsou obě vstupní množiny seřazeny podle **všech** sloupců, je možno použít poměrně rychlou operaci MERGE JOIN. Operace MERGE JOIN je popsána v kapitole 2.2.2 (JOIN HINTS). Plán vykonávání této operace je následující:

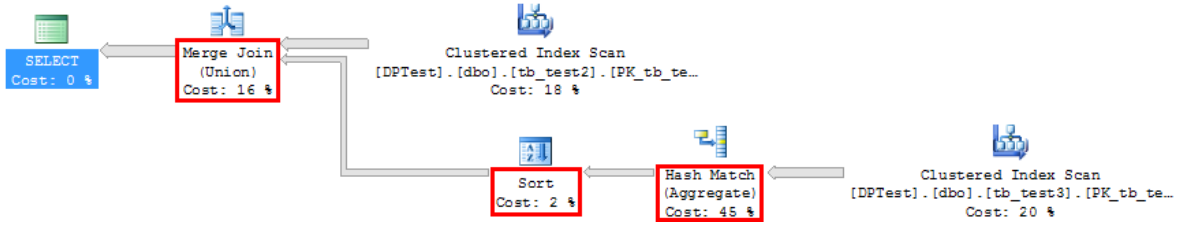


Obrázek 10 – Operace MERGE JOIN (UNION) použitá ke spojení dvou množin dat

Operace MERGE JOIN mohla být v tomto případě použita jen proto, že index je **unikátní**. MERGE JOIN totiž vyžaduje **řazení podle všech sloupců**, které jsou ve slučovaných množinách. V tomto případě je řazení zajištěno pouze na jednom sloupci, ale ve výběru jsou všechny sloupce (SELECT *….). Unikátní index zajistí, že pokud se řádky neshodují v hodnotě primárního klíče, nemohou být nalezeny další shody v rámci všech sloupců. Řazení podle dalších sloupců je tedy v tomto případě nepodstatné.

Příklad 3 ukazuje, že se v plánu vykonávání pro operaci UNION může vyskytnout řazení (SORT) spojené s operací MERGE JOIN. Někdy SQL optimalizér rozhodne, že je výhodnější data nejprve seřadit a pak aplikovat operaci MERGE JOIN, než provádět HASH AGGREGATE:

```
-- UNION - Příklad 3:
SELECT id, cisloA FROM tb_test2              -- unikátní klastrovaný index na sloupci id
UNION
SELECT cisloB, cisloC FROM tb test3          -- bez výběru primárního klíče, řazení není
                                               specifikováno
```



Obrázek 11 – Řazení a MERGE JOIN pro operaci UNION

Na předchozím obrázku je vidět, že data se nejenom seřadí, ale před seřazením se ještě agregují (HASH MATCH AGGREGATE). Optimalizér se tímto krokem rozhodl eliminovat duplicity už ve výchozí množině dat, aby redukoval počet řádků pro následné řazení.

Pokud je jisté, že řádky v obou sjednocovaných množinách jsou jedinečné, pak je rozumné využít operaci UNION ALL, jejíž režie v rámci dotazu jsou zanedbatelné (viz operace CONCATENATION - Obrázek 9).

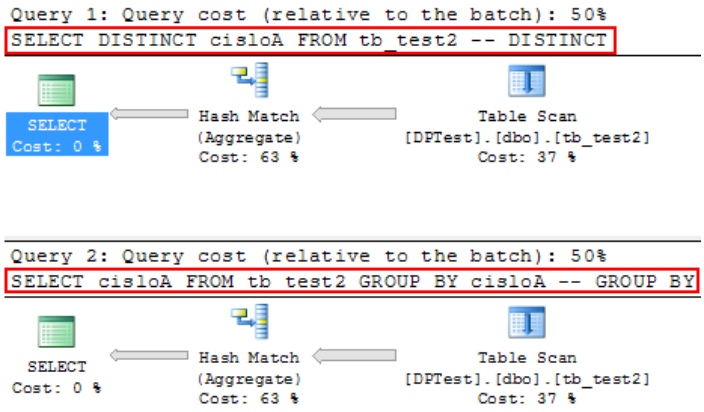
1.3.6 Klauzule DISTINCT

Klíčové slovo DISTINCT (v překladu „rozdílný“) se používá pro příkaz SELECT k odstranění duplicit ve výstupním recordsetu[1]. Je možno jej také použít v agregačních funkcích (SUM, COUNT…) pro určení, že výsledná agregovaná hodnota má být získána jen z hodnot, které jsou v recordsetu v daném sloupci unikátní. Operace DISTINT je stejně jako klauzule GROUP BY založena na agregaci. Pokud je stejný SQL dotaz napsán pomocí klauzule GROUP BY a pomocí klíčového slova DISTINCT, získáme velmi pravděpodobně stejné plány vykonávání. Následující jednoduchý příklad dokazuje výše uvedené tvrzení:

```
SELECT DISTINCT cisloA FROM tb_test2                    -- DISTINCT
SELECT cisloA FROM tb_test2 GROUP BY cisloA             -- GROUP BY
```

Výše uvedené dotazy SELECT jsou logicky zcela identické a získáváme také identické plány vykonávání:

[1] Dotaz SELECT implicitně používá klíčové slovo ALL

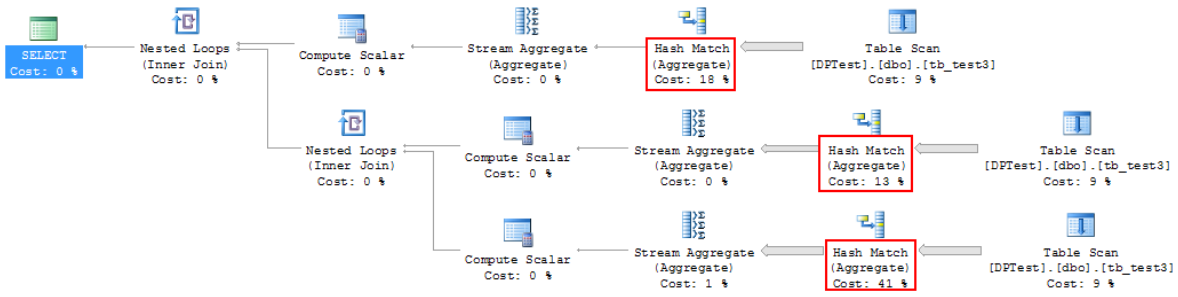


Obrázek 12 – Shodné plány vykonávání pro DISTINCT a GROUP BY

Při použití klíčového slova DISTINCT uvnitř agregačních funkcí je třeba mít na paměti, že pro každý agregovaný sloupec, je nutno provést samostatnou agregaci. Naopak pokud používáme více agregačních „DISTINCT“ funkcí, ale nad stejným sloupcem, stačí provést pouze jednu agregaci podle daného sloupce. Rozdíl je vidět na následujícím příkladu:

```
-- Příklad 1 - stejná agregační funkce na různých sloupcích:
SELECT COUNT(DISTINCT cisloB), COUNT(DISTINCT id), COUNT(DISTINCT cisloC) FROM tb_test3
```

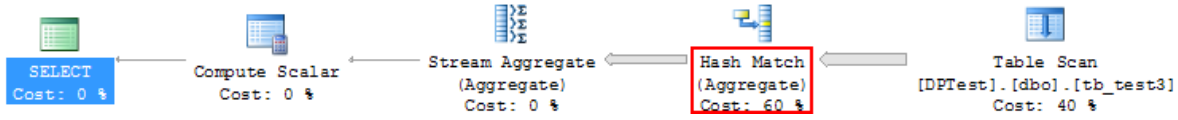
Při použití stejné agregační funkce (COUNT(…)) pro **různé** sloupce v parametru funkce (Příklad 1) je sestaven následující (relativně komplikovaný) plán vykovávání:



Obrázek 13 – Plán vykovávání pro COUNT (DISTINCT…) nad různými sloupci

Naopak pokud jsou použity různé „DISTINCT“ agregační funkce nad **stejným** sloupcem, pak stačí agregovat pouze jedenkrát. Situaci zobrazuje Příklad 2:

```
-- Příklad 2 - různé agregační funkce nad stejným sloupcem:
SELECT COUNT(DISTINCT cisloB), SUM(DISTINCT cisloB), AVG(DISTINCT cisloB) FROM tb_test3
```



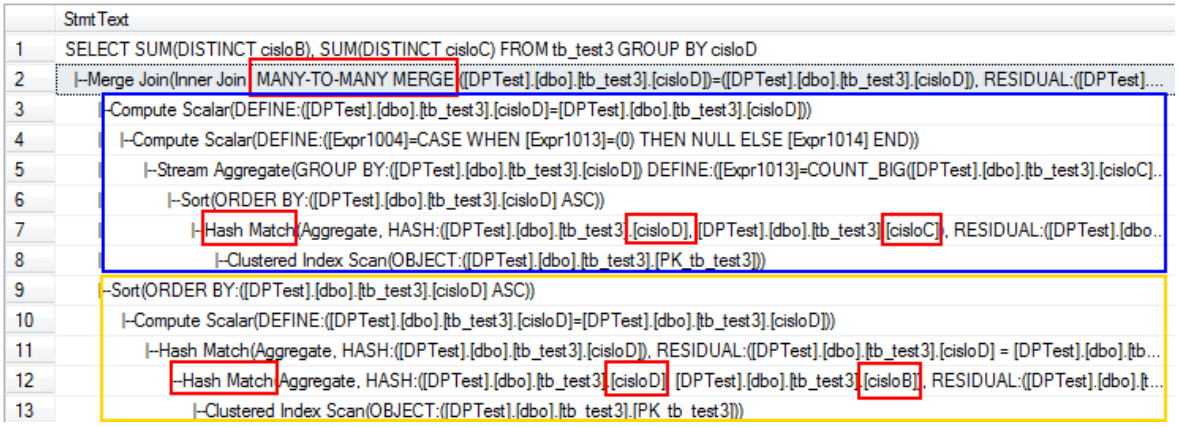
Obrázek 14 – Plán vykovávání pro agreg. DISTINCT funkce nad stejným sloupcem

Optimalizační zásahy se budou ubírat stejným směrem jako u klauzule GROUP BY (viz kapitola 1.3.1). V případě komplikovanějších DISTINCT dotazů můžeme výkon vylepšit

například rozdělením na vnořené dotazy. Následující příklad (Příklad 3) demonstruje, jak můžeme ušetřit výkon pro dotaz, kde jsou agregační DISTINCT funkce spojeny s agregací pomocí GROUP BY:

```
-- Příklad 3 - první část
SELECT SUM(DISTINCT cisloB), SUM(DISTINCT cisloC) FROM tb_test3 GROUP BY cisloD
```

Dotaz má za úkol vypočítat sumu unikátních hodnot ze sloupců [cisloB] a [cisloC]. Dané sumy mají však být vypočítány pro každou unikátní hodnotu ve sloupci [cisloD]. Při vykonání tohoto dotazu je získán tento plán vykonávání:



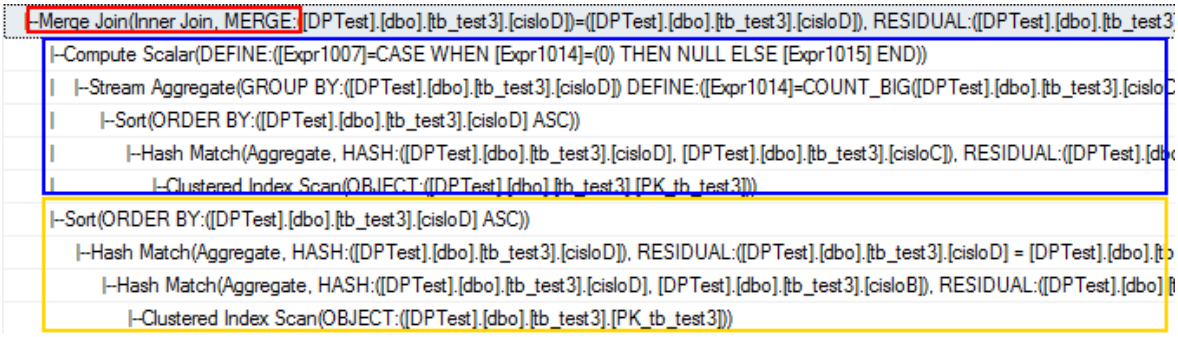
Obrázek 15 - MANY-TO-MANY MERGE JOIN (klauzule DISTINCT) [1]

Nejprve se provede agregace na sloupcích [cisloB] a [cisloD] (žlutý obdélník) a poté na sloupcích [cisloC] a [cisloD] (modrý obdélník). Tyto dvě množiny se do výsledného recordsetu spojí pomocí operace MERGE JOIN. MERGE JOIN je však proveden jako typ MANY-TO-MANY, který je méně výkonný než operace MERGE JOIN typu ONE-TO-MANY (více o typech operace MERGE JOIN v kapitole 2.2.2). Optimalizér v tomto případě není schopen určit, zda jsou hodnoty v jednotlivých množinách unikátní. Dotaz však můžeme přepsat poněkud komplikovanějším zápisem, ale získáme vyšší výkon, protože ke spojení množin bude použít výkonnější ONE-TO-MANY MERGE JOIN:

```
-- Příklad 3 - druhá část (přepis původního dotazu se zachováním stejné logiky)
SELECT
        sum b, sum c
FROM
        (SELECT cisloD, SUM(DISTINCT cisloB) AS sum_b FROM tb_test3 GROUP BY cisloD) R
        INNER JOIN
        (SELECT cisloD, SUM(DISTINCT cisloC) AS sum_c FROM tb_test3 GROUP BY cisloD) S
            ON R.cisloD = S.cisloD
```

[1] Kvůli velkému rozsahu není uveden grafický plán vykonávání, ale pouze textová stromová interpretace

V tomto případě je pomocí GROUP BY klauzule ve vnořených SELECT dotazech zajištěna unikátnost spojových sloupců v obou množinách. Plán vykonávání bude pro tento přepis následující:



Obrázek 16 – ONE-TO-MANY MERGE JOIN (klauzule DISTINCT)

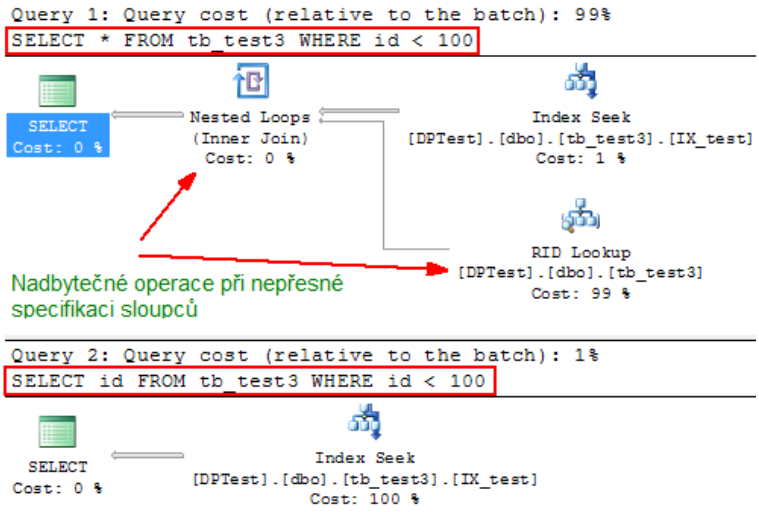
1.3.7 Používání symbolu „*" místo výčtu sloupců

Pevný výčet sloupců za klauzulí SELECT může zvýšit výkon SQL dotazu díky následujícím faktům:

- **Sníží se počet vstupně výstupních operací** – tj. není třeba načítat zbytečná data z fyzických tabulek a snižuje se také objem dat při ukládání do dočasných struktur v případě složitějších dotazů.
- **Sníží se riziko stavu „OUT OF MEMORY"**, po kterém následuje odkládání dat do tempdb a dochází ke zvyšování počtu IO operací.
- **Zjednoduší se plán vykonávání** - například v případě, že jsou použity jen sloupce pokryté indexem, odpadne operace typu RID LOOKUP, pokud je v k výběru řádků použita operace INDEX SEEK nebo INDEX SCAN.
- **Sníží se objem dat přenášených po síti.**
- **Zůstane zachován téměř stejný výkon** i po přidání nových sloupců do tabulek, které však nejsou potřeba ve výsledku daného dotazu.

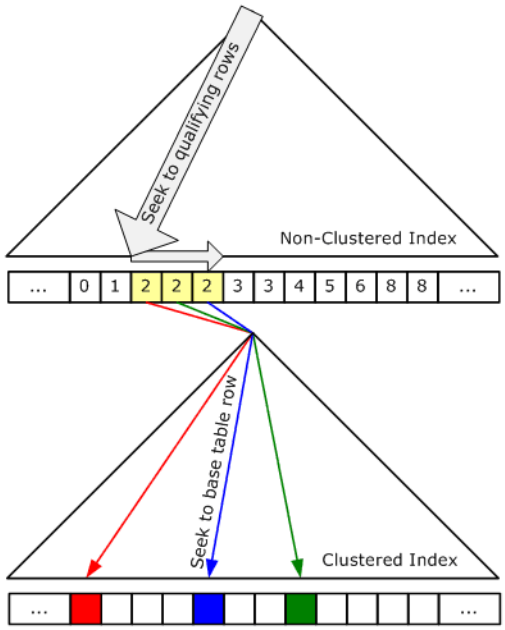
Následující příklad ukazuje zjednodušení plánu vykonávání, pokud vybíráme jen sloupce, které je možné přečíst z indexu (resp. sloupce pokryté indexem):

```
-- Příklad předpokládá neunikátní a neklastrovaný index nad tabulkou tb test3
SELECT * FROM tb_test3 WHERE id < 100
SELECT id FROM tb_test3 WHERE id < 100
```



Obrázek 17 – Rozdílné plány vykonávání pro výčet sloupců a pro použití „“*

V souvislosti s pokrytím sloupců indexem je nutno zmínit možnost zahrnout do indexu „neklíčové“ přídavné sloupce. Tyto sloupce jsou součástí indexu, ale není udržováno jejich řazení. Přidávání „neklíčových“ sloupců do indexu má za úkol eliminovat používání operace typu RID LOOKUP do tabulky, nad kterou je index vytvořen**. Operace RID LOOKUP** je výkonově poměrně náročná, protože provádí náhodné IO operace do klastrovaného indexu nebo do „heap table“. Princip operace RID LOOKUP je zobrazen na následujícím obrázku. [7]



Obrázek 18 – Princip operace RID LOOKUP

Přidávání „neklíčových“ sloupců do INDEXU se provádí pomocí klíčového slova „INCLUDE“ v příkazu CREATE INDEX. Příklad použití je následující:

```
CREATE INDEX IX_test ON tb_test3 (id) INCLUDE (cisloB)
```

Tento SQL kód vytvoří neklastrovaný neunikátní index nad sloupcem [id], ale zahrne do něj také sloupec [cisloB]. Říkáme, že index **pokrývá** sloupce [id] a [cisloB]. Pokud bychom nyní uvedli v SELECT dotazu výčet sloupců „[id], [cisloB]", došlo by pouze k operaci INDEX SEEK, protože veškerá požadovaná data je možno přečíst z indexu. SQL optimalizér dokáže analyzovat pokrytí sloupců indexem a vybírá nejlepší možnou variantu. Tedy i v případě, že v tabulce budou existovat jiné indexy (například separátně nad sloupcem [id] a separátně nad sloupcem [cisloB]) bude pravděpodobně použit index, který pokrývá všechny sloupce[1].

1.3.8 Používání COUNT(*)

Agregační funkce COUNT umožňuje mimo jiné zjistit celkový počet záznamů v tabulce. Pokud nepotřebujeme specifikovat žádné další parametry a zajímá nás skutečně jen počet záznamů v tabulce, pak je možné podstatně ušetřit výkon použitím systémové tabulky **[sysindexes]**. Princip použití demonstruje následující příklad. [8] [9]

```
SET STATISTICS IO ON                -- povolení výpisu statistik pro IO operace
GO
SELECT COUNT(*) FROM tb test3
GO
SELECT rows FROM sysindexes WHERE id = OBJECT ID('tb test3') AND indid < 2
GO
SET STATISTICS IO OFF               -- vypnutí výpisu statistik pro IO operace
```

Nyní můžeme vyhodnotit statistiky vstupně-výstupních operací, které jsme aktivovali pomocí příkazu SET STATISTICS IO ON:

Tabulka 2 – Vyhodnocení IO statistik pro porovnání COUNT(*) vs. [sysindexes]

Table	Scan count	Logical reads	Physical reads	Read-ahead reads	Lob Logical reads	Lob Physical reads	Lob read-ahead reads
[tb_test3]	1	**4952**	1	**4945**	0	0	0
[sysindexes]	1	3	2	0	0	0	0

Význam jednotlivých hodnot je popsán v kapitole 1.4.2 (Vstupně výstupní operace).

[1] Výběr indexu je řízen mnoha jinými parametry – tvrzení je platné pro tento jednoduchý příklad.

1.4 Měření a porovnávání výkonu SQL vrstvy

1.1.1 Obecné požadavky

Při měření je třeba eliminovat veškeré možné vlivy jiných aplikací a služeb. Pro přesné porovnání je vhodné mít samostatný stroj s operačním systémem a se službou SQL Server. Vhodné je pozastavit co nejvíce ostatních služeb, které mohou ovlivňovat měření – například služby indexující obsah disku nebo provádějící jiné náročné operace. Dále je třeba zajistit stejné výchozí podmínky pro měření v rámci služby SQL Server. Především je nutné před měřením vyprázdnit paměť keš SQL serveru. Pro vyprázdnění veškerých dat v paměti keš je možno použít kombinaci těchto dvou příkazů:

```
DBCC DROPCLEANBUFFERS          -- vyprázdnění buffer pool - uvolní veškerá přednačtená data
DBCC FREEPROCCACHE             -- vyprázdnění paměti keš - plány vykonávání apod.
```

Voláním příkazu „DBCC DROPCLEANBUFFERS“ je zajištěn stejný stav, jako by služba SQL server byla právě restartována. V paměti nejsou přednačtena žádná data. Veškeré datové stránky se musí znova načíst z disku. Je třeba si uvědomit, že pokud je po zavolání tohoto příkazu vykonán nějaký SQL dotaz několikrát po sobě, pak první z těchto volání trvá mnohanásobně delší dobu než volání následující (během prvního volání se data opět nakešují).

1.4.1 Časová náročnost

Pokud máme možnost porovnávat SQL dotazy v relativně neměnných podmínkách, zajímá nás především absolutní čas spotřebovaný SQL serverem na vykonání daného dotazu. Princip měření může být následující:

- Zjistíme časovou značku před spuštěním měřeného dotazu
- Vykonáme dotaz
- Zjistíme časovou značku po ukončení měřeného dotazu
- Stanovíme rozdíl časových značek (například v milisekundách)

Implementace měření času může být následující:

```
DECLARE @start DATETIME, @stop DATETIME
SET @start = GETDATE()

-- zde je měřený SQL dotaz --

SET @stop = GETDATE()
PRINT CAST(DATEDIFF(ms, @start, @stop) AS VARCHAR) + ' ms'
```

Měření časové náročnosti je možno provádět také sofistikovanější formou. MS SQL server nabízí příkaz **SET STATISTIC TIME**. Pokud je tato volba zapnuta, získáváme detailní informace o času potřebném na vykonání dotazu:

```
SET STATISTICS TIME ON
GO
-- zde je měřený SQL dotaz --
GO
SET STATISTICS TIME OFF
```

Informace jsou vypisovány do messages (standardní výstup, na který jsou zasílány zprávy funkcí PRINT - bohužel neexistuje možnost získat tyto hodnoty formou tabulky). Získáváme tyto hodnoty:

SQL server parse and compile time – čas, který byl potřeba na vyparsování SQL dotazu z řetězce a čas na sestavení plánu vykonávání.

SQL server execution time – celkový čas na vykonání SQL dotazu.

Získané hodnoty jsou dále rozděleny na „**CPU time“** a „**elapsed time“**. CPU time je čas spotřebovaný procesorem – tedy čas, po který byl procesor zaneprázdněn vykonáváním SQL dotazu. Elapsed time je celkový čas na vykonání dotazu. Rozdíl mezi „elapsed time“ a „CPU time“ je čas spotřebovaný na IO operace a jiné. Všechny hodnoty jsou udávány v milisekundách.

1.4.2 Vstupně výstupní operace

MS SQL server umožňuje kromě časové náročnosti monitorovat také náročnost na vstupně výstupní operace. Pro porovnávání výsledků IO statistik je nepodstatné aktuální vytížení serveru. Důležité je před měřením vyčistit paměť keš, protože kešování může výrazně ovlivnit počty IO operací. Pro sledování IO statistik se používá příkaz **SET STATISTICS IO**. Pokud je tato volba zapnutá, pak se do kanálu messages vypisují informace o počtech IO operací. Použití je následující. [10]

```
SET STATISTICS IO ON
GO
-- zde je měřený SQL dotaz --
GO
SET STATISTICS IO OFF
```

Význam hodnot, které nám poskytují IO statistiky, je následující:

- **Scan count** – počet provedených operací TABLE SCAN nebo INDEX SCAN
- **Logical reads** – počet stránek přečtených z paměti keš
- **Physical reads** – počet stránek přečtených z disku
- **Read-ahead reads** – počet stránek uložených do paměti keš pro tento dotaz

- **LOB logical reads** – počet stránek obsahujících rozsáhlé (LOB) datové typy (text, ntext, image, varchar(max), nvarchar(max), varbinary(max)), které byly přečteny z paměti keš v rámci dotazu
- **LOB physical reads** – počet stránek obsahujících rozsáhlé datové typy, které byly přečteny z disku v rámci dotazu
- **LOB read-ahead reads** – počet stránek obsahujících rozsáhlé datové typy, které byly uloženy do paměti keš pro tento dotaz

2 VLASTNOSTI JAZYKA SQL DŮLEŽITÉ PRO OPTIMALIZACI VÝKONU

Jazyk SQL se diametrálně liší od běžných programovacích jazyků (jako například C). Tato kapitola přináší zevrubný popis největších rozdílů, jejichž znalost je nezbytná pro psaní optimálních SQL dotazů.

2.1 SQL jako „COST BASED“ systém

Programovací jazyky jako například C, C++, C#, VB apod. využívají při vyhodnocování logických výrazů tzv. „short-circuiting“. Jedná se o ukončení vyhodnocování daného logického výrazu ve chvíli, kdy je již známá jeho výsledná hodnota a kdy ostatní operandy a operátory nemohou mít na výslednou hodnotu vliv. Programátoři této vlastnosti programovacích jazyků hojně využívají a to nejenom k optimalizaci zdrojových kódů. Často jsou logické výrazy konstruovány tak, že v případě vyhodnocování **bez** „short-circuiting“ budou způsobovat chyby. Klasický příklad využívání „short-circuiting“ je následující:

```
delitel = 0
If (delitel <> 0 AND delenec/delitel < 1) Then Do Something;
```

Výše uvedený kód (napsaný v pseudojazyce) by v případě vyhodnocení bez „short-circuiting“ způsoboval chybu „dělení nulou“. Naopak při využití „short-circuiting“ je po vyhodnocení první části logického výrazu zřejmé, že hodnota celého výrazu je FALSE a tudíž se neprovádí vyhodnocování dalších částí. Při vyhodnocování logických výrazů se postupuje zleva doprava v závislosti na prioritách jednotlivých operátorů.

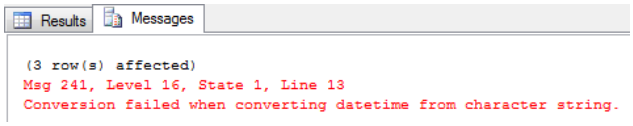
Jazyk SQL se však chová jinak. Přestože také využívá „short-circuiting“ (viz příklady), není možno nikdy spoléhat na pořadí vyhodnocování logických výrazů, jaké je obvyklé u jiných programovacích jazyků. SQL server interpretuje jazyk SQL v závislosti na „nákladech“ (cost based system). Při hledání nejvhodnějšího způsobu vyhodnocování postupuje SQL server následovně:

- Vyhodnotí se všechny podmínky a přiradí se jim „náklady“
- Vytvoří se suma „nákladů“ pro celý SQL dotaz
- Výsledná suma se porovnává s prahovou hodnotou, kterou určí SQL server a která udává hranici, od které je možno považovat navrhovaný plán za optimální
- Pokud je suma nákladů větší než dovolený práh, je vytvořena nová varianta vyhodnocení SQL dotazu a je připraven nový plán

Pro SQL jazyk obecně platí, že **pořadí** vyhodnocování predikátů **není předem přesně dáno a neexistuje žádný způsob, jak specifické pořadí vynutit** [11]. Chování jazyka SQL je demonstrováno na následujících příkladech. [11]

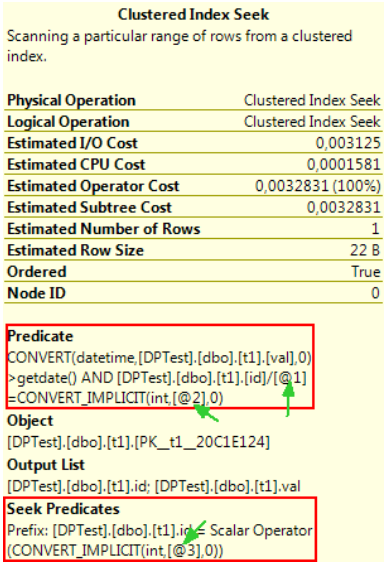
```
-- Příklad 1:
IF OBJECT ID('sc test') IS NOT NULL
        DROP TABLE sc_test

CREATE TABLE sc test(id INT PRIMARY KEY, textValue VARCHAR(20), textValue2 VARCHAR(10))

INSERT INTO sc test
SELECT 1, '20090402', 'a'
UNION ALL
SELECT 2, 'neplatná hodnota', 'b'
UNION ALL
SELECT 3, '20090401', 'c'

SELECT * FROM t1
WHERE id / 0 = 1                                    -- dělení nulou
       AND id = 2                                   -- výběr řádku
       AND CONVERT(DATETIME, textValue) > GETDATE() -- neplatný převod pro id = 2

DROP TABLE sc_test
```

Při vykonání kódu z příkladu č. 1 je očekávána chyba „dělení nulou“, protože první část predikátu ve WHERE klauzuli je „id / 0 = 1”. Nastane však následující chyba:



Obrázek 19 – Neočekávaná chyba při testu pořadí vyhodnocování predikátů

Pro porozumění tomuto stavu je nutno nahlédnout do plánu vykonávání SELECT dotazu. Protože je v predikátu uvedeno „AND id = 2” provede SQL server nejprve operaci CLUSTERED INDEX SEEK. Pro tuto operaci jsou predikáty rozděleny na tzv. „seek predicates“ a na „predicate“. „Seek predicates“ jsou predikáty, které se používají k vyhledávání v indexu. „Predicate“ jsou další predikáty, které není možné vyhodnotit přímo z indexovaných sloupců. Následující obrázek ukazuje vlastnosti operace CLUSTERED INDEX SEEK, která byla použita pro načtení dat.



Obrázek 20 – Vlastnosti operace CLUSTERED INDEX SEEK

Zelené šipky označují místa, kde došlo k tzv. „jednoduché parametrizaci“[1]. Jednoduchá parametrizace je nahrazení konstant parametry a její význam je v možnosti využití stejných plánů vykonávání pro příkazy, které se liší pouze v hodnotách parametrů. Vidíme, že náš predikát „AND id = 2” byl zvolen jako „seek predicate“ a bude tedy vyhodnocen nejdříve. Další predikáty jsou zařazeny do „predicates“. Pořadí, které vidíme ve vlastnostech operace INDEX SEEK (Obrázek 20) však nereflektuje skutečné pořadí vyhodnocování. Pořadí predikátů v sekci „predicates“ opět není nijak stanoveno. Fakt, že došlo k parametrizaci, neumožní optimalizéru určit, že daný výraz v predikátu „id / @1 = CONVERT_IMPLICIT(int, [@2], 0)“ je nesmyslný, protože hodnota parametru @1 není v době sestavování plánu známá. Optimalizér musí s predikátem počítat jako s každým jiným platným výrazem.

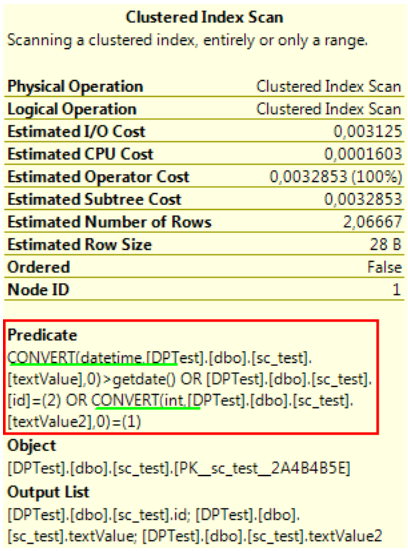
Příklad 2 je obdobou příkladu 1 – pouze je zaměněn SELECT uvnitř příkladu za následující:

```
-- Příklad 2:
SELECT * FROM sc_test
WHERE CONVERT(DATETIME, textValue) > GETDATE()
       OR id = 2                                    -- záměna AND za OR
       OR CONVERT(INT, textValue2) = 1              -- záměna AND za OR
```

V tomto případě je díky operátoru OR v predikátu použita operace CLUSTERED INDEX **SCAN**. Při pohledu do plánu vykonávání zjistíme, že dotaz nebyl automaticky parametri-

[1] V SQL serveru 2000 je označována jako „auto-parameterization“, v SQL server 2005 jako „simple parameterization“

zován (v predikátech jsou místo parametrů skutečné hodnoty). Při pokusu o vykonání je vrácena chyba konverze hodnoty „a“ na celé číslo. Když se podíváme na vlastnosti operace INDEX SCAN, vidíme následující situaci:



Obrázek 21 – Vlastnosti operace CLUSTERED INDEX SCAN

I když je mezi predikáty uvedena jako první konverze hodnoty [textValue] na DATETIME, je ve skutečnosti nejprve prováděna konverze hodnoty [textValue2] na INT. Opět tedy zůstává platné tvrzení, že pořadí predikátů v SQL dotazu nehraje roli.

Příklad 3 demonstruje fakt, že i SQL server provádí určitou formu „short-circuiting“. Některé zdroje uvádí, že jazyk SQL „short-circuiting“ vůbec nemá. Vždy však záleží na konkrétní realizaci v rámci daného SQL serveru a samozřejmě na samotném dotazu. MS SQL server 2005 vyhodnotí následující dva SELECT dotazy, aniž by nastala chyba „dělení nulou“ (která je nevyhnutelná u jazyků bez „short-circuiting“). V obou dotazech se vyskytuje predikát „0 = 1“, který dává celému výrazu za WHERE klauzulí výslednou logickou hodnotu FALSE. Vyhodnocení tohoto predikátu zamezí vyhodnocení nesmyslného predikátu „1 / 0 = 1“. Vidíme však, že nezáleží na pořadí predikátů – nejedná se tedy o „short-circuiting“ v pravém slova smyslu.

```
-- Příklad 3:
SELECT 'test' WHERE 1 = 0 AND 1 / 0 = 1      -- vyhodnotí se bez chyby a nic nevrátí
SELECT 'test' WHERE 1 / 0 = 1 AND 0 = 1      -- vyhodnotí se bez chyby a nic nevrátí
```

V tomto jednoduchém příkladu použije SQL optimalizér operaci CONSTANT SCAN.

2.1.1 Case statement

Pro úplnost je nutné zmínit způsob vyhodnocování podmínek ve výrazu CASE. Výraz CASE je vyhodnocován podobně jako například v jazyce C#. Postupně se vyhodnocují podmínky shora dolů a při prvním nalezeném logické výrazu, který se vyhodnotí jako TRUE, se provádění ukončí a další podmínky nejsou vyhodnoceny. Toto pravidlo platí pro oba typy CASE výrazů. Oba následující příklady budou vyhodnoceny bez chyby (a bude vrácena hodnota 1):

```
-- Příklad 1:
SELECT
	CASE
		WHEN 1 = 1 THEN 1
		WHEN 1 / 0 = 1 THEN 2		-- potenciálně chybný kód
		ELSE 3
	END

-- Příklad 2:
DECLARE @a INT
SET @a = 1
SELECT
	CASE @a
		WHEN 1 THEN 1
		WHEN 1 / 0 THEN 2		-- potenciálně chybný kód
		ELSE 3
	END
```

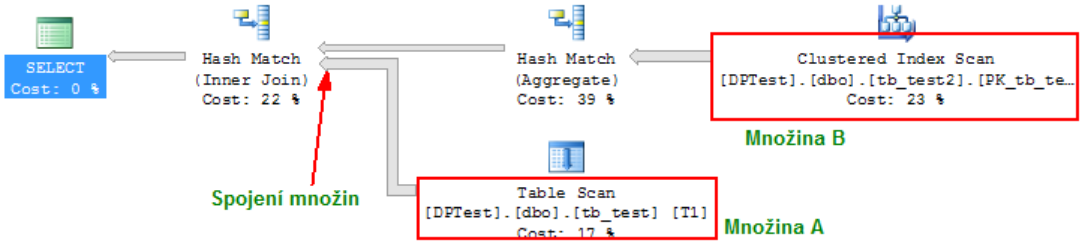
2.1.2 Množinové vyhodnocování

Základní vlastností jazyka SQL je zpracovávání dotazů pomocí množin. Během vykonávání dotazu jsou pomocí jednoduchých operací sestaveny **množiny**, které se následně spojují, slučují apod. Sestavování potřebných množin má na starosti optimalizér.

Příklad 1 ukazuje jednoduchý SELECT dotaz s vnořeným dotazem (pod-dotazem). V tomto případě není vnořený dotaz nijak závislý na vnějším dotazu (lze jej vykonat samostatně). Při vykonání tedy vzniknou dvě množiny – jedna pro vnější dotaz a druhá pro dotaz vnitřní. Tyto dvě množiny se posléze spojí a vznikne výsledný recordset.

```
-- Příklad 1:
SELECT                          -- vnější dotaz - MNOŽINA A
	*
FROM [tb test] T1
WHERE T1.[cisloA] IN
(
	SELECT                  -- vnitřní dotaz - MNOŽINA B
		T2.[cisloB]
	FROM [tb_test2] T2
)
```

Plán vykonávání dotazu z příkladu 1 je následující:

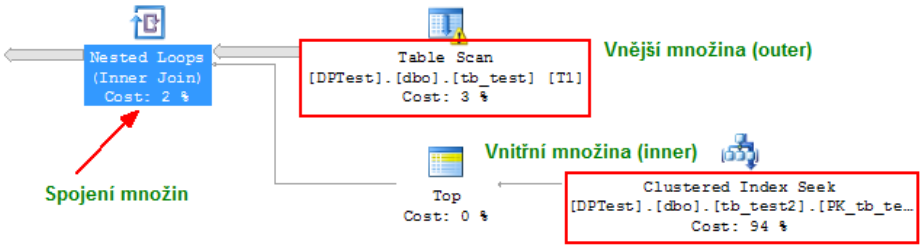


Obrázek 22 – Jednoduché spojování množin

Příklad 2 ukazuje komplikovanější situaci. V tomto případě je pod-dotaz provázán na vnější tabulku (tj. pod-dotaz nelze vykonat samostatně). Jedná se o takzvaný **korelovaný pod-dotaz**. Korelovaný pod-dotaz se vykonává pro každý řádek z vnější tabulky.

```
-- Příklad 2:
SELECT                                  -- vnější množina
       *
FROM [tb_test] T1
WHERE T1.turnover =
(
       SELECT TOP 1                     -- vnitřní množina
              cisloA
       FROM [tb_test2] T2
       WHERE T2.[id] = T1.[year]        -- vazba na vnější "množinu" (korelace)
)
```

Plán vykonávání dotazu z příkladu 2 je následující:



Obrázek 23 – Spojování množin pomocí operace NESTED LOOPS

V plánu vykonávání vidíme operaci **NESTED LOOPS** (dále jen NL). Tato operace pro každý řádek z vnější množiny (horní větev) provede nový výběr vnitřní množiny (spodní větev) a poté hledá ve vnitřní množině řádky, které se spojí s aktuálním řádkem z vnější množiny (také viz kapitola 2.2.2). V tomto konkrétním případě je pro každý řádek z tabulky [tb_test] provedena operace CLUSTERED INDEX SEEK nad tabulkou [tb_test2].

Je zřejmé, že operace NL bude používána především pro korelované pod-dotazy – tj. pro situace, kde se vnitřní množina mění. NL může však být použita i u nekorelovaných pod-dotazů. Stále však platí zásada, že vnitřní množina je sestavována znovu pro každý řádek z vnější množiny. Opakované sestavování vnitřní množiny pochopitelně představuje riziko ztráty výkonu. Pokud je vnitřní množina neměnná a je přesto použita operace NL, může

docházet u vnitřní množiny k optimalizaci pomocí operace TABLE SPOOL. Tato operace ukládá data z vnitřní množiny do dočasné struktury, aby nemuselo opakovaně docházet k operacím typu TABLE SCAN, které mohou být náročné na diskové operace.

2.2 SQL server HINTS

O tom, jakým způsobem bude SQL dotaz vykonán, rozhoduje **optimalizér**. Optimalizér má za úkol nalézt co nejefektnější plán vykonání dotazu. Svou činnost řídí v závislosti na indexech, klíčích, statistikách apod. Rozhodování optimalizéru je možné ovlivnit pomocí tzv. „hints“ (pokynů). Obecně se používání „pokynů“ nedoporučuje – optimalizér má k dispozici více informací, které může vyhodnotit a většinou produkuje nejlepší možné výsledky [8] [12]. Problém v použití „pokynů“ v SQL dotazu je ten, že je optimalizér nemůže nikdy překrýt, dokud je programátor z dotazu sám nevyloučí. Výkon dotazu bez specifikace „pokynů“ se může v čase vylepšit – například přibude nový index, aktualizují se statistiky nebo se vytvoří nové statistiky. Vždy by měly být stanoveny procesy, které zajistí zpětnou kontrolu použitých pokynů. Tyto procesy by měly kontrolovat, zda „pokyny“ stále plní svoji funkci a poskytují vylepšení výkonu.

Předtím než zavedeme „pokyn“ do SQL dotazu je lépe se zaměřit na následující body:

- Vynutit aktualizaci statistik nad relevantními tabulkami.
- Pokud se problém nachází uvnitř procedury, překompilovat danou proceduru.
- Revidovat stav indexů a provést změny, pokud jsou potřeba.

2.2.1 TABLE HINTS

„Table hints“ umožňují vynutit použití specifického indexu při vykonávání dotazu. Syntaxe pro použití je následující:

```
SELECT column_list FROM table_name
       WITH (INDEX (index_name))                    -- pokyn pro vynucení použití indexu
-- Příklad použití:
SELECT * FROM tb_test WITH (INDEX (PK_tbTest))      -- bude použit index PK_tbTest
```

Pro vynucení operace TABLE SCAN (tj. zakázání použití jakéhokoliv indexu) je možno místo „index_name“ použít číslo 0. Vynucení použití specifického indexu má smysl v případě, kdy nad tabulkou existuje více indexů, které mohou být použity pro počáteční načtení dat. Je nutno si uvědomit, že pro jednu tabulku je možno použít vždy pouze jeden index.

2.2.2 JOIN HINTS

„JOIN HINTS" umožňují specifikovat typ spojení dvou množin. Spojování se nepoužívá jen při „explicitním“ spojování tabulek pomocí JOIN příkazů v SQL dotazu, ale také v některých jiných případech – např. UNION, pod-dotazy apod. Spojení může být realizováno pomocí operací „LOOP JOIN“, „MERGE JOIN“ nebo „HASH JOIN“. Následuje popis vnitřní funkcionality JOIN operací, jejichž znalost je nezbytná pro správné rozhodování při volbě JOIN HINTS.

LOOP JOIN (LJ)

Pro každý řádek vnější tabulky se hledají shody ve všech řádcích vnitřní tabulky. Za shodné řádky považujeme takové, které se shodují v predikátech spojení. Fyzicky je operace realizována pomocí dvou vnořených smyček (loops) typu „for“. Pseudokód může vypadat následovně. [13]

```
Foreach row RO in the outer table
       Prepare new inner table
       Foreach row RI in the inner table
              If RO joins RI Then send {RO, RI} to the output
```

LJ poskytuje nejlepší výkon pro množiny, které obsahují malé množství záznamů. V některých případech je použití LJ nevyhnutelné – korelované pod-dotazy, operace CROSS JOIN, CROOS APPLY apod. LJ je jediný typ spojení, který může být proveden bez specifikace predikátů spojení.

MERGE JOIN (MJ)

Operace MJ vyžaduje, aby obě spojované množiny byly **seřazeny** podle všech spojových sloupců. Operace pak probíhá tak, že z každé množiny se vezme jeden řádek a tyto dva řádky se porovnají. Pokud jsou řádky shodné ve všech spojových sloupcích, pak je dvojice zaslána do výstupu a v druhé množině načteme další řádek. Pokud se řádky neshodují, je řádek s „nižší“ hodnotou zahozen (nemůže existovat žádný řádek v druhé množině, který by se spojil). „Zahozený“ řádek je nahrazen dalším řádkem z příslušné množiny. Pseudokód je následující. [14]

```
Vezmi první řádek R1 ze vstupu 1
Vezmi první řádek R2 ze vstupu 2
While není dosaženo konce jednoho ze vstupů
Begin
       If R1 == R2
       Begin
              Return (R1, R2)
              Vezmi další řádek R2 ze vstupu 2     -- pouze pro ONE-TO-MANY!
       End
       Else If R1 < R2
              Vezmi další řádek R1 ze vstupu 1
       Else
              Vezmi další řádek R2 ze vstupu 2
End
```

Výše uvedený kód platí pro MJ typu **ONE-TO-MANY**. Při nalezené shodě načteme nový záznam ze vstupu 2. To je možno udělat jen tehdy, pokud už nemůže na vstupu 1 existovat řádek, který by se spojil se „zahozeným" řádkem ze vstupu 2. Tento předpoklad naplníme jedině zajištěním unikátnosti řádků na vstupu 1 na spojových sloupcích. Pokud unikátnost není zajištěna, pak musí probíhat MJ typu **MANY-TO-MANY**. V tomto případě se uchovává kopie „zahozených" řádků ze vstupu 2, dokud se nezmění klíčová hodnota spojových sloupců na vstupu 1 (tj. než se přečtou všechny duplicitní řádky pro daný „klíč" ze vstupu 1). Typ MANY-TO-MANY je náročnější na výkon, protože používá pomocnou tabulku v tempdb pro odkládání záznamů. Pro použití typu ONE-TO-MANY musí být optimalizér schopen určit, že řádky v jedné z množin jsou unikátní. Toho je možno docílit použitím klauzulí typu DISTINCT nebo GROUP BY nebo musí existovat unikátní index na jednom ze spojových sloupců. Příklad převedení typu MANY-TO-MANY na ONE-TO-MANY je uveden v kapitole 1.3.6.

HASH JOIN (HJ)

Spojení typu HJ vyniká pro velmi velké množiny dat. HJ má také nejlepší schopnosti paralelizace[1]. HJ probíhá ve dvou fázích. První je „fáze sestavování" (build phase) a druhá je „fáze hledání" (probe phase). Při první fázi se přečtou všechny řádky z jedné tabulky (tabulka se nazývá „left" nebo „build" input) a počítají se HASH hodnoty ze spojových[2] sloupců. Výsledné HASH hodnoty jsou uloženy do HASH tabulky. Ve druhé fázi se postupně čtou řádky z druhé tabulky („right" nebo „probe" input) a opět se počítá HASH hodnota spojových sloupců. Poté se prohledává HASH tabulka sestavená v první fázi a hledají se shody. Protože hashovací funkce může vracet stejné HASH kódy pro různé vstu-

[1] Paralelizace – rozdělení práce na operaci mezi více procesorů. V nastavení SQL serveru je možno určit stupeň paralelizace (<= počet procesorů, 0 = max)
[2] V anglické literatuře se spojové sloupce označují „equijoin keys"

py, je u nalezené shody HASH kódů zjišťována ještě shoda hodnot spojových sloupců. Pseudokód operace HJ je následující:

```
ForEach řádek R1 v "build table"
Begin
       Vypočítej HASH kód ze spojových sloupců na řádku R1
       Vlož HASH kód a odkaz na řádek do HASH tabulky
End
ForEach řádek R2 v "probe table"
Begin
       Vypočítej HASH kód ze spojových sloupců na řádku R2
       ForEach řádek R1 v HASH tabulce
              If R1 odpovídá R2
                     Return (R1, R2)
End
```

Oproti LOOP JOIN a MERGE JOIN nemůže HASH JOIN v průběhu první fáze vracet řádky a dochází k blokování, dokud se nesestaví celá HASH tabulka. HASH JOIN také spotřebovává více paměti právě pro uložení HASH tabulky. Během vytváření HASH tabulky může dojít ke stavu „out of memory“. V takovém případě jsou malé části HASH tabulky ukládány na disk, což vede ke snížení výkonu. Optimalizér se vždy snaží snížit paměťové nároky zvolením menšího ze vstupů za „build input“.

Vynucení konkrétního typu spojení je možno realizovat pomocí „pokynu“, který má následující syntaxi:

```
…FROM table_one INNER [LOOP | MERGE | HASH] JOIN table_two
-- Příklad:
SELECT * FROM tb_test T1 INNER LOOP JOIN tb_test2 T2 ON T1.id = T2.id
```

Výše uvedený příklad je demonstrován na operaci INNER JOIN. „Pokyny“ lze aplikovat pochopitelně i na další typy spojení. Pokyn se uvádí mezi klíčové slovo identifikující typ spojení (INNER) a klíčové slovo (JOIN).

Typ spojení je možno určit také pomocí příkazu OPTION. OPTION určuje typ spojení pro celý dotaz. Použití je následující:

```
SELECT * FROM tb_test T1
       INNER JOIN tb_test2 T2 ON T1.year = T2.id
       INNER JOIN tb_test3 T3 ON T3.id = T2.id
OPTION(MERGE JOIN)            -- všechna spojení v dotazu budou typu MERGE JOIN
```

Následující tabulka srovnává vlastnosti výše uvedených fyzických operací pro realizaci spojování množin. [15]

Tabulka 3 – Srovnání fyzických operací pro realizaci spojování množin (JOINS)

	NESTED LOOPS JOIN	MERGE JOIN	HASH JOIN
Typ vstupů	Relativně malé vstupy s indexem na vnitřním (inner) vstupu	Pro střední až velké vstupy s indexem zajišťujícím řazení na spojových sloupcích nebo situace, kde jsou potřeba seřazená data po spojení	Střední až velké vstupy. Paralelní zpracování s lineárním rozdělením
Souběžné zpracování	Podpora velkého počtu souběžně pracujících uživatelů (procesů)	Podpora velkého počtu souběžně pracujících uživatelů (procesů) – pouze pro typ ONE-TO-MANY s indexem udávajícím pořadí spojových sloupců	Nejlepší pro malý počet souběžně pracujících uživatelů, kde jsou požadavky na vysokou propustnost
Vyžaduje equijoin[1] predikáty	Ne	Ano (s výjimkou pro FULL OUTER)	Ano
Podpora OUTER a SEMI JOIN	Pouze LEFT JOINS (FULL OUTER JOIN pomocí transformace)	Všechny typy JOIN operací	Všechny typy JOIN operací
Využívá paměť	Ne	Ne (ale může používat řazení, které využívá paměť)	Ano
Používá tempdb	Ne	Ano (pouze typ MANY-TO-MANY)	Ano (pokud nastane stav „out of memory“ a dochází k operaci „spill“)
Vyžaduje seřazení vstupů	Ne	Ano	Ne
Zachovává pořadí záznamů	Ano (pouze pro vnější (outer) vstup)	Ano	Ne

[1] Equijoin predikát – predikát s operátorem „rovnost“; mezi equijoin predikáty nepatří relace typu „větší než“, „menší než“ apod.

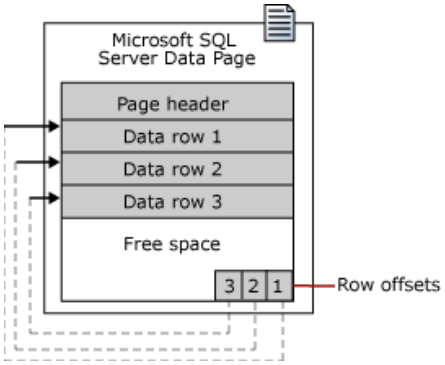
3 STRUKTURY DŮLEŽITÉ PRO OPTIMALIZACI VÝKONU

3.1 Fyzické uložení dat na disku

Základní jednotkou pro uložení dat na SQL serveru je **stránka** (page). Místo na disku alokované pro datový soubor (.mdf nebo .ndf) je logicky rozděleno na stránky, které jsou číslovány od 0 do *n*. Všechny diskové (IO) operace probíhají vždy na úrovni stránek – SQL server čte nebo zapisuje vždy celé stránky najednou. Velikost stránky je 8 KB (tj. 128 stránek na jeden MB). Každá stránka obsahuje na začátku hlavičku o velikosti 96 B a jsou v ní uloženy systémové informace o stránce (číslo stránky, typ stránky, velikost nevyužitého místa, ID objektu, který vlastní stránku). Typy stránek mohou být následující:

- **Data** – všechna data kromě rozsáhlých (LOB) datových typů (text, ntext, image, nvarchar(max), varchar(max), varbinary (max)) a také XML data, kde volba TEXT IN ROW je nastavena na ON
- **Index** – záznamy indexů
- **Text/Image** – data rozsáhlých (LOB) datových typů - text, ntext, image, nvarchar(max), varchar(max), varbinary(max), xml data a také datové typy varchar, nvarchar, varbinary a sql_variant, kde došlo k překročení velikosti 8 KB
- **Global Allocation Map, Shared Global Allocation Map** – informace o tom, kde jsou uloženy jednotlivé prostory (extents - viz dále)
- **Page Free Space** – informace o rozložení stránek a o volném místě ve stránkách
- **Index Allocation Map** – informace o prostorech využívaných tabulkou nebo indexem
- **Bulk Changed Map** – informace o prostorech změněných BULK operacemi od posledního BACKUP LOG příkazu
- **Differential Changed Map** – informace o prostorech změněných od posledního příkazu BACKUP DATABASE

Schéma stránky je na následujícím obrázku (Obrázek 24). Řádky jsou uloženy sériově hned za hlavičkou stránky. Stránka obsahuje také tabulku offsetů – ta udává vzdálenosti prvních bajtů všech řádků od začátku stránky. Tabulka offsetů je uložena na konci stránky a záznamy jsou uloženy v obráceném pořadí, než je skutečné pořadí řádků ve stránce.



Obrázek 24 - Schéma datové stránky SQL serveru

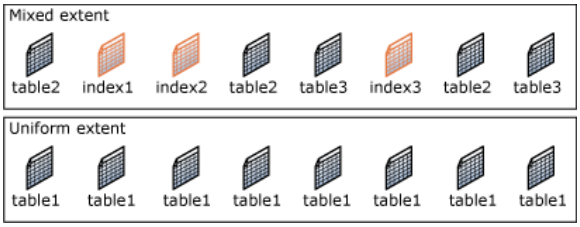
Maximální velikost řádku v jedné datové stránce nesmí přesáhnout velikost 8060 bajtů. Toto omezení není platné, pokud tabulka obsahuje sloupce typu varchar, nvarchar, varbinary nebo sql_variant. V takovém případě se po překročení hranice přesouvá jeden nebo více sloupců s proměnnou délkou do alokační jednotky ROW_OVERFLOW_DATA. Nejprve dochází k přesunu sloupců s největší velikostí. V původní datové stránce (v alokační jednotce IN_ROW_DATA) zůstane pouze 24-bitový ukazatel na přesunutá data. Přesun dat se provádí automaticky při operacích INSERT nebo UPDATE. V případě, že se velikost řádku sníží, jsou data přesunuta zpět do původní alokační jednotky.

Stránky jsou dále logicky seskupeny do tzv. „extents“[1] „Extent“ se skládá z osmi stránek[2], které jsou fyzicky uloženy za sebou. Existují dva typy „extents“:

- **Uniform extents** – všech 8 stránek je použito jedním objektem (tabulkou, indexem)
- **Mixed extents** – každá ze stránek může náležet jinému objektu

Nové tabulky nebo indexy obvykle alokují stránky v „mixed extents“. Pokud tabulka roste a obsadí alespoň 8 stránek, pak jsou data přeskupena a je vytvořen „uniform extent“ pro danou tabulku. Při vytváření nového indexu nad tabulkou, která má dostatečnou velikost, jsou automaticky vytvářeny „extents“ typu „uniform“ [16].

[1] Extent – v českém překladu „rozsah“, „rozloha“, „prostor“…
[2] Velikost „extent“ je 64 KB – tj. 16 „extents“ v 1 MB



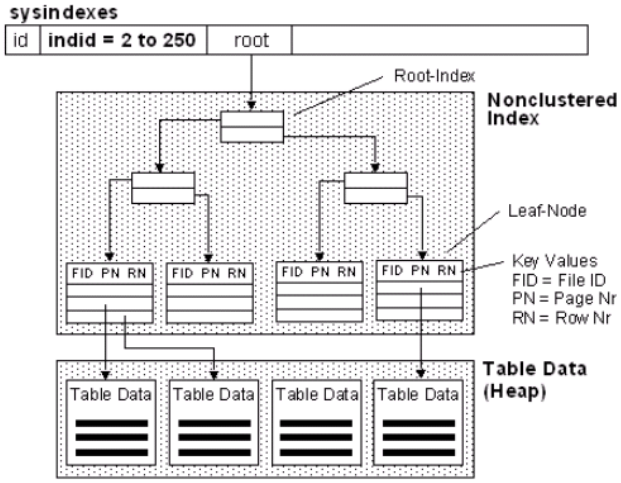
Obrázek 25 – Mixed a uniform extents

3.2 Indexy

Základní rozdělení indexů je na **klastrované** a **neklastrované** a je mezi nimi zásadní rozdíl. Nejprve je nutno objasnit pojem „heap[1] table“. **„Heap table“** je tabulka, jejíž řádky nejsou nijak uspořádány. Řádky jsou do datových stránek ukládány v pořadí, v jakém byly přidány do tabulky. Jinak řečeno se jedná o skupinu datových stránek, mezi kterými není žádný systém. Pokud je vytvořena nová tabulka bez indexu, tak je uložena právě jako „heap table“.

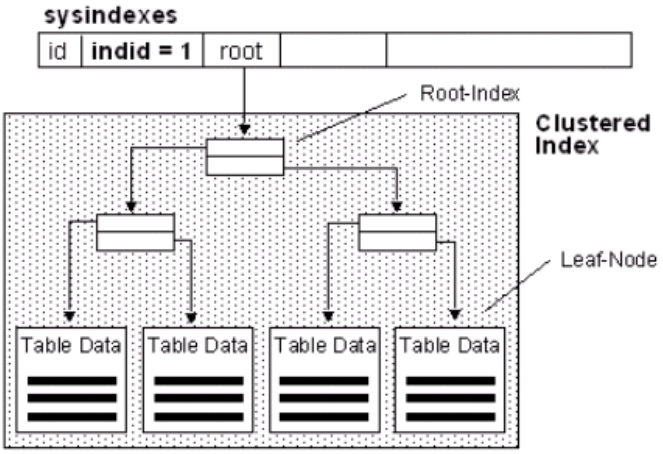
Neklastrovaný index je objekt, který umožňuje „systematizovat“ tabulku typu „heap table“. Původní data zůstávají uložena v „heap table“ ve stejném „chaotickém“ stavu, ale je k nim vytvořen rejstřík (index). Je zde analogie například s klasickou knihou, která na konci obsahuje rejstřík slov s odkazem na stránku, na které se dané slovo nachází. Podstatnou vlastností rejstříku v knize je abecední řazení pojmů. Z toho plyne opět analogie mezi vyhledáváním v tabulce a vyhledáváním v knize – buď můžeme přečíst všechny stránky knihy, abychom nalezli hledanou informaci („book scan“ = table scan) nebo se podíváme do rejstříku a nalezneme hledanou klíčovou hodnotu podle abecedního řazení – například přeskakujeme písmena, která nás nezajímají (index seek). Index je implementován datovou strukturou typu B-Tree[2], která umožňuje rychlé vyhledávání (resp. zajišťuje stejný (nízký) počet vyhledávacích skoků ve stromu pro vyhledání jakékoliv hodnoty – jinak řečeno: všechny listy stromu jsou ve stejné hloubce). B-Tree obsahuje jeden kořenový uzel (root node), dále pak libovolný počet mezilehlých uzlů (intermidiate leaf level nodes) a nakonec koncové uzly (leaf level nodes). V případě **neklastrovaného indexu** obsahují koncové uzly pouze ukazatele na datové stránky v „heap table“. Situaci ukazuje následující obrázek. [17] [18]

[1] Heap – halda, hromada
[2] B – od slova „balanced“ (vyvažovaný) nikoliv od „binary“ (binární)



Obrázek 26 – Schéma neklastrovaného indexu

Klastrovaný index naopak přímo organizuje data do konkrétní struktury. Nejedná se již o „heap table“, ale o tzv. „**index based table**“. Data v takové tabulce jsou již fyzicky seřazena přímo na disku. Analogií ke klastrovanému indexu je například telefonní seznam – adresa a číslo účastníka je uloženo přímo v „rejstříku“ a není nutno hledat podrobné informace na dalších stránkách. Schéma klastrovaného indexu ukazuje následující obrázek. [17]



Obrázek 27 – Schéma klastrovaného indexu

Ze schématu je vidět, že v koncových listech klastrovaného indexu jsou uloženy přímo data tabulky. Z tohoto principu vyplývají tato tvrzení:

- Nad tabulkou může existovat **pouze jediný** klastrovaný index – index je sám o sobě organizovanou tabulkou.
- Klastrovaný index **pokrývá všechny sloupce** tabulky – to znamená, že všechny sloupce tabulky jsou dostupné po operaci CLUSTERED INDEX SCAN bez nutnosti provádět operace typu RID LOOKUP do jiných datových stránek.

Dále můžeme dělit indexy na **unikání** a **neunikátní. Unikátní indexy** nepovolují duplicitní hodnoty na indexovaném sloupci (sloupcích). **Neunikátní indexy** duplicity povolují.

Klíčová hodnota indexu nemusí být pouze z jediného sloupce. Například telefonní seznam má dva klíčové sloupce – příjmení a jméno. Pokud je v indexu zahrnuto více sloupců, pak hovoříme o **složených indexech (composite indexes).** Složené indexy umožňují:

- Zvýšení pokrytí tabulky indexem – urychlí vracení sloupců, které jsou pokryté indexem
- Optimalizaci vyhledávání pro specifické dotazy

Srovnejme dotazy v následujícím příkladu. Předpokládáme tabulku [tb_phoneBook], která představuje telefonní seznam. Nad sloupci [surname] a [name] existuje klastrovaný index. [19]

```
-- Příklad 1:
SELECT * FROM tb_phoneBook WHERE surname = 'Kymla'
-- Příklad 2:
SELECT * FROM tb phoneBook WHERE surname = 'Kymla' and name = 'Vit'
-- Příklad 3:
SELECT * FROM tb_phoneBook WHERE name = 'Vit' and surname = 'Kymla'
-- Příklad 4:
SELECT * FROM tb_phoneBook WHERE name = 'Vit'
```

U příkladů 1, 2 a 3 umožňuje složený index zlepšit výkon vyhledávání, protože je možno aplikovat operaci CLUSTERED INDEX SEEK. Příklady 2 a 3 jsou navíc zcela rovnocenné – nezáleží na pořadí predikátů, v jakém je uvádí programátor. U příkladu 4 není možno využít INDEX SEEK byť je sloupec [name] plnohodnotnou součástí indexu. Sloupec [name] totiž udává řazení až ve druhé úrovni. Je to stejný případ, jako bychom v klasickém telefonním seznamu vyhledávali podle křestního jména. Pokud bychom chtěli optimalizovat vyhledávání podle křestního jména, museli bychom přidat další index, kde sloupec [name] je na prvním místě.

Složený index se může skládat maximálně ze 16 sloupců a maximální velikost všech sloupců nesmí přesáhnout 900 bajtů. V klíči indexu nemůžeme použít datové typy pro velké objekty (LOB), což jsou varchar(max), nvarchar(max), varbinary(max), xml, text, ntext a image. [20]

O možnosti zvýšení pokrytí tabulky indexem tzv. „neklíčovými sloupci“ je podrobně psáno v kapitole 1.3.7.

Indexy bezesporu poskytují obrovské úspory výkonu při provádění SQL dotazů pro výběr dat (SELECT). Kromě těchto výhod přináší indexy následující nevýhody:

- Indexy zabírají místo na disku – hlavně neklastrované indexy, protože duplikují některé sloupce. Tato nevýhoda je většinou nepodstatná, protože místo na disku je poměrně levné.
- Indexy zpomalují operace pro modifikaci dat – tj. INSERT, UPDATE a DELETE. V jednom ohledu umožní rychle nalézt záznam, který se modifikuje. Na druhou stranu je však nutné přestavět všechny indexy, pokud došlo k modifikaci klíčových hodnot. Tato nevýhoda je nepodstatná pro datová centra, která poskytují přehledy a reporty z existujících dat. Naopak v systémech pro transakční zpracování, kde neustále dochází k vkládání nových záznamů a k jejich modifikaci, se může jednat o podstatný problém.

Pokud chceme zjistit, kolik místa zabírá tabulka a její indexy na disku, můžeme použít následující příkaz:

```
EXEC sp spaceused název tabulky
-- Například:
EXEC sp_spaceused tb_test
```

Výsledek je ve formátu recordsetu s jedním řádkem. Sloupce jsou: „rows“ – počet řádků v tabulce, „reserved“ – počet KB rezervovaných pro tabulku, „data“ – počet KB zabraných daty tabulky, „index_size“ – velikost indexů v KB, „unused“ – počet KB nevyužitého místa.

3.3 Statistiky

Pro správnou funkcionalitu optimalizéru jsou potřeba tzv. statistiky. Statistiky udržují určité statistické informace o hodnotách v daných sloupcích tabulky. Tyto statistiky jsou nezbytné pro správné rozhodování optimalizéru při vytváření plánu vykonávání. Statistiky umožňují **odhadnout** „náklady“ pro různé navrhované plány vykonávání a následně vybrat nejlepší z nich. Statistické informace jsou uloženy jednak v tabulce [sysindexes] a dále v tzv. „statistics binary large object“ (statblob), který je uložený v interní tabulce SQL serveru.

Jednou z vlastností statistik je jejich automatické generování (volba AUTO_CREATE_STATISTICS). Tato volba je ve výchozím stavu zapnuta pro SQL SERVER 2000 i 2005 a pro 98% instalací zůstává povolena i při provozu. SQL SERVER umožňuje i manuální vytváření a aktualizaci statistik, nicméně tento postup není doporučovaný. Novou vlastností SQL SERVER 2005 je asynchronní generování statistik. Pokud je tato volba zapnuta, pak se statistiky negenerují ve vlákně, jež vykonává dotaz, ale jsou

generovány pomocí jiného vlákna na pozadí. Toto je výhodné pro aplikace s požadavkem na vysokou propustnost, protože vytváření statistik neblokuje samotný příkaz.

Automatické vytváření statistik pro sloupce probíhá například v těchto případech [21]:

- Použití sloupce v predikátu SQL dotazu – tj. ve WHERE nebo ON klauzuli.
- Vytvoření indexu nad sloupcem (nebo sloupci).

Se statistikami souvisí pojmy „selektivita“ a „kardinalita“. **Kardinalita sloupce** je počet různých hodnot, které se mohou ve sloupci objevit. Například sloupec typu BIT NOT NULL má kardinalitu 2 (hodnoty TRUE, FALSE). Uvádí se i pojem **kardinalita tabulky** – pak je ovšem míněn celkový počet řádků tabulky (spíše ve smyslu „mohutnost“ tabulky). **Selektivita indexu** je pak hodnota určená podle vztahu: [22]

$$selektivita\ indexu = \frac{počet\ vyhovujících\ řádků}{celkový\ počet\ řádků}$$

Rovnice 1 - Výpočet selektivity indexu

Selektivitu je vždy nutné vztáhnout k nějakému predikátu, abychom určili počet vyhovujících řádků. V podstatě je definice selektivity velmi nejasná. Některé zdroje uvádí, že selektivita indexu je:

$$selektivita\ indexu = \frac{celkový\ počet\ řádků}{kardinalita\ sloupce}$$

Rovnice 2 – Výpočet selektivity indexu (varianta 2)

Pokud budeme uvažovat selektivitu podle prvního ze vzorců, pak za vysoce selektivní indexy považujeme takové, které mají co nejnižší hodnotu selektivity (v tomto případě je selektivita definovaná jako hodnota z intervalu *(0, 1>*). Unikátní index má nejvyšší možnou selektivitu (neobsahuje duplicitní hodnoty). Při výběru indexů, které se mají použít pro vykonání dotazu, se pomocí statistik vybírají indexy s nejvyšší možnou selektivitou, protože umožní nejvíce redukovat objem dat pro následné zpracování [22].

Pro zobrazení **seznamu statistik** pro určitý objekt (tabulku, indexovaný pohled) můžeme použít proceduru [sp_helpstats]. Je třeba uvést parameter @objName, který identifikuje příslušný objekt. Následuje příklad použití:

```
-- vypsání seznamu všech statistik nad tabulkou [tb_test]
sp_helpstats @objname = N'tb_test'
```

Výše uvedený příklad vrací recordset se dvěma sloupci – [statistics_name] – udává jméno statistiky a [statistics_keys] – udává, na které sloupce se statistika vztahuje.

Zobrazení konkrétních statistických hodnot se provádí pomocí příkazu **DBCC SHOW_STATISTICS.** Je třeba uvést název objektu a název statistiky, kterou chceme zobrazit. Příklad použití je následující:

```
-- zobrazení statistiky s názvem _WA_Sys_00000004_023D5A04 na tabulce [tb_test]
DBCC SHOW_STATISTICS (N'tb_test', _WA_Sys_00000004_023D5A04)
```

Výše uvedené volání vrací tři recordsety se statistickými informacemi. **První** z nich obsahuje vždy jeden řádek, ve kterém jsou zobrazeny tyto informace: [23]

- **Name** – název statistiky
- **Updated** – datum a čas poslední aktualizace statistiky
- **Rows** – celkový počet řádků v tabulce v době poslední aktualizace statistik
- **Rows sampled** – počet řádků, ze kterých byly statistiky vyhotoveny (vždy je menší nebo rovno hodnotě „rows“)
- **Steps** – udává počet záznamů v histogramu rozdělení hodnot (viz dále); maximální hodnota je 200
- **Density** – hustota vypočítaná jako 1 / počet různých hodnot v prvním ze sloupců, které statistika pokrývá[1]
- **Average key length** – průměrný počet bajtů na hodnotu pro všechny sloupce, které statistika pokrývá
- **String index** – hodnota „Yes“ znamená, že v rámci této statistiky jsou vytvořeny také tzv. „string summary statistics“ – jedná se o statistiky pro zlepšení odhadu mohutnosti dotazů využívajících operátor LIKE (např. WHERE productName like ‘%Bike’); tyto statistiky jsou vytvářeny pro první ze sloupců, které statistika pokrývá (pokud je typu char, varchar, nvarchar apod.)

Druhý recordset obsahuje tzv. vektor hustot. Vrací jeden záznam pro každý „prefix“ vytvořený ze sloupců pokrytých statistikou. Například pokud statistika pokrývá sloupce (A, B, C), jsou zobrazeny hodnoty pro „prefixy“ (A), (A, B) a (A, B, C). Recordset obsahuje následující hodnoty:

[1] Od verze MS SQL server 2008 není tato hodnota optimalizérem používána a je zobrazena pouze kvůli kompatibilitě se staršími verzemi

- **All density** – je hodnota hustoty vypočítaná jako 1 / počet rozdílných hodnot; počet rozdílných hodnot je počet unikátních řádků pro výběr z daných sloupců statistiky (viz hodnota [columns])
- **Average length** – průměrný počet bajtů potřebný pro uložení hodnot pro dané sloupce statistiky (viz hodnota [columns])
- **Columns** – seznam sloupců, na které se příslušný řádek vztahuje

Třetí recordset vrací tzv. histogram. Histogram rozdělí hodnoty tabulky do maximálně 200 skupin a pro každou skupinu dat zobrazuje následující hodnoty (co jedna skupina, to jeden řádek v recordsetu). Histogram se počítá pouze pro první ze sloupců pokrytých statistikou. [24]

- **RANGE_HI_KEY** – horní hranice hodnot pro danou skupinu dat
- **RANGE_ROWS** – odhadovaný počet řádků tabulky, které mají hodnotu sloupce spadající do této skupiny dat (tj. menší než je hodnota RANGE_HI_KEY na tomto řádku a zároveň větší než hodnota RANGE_HI_KEY na předchozím řádku)
- **EQ_ROWS** – odhadovaný počet řádků tabulky, které mají hodnotu sloupce rovnu horní hranici této skupiny dat (RANGE_HI_KEY)
- **DISTINCT_RANGE_ROWS** – odhadovaný počet různých hodnot daného sloupce v této skupině dat
- **AVG_RANGE_ROWS** – průměrný počet řádků s duplicitní hodnotou sloupce v dané skupině dat; je vypočítána jako podíl RANGE_ROWS a DISTINCT_RANGE_ROWS (pro DISTINCT_RANGE_ROWS > 0)

Statistiky je možno kdykoliv explicitně aktualizovat příkazem **[sp_updatestats]**. V messages je možno vidět přehled všech statistik v databázi s informací, zda bylo třeba danou statistiku aktualizovat.

Statistiky se počítají buď z celkového počtu řádků v tabulce, nebo jen ze „vzorku“. Vzorek je možno určit buď procentem řádků, nebo absolutním počtem řádků. Tato nastavení je možno aplikovat pouze při ručním vytváření statistik pomocí příkazu CREATE STATISTICS[1].

[1] Více informací je možno nalézt na http://msdn.microsoft.com/en-us/library/ms188038.aspx

II. PRAKTICKÁ ČÁST

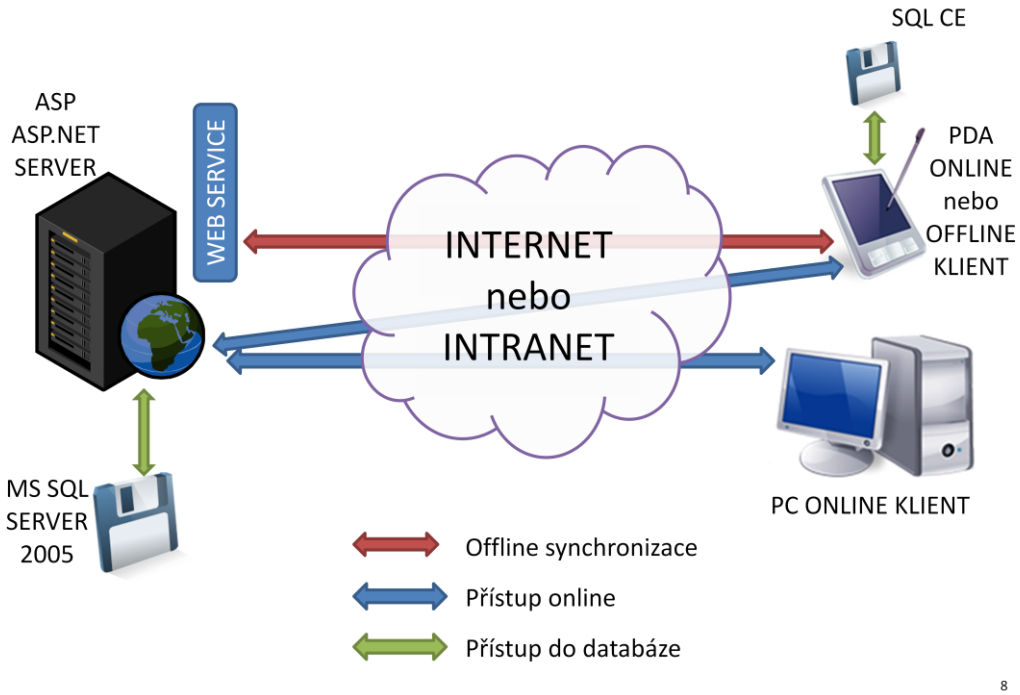
1 POPIS CLIENT-SERVER APLIKACE GETMORESYSTEM

Aplikace GetmoreSystem (dále jen GMS) je informační sytém určený především pro společnosti zabývající se splátkovým prodejem. Využívají jej však také jiné společnosti z bankovního sektoru i z jiných oblastí. Systém nabízí tyto základní aplikace:

- **CRM** (Customer relationship management) – aplikace pro řízení vztahů se zákazníky (správa kontaktů, aktivit…)
- **ERP** (Enterprise Resource Planning) – aplikace pro sledování obratů, nákladů, odměn, plánování obratů apod.
- **HRM** (Human Resources Management) – aplikace určená pro sféru HR (personalistika) – hodnocení zaměstnanců, sledování vzdělávacích aktivit a jejich úspěšnosti…
- **Workflow** – aplikace pro řízení firmy - správa požadavků, úkolů apod.
- **E-learning** – aplikace pro elektronické testování znalostí

1.2 Použité technologie

GMS je z velké části postaven na principu „client – server“ aplikace, kde klientem je prohlížeč Internetu. Serverová část je vytvořena na platformě ASP a ASP.NET.



Obrázek 28 – Struktura GMS

Pro uložení dat na straně serveru je využit MS SQL Server 2005. Hlavním klientem je tzv. PC ONLINE klient, který funguje v prohlížeči Internet Explorer (6+) a poskytuje plnou funkcionalitu systému. Dále je k dispozici PDA ONLINE klient. Tento klient využívá prohlížeč Internet Explorer CE na zařízeních typu Pocket PC. Klient poskytuje některé zjednodušené moduly z primárního PC ONLINE klienta. Posledním klientem je tzv. PDA OFFLINE klient, který umožňuje pracovat s daty[1] systému bez nutnosti aktivního připojení k serverové části. Klient je aplikace postavená na technologii MS .NET Compact Framework. Pro uchovávání offline dat na PDA je využit MS SQL Server CE. Synchronizace offline klienta probíhá pomocí webových služeb založených na ASP.NET.

1.2.1 Konverze ASP na ASPX

V současné době je dokončován projekt na konverzi všech ASP součástí GMS na ASP.NET. Tato konverze probíhá automaticky pomocí překladu zdrojových kódů z jazyka VBScript do C#. Z hlediska optimalizace SQL vrstvy se tímto otevírají nové možnosti. Především jde o jednodušší možnost kešování dat na straně aplikace nebo kešování datových zdrojů založeném na závislostech vypršení platnosti obsahu.

1.3 Požadavky na SQL vrstvu aplikace GetmoreSystem

GMS je značně rozsáhlý systém a nároky na SQL vrstvu jsou různé v jeho dílčích částech. Některé aplikace jsou zaměřeny spíše na analýzu dat (OLAP) a u jiných je naopak požadováno rychlé zpracovávání dotazů pro vytváření a modifikaci dat (OLTP). Například aplikace ERP funguje víceméně jako sekundární systém, který čerpá data z primárních účetních systémů. Uživatelé nepotřebují vidět aktuální data ihned po změnách, které provádí. Je zde proto velký prostor pro zpracovávání SQL operací na pozadí (importy dat, přepočty, kešování).

Zpracování SQL příkazů na pozadí je již v GMS aplikováno. Princip je takový, že náročné SQL operace se neprovádí přímo v kontextu pracujícího uživatele, ale jsou ukládány do pomocné tabulky jako „operace ke zpracování“. Ukládá se SQL dotaz, který má být vykonán a také HASH kód parametrů dotazu (pro zabránění zbytečného ukládání totožné operace vícekrát za sebou do fronty). Na SQL serveru je spuštěna služba SQL Server Agent,

[1] Synchronizuje se jen zlomek dat ze serverové databáze (tj. pouze data potřebná pro uživatele PDA)

která periodicky spouští úlohu (job), která kontroluje obsah tabulky s frontou operací a provádí vykonávání těchto operací. Tento princip má výrazný vliv na zvýšení odezvy aplikace pro uživatele. Uživatelé jsou informováni o počtu operací ve frontě. Uživatelé jsou poučeni, že pokud nejsou zpracovány veškeré operace na pozadí, nemusí být zobrazená data aktuální.

1.4 Směrování GMS na hostingový provoz

V současné době probíhá zpřístupnění systému také pro zákazníky, kteří nechtějí provozovat vlastní aplikační a databázové servery v rámci vnitřní sítě, ale chtějí používat aplikace na hostingovaných serverech přes Internet. Na jednom hostingovém serveru budou provozovány desítky současně běžících aplikací. Směrování systému do oblasti Internetu s sebou bezpochyby ponese zpomalení odezvy aplikace už jen díky změně „přenosového média“ (místní intranet → Internet). Optimalizace je tedy nevyhnutelná téměř na všech vrstvách. Tato práce pokrývá optimalizaci nejvrchnější z nich. Dále může následovat např. optimalizace objemu přenášených dat mezi serverem a klientem.

2 ZADÁNÍ OD SPOLEČNOSTI GETMORE

Je požadováno provedení analýzy SQL vrstvy GMS, zjištění nejzávaznější výkonnostních problémů a jejich následné odstranění. Cílem je maximalizovat výkon SQL vrstvy, aniž by došlo k výrazným úbytkům výkonu na nižších vrstvách. Některé případy je možno řešit přesunem operací z SQL do aplikační vrstvy, pokud tento stav přinese celkové zvýšení výkonu aplikace. Pro usnadnění optimalizačních prací bude připravena utilita, která umožní porovnávat vybrané fragmenty SQL kódu, resp. umožní změřit výkon aplikace před a po optimalizaci.

2.1 Požadované technologie

Utilita pro testování optimalizačních prací bude realizována jako tlustý klient pomocí knihovny Microsof.Windows.Forms.

Tabulka 4 – Permon – použité technologie

Verze Microsoft .NET Framework:	3.5
Databáze pro ukládání dat:	MS SQL Server 2005
Použitý vývojový nástroj:	Microsoft Visual Studio 2008
ORM vrstva:	Microsoft DBML
Zobrazování výsledků	Microsoft Reporting

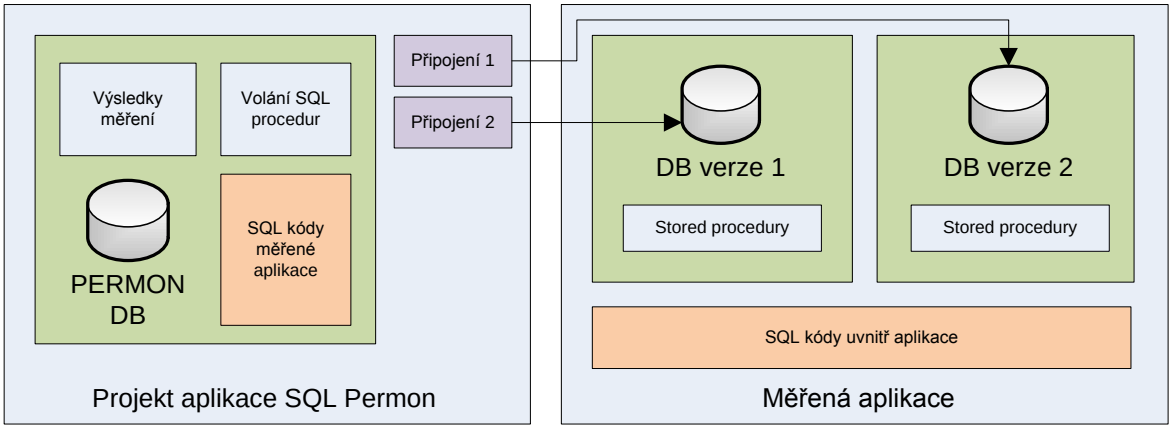
3 POMOCNÁ APLIKACE NA MĚŘENÍ VÝKONU SQL VRSTVY

Aplikace byla pracovně pojmenována „Getmore SQL Permon“ (SQL Performance Monitor – dále jen Permon). Prozatím je považována za interní pomocnou utilitu, které nebude dále nabízena zákazníkům společnosti Getmore.

3.1 Architektura aplikace

3.1.1 Vnější schéma aplikace

Na následujícím obrázku je zobrazeno „vnější“ schéma aplikace Permon:



Obrázek 29 – Vnější schéma aplikace Permon

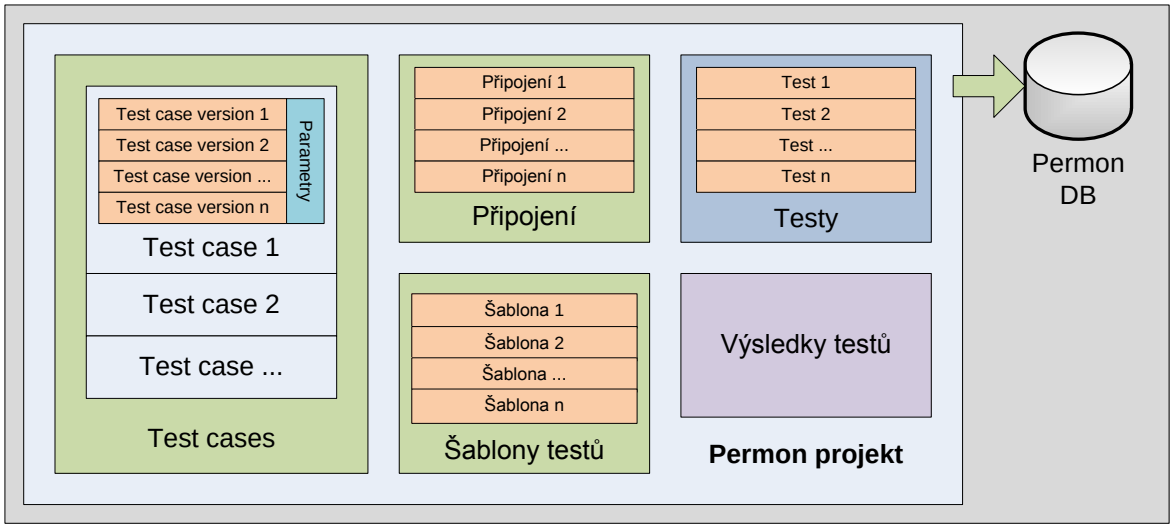
Obrázek 29 ukazuje, jakým způsobem bude aplikace Permon propojena s měřenou aplikací. Do projektu v aplikaci Permon bude nutno „ručně“ navést konkrétní měřené SQL kódy a také volání „stored procedur“ (myšleno EXEC sp_name…). Různé verze měřených databází budou v Permonu realizovány jako různá připojení – tedy co jedna verze databáze, to jedno připojení. Verze databáze pevně udává strukturu databáze. Do struktury patří tabulky, views, indexy, ale také zdrojové kódy SQL procedur. Povyšování struktury databáze na nové verze je realizováno pomocí tzv. „update“ balíčků.

V zásadě bude vždy potřeba mít alespoň dvě měřené databáze. Jedna z nich je výchozí – nebo také referenční. Druhá je pak databáze s aplikovanými optimalizačními úpravami (např. přidané indexy nebo upravené zdrojové kódy stored procedur). Je možné mít i více verzí měřených databází. Aplikace pak umožňuje měřit zadané SQL kódy a volání stored procedur na jednotlivých připojeních (verzích databázové struktury). Výše popsaná způsob umožní jednoduše vyhodnotit, jak optimalizační zásahy do struktury databáze ovlivnily

celkový výkon SQL vrstvy. Aplikace bude umožňovat porovnání výkonu pro jednotlivá připojení.

3.1.2 Vnitřní schéma aplikace – základní elementy

Aplikace Permon je vnitřně členěná do projektů. Pro každou měřenou aplikaci je nutno založit nový projekt. Schéma Permon projektu je na následujícím obrázku:



Obrázek 30 – Vnitřní schéma aplikace Permon

Veškerá data projektu jsou uložena přímo v databázi Permon - aplikace funguje jako tlustý klient.

Test case je základní element projektu. Představuje jeden testovací případ. Testovacím případem může být libovolný SQL kód (SELECT dotaz, volání stored procedury…). U každého testovacího případu může existovat více **verzí** (variant) jak lze daný dotaz implementovat. Vždy budou existovat alespoň dvě verze testovacího případu – výchozí a optimalizovaná. Teoreticky však může existovat *n* variant - aplikace pak umožní vybrat nejlepší variantu. SQL kód testovacího případu může obsahovat také SQL parametry (ve tvaru „@ParamName“). Pro test case pak může být specifikována tabulka hodnot pro jednotlivé parametry. Tato tabulka je dána SQL dotazem, který vrací recordset. Každý sloupec tohoto recordsetu představuje jeden parametr a každý řádek představuje jednu množinu parametrů pro dotaz. Názvy parametrů se musí shodovat s názvy sloupců.

Připojení již bylo částečně vysvětleno v předchozí kapitole (Vnější schéma aplikace). V projektu je připojení reprezentováno pouze připojovacím řetězcem. Všechna připojení v rámci jednoho projektu ukazují na stejnou databázi v různých verzích (tj. databáze obsahují stejná data a liší se pouze strukturou).

Testy umožňují logické seskupení naměřených výsledků. V podstatě každé měření probíhá jako test – každá naměřená hodnota se pak vztahuje k nějakému testu. Test je popsán svým názvem, popisem a časovou značkou vykonání. Testy se liší například počtem testovacích případů, počtem měřených variant testovacích případů nebo rozsahem měřených připojení. Test je možné chápat také jako množinu logicky spojených výsledků měření.

Šablona testu umožňuje specifikovat rozsah testu. Šablona udává, které testovací případy se měří v rámci testu, a specifikuje další parametry testování. Test daný šablonou je pak možné vícekrát opakovat se stejným objemem testovacích případů a se stejným nastavením (šablony testů nejsou realizovány v rámci této DP).

3.2 Aplikační logika, feature-set

Aplikace umožňuje spravovat projekty, spravovat testovací případy a jejich varianty apod. Tyto funkcionality jsou zřejmé a intuitivní. Jejich popis by byl nad rámec této práce. Dále jsou uvedeny jen významné funkcionality související s měřením výkonu.

3.2.1 Způsob měření výkonu

Měření testovacích případů probíhá podle následujícího pseudokódu. Testování je ještě rozšířeno o statistickou složku – každé měření se *n*-krát opakuje, aby se eliminovaly náhodné chyby měření:

```
ForEach TestCase do
Begin
        For TestNumber from 1 to N Do        -- statistické roznásobení
        Begin
                ForEach TestCaseVersion
                Begin
                        ForEach ParameterSet in ParameterTable
                                DoTest (TestCase, TestCaseVersion, ParameterSet)
                End
        End
End
```

Měření popsané v pseudokódu se ještě opakuje na všech připojeních definovaných v projektu. Výsledná granularita výsledků měření je zobrazena v následující tabulce.

Tabulka 5 – Výsledná granularita měřených dat

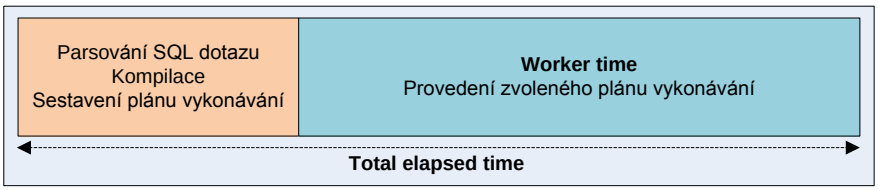
Připojení	Testovací případ	Verze testovacího případu	Číslo měření	Výsledky měření
…	…	…	…	…

Z tabulky je vidět, že měření pro jednotlivé sady parametrů se již nerozlišují. Vždy se změří daná varianta testovacího případu pro všechny sady parametrů a tento test se považuje za „black box“, pro který jsou určeny měřené veličiny.

Pro výše uvedenou granularitu záznamů jsou měřeny tyto veličiny:

- **Total elapsed time** – celkový čas, který zabralo procesování dotazu, včetně parsování SQL, sestavení plánu vykonávání a samotného vykonání.
- **Total worker time** – část celkového času (total elapsed time), který byl potřeba na samotné vykonání dotazu.
- **Total physical reads** – počet fyzických diskových operací typu „čtení".
- **Total logical reads** – počet logických operací typu „čtení" (tj. čtení z paměti keš).
- **Total logical writes** – počet logických operací typu „zápis".

Výše popsané hodnoty se vždy určují jako **průměr** z *n* měření. Časové hodnoty jsou určeny v milisekundách (teoreticky je možno zjišťovat s přesností na mikrosekundy – vychází se z počtu taktů procesoru).



Obrázek 31 – Elapsed time a worker time

3.2.1.1 Princip zjišťování měřených hodnot

Způsoby měření časové náročnosti a jiných veličin jsou popsány v kapitole 1.4. Tyto metody však neumožňují kompletní měření na úrovni systému. Jedná se spíše o uživatelské metody, které poskytují výsledky ve formě textových zpráv. Tento formát informací je však nevhodný pro systémové zpracování (parsování hodnot z řetězců apod.). MS SQL server 2005 nabízí další metodu měření statistických veličin pomocí systémových pohledů (views). Tato metoda byla zvolena i pro testovací aplikaci. Následující příklad ukazuje, jak je možno pomocí systémových view získat statistické informace o výkonu SQL dotazů, které byly vykonány na serveru:

```
SELECT
        creation_time
        , last_execution_time
        , total physical reads
        , total logical reads
        , total logical writes
        , execution_count
        , total_worker_time
        , total_elapsed_time
        , total elapsed time / execution count avg elapsed time
        , st.dbid               -- určuje ID databáze, nad kterou byl dotaz spuštěn
        ,SUBSTRING(st.text, (qs.statement_start_offset/2) + 1,
        (
                (
                        CASE statement end offset
                                WHEN -1 THEN DATALENGTH(st.text)
                                ELSE qs.statement_end_offset
                        END
                        - qs.statement_start_offset
                ) / 2
        ) + 1) AS statement text
FROM sys.dm exec query stats AS qs
        CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) st
ORDER BY total_elapsed_time DESC
```

Dotaz využívá systémový pohled [sys].[dm_exec_query_stats][1] pro získání statistických informací o čase vykonávání dotazu a o počtu IO operací. Dále je použita systémová funkce [sys].[dm_exec_sql_text] pro zjištění SQL dotazu z hodnoty [sql_handle] (a pro následnou filtraci podle SQL kódu dotazu).

Tato metoda měření má však také své nevýhody. Při měření je třeba dávat pozor na tato fakta:

- Do view [sys].[dm_exec_query_stats] vstupují informace ze všech databází na daném SQL serveru. Získávaná data je nutno filtrovat také podle databáze, na které se měří. Bohužel však existují omezení (viz následující body).
- Pokud je měřeným dotazem kód typu „EXECUTE [proc_name]“ (vykonání SQL procedury), pak není možno tento kód identifikovat v tabulce [sys].[dm_exec_query_stats] přímo přes řetězec s SQL kódem. Do tabulky jsou totiž ukládány až konkrétní dotazy vykonané uvnitř SQL procedury. Proto musí být měřené veličiny vypočítány součtem všech těchto „pod-dotazů“.
- Pokud je v měřené SQL proceduře vykonán dynamicky složený SQL kód (např. sp_execute), pak nelze zjistit, uvnitř které procedury a nad kterou databází byl dotaz vykonán.
- U dotazů typu „AdHoc“ (tj. u dotazů, které nejsou uvnitř SQL procedury – jsou volány napřímo) je nutno zjišťovat id databáze pomocí systémové funkce

[1] Bližší informace na: http://technet.microsoft.com/en-us/library/ms189741.aspx

[sys].[dm_exec_plan_attributes], protože hodnota [st].[dbid] (uvedená v předchozím příkladu) je pro tyto dotazy NULL[1].

Vzhledem k výše uvedeným bodům je výkon v aplikaci Permon měřen dvěma způsoby. „AdHoc“ dotazy jsou měřeny procedurou [dbo].[GetQueryStatisticData] a dotazy typu „EXECUTE [proc_name]“ jsou měřeny procedurou [dbo].[GetStoredProcedureStatisticData]. Obě procedury je možno najít v databázi aplikace Permon. Kromě toho je k dispozici také metoda „dummy measurement“, která funguje na principu časových značek před vykonáním a po vykonání dotazu. Následně se určí rozdíl časových značek v milisekundách. Tato metoda však umožňuje určit pouze hodnotu „total elapsed time“.

Měření výkonu procedur je ještě dále rozděleno na dvě varianty. Jedna umožňuje měřit jen procedury, které **nevyužívají** dynamické SQL a druhá i takové, které dynamické SQL využívají. Druhá z nich však nemůže filtrovat data podle databáze a názvu procedury (důvody viz výše). Postup je tedy takový, že se agregují všechny řádky, které se nachází ve view se statistikami bez ohledu na to, ke které databázi nebo proceduře patří. Před spuštěním měřené SQL procedury je view pochopitelně vyprázdněno. Je nutno zajistit, aby nad SQL serverem během měření neprobíhaly žádné jiné procesy. Ideální je stav, kdy na SQL serveru jsou pouze měřené databáze a všechny jsou v SINGLE USER módu, který zajistí přístup pouze pro aplikaci Permon.

3.2.2 Quick test

Aplikace umožňuje spustit okamžité porovnání dvou nebo více vzorků SQL kódu. Měřené kódy jsou navedeny do aplikace jako varianty jednoho testovacího případu. Uživatel pak může spustit funkcionalitu „Změřit“ nad daným testovacím případem. Možné parametry porovnání jsou následující:

- **Test name** – název testu (není povinný, resp. je generován automaticky).
- **Test description** – popis - není povinný (může sloužit pro jakoukoliv poznámku pro pozdější vyhodnocení testu,
- **Test count** – číslo *n* udávající počet opakování každého měření. Výchozí hodnota je 10.

[1] Více na: http://connect.microsoft.com/SQLServer/feedback/ViewFeedback.aspx?FeedbackID=374600

- **Connections** – výběr připojení, na kterých se má provádět měření. V sezamu jsou nabídnuta všechna připojení definovaná v projektu.
- **TC versions** – varianty testovacího případu, které se mají měřit.
- **Buffer cleaning** – indikace, že se má před každým měřením provést vyčištění bufferů. Pokud je nastaveno na TRUE, pak se před každým SQL příkazem volá příkaz DROPCLEANBUFFERS (viz kapitola 1.4). Zapnutí této volby zajistí nejpřesnější možné měření. Měření však trvá poměrně dlouho. Pokud je volba vypnuta, pak se vždy ignorují výsledky prvního měření (první měření může být zkresleno načítáním dat z disku – experimentálně ověřeno).

Test měří všechny vybrané varianty na všech vybraných připojeních a toto měření *n*-krát opakuje. Výsledky měření jsou přehledně zobrazeny formou tabulek (zdrojová naměřená data a statistické výsledky) a je vyhodnocena nejlepší a nejhorší varianta. Do databáze se po provedení testu ukládá informace o tom, která varianta testovacího případu byla vyhodnocena pro dané připojení jako nejlepší (má minimální hodnotu „total elapsed time“). Je také vyhodnoceno **procento optimalizace**. Procento optimalizace udává poměr rozdílu časů mezi nejhorší a nejlepší variantou vůči času nejhorší varianty:

$$procento\ optimalizace = \frac{nejhorší\ čas - nejlepší\ čas}{nejhoší\ čas} \cdot 100\%$$

Rovnice 3 – Výpočet procenta optimalizace

Čím vyšší procento, tím úspěšnější optimalizace SQL dotazu proběhla. Hodnota je vždy větší nebo rovna nule a zároveň menší než 100.

Výsledky je možno zobrazit také formou grafu, který srovnává kombinace všech variant a připojení, které byly měřeny v rámci testu.

3.2.3 Final test

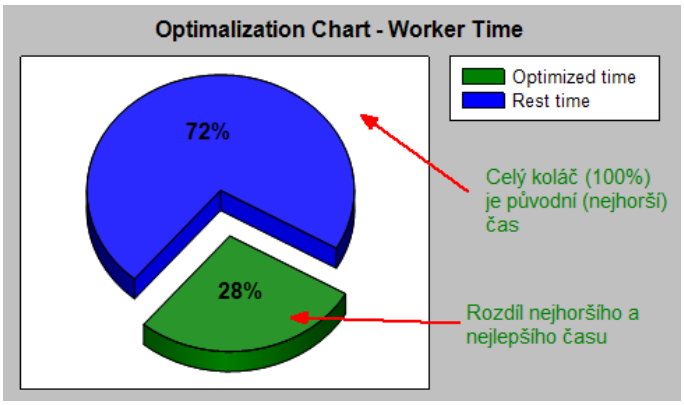
Final test umožňuje vyhodnotit optimalizační práce jako celek. Poskytuje ucelený přehled o tom, jakých výsledků bylo při optimalizaci dosaženo. Final test má následující nastavení:

- **Default connection** – připojení do referenční (srovnávací) databáze. Toto je většinou připojení do výchozí databáze, nad kterou nebyly provedeny žádné optimalizační zásahy.
- **Final connection** – připojení do finální verze databáze, která obsahuje všechny provedené optimalizační zásahy (indexy, upravené verze SQL procedur…).

- **Count** – počet opakování každého měření pro minimalizaci chyb měření.
- **Description** – popis testu.

Final test probíhá jako každý jiný test, ale má svá specifika. Do testu vstupují testovací případy, které mají nastaveno, že vstupují do „final“ testu (hodnota „Count for final test“ je větší než 0). Nad výchozím připojením (default connection) se však měří jen výchozí varianty testovacích případů (výchozí varianta je první uvedená u testovacího případu). Na konečném připojení (final connection) se měří všechny varianty testovacích případů. Final test pak provede srovnání sumárních hodnot měřených veličin takto:

- Vytvoří se sumární hodnoty (tj. suma celkového času, suma počtu diskových operací apod.) z výsledků testů, které byly provedeny na **výchozím** připojení – získáme informaci o výkonu databáze před optimalizací.
- Z testů provedených nad **konečným** připojením se vyberou jen ty **nejlepší** varianty testovacích případů. Z těchto nejlepších variant se vytvoří stejné sumární hodnoty jako v předchozím bodě. Získáme informaci o výkonu databáze po všech optimalizačních zásazích.
- Sumární hodnoty získané z výchozího a z konečného připojení jsou srovnány formou grafů a je vypočítáno procento optimalizace. Výpočet procenta optimalizace se provádí buď z hodnot „total worker time“ nebo „total elapsed time“. Vzorec pro výpočet je uveden v kapitole 3.2.2 (Rovnice 3). Procento optimalizace je pro názornost zobrazeno formou koláčového grafu, který vypadá následovně:



Obrázek 32 – Ukázka výstupu z final test

Zobrazení výsledků konečného testu je realizováno pomocí technologie Microsoft Reporting.

3.2.4 Automatické vytvoření projektu z trace logu

Pro dokonalé otestování (porovnávání) výkonu SQL vrstvy by bylo nutné do projektu navést všechny možné SQL kódy, které se v aplikaci vyskytují. To by však bylo nerealizovatelné pro aplikaci takového rozsahu, jako je například GMS. Proto byla zavedena funkcionalita, která umožní automaticky vygenerovat projekt a naplnit jej testovacími případy podle zachyceného provozu (trace) z programu Microsoft SQL Server Profiler. Před importem trace je vhodné z něj odfiltrovat nežádoucí SQL kódy, které není potřeba měřit. Při samotném logování SQL provozu je také nutné nastavit zachytávání správných událostí tak, aby bylo možné provoz opětovně přehrát („playback“). Šablona s vhodným nastavením je součástí elektronické přílohy této práce.

Nad takto automaticky vygenerovaným projektem je pak možné spustit „final test“, který v podstatě pouze provede „playback“ zachycených událostí a změří jednotlivé hodnoty. Není vhodné nastavit vícenásobné opakování měření, protože projekt samotný obsahuje dostatečně velký vzorek kódů pro měření. Při zachytávání provozu pro tento typ automatického projektu je vhodné zachytávat práci jednoho uživatele, který v aplikaci simuluje běžné pracovní úkony od přihlášení až po odhlášení.

4 OPTIMALIZACE SQL VRSTVY APLIKACE GETMORESYSTEM

4.1 Analýza SQL vrstvy aplikace

Prvním krokem byla analýza databázové vrstvy aplikace. Bylo nutno zjistit, které dotazy způsobují největší výkonnostní problémy. Analýza byla sestavena z následujících kroků:

4.1.1 Analýza statistik dotazů

Systémové view [sys].[dm_exec_query_stats], které je používáno aplikací Permon k měření výkonu SQL dotazů, může být použito také jako analytický prostředek pro identifikaci „pomalých“ dotazů. Je nutné si uvědomit, že toto view se vyprázdní při každém restartu služby SQL server a také při explicitním volání vyčištění paměti keš. Pokud je zaručen dostatečně dlouhý provoz bez restartu a bez vyčištění keše, pak máme okamžitě k dispozici cenná statistická data.

Získání nejpomalejších dotazů je možno provést SQL dotazem, který je uveden jako příklad v kapitole 3.2.1.1. Tento dotaz vrátí statistické informace o všech samostatně vykonaných SQL dotazech – tj. zahrnuje všechny dotazy uvnitř SQL procedur, všechny dynamicky složené dotazy apod. Je pak poměrně náročné zpětně určit, kde je daný SQL kód v aplikaci používán. Někdy není dohledání daného SQL kódu v aplikaci podstatné, protože není vždy nutné provést zásah přímo do kódu, ale k optimalizaci dojde například zavedením indexu. SQL kód nám však bude sloužit jako prostředek pro měření. Dalším úskalím jsou parametry uvnitř dotazů. Často jsou zachyceny parametrizované SQL dotazy, kde neznáme hodnoty parametrů (data jsou ve view ukládána pro potřeby optimalizéru, který nepotřebuje znát pro svoji činnost konkrétní hodnoty). Pokud chceme pak kód použít pro měření, je nutno parametry specifikovat.

Dále můžeme ze systémových view zjistit nejpomalejší a nejčastěji používané SQL procedury. Aplikace GMS využívá SQL procedury ve velké míře, a proto má tento bod velký význam. SQL procedura je navíc snadno identifikovatelná a pro optimalizační práce představuje dobře měřitelný element. Následujícím SQL kódem byly identifikovány „nejpomalejší“ SQL procedury v databázi aplikace GMS:

```
SELECT
        DB_NAME(st.dbid) DBName
        ,OBJECT_SCHEMA_NAME(objectid,st.dbid) SchemaName
        ,OBJECT_NAME(objectid,st.dbid) StoredProcedure
        ,MAX(cp.usecounts) execution count
        ,SUM(qs.total_elapsed_time) total elapsed time
        ,SUM(qs.total_elapsed_time) / MAX(cp.usecounts) avg_elapsed_time
FROM sys.dm_exec_query_stats qs
        CROSS APPLY sys.dm_exec_sql_text(qs.plan_handle) st
        INNER JOIN sys.dm_exec_cached_plans cp ON qs.plan_handle = cp.plan_handle
WHERE DB_NAME(st.dbid) IS NOT NULL AND cp.objtype = 'proc'
GROUP BY DB_NAME(st.dbid),OBJECT_SCHEMA_NAME(objectid,st.dbid),
          OBJECT_NAME(objectid,st.dbid)
ORDER BY SUM(qs.total_elapsed_time) DESC
```

Následujícím kódem byly nalezeny nejčastěji používané SQL procedury:

```
SELECT
        DB_NAME(st.dbid) DBName
        ,OBJECT_SCHEMA_NAME(st.objectid,dbid) SchemaName
        ,OBJECT_NAME(st.objectid,dbid) StoredProcedure
        ,MAX(cp.usecounts) Execution count
FROM sys.dm_exec_cached_plans cp
        CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
WHERE DB_NAME(st.dbid) IS NOT NULL AND cp.objtype = 'proc'
GROUP BY cp.plan_handle, DB_NAME(st.dbid), OBJECT_SCHEMA_NAME(objectid,st.dbid),
          OBJECT_NAME(objectid,st.dbid)
ORDER BY MAX(cp.usecounts) DESC
```

Výsledky jsou součástí elektronické dokumentace k této diplomové práci ve formátu XLS.

4.1.2 Záznam SQL provozu

Pomocí programu Microsoft SQL Server Profiler byl zachycen provoz na produkční databázi společnosti Getmore během jednoho pracovního dne. Tento „trace“ pokrývá poměrně dobře aplikaci Workflow a také obecné součásti. Pro zachycení provozu jiných aplikací (CRM, HRM…) bylo použito trasování na testovací aplikaci, nad kterou byly provedeny obvyklé uživatelské postupy (doba používání v řádu několika hodin). Výsledky trasování jsou elektronickou přílohou této práce ve formátu XLS.

Pro zachytávání byla vytvořena speciální šablona, která je také k dispozici v elektronické příloze této práce. Výstup profilování je vhodné ukládat do databázové tabulky pro pozdější jednodušší analýzu naměřených dat. Tato metoda analýzy poskytuje přehlednější data, protože je můžeme filtrovat již při profilování podle různých parametrů (čas vykonávání, IO operace, název databáze…). Nevýhodou je nutnost nechat probíhat profilování po dostatečně dlouhý čas. Profilování částečně snižuje výkon SQL serveru.

4.2 Průběh optimalizace

Z výstupů analýzy byli postupně vytipováni „kandidáti“ pro optimalizační zásahy. Tito „kandidáti“ byli navedeni do aplikace Permon jako testovací případy. Pro testovací případ je vždy nutné zvolit správnou množinu vstupních parametrů a také nastavit správný typ

měření (tj. zda se jedná o SQL proceduru nebo volný SQL kód…). Není nutné navádět případy, ve kterých není předpokládána optimalizace – jejich výkon bude otestován až pomocí automaticky vygenerovaného projektu ze zachyceného provozu (viz kapitola 3.2.4).

Pro každý navedený testovací případ pak byly provedeny následující kroky:

- Rozbor SQL kódu testovacího případu – zjištění případných problémů dle fakt uvedených v teoretické části této práce a rozbor plánu vykonávání daného případu. Totéž pak pro všechny vnořené SQL funkce (nebo procedury).
- Navržení nových variant SQL kódů a jejich navedení do aplikace Permon.
- Proměření všech variant pomocí aplikace Permon a výběr nejoptimálnější varianty.
- Analýza SQL kódu pomocí aplikace Microsoft Database Engine Tuning Advisor (dále jen DETA) pro navržení nových indexů a statistik. Často je nutné kód přeformulovat, aby jej byl schopen program analyzovat.
- Aplikování navrhovaných změn (přidání indexů a statistik) do testovací databáze a opětovně ověření výkonu. Tento krok je nezbytný, protože často bylo zjištěno, že struktury navrhované programem DETA nepřináší téměř žádný zisk, ačkoliv program odhadoval např. 95% zvýšení výkonu.
- Aplikace všech ověřených změn do finální verze databáze.

4.3 Popis vybraných optimalizačních zásahů

Tato kapitola popisuje jen demonstrativně několik příkladů, které byly řešeny v rámci optimalizace. Rozsah práce neumožňuje popis a rozbor všech zajímavých optimalizačních zásahů. Kompletní přehled je možno nalézt v databázi aplikace Permon. Veškeré uváděné příklady vycházejí z databázové struktury aplikace GMS.

4.3.1 Optimalizace funkce dbo.gm_fce_getRight(…)

Velmi zásadní úsporu výkonu přinesla optimalizace SQL funkce gm_fce_getRight. Tato funkce umožňuje ověřit, zda specifikovaný uživatel má uděleno dané oprávnění v rámci systému. Na vstupu funkce jsou tedy parametry „uživatel“ a „oprávnění“. Výsledkem je bitová hodnota. Funkce se používá napříč celou aplikací a na mnoha místech způsobovala „úzká hrdla“. Funkce musí vyhodnotit oprávnění na uživatele a na skupinu uživatelů, do které uživatel náleží. Výsledná hodnota je TRUE, pokud má uživatel právo udělené z po-

hledu uživatele nebo skupiny a zároveň jej nemá zakázané z pohledu uživatele. Původní kód řešil tuto situaci následujícím relativně komplikovaným dotazem:

```
IF EXISTS(SELECT P.id FROM tb prava P INNER JOIN tb uzivatele U ON U.id = @iduziv
        WHERE P.idprava = @idpravo AND
                                (P.iduziv = U.id OR P.idpravaUzivSk = U.idpravaUzivSk))
BEGIN
        IF EXISTS(SELECT tP.id FROM cis_prava tP
                INNER JOIN tb_uzivatele U ON U.id = @iduziv
                LEFT OUTER JOIN tb prava P1 ON P1.idprava = tP.id AND P1.iduziv = U.id
                LEFT OUTER JOIN tb prava P2 ON P2.idprava = tP.id
                                                AND P2.idpravaUzivSk = U.idpravaUzivSk
                WHERE (tP.id = @idpravo) AND (P1.pristup = 1 OR P2.pristup = 1) AND
                                (ISNULL(P1.pristup,1)<>0)   AND (ISNULL(P2.pristup, 1)<>0)
                SET @ret = 1
        ELSE
                SET @ret = 0
END
```

Kód byl přeformulován do dvou jednoduchých dotazů, které nejprve zjistí oprávnění na uživatele, poté oprávnění na skupinu a tyto dvě hodnoty vyhodnotí jednoduchým logickým výrazem pomocí operace CASE:

```
SELECT @pristupByUser = pristup FROM tb_prava
WHERE idUziv = @iduziv AND idprava = @idpravo
IF @IdPravaUzivSk IS NOT NULL
        SELECT
                @pristupByPravaUzivSk = pristup
        FROM tb_prava
        WHERE idPravaUzivSk = @IdPravaUzivSk AND idprava = @idpravo
ELSE
        SELECT @pristupByPravaUzivSk = 0

SET @ret =      CASE
                        WHEN (@pristupByUser = 1 OR @pristupByPravaUzivSk = 1)
                                        AND (ISNULL(@pristupByUser, 1) <> 0) THEN 1
                        ELSE 0
                END
```

Ve spojení s nově přidanými indexy na tabulku [tb_prava] přinesl tento optimalizační zásah úsporu cca 93% času na vykonání funkce.

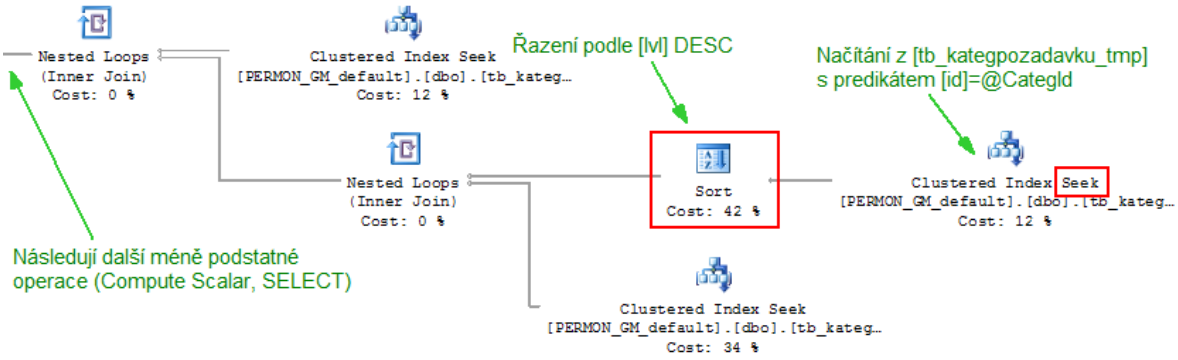
4.3.2 Ukázka nevhodné optimalizace (TP2)

Během optimalizace může docházet k paradoxním situacím. V tomto případě byl optimalizován následující SQL dotaz:

```
DECLARE @out NVARCHAR(4000), @CategId INT
SET @out = NULL
SET @CategId = 10       -- náhodná vstupní hodnota

SELECT
        @out = ISNULL(@out + ' - ', '') + KP.text 1
FROM dbo.tb_kategpozadavku K
        INNER JOIN dbo.tb_kategpozadavku_tmp T ON T.id = K.id
        INNER JOIN dbo.tb kategpozadavku KP ON KP.id = T.ancestor
WHERE K.id = @CategId
ORDER BY T.lvl DESC

SELECT @out
```

Dotaz je sám o sobě velmi zajímavý. Funguje víceméně jako cyklus, který postupně přidává nové řetězce do globální proměnné [@out]. Jinak řečeno – postupně se prochází řádky

výsledného recordsetu a pro každý z nich se do proměnné [@out] přičte „pomlčka“ a hodnota sloupce KP.[text_1] z aktuálního řádku. Výsledkem je nakonec skalární hodnota typu NVARCHAR. Konkrétně se zde zjišťuje řetězec s cestou ve tvaru „root – level 1 – level2 – leaf“ ve stromové tabulce kategorií. Příkaz se vykonával původně podle tohoto plánu:

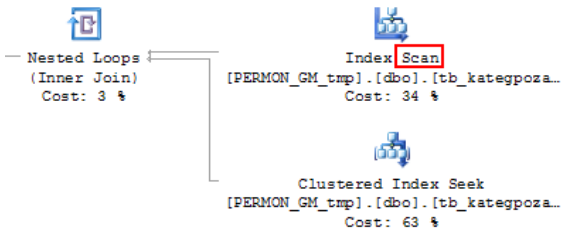


Obrázek 33 – Plán vykonávání pro TP2 – původní

Nejprve byl mírně optimalizován SQL kód. Tabulka s aliasem „K“ je v dotazu zbytečná – její hodnoty se nikde nepoužívají (tedy pouze v predikátu spojení s tabulkou T – tam ale může být použita přímo filtrační konstanta @CategId). Přepis SELECT dotazu je následující:

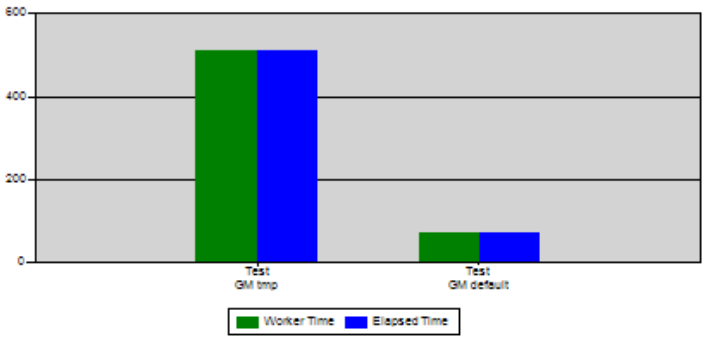
```
SELECT
        @out = ISNULL(@out + ' - ', '') + KP.text 1
FROM dbo.tb_kategpozadavku_tmp T
        INNER JOIN dbo.tb_kategpozadavku KP ON KP.id = T.ancestor
WHERE T.id = @CategId
ORDER BY T.lvl DESC
```

Tato změna v kódu poskytla optimalizaci cca **8%**. Dále byl učiněn pokus o optimalizační zásah pomocí nového indexu. Byl přidán nový index nad tabulku [tb_kategpozadavku_tmp] nad sloupec **[lvl] (desc)** a byly do něj zahrnuty také sloupce **[ancestor]** a **[id]** (formou INCLUDE). Tento zásah byl učiněn za účelem odebrat řazení z plánu vykonávání. Po zavedení nového indexu se jej SQL optimalizér rozhodl využívat místo původního klastrovaného PK (nad sloupcem [id]). Výsledek vedl podle očekávání ke zjednodušení plánu vykonávání:



Obrázek 34 - Plán vykonávání pro TP2 – výsledný

Řazení bylo z plánu odebráno. Nová verze je však výkonnostně podstatně horší (až 6x pomalejší). Graf ukazuje celkový čas vykonávání pro několik tisíc cyklů – vlevo nová „optimalizovaná“ verze, vpravo původní verze:



Obrázek 35 – Porovnání TP2

Optimalizační zásah obsahuje totiž jeden zásadní problém. Použitím jiného indexu již není zajištěno řazení nad sloupcem [id] v tabulce [tb_kategpozadavku_tmp]. Proto je pro její načtení použita operace INDEX SCAN místo INDEX SEEK (vyhledává se s predikátem [id] = @CategId). Index byl zvolen nevhodně. Situaci je možno napravit – sloupec [id] musí být zvolen jako první klíčový sloupec a za něj teprve bude přidán sloupec [lvl] (desc). V tomto případě je zajištěno správné řazení pro predikát v operaci INDEX SEEK a také pro klauzuli ORDER BY – řazení je vynecháno a načítání probíhá pomocí INDEX SEEK. Po této úpravě je již dosaženo celkové optimalizace cca **40%.**

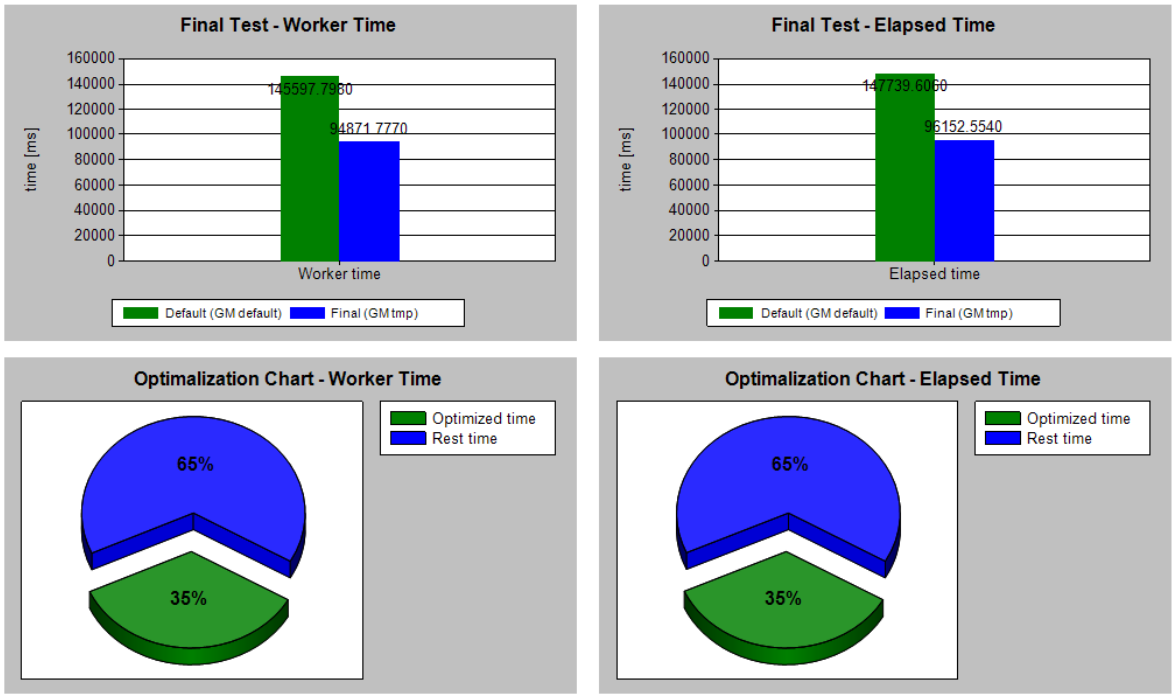
SQL kód tohoto případu je částí funkce, která je často používána při vracení velkého množství dat. Její optimalizace tedy urychlila spoustu jiných SQL procedur, kde tato funkce „hrála roli“ úzkého hrdla.

4.4 Celkové výsledky optimalizace

4.4.1 Test na optimalizovaných součástech

Do tohoto testu byly zahrnuty jen ty SQL fragmenty, které byly optimalizovány. Jedná se o poměrně malý výsek z celkového objemu SQL kódu v GMS. Proto je vidět také největší procento optimalizace. Následující grafy[1] tedy neposkytují informaci o celkové optimalizaci databázové vrstvy, ale spíše o úspěšnosti prováděných dílčích optimalizačních zásahů.

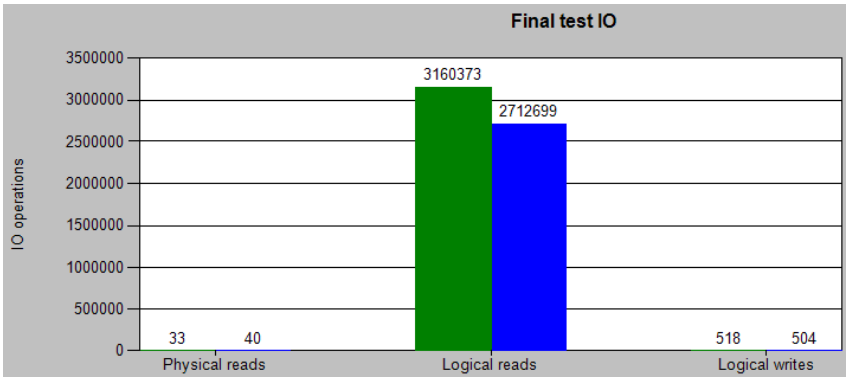
[1] Všechny grafy jsou generovány aplikací Permon



Obrázek 36 – Výsledná průměrná optimalizace (elapsed a worker time)

Sloupcové grafy ukazují celkovou sumu času, který byl potřeba na vykonání všech měřených případů. Zelené sloupce ukazují trvání výchozích variant testovacích případů na výchozím připojení. Modré sloupce pak trvání nejlepších variant na finálním připojení. Koláčové grafy ukazují procento optimalizace. Celek je v tomto případě suma časů na výchozím připojení a zelená výseč je rozdíl časů mezi výchozím a konečným připojením. Zelená výseč tedy ukazuje procento optimalizace.

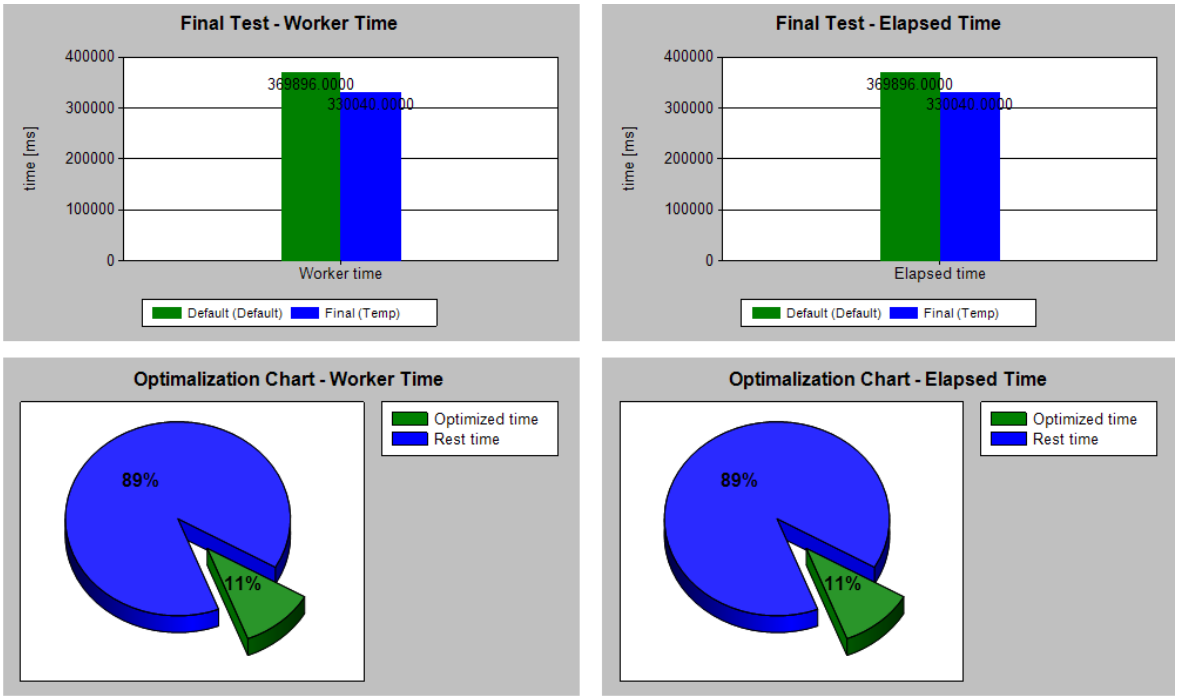
Následující sloupcový graf vyhodnocuje počty IO operací nutných pro provedení SQL kódů. Logická čtení dominují, protože testy probíhají většinou na „nedestruktivních“ dotazech a k zápisům do fyzických tabulek téměř nedochází. Je vidět že skrze optimalizaci došlo ke snížení počtu logických i fyzických čtení.



Obrázek 37 - Výsledná průměrná optimalizace (IO operace)

4.4.2 Celkové testování aplikace Workflow

Následující test už podává ucelenější představu o optimalizaci. Jedná se o automaticky vygenerovaný test aplikace Workflow. Test zachycuje téměř všechny operace, které může jeden uživatel v aplikaci provádět během cca 20 minut (1730 testovacích případů).



Obrázek 38 – Výsledná optimalizace aplikace Workflow (elapsed a worker time)

Grafy mají stejný význam jako v kapitole 4.4.1. Graf IO operací není pro automatický test zachycen, protože veškeré testovací případy se pro jednoduchost měří metodou „dummy measurement“ – tj. zachytává se jen celkový čas vykonávání.

4.4.3 Celkové testování aplikace CRM

Následující bod ukazuje výsledky testu, který byl proveden na aplikaci CRM. Průběh testování je stejný jako v kapitole 4.4.2.

Obrázek 39 - Výsledná optimalizace aplikace CRM (elapsed a worker time)

Podrobné výsledky testů a další informace je možno nalézt v databázi aplikace Permon, která je součástí elektronické přílohy této práce. K prohlížení výsledků slouží aplikace Permon, která je taktéž k dispozici na přiloženém CD v podobě zdrojových kódů a také v kompilované podobě.

ZÁVĚR

V teoretické části práce se podařilo podat ucelený přehled základních principů optimalizace DB vrstvy aplikace. Literatura k tématu „Optimalizace SQL“ v českém jazyce téměř neexistuje. Tento text může tedy v tomto ohledu fungovat jako stručná učebnice nebo základní návod. Je vhodné se seznámit také s citovanou literaturou. Celá teoretická část je na mnoha místech zaměřena prakticky. Často bylo vhodnější uvést názorný příklad než zahrnout čtenáře zdlouhavým teoretickým popisem. Příklady byly kvůli přehlednosti záměrně voleny triviálně a čtenář si je může jednoduše vyzkoušet. Informace v teoretické části jsou platformě určeny především pro MS SQL Server 2005. Mnohé z nich však mají obecnou váhu a je možno je aplikovat nezávisle na databázovém serveru.

V rámci praktické části byla naprogramována utilita Permon. Pro „optimalizéra“ je velmi dobrou pomůckou, protože dokáže podat reálný pohled na úspěšnost optimalizačních zásahů. Podává skutečné a nikoliv pouze odhadované informace (jako je tomu například u DETA). Utilita má další směřování do podoby testovací aplikace, která umožní nejenom změřit výkon (např. po release nové verze aplikace), ale umožní také otestovat, zda SQL dotazy vracejí stejná data pro starou i novou verzi. Optimalizace DB vrstvy GMS byla provedena nad obecnými součástmi a na aplikacích CRM a Workflow. Optimalizaci komplikoval fakt, že stěžejní filtrační mechanizmy jsou řešeny SQL procedurami, které slouží jako „sestavovače“ dynamických SQL dotazů. Procedury pracují na bázi dočasných tabulek a často bylo obtížné vyseparovat SQL kód k optimalizaci. Většina SQL kódů byla psána již optimálně a pokusy o přepis nepřinášely výrazné výsledky. I přesto byla nalezena úzká hrdla (nejčastěji ve formě skalárních nebo tabulkových funkcí) nebo nešikovně formulované dotazy, jejichž přepsání přinášelo někdy až 90% optimalizaci. Největším nedostatkem se ukázala naprostá absence indexové politiky. V podstatě existovaly jen primární klíče. Nové indexy byly navrhovány ručně nebo pomocí nástroje DETA. Mnohé automaticky navrhované indexy musely být zamítnuty, protože nepřinášely očekávané výsledky a pouze navyšovaly velikost databáze. Nepodařilo se naplnit původní plán 30% optimalizace v celkovém měřítku. Nicméně u kódů, které byly detailně analyzovány a optimalizovány, je průměrná optimalizace cca 35%. Optimalizační práce budou dále pokračovat aplikacemi HRM a E-learning. V refaktoringu SQL kódů se počítá s aktivním využíváním utility Permon.

CONCLUSION

In the theoretical part of the work was given a comprehensive overview of basic principles of application DB-layer optimization. There are almost no books about "Optimizing SQL" in the Czech language. This text can therefore in this respect act as a short textbook or basic manual. It is appropriate to study also featured sources. The entire theoretical part is in many places focused also practically. Often was better to give an illustrative example than heap the reader with tedious theoretical description. Examples have been chosen purposely trivially for the sake of clarity, and the reader can simply try it. Information in the theoretical part was designed especially for the MS SQL Server 2005. But many of them are of general validity and can be applied independently of the database server.

In the practical part was programmed the Permon utility. It is a very good tool for an "optimizer", because it can give a real overview of optimization work success. It presents the real and not only estimated information (as in the case of DETA). This utility will be used in the future as a test application, which allows not only to measure performance (eg, after new version release), but also to test whether SQL queries return the same data for the old and the new version. GMS DB layer optimization was conducted over the general components and over CRM and Workflow applications. Optimization was complicated by the fact that all of the central filter mechanisms are designed as SQL procedures which serve as dynamic SQL “compilers”. Procedures are working on the basis of temporary tables, and often it was difficult to parse the SQL code to optimize. Most of the SQL code has been written already in the optimal way, and attempts to rewrite them gave no significant results. Anyhow many bottlenecks (usually in the form of scalar or table functions), or clumsily formulated SQL queries were found, theirs rewriting produces sometimes up to 90% optimization. The biggest deficiency was complete absence of the index policy. In fact, there were only table primary keys. The new indexes have been designed manually or using the DETA tool. Many of the automatically recommended indexes had to be rejected because they didn’t give the expected results, and because they only increased the size of the database. The original plan of 30% optimization in the overall scale was not fulfilled. However, for codes which have been analyzed and optimized in detail, the average optimization is about 35%. Optimization work will continue by HRM and E-learning applications. The Permon utility will be actively used for SQL code refactoring.

SEZNAM POUŽITÉ LITERATURY

[1] **Wikipedia.** Online Transaction Processing. *Wikipedia.* [Online] 10. 3 2009. http://cs.wikipedia.org/wiki/OLTP.

[2] —. Bottleneck (engineering). *Wikipedia.* [Online] 29. 12 2008. http://en.wikipedia.org/wiki/Bottleneck_(engineering).

[3] **Freedman, Craig.** Stream Aggregate. *Craig Freedman's SQL Server Blog.* [Online] 19. 3 2008. http://blogs.msdn.com/craigfr/archive/2006/09/13/752728.aspx.

[4] —. Hash Aggregate. *Craig Freedman's SQL Server Blog.* [Online] 18. 2 2008. http://blogs.msdn.com/craigfr/archive/2006/09/20/hash-aggregate.aspx.

[5] **Pinal, Dave.** Journey to SQL Authority with Pinal Dave. *sqlauthority.com.* [Online] 30. 3 2007. http://blog.sqlauthority.com/2007/03/30/sql-server-index-seek-vs-index-scan-table-scan/.

[6] **Chigrik, Alexander.** Using SQL Server Cursors. *Database Journal.* [Online] WebMediaBrands Inc., 20. 5 2002. http://www.databasejournal.com/features/mssql/article.php/1439731/Using-SQL-Server-Cursors.htm.

[7] **Freedman, Craig.** Bookmark Lookup. *Craig Freedman's SQL Server Blog.* [Online] 30. 6 206. http://blogs.msdn.com/craigfr/archive/2006/06/30/652639.aspx.

[8] **SSWUG.org.** Transact-SQL Optimization Tips. *www.mssqlcity.com.* [Online] Bits on the Wire, 2005. http://www.mssqlcity.com/Tips/tipTSQL.htm.

[9] **Chigrik, Alexander.** Alternative way to get the table's row count. *MS SQL City.* [Online] http://www.mssqlcity.com/Articles/KnowHow/RowCount.htm.

[10] **Hedgate, Christoffer.** Measuring SQL Performance. *sqlservercentral.com.* [Online] Simple Talk Publishing, 23. 12 2005. http://www.sqlservercentral.com/articles/Performance+Tuning/measuringperformance/1323/.

[11] **Prajdić, Mladen.** Mladen Prajdić blog. *SQLTeam.com.* [Online] 25. 2 2008. http://weblogs.sqlteam.com/mladenp/archive/2008/02/25/How-SQL-Server-short-circuits-WHERE-condition-evaluation.aspx.

[12] **McGehee, Brad.** SQL Server Optimizer Hints. *sql-server-performance.com.* [Online] 13. 9 2006. http://www.sql-server-performance.com/tips/hints_general_p1.aspx.

[13] **Freedman, Craig.** Nested Loops Join. *Craig Freedman's SQL Server Blog.* [Online] 26. 7 2006. http://blogs.msdn.com/craigfr/archive/2006/07/26/679319.aspx.

[14]. —. Merge Join. *Craig Freedman's SQL Server Blog.* [Online] 3. 8 2006. http://blogs.msdn.com/craigfr/archive/2006/08/03/merge-join.aspx.

[15] —. Summary of Join Properties. *Craig Freedman's SQL Server Blog.* [Online] 16. 8 2006. http://blogs.msdn.com/craigfr/archive/2006/08/16/702828.aspx.

[16]. **Community.** Understanding Pages and Extents. *msdn.microsoft.com.* [Online] Microsoft, 2009. http://msdn.microsoft.com/en-us/library/ms190969.aspx.

[17] **Surkov, Konstantin.** SQL Server Optimization. *SQL Server Developer Center.* [Online] 6 2006. http://msdn.microsoft.com/en-us/library/aa964133(SQL.90).aspx.

[18] **Scott, Allen.** SQL Server Indexes. *odetocode.com.* [Online] 31. 12 2003. http://www.odetocode.com/Articles/70.aspx.

[19] **Freedman, Craig.** Seek Predicates. *Craig Freedman's SQL Server Blog.* [Online] 5. 8 2008. http://blogs.msdn.com/craigfr/archive/2006/07/07/652668.aspx.

[20] **Sack, Joseph.** *Velká kniha T-SQL & SQL Server 2005.* [překl.] Jan Pokorný. místo neznámé : Zoner Press,, 2007. str. 864. 978-80-806815-57-2.

[21] **Community.** Using Statistics to Improve Query Performance. *SQL Server Developer Center.* [Online] Microsoft, 26. 2 2009. http://msdn.microsoft.com/en-us/library/ms190397.aspx.

[22] —. Query Performance Tuning. *SQL Server Developer Center.* [Online] Microsoft. http://msdn.microsoft.com/en-us/library/ms172984.aspx.

[23] —. DBCC SHOW_STATISTICS. *SQL Server Developer Center.* [Online] Microsoft, 26. 2 2009. http://msdn.microsoft.com/en-us/library/ms174384.aspx.

[24] **Hanson, Eric N. a Kollar, Lubor.** Statistics Used by the Query Optimizer in Microsoft SQL Server 2005. *Microsoft Technet.* [Online] Microsoft. http://technet.microsoft.com/cs-cz/library/cc966419(en-us).aspx.

[25] **Gruber, Martin.** *Mistrovství v SQL.* [překl.] RNDr. Jan Pokorný Dušan Juhas. Praha : SoftPress, 2004. str. 480. Sv. 1. 80-86497-62-3.

[26] **Brust, Andrew J. a Forte, Stephen.** *Mistrovství v programování SQL Serveru 2005.* [překl.] Jiří Fadrný Petr Matějů. Brno : Computer Press, 2007. str. 847. 978-80-251-1607-4.

[27] **Whalen, Edward, a další.** *Microsoft SQL Server 2005 - Velký průvodce administrátora.* místo neznámé : Computer Press, 2008. str. 1080. 9788025119495.

[28] **Stanek, William R.** *SQL Server 2005 Kapesní rádce administrátora.* [překl.] Luděk Horčička. Brno : Computer Press, a.s., 2006. str. 542. ISBN 80-251-1211-X.

[29] **Nagel, Christian, a další.** *C# 2005 Programujeme profesionálně.* [překl.] Petr Dokoupil Jakub Mikulaštík. 1. vyd. Brno : Computer Press, a.s., 2006. str. 1398. ISBN 80-251-1181-4.

[30] **Chigrik, Alexander.** SQL Server Optimization Tips for Designing Tables. *Database journal.* [Online] WebMediaBrands Inc., 29. 2 2003. http://www.databasejournal.com/features/mssql/article.php/1576231/SQL-Server-Optimization-Tips-for-Designing-Tables.htm.

[31] **Novotný, Karel.** Optimalizace odezev pro globální přehledy. *SestemOnLine.* [Online] http://www.systemonline.cz/business-intelligence/optimalizace-odezev-pro-globalni-prehledy.htm.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

3NF Třetí normální forma - soubor doporučení (metodika) pro návrh datové struktury databáze, jehož dodržení vede k optimálnímu využití vlastností systému OLTP při tvorbě databázových aplikací.

ASP Active Server Pages - technologie společnosti Microsoft pro vytváření webových aplikací.

AVG Average - průměr.

CPU Central Processing Unit - centrální výpočetní jednotka počítače. Procesor počítače.

DETA Zkratka pro program Database Engine Tuning Advisor od společnosti Microsoft, který je součástí instalace MS SQL Server 2005.

GMS Zkratka pro aplikaci GetmoreSystem od společnosti Getmore s.r.o.

HJ Hash Join - princip spojení dvou množin při vykonávání SQL dotazu.

IO Input - Output - vstupně výstupní.

KB KiloByte - kilobajt (1024 bajtů).

LJ Loop Join - princip spojení dvou množin při vykonávání SQL dotazu.

LOB Large Objects - v SQL například hodnoty datových typů ntext, nvarchar(max) apod.

MB MegaByte - megabajt (1024 kilobajtů).

MJ Merge Join - princip spojení dvou množin při vykonávání SQL dotazu.

OLAP Online Analytical Processing - označení databázového modelu určeného pro získávání analytických výsledků z velkoobjemových databází.

OLTP Online Transaction Processing - označení databázového modelu určeného do mnohouživatelského prostředí, kde převládají transakční operace pro modifikaci a vkládání dat.

PK Primary key - primární klíč tabulky v relační databázi.

RAID Redundant Array of Independent Disks - vícenásobné diskové pole nezávislých disků. Typ diskových řadičů, které zabezpečují pomocí speciálních funkcí ko-

ordinovanou práci dvou nebo více fyzických diskových jednotek.

RID Row identifier - identifikátor řádku.

SQL Structured Query Language - strukturovaný dotazovací jazyk používaný pro práci s daty v relačních databázích.

VB Visual Basic - programovací jazyk od společnosti Microsoft.

XML eXtensible Markup Language - rozšiřitelný značkovací jazyk.

SEZNAM OBRÁZKŮ

SEZNAM TABULEK