



BAKALÁŘSKÁ PRÁCE

**Optimalizace SQL dotazů v
databázi Oracle**

**SQL query optimization in
Oracle database**

Bohdan Blaha

Unicorn College © 2010

Unicorn College, V Kapslovně 2767/2, Praha 3, 130 00

Název práce v ČJ:	SQL Optimalizace v Oracle
Název práce v AJ:	SQL Optimization in Oracle



Autor:	Bohdan Blaha
--------	--------------

Akademický rok:	2009/2010
-----------------	-----------

Kontakt:	E-mail: blaha@bohdan.org Tel.: (+420) 737 887 715
----------	--

1. ZADÁNÍ

 2009.10.12 - 20:47 - Blaha Bohdan - Zadání BP - Optimalizace databázových systémů (ZBP/20091012_001)

Artefakt  DIT Zadání závěrečné bakalářské práce
Zadavatel  Student (Blaha Bohdan)

ZADÁNÍ ZÁVĚREČNÉ BAKALÁŘSKÉ PRÁCE (ZBP)

Název ZBP v češtině	Optimalizace databázových systémů
Název ZBP v angličtině	Optimization of database systems
Studijní obor	 Information Technologies
Akademický rok	2009/2010
Vedoucí závěrečné práce	Miroslav Žďárský
Termín odevzdání Zadání ZBP	16.10.2009
Termín odevzdání práce ZBP	07.05.2010

Cíl závěrečné bakalářské práce

Cílem bakalářské práce je popsat postupy jak optimalizovat SQL dotazy pro databázový systém Oracle

Základní literatura

Mistrovství v Oracle Database 11g - (Bob Bryla, Kevin Loney)
SQL Server 2008 Query Performance Tuning Distilled - (Grant Fritchey, Sajal Dam)
and Oracle materials (f.i. asktom.oracle.com)

2. ABSTRAKT

Tato bakalářská práce se věnuje problematice optimalizace SQL dotazů v databázi Oracle, která povede ke zvýšení jejího výkonu. Nutno podotknout, že optimalizace SQL dotazů je pouze jedna část komplexního problému, jakým ladění výkonu databází bezesporu je.

Mnoho firem v dnešní době neklade na optimalizaci příliš velký důraz, nejen proto, že ji nepovažují za důležitou, ale často nemají potřebné informace o tom, co, jak a kdy optimalizovat. Ve své praxi jsem se setkal s řadou odstrašujících příkladů, které mnohdy plynuly z neznalosti principů optimalizace. Z tohoto důvodu v první části práce popisuji důležité objekty a pojmy spojené s laděním výkonu, jako jsou různé typy indexů, statistiky, materializované pohledy, metody přístupů do tabulek a typy spojení tabulek. Tuto část provázejí názorné příklady z praxe, které je možno si vyzkoušet, neboť uvedené SQL dotazy a DB struktury jsou součástí proloženého CD.

Druhá část práce je zaměřena na detekci SQL dotazů, které mohou způsobovat snížení výkonu databáze. Proces hledání slabých míst, je jak časově tak finančně náročný, neboť neexistuje žádné, zaručeně funkční řešení. Každý systém či aplikace se liší jak architekturou, tak vlastní logikou. Odhalení problému, bez pomoci vhodných nástrojů se mnohdy může zdát nemožné a v tomto případě je to spíše o štěstí než logickém postupu. Proto se v druhé části věnuji velmi užitečným nástrojům, které databáze Oracle k usnadnění detekce slabých míst přináší. Těmito nástroji je Automatic Database Diagnostic Monitor, Trace nástrojům a samozřejmě také exekučnímu plánu.

Závěr práce pak nabízí několik užitečných rad, kterým je vhodné věnovat pozornost při psaní SQL dotazů.

Klíčová slova: Optimalizace, analytické nástroje, exekuční plán, indexy, statistiky, Oracle, ladění SQL dotazů, pohledy, výkon

3. ABSTRACT

This thesis describes an SQL query-optimization process in Oracle database, which has positive impact on its performance. I have to mention that the SQL query-optimization process is only one part of a complex issue, which database performance tuning undoubtedly is.

Many companies do not spend much time on performance tuning today, not only because they do not consider this as an important issue, but they often do not have proper information about what, how and when to optimize. I have seen many deterrent examples, which often stemmed from ignorance of the optimization principles. That is the reason, why I describe important objects and ideas related to the performance tuning, such as various types of indexes, statistics, materialized views, access methods types and joins in the first part of this thesis. This part is full of illustrative examples that could be practically tried, as all SQL statements and DB structures are stored on the attached CD.

The second part focuses on a detection of SQL queries, which could cause poor performance. The process of finding bottlenecks is both time consuming and financially expensive, since there is no certainly workable solution. Each system or application varies both architecture and its own logic. Bottleneck detection without the help of appropriate tools may be often almost impossible. In this case it's more about luck than logical process. Hence, the second part concentrates on very useful tools that Oracle database brings to facilitate the detection of bottlenecks. These tools consist of Automatic Database, Diagnostic Monitor, trace tools and Execution plan too.

You can find some useful tips at the end, which you should pay attention to, due to effective writing SQL queries.

Keywords: Optimization, analytics tools, indexes, statistics, execution plan, Oracle, SQL query tuning, materialized view, performance

4. PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma Optimalizace SQL dotazů v databázi Oracle jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou v práci citovány a jsou též uvedeny v seznamu literatury a použitých zdrojů.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb.

V Praze dne

.....

Jméno Příjmení

5. PODĚKOVÁNÍ (DOBROVOLNÉ)

Děkuji vedoucímu bakalářské práce Miroslavu Žďárskému za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé bakalářské práce.

6. OBSAH

1. Zadání.....	3
2. Abstrakt.....	4
3. Abstract.....	5
4. Prohlášení.....	6
5. Poděkování (dobrovolné).....	7
6. Obsah.....	8
7. Úvod.....	9
8. Optimalizace SQL dotazů.....	11
9. Základní pojmy.....	13
9.1 Přístupy do databáze.....	13
9.2 Indexy.....	14
9.2.1 B-Tree index.....	16
9.2.2 Bitmap index.....	17
9.2.3 R-Tree index.....	18
9.2.4 Index organized table.....	20
9.3 Statistiky.....	20
9.4 Spojování tabulek.....	22
9.4.1 Nested loop joins.....	22
9.4.2 Hash joins.....	23
9.4.3 Sort merge joins.....	23
9.5 Materializované pohledy.....	24
10. Detekce problematických SQL.....	27
10.1 Nástroje na ladění.....	28
10.1.1 ADDM.....	28
10.2 Trace nástroje.....	29
10.2.1 Oracle TRACE.....	29
10.2.2 Oracle AUTOTRACE.....	29
10.2.3 End to End Application tracing.....	31
10.3 Exekuční plán.....	31
11. BULK OPERACE.....	35
12. Závěr.....	36
13. Conclusion.....	39
14. Seznam použité literatury.....	42
15. Seznam použitých symbolů a zkratek.....	43
16. Seznam obrázků.....	44
17. Seznam tabulek.....	45
18. Seznam příloh.....	46

7. ÚVOD

Cílem této práce je upozornit na úskalí spojená s optimalizací SQL dotazů v databázi Oracle. První část práce popisuje důležité objekty a pojmy spojené s laděním výkonu. Druhá část práce se zaměřuje na nalezení problematických dotazů, odhalení možností při hledání slabých míst v dotazech a představení možností, které databáze Oracle nabízí.

Existuje mnoho možností zvýšení výkonu aplikací, které pracují s databází Oracle. Výkon se dá ovlivnit nejenom na hardwarové úrovni (například instalací rychlejších disků, větší paměti, atd.), ale i pomocí vlastní změny konfigurace databáze Oracle (například změnou inicializačních parametrů), správnou implementací dotazů, indexů, použitím materializovaných pohledů, pomocí pravidelné údržby, apod. V této práci se zaměřím pouze na optimalizaci SQL dotazů, přestože hardware a konfigurace Oracle je neméně důležitá.

Správná optimalizace SQL dotazů má významný pozitivní dopad na celkový běh aplikací a výrazně ovlivňuje jejich úspěšné nasazení do produkčních prostředí. Softwarové firmy mnohdy optimalizaci dotazů podceňují a nekladou na ni velký důraz, ovšem při následném odstraňování případných pomalých dotazů přicházejí o nemalé finanční prostředky. Problémy spojené s výkonem se nemusí projevit okamžitě po nasazení. Ovšem v momentě, kdy databáze obsahuje větší množství dat, se mohou objevit zcela nečekaně ze dne na den. Projevy těchto problémů mohou být různé. Jedná se například o pomalou odezvu z databáze, způsobující dlouhé čekání na výsledek běhu aplikace a tím znepríjemňující uživatelům práci s aplikací. Dalším, mnohem negativnějším projevem je ovlivňování běhu jiných, mnohdy důležitějších programů nebo zapříčinění nestandardního chování aplikace, případně i zhroucení celého programu nebo systému. Vyhledání a následná oprava těchto problémů je náročná na kapacitu, úsilí a finance.

Abychom těmto problémům předešli, případně je eliminovali na minimum, je důležité připravit si maximálně reálná testovací data. Při přípravě testovacích dat musíme zohlednit nejenom kvalitu, ale i množství dat, se kterými bude aplikace v produkčním prostředí pracovat. Pro generování dat je možno využít možnosti PL/SQL, kterou databáze Oracle nabízí.

Aplikace se dají ladit téměř do nekonečna, nicméně je důležité si předem odpovědět na následující otázky:

1. Kdy je aplikace dostatečně vyladěna a optimalizována?
2. Přinese investice do výkonu požadovaný výsledek?

Vše je pouze otázkou priorit a financí. Čím větší bude potřeba optimalizace dotazů, tím více bude vynaloženo finančních prostředků a zde platí pravidlo, že rozdíl nárůstu výkonu

a vynaložených financí se v jistém okamžiku začne prohlubovat. Je proto velmi důležité odhadnout, kdy je aplikace dostatečně optimalizována v poměru výkon/cena. Platí zde pravidlo 80:20, kdy investováním 20% zdrojů je výkon zvýšen o 80%, ale pro získání zbylých 20% výkonu je potřeba vynaložit 80% zdrojů (DAM, Sajal. *SQL Server Query Performance Tuning Distilled*. Second edition. New York : Apress, 3004. 597 s. ISBN 1-59059-421-5.).

Je nepříjemným zjištěním, že obecný návod na zajištění vysokého výkonu databáze prostě neexistuje, jelikož na trhu existují různé aplikace a systémy s různými parametry a proto se jejich potřeby ladění výkonu liší. Výkony aplikací se mohou změnit i po upgrade na jinou verzi databáze Oracle. Z tohoto důvodu je velmi důležité takovouto zásadní změnu vždy důkladně otestovat.

Ve snaze předejít nečekaným problémům, je nutné výkon databáze pravidelně monitorovat. Monitorování by mělo probíhat i v systémech, kde neprobíhají žádné konfigurační změny, nicméně toto monitorování může včas odhalit problémy s výkonem, které jsou způsobeny například přírůstkem dat, a následná změna konfigurace bude naopak nutná. Bohužel je systém mnohdy monitorován a řádně testován, až po nasazení do produkčního prostředí.

Dojde-li k poklesu výkonu na neúnosnou hranici, je nezbytně nutné identifikovat příčinu tohoto poklesu. Ve většině případů dochází ke změně výkonu v důsledku nadměrného používání systémových prostředků. Tato situace je mnohdy řešena jednou z následujících možností:

- Změnou hardware
- Změnou konfigurace systému/databáze
- Změnou v aplikaci

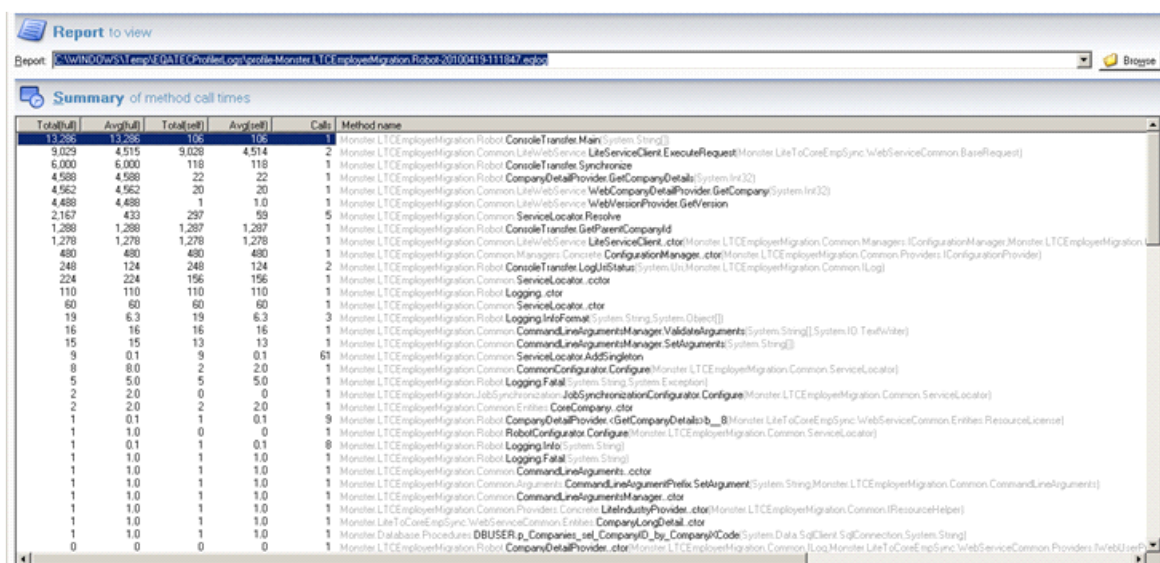
Nejčastěji jsou problémy způsobeny aplikací.

Velmi cenné informace ohledně databáze lze získat z Enterprise Manager (EM), dodávaného přímo s databází, kde v záložce *Performance* nalezneme informace o Top Relacích, největších konzumentech prostředků, a podobně.

8. OPTIMALIZACE SQL DOTAZŮ

V současnosti se od vyvíjených systémů očekává především velký výkon, dále příjemné uživatelské rozhraní, bezpečnost a stabilita. Při vývoji systému není kladen důraz pouze na jejich výkon, ale i na příjemné uživatelské rozhraní, bezpečnost a stabilitu. V praxi se můžeme setkat s aplikacemi typu klient server, kdy uživatel využívá možnosti webového prohlížeče a logika je uložena na aplikačním serveru. Tvůrci webových stránek se často snaží stránky vylepšit různými grafickými prvky, aktive-X prvky apod., tím mohou být pro uživatele atraktivní, ale jakákoliv akce může trvat dlouho a uživatele odradit od používání. V důsledku toho vsadilo mnoho společností na jednoduchost a rychlost. Jako příklad můžeme uvést společnost Google, ta vytvořila jednoduchou stránku, která má za úkol, co nejrychleji vrátit požadovaná data (výsledek hledání).

Budeme-li optimalizovat nějakou aplikaci, musíme nejprve najít nejslabší místo systému a k tomuto účelu slouží profily. Následující obrázek ukazuje výstup z profileru EQATEC (detailní informace na stránkách výrobce: <http://www.eqatec.com/tools/profiler>). (Originál obrázku je umístěn na příloženém CD v adresáři /images/profiler.png)



Total[full]	Avg[full]	Total[set]	Avg[set]	Calls	Method name
13,296	13,296	106	106	1	Monster.LTCEmployeeMigration.Robot.Console.Transfer.Main(System.String)
9,029	4,515	9,029	4,514	2	Monster.LTCEmployeeMigration.Common.LifeWebService.LifeServiceClient.ExecuteRequest(Monster.LifeToCoreEmpSync.WebService.Common.BaseRequest)
6,000	6,000	118	118	1	Monster.LTCEmployeeMigration.Robot.Console.Transfer.Synchronize
4,599	4,599	22	22	1	Monster.LTCEmployeeMigration.Robot.Console.Transfer.GetCompanyDetails(System.Int32)
4,562	4,562	20	20	1	Monster.LTCEmployeeMigration.Common.LifeWebService.WebCompanyDetailProvider.GetCompany(System.Int32)
4,489	4,489	1	1,0	1	Monster.LTCEmployeeMigration.Common.LifeWebService.WebVersionProvider.GetVersion
2,167	433	257	59	5	Monster.LTCEmployeeMigration.Common.ServiceLocator.Resolve
1,288	1,288	1,287	1,287	1	Monster.LTCEmployeeMigration.Robot.Console.Transfer.GetParentCompanyId
1,279	1,279	1,279	1,279	1	Monster.LTCEmployeeMigration.Common.LifeWebService.LifeServiceClient.ctor(Monster.LTCEmployeeMigration.Common.Managers.IConfigurationManager;Monster.LTCEmployeeMigration.Common.Managers.IConfigurationManager;Monster.LTCEmployeeMigration.Common.Providers.IConfigurationProvider)
480	480	480	480	1	Monster.LTCEmployeeMigration.Common.Managers.ConfigurationManager.ctor(Monster.LTCEmployeeMigration.Common.Providers.IConfigurationProvider)
248	124	248	124	2	Monster.LTCEmployeeMigration.Robot.Console.Transfer.LogInStatus(System.UInt32;Monster.LTCEmployeeMigration.Common.ILog)
224	224	156	156	1	Monster.LTCEmployeeMigration.Common.ServiceLocator.ctor
110	110	110	110	1	Monster.LTCEmployeeMigration.Robot.Logging.ctor
60	60	60	60	1	Monster.LTCEmployeeMigration.Common.ServiceLocator.ctor
19	6,3	19	6,3	3	Monster.LTCEmployeeMigration.Robot.Logging.InfoFormat(System.String;System.Object[])
16	16	16	16	1	Monster.LTCEmployeeMigration.Common.CommandLineArgumentsManager.ValidateArguments(System.String[];System.ID TextWriter)
15	15	13	13	1	Monster.LTCEmployeeMigration.Common.CommandLineArgumentsManager.SetArguments(System.String[])
9	0,1	9	0,1	61	Monster.LTCEmployeeMigration.Common.ServiceLocator.AddSingleton
8	8,0	2	2,0	1	Monster.LTCEmployeeMigration.Common.CommandConfigurator.Configure(Monster.LTCEmployeeMigration.Common.ServiceLocator)
5	5,0	5	5,0	1	Monster.LTCEmployeeMigration.Robot.Logging.Fatal(System.String;System.Exception)
2	2,0	0	0	1	Monster.LTCEmployeeMigration.JobSynchronizationConfigurator.Configure(Monster.LTCEmployeeMigration.Common.ServiceLocator)
2	2,0	2	2,0	1	Monster.LTCEmployeeMigration.Common.Entities.CoreCompany.ctor
1	0,1	1	0,1	9	Monster.LTCEmployeeMigration.Robot.CompanyDetailProvider.GetCompanyDetailsBy_B(Monster.LifeToCoreEmpSync.WebService.Common.Entities.ResourceLicense)
1	1,0	0	0	1	Monster.LTCEmployeeMigration.Robot.RobotConfigurator.Configure(Monster.LTCEmployeeMigration.Common.ServiceLocator)
1	0,1	1	0,1	8	Monster.LTCEmployeeMigration.Robot.Logging.Info(System.String)
1	1,0	1	1,0	1	Monster.LTCEmployeeMigration.Robot.Logging.Fatal(System.String)
1	1,0	1	1,0	1	Monster.LTCEmployeeMigration.Common.CommandLineArguments.ctor
1	1,0	1	1,0	1	Monster.LTCEmployeeMigration.Common.CommandLineArguments.PrefixSetArgument(System.String;Monster.LTCEmployeeMigration.Common.CommandLineArguments)
1	1,0	1	1,0	1	Monster.LTCEmployeeMigration.Common.CommandLineArgumentsManager.ctor
1	1,0	1	1,0	1	Monster.LTCEmployeeMigration.Common.Providers.ConcreteLifeWebService(Monster.LTCEmployeeMigration.Common.ResourceHelper)
1	1,0	1	1,0	1	Monster.LifeToCoreEmpSync.WebService.Common.Entities.CompanyLongDetail.ctor
1	1,0	1	1,0	1	Monster.Database.Procedures.DBUSER_p_Companies_sel_CompanyID_by_CompanyCode(System.Data.SqlClient.SqlConnection;System.String)
0	0	0	0	1	Monster.LTCEmployeeMigration.Robot.CompanyDetailProvider.ctor(Monster.LTCEmployeeMigration.Common.ILog;Monster.LifeToCoreEmpSync.WebService.Common.Providers.WebServiceClient)

Obrázek 1: Výstup profileru

Jestliže má aplikace plnit svůj účel, musí v ní proběhnout mnoho činností, počínaje inicializací objektů a tříd, přes volání různých metod, či externích služeb. Výstup profileru (obrázek 1) zobrazuje seznam metod .NET aplikace, které jsou volány. U každé metody jsou zobrazeny informace o celkovém čase (v milisekundách), který daná metoda potřebovala a průměrném čase, jež je odlišný od celkového v případě, že je metoda volána vícekrát. Jak již bylo v úvodu zmíněno, je nutno se zaměřit na optimalizaci kroků, které přinesou viditelné výsledky.

Pokud aplikace pracuje s databází, tím nejslabším místem většinou bývá přístup do databáze a práce s daty (což je i náš případ). Zde je vidět, že aplikace volá metody *GetCompany* a *GetCompanyDetails*. První metoda vrací základní informace o společnosti a druhá potom vrací její úplný detail. Obě metody potřebují přes 4500 milisekund (4,5 sekundy) k tomu, aby vykonaly své úkoly. V tomto případě musíme nejprve zjistit, proč je získávání dat z databáze pomalé a následně se pokusit čas potřebný na vykonání metody snížit. Nemá význam se zaměřit na optimalizaci metod, které zabírají pouze jednu milisekundu (tedy aspoň v našem případě), pokud není takových metod velké množství. Problémem bude patrně načtení dat z databáze.

Pro úspěšnou optimalizaci SQL je nutné nejprve identifikovat nejproblematictější oblasti. V níže uvedených odstavcích se zaměřím na následující oblasti:

1. Indexy
2. Statistiky
3. Skladbu SQL dotazů
4. Design datového modelu

Než ale podrobněji popíši indexy používané v databázi, zmíním se o možných přístupech do tabulek, které jsou pro pochopení optimalizace důležité.

9. ZÁKLADNÍ POJMY

9.1 Přístupy do databáze

Full table Scan (FTS)

Metoda FTS prochází v databázi všechny řádky tabulky a vrací výsledky, které splňují podmínky výběru. Databáze používá FTS v případech:

1. Chybějící/nepoužitelný index
2. Potřeby načtení velkého množství dat z tabulek
3. Malé tabulky

Databáze může použít FTS dokonce i pro nalezení pouze jednoho řádku tabulky. Pokud FTS načítá velké množství dat, může být rychlejší než Index Range Scan (IRS). FTS prochází najednou více bloků diskových dat (což znamená méně diskových operací) a proto je rychlejší než metoda IRS, která provádí mnoho diskových operací.

Index Range Scan (IRS)

Metoda IRS začíná procházet tabulky vzestupně od kořene indexu a pokračuje přes jeho uzly a listy, vráceným výsledkem je jeden či více ROWID. IRS je jedna z nejpoužívanějších metod pro přístup do tabulek v OLTP databázi Oracle.

Fast Full Index Scan (FFIS)

Metoda FFIS získává data pouze čtením z indexu, aniž by databáze musela přistoupit k datům tabulky pomocí ROWID. K úspěšnému proběhnutí FFIS je potřeba splnit několik podmínek, například existence všech sloupců v indexu (jak sloupce požadované v klauzuli SELECT tak i WHERE).

Index Unique Scan (IUS)

Metoda IUS většinou navrácí pouze jedinou hodnotu. Používá se tehdy, pokud chceme načíst data na základě unikátní hodnoty (například ID uživatele, rodné číslo nebo jakoukoliv unikátní hodnotu).

9.2 Indexy

Indexy jsou klíčovou součástí všech databází. Jakmile databáze potřebuje načíst data z tabulky, má dvě možnosti. Buď projde podle použitého vyhledávacího klíče všechny řádky tabulky (FTS) ¹, nebo přistoupí pouze k několika málo hodnotám díky mapování hodnot na ROWID, které pak odkazují na konkrétní fyzické umístění. Vhodně zvolené indexy výrazně snižují dobu potřebnou k načtení dat z databáze. Důležitým aspektem pro výběr vhodného indexu je selektivita (tj. množství unikátních hodnot v tabulce). V případě velké selektivity bude množství navrácených dat malé. Mnohdy může být navrácen pouze jeden řádek, hledáme-li podle unikátního identifikátoru, například id nebo telefonního čísla.

Indexy je možno si představit jako rejstřík v zadní části knihy. Rejstřík obsahuje informaci, kde se námi hledaná data (v našem případě to může být například text, číslo nebo jiný datový typ) nachází a řekne nám číslo stránky (v případě indexu ROWID). Bez tohoto rejstříku bychom museli procházet celou knihu stránku po stránce a to by jistě zabralo mnohem více času.

Indexy jsou fyzicky i logicky odděleny od dat v asociovaných tabulkách. Díky tomu je možné indexy smazat bez toho, aby byly ovlivněny data, ke kterým se vztahují, jiné indexy nebo aplikace, která přistupuje k datům. Databáze automaticky udržuje indexy během operací `INSERT`, `UPDATE` a `DELETE` v asociovaných tabulkách. Pokud smažeme index, všechny aplikace budou fungční, nicméně přístup k dříve indexovaným datům se může zpomalit.

Nevhodně zvolené nebo špatně nastavené indexy mohou způsobit nemalé problémy. Jakmile je databáze pomocí SQL dotazu `SELECT` požádána o data, Oracle přistoupí k tabulce a v případě, že nenalezne vhodný index, provede FTS. Databáze obsahuje několik druhů indexů. Nejčastěji používanými indexy v databázi Oracle jsou B-Tree index a Bitmap index. Mezi další používané indexy patří CTX indexy, Quadtree index a R-Tree index.

Přestože máme indexy správně definovány, může nastat mnoho případů, kdy vzhledem k nesprávně napsanému SQL dotazu bude index ignorován a proběhne FTS. Například použití funkcí nad indexovaným sloupcem zapříčiní FTS. Následující konkrétní příklad vyhledává v tabulce *tbllobcan* ve sloupci *prijmeni*. Tabulka obsahuje následující tři sloupce:

```
RodneCislo Numeric(10, 0) Not Null
```

```
Jmeno NVarChar(50) Null
```

```
Prijmeni NVarChar(80) Not Null
```

1 většina tabulek v databázi není seřazená, výjimkou mohou být například index-organized tabulky

Níže uvedené řádky vytvoří index nad sloupcem *prijmeni* v tabulce *tblobcan*, následně sesbírají statistiku a spustí dotaz nad tabulkou *tblobcan*. Jako výsledek je navraceno 6021 řádků obsahující příjmení 'NEUBAUER'. Pro získání výsledku byla využita metoda IRS (s použitím vytvořeného indexu *name_idx*).

```
CREATE INDEX name_idx ON tblobcan (prijmeni);

EXEC DBMS_STATS.gather_table_stats('bohdan', 'tblobcan');

SELECT jmeno, prijmeni, rodnecislo FROM tblobcan WHERE

prijmeni='NEUBAUER';
```

Nalezeno celkem: 6021

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		5930	121K	5099 (1)	00:01:02
1	TABLE ACCESS BY INDEX ROWID	TBLOBCAN	5930	121K	5099 (1)	00:01:02
* 2	INDEX RANGE SCAN	NAME_IDX	5930		17 (0)	00:00:01

Obrázek 2: Názorný příklad – Function based index

Co se ovšem stane, pokud všechna příjmení v tabulce nebudou zadána velkými písmeny, ale budou 'lowercase', případně 'mixcase'? V případě spuštění následujícího dotazu, bude vyhledáván řádek s příjmením 'Neubauer'. Ovšem databáze vrátí 0 řádků, jelikož se v databázi žádný 'Neubauer' (první písmeno velké, zbytek malá) nevyskytuje.

```
SELECT jmeno, prijmeni, rodnecislo FROM tblobcan WHERE

Prijmeni='Neubauer';
```

Nalezeno celkem: 0

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		5930	121K	5099 (1)	00:01:02
1	TABLE ACCESS BY INDEX ROWID	TBLOBCAN	5930	121K	5099 (1)	00:01:02
* 2	INDEX RANGE SCAN	NAME_IDX	5930		17 (0)	00:00:01

Obrázek 3: Názorný příklad – Function based index

Jak tedy hledat příjmení napsaná různými způsoby (rozdílná velikost písma)? Mohli bychom využít funkci `Lower`, které převede všechny velké písmena na malá a vyhledá řádek s příjmením 'Neubauer'. Aplikujeme následující dotaz:

```
SELECT jmeno, prijmeni, rodnecislo FROM tblobcan WHERE
Lower(prijmeni)='Neubauer';
```

Nalezeno celkem: 6021

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		59245	1214K	7501 (2)	00:01:31
* 1	TABLE ACCESS FULL	TBLOBCAN	59245	1214K	7501 (2)	00:01:31

Obrázek 4: Názorný příklad – Function based index

Databáze našla správně všechny Neubauery, ale na výše uvedeném obrázku můžete vidět, že **byl použit FTS na místo IRS!** Provedeme tedy následující změnu. Smažeme index a vytvoříme nový, založený na funkci `Lower`. Všechny hodnoty indexu budou převedeny na malá písmena. Spustíme znovu dotaz:

```
DROP INDEX name_idx;
CREATE INDEX name_idx ON tblobcan (lower(prijmeni));
EXEC DBMS_STATS.gather_table_stats('bohdan', 'tblobcan');
SELECT jmeno, prijmeni, rodnecislo FROM tblobcan WHERE
Lower(prijmeni)='neubauer';
```

Nalezeno celkem: 6021

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		5930	162K	5448 (1)	00:01:06
1	TABLE ACCESS BY INDEX ROWID	TBLOBCAN	5930	162K	5448 (1)	00:01:06
* 2	INDEX RANGE SCAN	NAME_IDX	5930		18 (0)	00:00:01

Obrázek 5: Nazorný příklad – Function based index

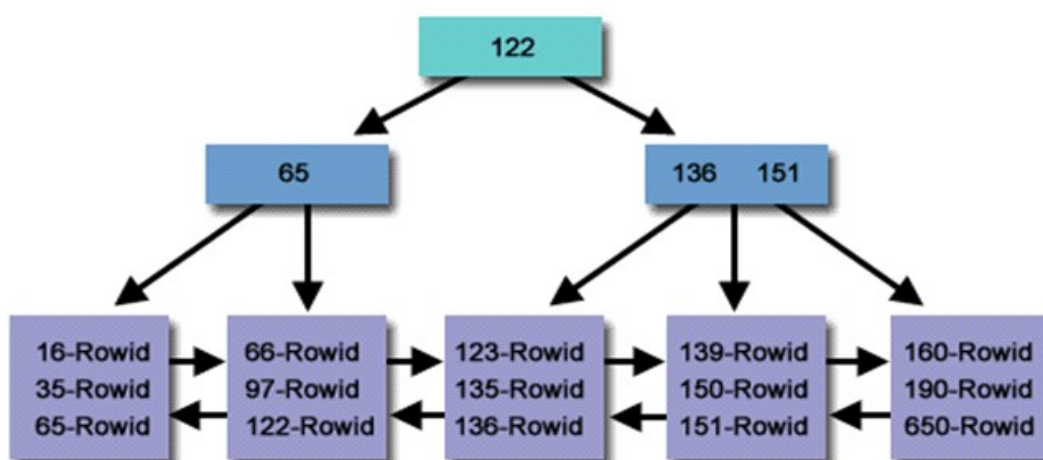
Jak je z Explain Plan patrné, byl znovu použit IRS a byli nalezeny všechny řádky, nezáleží na velikosti písmen.

9.2.1 B-Tree index

Indexy jsou často veliké, tudíž se nevejdou do cache paměti a z tohoto důvodu musí být uloženy na disku, přestože diskové operace jsou mnohem pomalejší, než operace v paměti. Proto musí být indexy uloženy tak, aby bylo jejich využití co nejefektivnější. *B-Tree* index se skládá ze tří částí: kořen (Root), uzel (Node) a list (Leaf). Index B-Tree vytvoří uzel (Node) o velikosti fyzického

bloku disku, tak aby splnil podmínku efektivnosti čtení z disku, čímž přechod o úroveň níže zabere pouze jednu I/O operaci.

B-Tree (vyvážený) index má podobu vyváženého stromu a proto jsou operace typu `SELECT`, `INSERT` a `DELETE` prováděny v čase $\log(n)$. Databáze jej používá jako základní typ indexu. Tyto indexy mohou být jednoduché (obsahující pouze jednu hodnotu) nebo kompozitní (obsahující dvě až 32 hodnot).

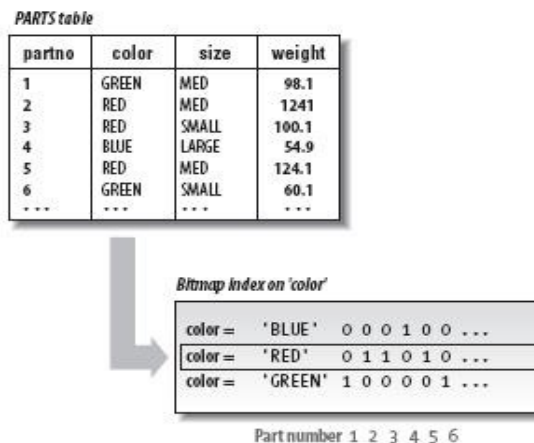


Obrázek 6: B-Tree index

9.2.2 Bitmap index

Tento typ indexu je hojně používán v tabulkách s nízkou selektivitou. Existují dva typy : „pravý“ a „dynamický“. Oracle umožňuje použít více typů indexů k získání dat z jedné tabulky. Použije li se kombinace bitmapového indexu a *B-Tree* indexu, je *B-Tree* index převeden na dynamický bitmapový. „Pravé“ bitmapové indexy mohou vrátit výsledek pouze přistoupením k indexu, kdežto „dynamické“ indexy vyžadují přístup k tabulce přes RowID.

Klasickým příkladem použití bitmapového indexu, je potřeba indexovat sloupec *Pohlaví*, který často může nabývat jen dvou hodnot: muž případně žena. V tomto případě bude velikost *Bimap* indexu mnohem menší než *B-Tree* indexu, což se projeví i na rychlosti. Svou opravdovou sílu ovšem Oracle skrývá ve vyhodnocování podmínek AND, OR a NOT, ale také v možnosti využití *Multi-Bitmap* indexů. Kdy Oracle dokáže spojit výsledky více *Bitmap* indexů do jednoho, což vede k výraznému posílení výkonu



Obrázek 7: Bimap index

9.2.3 R-Tree index

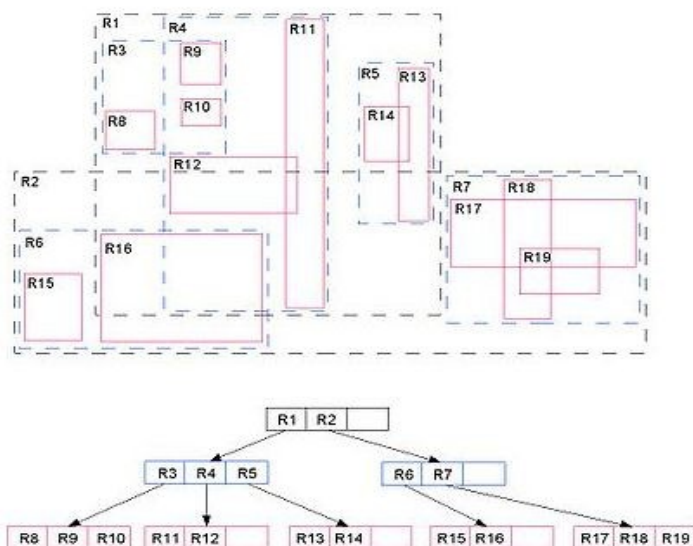
R-tree index je stromová datová struktura podobná B-stromům, ale používaná pro prostorové přístupové metody, například pro indexování vícerozměrných struktur v geografických informačních systémech. Datová struktura rozděluje místo na hierarchicky vkládané a potenciálně se překrývající, tzv. MBR (minimum bounding rectangles - minimální vázané obdélníky). (*R-tree In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, [cit. 2010-05-02]. Dostupné z WWW: <<http://en.wikipedia.org/wiki/R-tree>>.*)

Každý uzel R-stromu má proměnlivý počet záznamů. Každý záznam uvnitř uzlu, který není listem, ukládá další dvě informace:

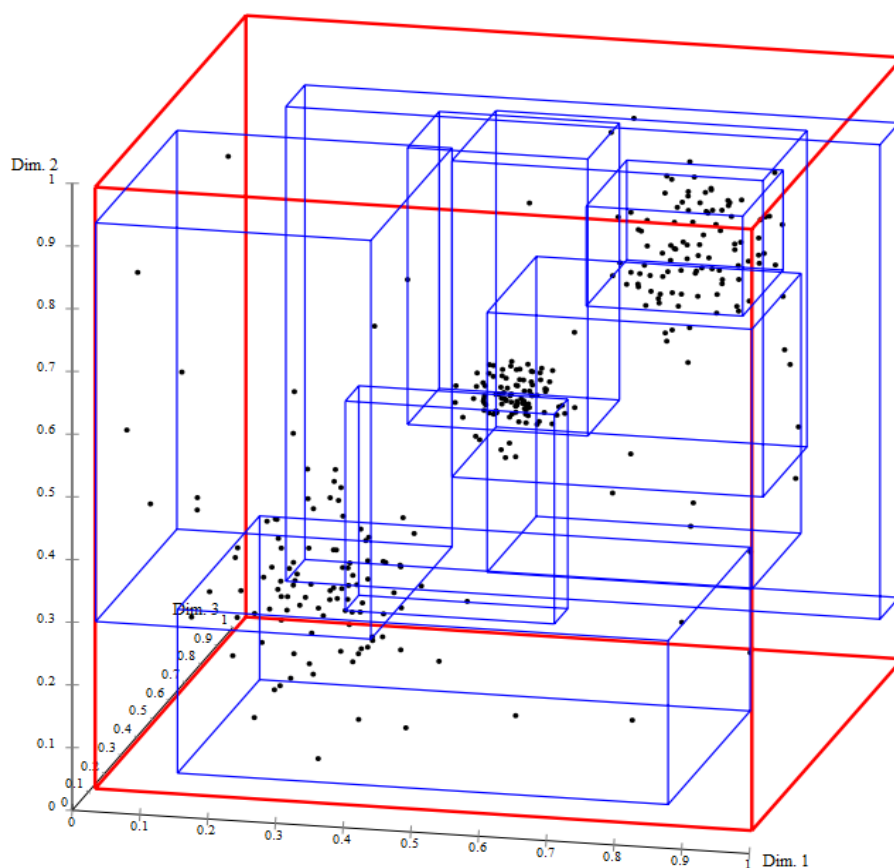
1. způsob identifikace dceřiného uzlu
2. MBR všech záznamů uvnitř tohoto dceřiného uzlu.

Algoritmy pro vložení nového a smazání stávajícího prvku používají MBR z uzlů ke kontrole, že prvky v geometrickém okolí jsou umístěny do stejných listových uzlů (konkrétně, nový prvek půjde do listového uzlu, který potřebuje nejmenší rozšíření svého MBR). Každý záznam uvnitř listového uzlu uloží dvě informace: identifikátor daného prvku (může být alternativně umístěn přímo do uzlu) a MBR datového prvku.

Podobně vyhledávací algoritmy používají MBR k rozhodnutí, zda-li hledat uvnitř daného uzlu. Tímto způsobem při hledání většina uzlů „netknutých“. Stejně jako B-stromy, jsou R-stromy vhodné pro databáze, kde se uzly podle potřeby mohou načítat do paměti.



Obrázek 8: R-Tree index



Obrázek 9: R-Tree index 3D vizualizace

9.2.4 Index organized table

Indexově orientovaná tabulka je v podstatě index, ve kterém je uložen celý řádek a nikoliv pouze klíčové hodnoty řádků. Místo identifikátoru ROWID je za primární klíč řádků považován logický identifikátor řádku. (*Mistrovství v oracle Database 11g. Brno : CPress, 2009. 700 s.*)

Přestože jsou indexy jistě důležitým aspektem ovlivňující výkon databáze, nejsou bohužel všemohoucí. Bez vhodného designu tabulek a stupně normalizace budou indexy takřka bezcenné. V takových případech mohou být jen na škodu a při vkládání nových dat zbytečně zatěžují databázi. V následujících dvou tabulkách je skutečný příklad z mé praxe. Jednalo se o aplikaci, která měla vyhledávat uživatele podle sloupce *RegionID*. Aplikace pracovala s tabulkami, které měly podobnou strukturu, jaká je znázorněna v Tab1. Tabulka obsahovala pouze dva sloupce: *UserID (Number)* a *RegionID (Varchar2)*. V tomto případě aplikace prohledávala sloupec *RegionID* pomocí LIKE, čímž pokaždé proběhl FTS nad několika milióny řádků. Nejenže výsledek byl vrácen v řádech sekund, ale obsahoval i data, které vrátit neměl. V případě hledání regionu „1“, vrátil i uživatele s regionem „11“. Pokud by byla použita struktura navržená v Tab 2, výsledek by byl vrácen okamžitě a zároveň by obsahoval pouze očekávaná data.

Tabulka 1: Schéma tabulky

UserID	RegionID
1254	1,12,16
6984	11,6

UserID	RegionID
1254	1
1254	12
1254	16
6984	11
6984	6

Zdroj: *Monster*

Bohužel výskyt tabulek, jako je znázorněno v tabulce 1, není ojedinělý.

9.3 Statistiky

Důležitým prvkem databáze jsou statistiky. Databáze generuje mnoho typů statistik. Mohou to být statistiky týkající se systému, relací, případně individuálních SQL dotazů, apod.

Statistiky slouží optimalizátoru pro výběr vhodného exekučního plánu. Výhodou statistik je jejich uložení v datových slovnících, takže je možné je exportovat a následně importovat do jiné

databáze. Díky tomu je možné dostat produkční statistiky do testovacího prostředí pro snadnější identifikaci možného problému. Databáze má dva optimalizátory: Rule Based Optimizer (RBO) a Cost Based Optimizer (CBO). Nejdůležitějším úkolem CBO je navrhnout exekuční plán pro SQL dotaz. CBO spustí SQL dotaz a snaží se k němu najít různé exekuční plány, u kterých porovnává jejich celkové náklady. Následně z těchto exekučních plánů vybere ten s nejmenšími celkovými náklady. A právě k tomuto úkolu potřebuje statistiky.

V případě absence statistik se použije dynamic sampling .

CBO (preferován od verze 8i.) je použit v případě, že statistiky naopak existují. CBO využije statistiky k určení selektivity a odhadne náklady exekučních plánů, z nichž vybere takový, který nejméně zatěžuje systémové prostředky (využití I/O a CPU).

Je několik možností získání statistik:

- ANALYZE
- DBMS_UTILITY
- DBMS_STATS

Oracle preferuje ke sběru statistik použití DBMS_STATS namísto ANALYZE. Ty pak obsahují údaje o rozložení dat, jejich charakteru, sloupcích, indexech apod. Následující tabulka zobrazuje část statistik.

Tabulka 2: Statistiky

Typ	
Tabulky	Počet řádků
	Počet bloků
	Průměrná délka řádku
	Sloupce
	Počet různých hodnot ve sloupci
	Počet NULL ve sloupci
	Distribuce dat
Indexy	Počet listů v bloku
	Úrovně
Systém	IO výkon a využití
	Výkon procesoru a využití

Oracle doporučuje pravidelný sběr statistik, hlavně v systémech, které se velmi často mění, jako například v systémech s přibývajícím velkým množstvím dat. Indexové statistiky se mohou generovat automaticky (REBUILD indexu, apod.).

Ke správnému použití indexů je potřeba mít také správně nastaveny statistiky, aktualizaci statistik lze zajistit spuštěním následujícího příkazu:

```
EXEC DBMS_STATS.GATHER_SCHEMA_STATS('NAZEV_SCHEMATU');
```

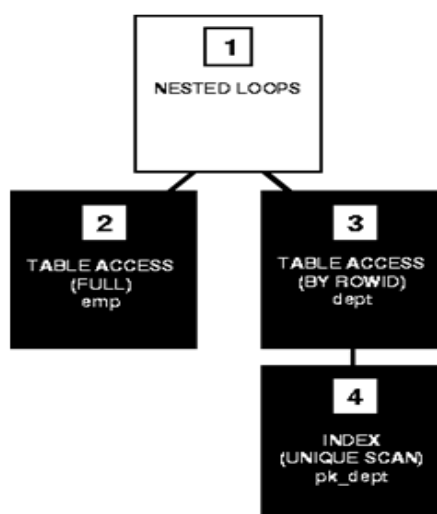
I správně nastavené statistiky nemusí v některých případech pomoci. Jako příklad lze uvést problém CBO s odhadem pořadí tabulek při použití klausule `WHERE`. Nevhodně zvolené pořadí tabulek může v mezivýsledku dotazu vrátit velké množství dat. Určení optimálního pořadí tabulek znamená „hádání“ takového pořadí, které v mezivýsledku vrátí co nejméně řádků.

9.4 Spojování tabulek

Při načítání dat z databáze je mnohdy potřeba načíst data ze dvou a více tabulek najednou. Nutnost spojení tabulek záleží na aplikaci a stupni normalizace tabulek databáze. Za tímto účelem se používá `JOIN` spojení. `JOIN` je syntaktická konstrukce jazyka SQL a slouží ke spojování výsledku dotazu `SELECT` ze dvou vstupních množin (typicky tabulek relační databáze). Některé dotazy poběží rychleji s Nested Loop spojením, některé zase s HASH spojením. Není jednoduché předem určit účinnější spojení, proto je vhodné zkusit, které spojení bude rychlejší. K přesnému změření času lze použít příkaz `SET TIMING ON` vypisující, kolik dotaz potřeboval času na dokončení.

9.4.1 Nested loop joins

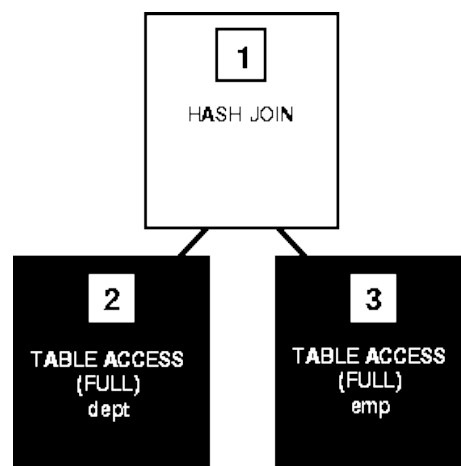
Nested loop join je vhodný pro malé množství dat (typicky menší než 10000 řádků), kde je spojení založeno na indexovaných sloupcích. Databáze určí jednu tabulku jako řídící (vnější) a druhou jako vnitřní. Vnější tabulka se prochází v cyklu a pro každý řádek se hledají řádky ve vnitřní tabulce, které odpovídají podmínce spojení.



Obrázek 10: Nested loop join

9.4.2 Hash joins

Tento typ sloučení je vhodný pro velké množství dat. Databáze určí menší tabulku jako řídící (vnější), druhou jako vnitřní a z klíče spojení vnější tabulky vytvoří HASH. Tato tabulka se pak načte do paměti. Poté se prochází větší tabulka a k řádkům se hledají odpovídající řádky. Tato metoda je vhodná, pokud se menší tabulka dokáže celá vlézt do paměti, v případě, že je tabulka na paměť moc velká, databáze ji rozdělí na několik oddílů (partitions).



Obrázek 11: Hash join

9.4.3 Sort merge joins

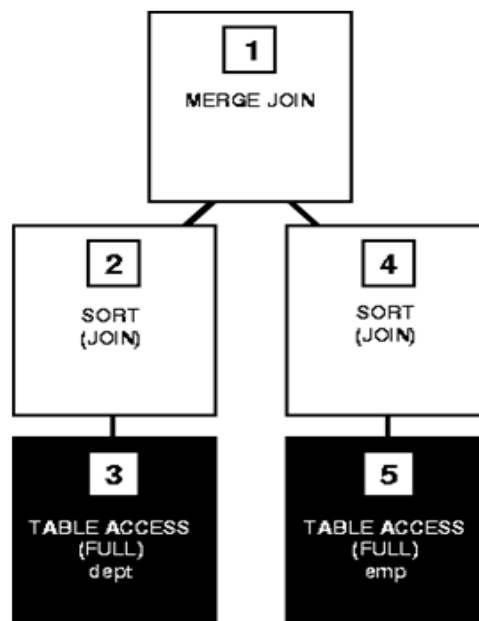
Databáze nejprve setřídí všechny řádky první tabulky podle sloupce uvedeného ve spojení, pak také setřídí odpovídající řádky druhé tabulky, podle sloupce uvedeného ve spojení. Jakmile je třídění hotovo, databáze sloučí setříděné řádky. Tento typ joinu je ovlivněn konfiguračními parametry:

1. `sort_area_size`
2. `db_file_multiblock_read_count`.

Hash spojení je obecně rychlejší než Sort Merge spojení, nicméně tomu tak nemusí být v případě, kdy:

- Řádky tabulek jsou již setříděny
- Operace třídění není potřeba

To z důvodu, že operace SORT velkého počtu dat, nelze provádět v paměti, ale je za potřebí i pomalých diskových operací.



Obrázek 12: Sort Merge join

9.5 Materializované pohledy

Materializované pohledy si můžeme představit jako virtuální tabulky v databázi, které reprezentují výsledky nějakého SQL dotazu. Po obdržení SQL dotazu databáze zjistí, zda existuje MV. Při jeho existenci a zapnutí `QUERY_REWRITE` databáze transparentně přepíše SQL dotaz, tak aby použila MV. Toto přepsání zvýší výkon, jelikož SQL dotaz přistupuje k menšímu množství dat (případně datům v jedné jediné tabulce), místo přístupu ke zdroji MV (případně čtení z více tabulek). Hlavní výhodou ovšem je pre-join tabulek. Kromě toho, databáze dokáže automaticky synchronizovat MV s daty ve zdrojových tabulkách.

Níže uvádím možnosti jejich využití:

- Data Warehousing
- Replikace

- Rozdělení tabulek na menší celky
- Bezpečnost
- Snížení síťové zátěže

V případě ladění výkonu nás bude především zajímat možnost snížení zátěže a rozdělení tabulek na menší celky. MV mohou být použity k distribuci globálních dat do regionálních databází. Místo toho, aby všichni používali jednu globální databázi, jsou požadavky rozděleny na lokální databáze. K zamezení velkých přenosů dat je možné nastavit jen malou část, na které bude probíhat refresh. Například máme-li tabulku, která obsahuje data z celého světa, můžeme vytvořit MV, který bude obsahovat jen data regionu, který nás zajímá. Můžeme také vytvořit MV pouze pro čtení:

```
CREATE MATERIALIZED VIEW obcane AS

SELECT jmeno, prijmeni FROM TBLOBCAN;
```

Vytvořený MV obsahuje pouze jméno a příjmení z tabulky *tblobcan*. Potřebujeme-li do něj zapisovat, stačí přidat `FOR UPDATE` během jeho vytvoření, což umožní změnu dat:

```
CREATE MATERIALIZED VIEW obcane FOR UPDATE AS

SELECT jmeno, prijmeni FROM TBLOBCAN;
```

K promítnutí změněných dat v MV zpět do zdrojových tabulek je nutné, aby MV byl součástí skupiny, kterou je možné vytvořit v aplikaci Replication management nebo pomocí PL/SQL:

```
DBMS_REPCAT.CREATE_MVIEW_REPGROUP (

gname          IN   VARCHAR2,

master         IN   VARCHAR2,

comment        IN   VARCHAR2      := ' ',

propagation_mode IN VARCHAR2      := 'ASYNCHRONOUS',

fname          IN   VARCHAR2      := NULL

gowner         IN   VARCHAR2      := 'PUBLIC');
```

Pokud MV není součástí skupiny, změna dat neovlivní zdrojové tabulky a navíc bude při

aktualizaci MV smazána.

Databáze nabízí několik typů MV:

- Primary Key Materialized Views
- Object Materialized Views
- ROWID Materialized Views
- Complex Materialized Views

Primary Key Materialized Views je základní MV databáze, který může obsahovat poddotaz, nebo komplexní dotaz obsahující `WHERE EXISTS` nebo jen jednoduchou klausuli `WHERE`. Např.:

```
CREATE MATERIALIZED VIEW obcane FOR UPDATE AS
```

```
SELECT jmeno, prijmeni FROM TBLOBCAN
```

```
WHERE prijmeni='BLAHA';
```

Tento MV je možné postupně aktualizovat (přidávání nových řádků ze zdrojových tabulek), případně je možné aktualizovat každý záznam, který se změnil od poslední aktualizace.

Data ve zdrojových tabulkách se mohou měnit, a proto databáze podporuje několik druhů aktualizací:

- Complete – MV je kompletně aktualizován
- Fast – Aktualizují se jen změny
- Force – Zkusí použít Fast metodu, pokud to není možné, použije Complete
- Never – MV se nikdy nezaktualizuje

Existuje několik způsobů aktualizací. MV může být aktualizován `ON DEMAND` (na požádání), případně `ON COMMIT`, kdy databáze automaticky aktualizuje data, pokud byla ukončena (`COMMIT`) transakce běžící na zdrojových tabulkách. V databázi máme možnost nadefinovat, jak často bude aktualizace probíhat. Následující příkaz vytvoří MV, který se bude

aktualizovat každých sedm dní od data vytvoření, metodou Fast:

```
CREATE MATERIALIZED VIEW obcane
```

```
REFRESH FAST NEXT sysdate + 7
```

```
AS SELECT jmeno, prijmeni FROM TBLOBCAN
```

Aktualizace MV v databázi významně zvyšuje jejich rychlost a použitelnost. Není proto nutné přistupovat k nepotřebným datům, což umožňuje mnohem rychlejší a efektivnější práci s daty.

10. DETEKCE PROBLEMATICKÝCH SQL

Klíčovým úkolem během optimalizace je nalezení SQL dotazů, které způsobují snížení výkonu databáze. Optimalizace těchto dotazů je nezbytná z důvodu jejich nadměrného využívání systémových prostředků. Pro nalezení těchto dotazů existuje několik způsobů:

Můžeme začít hledat nejnáročnější dotazy pomocí následujícího dotazu, kde 'N' je počet řádků(dotazů), které chceme zobrazit.

```
SELECT * FROM (SELECT hash_value,address,  
  
substr(sql_text,1,40) sql,  
  
buffer_gets "Gets",  
  
executions "Exec",  
  
buffer_gets/executions "Gets/Exec"  
  
FROM V$SQLAREA WHERE executions > 10 ORDER BY kritérium DESC)  
  
WHERE rownum <= N;
```

Nebo použijeme užitečné nástroje, které Oracle nabízí:

- Automatic Database Diagnostic Monitor (ADDM)
- Automatic Workload Repository
- V\$SQL view
- Custom Workload
- SQL Trace

V některých případech se dá příčina poklesu výkonu jednoduše vysledovat bez použití speciálních nástrojů databáze. Pravidelné projevy poklesu výkonu mohou mít různé příčiny. Snížení výkonu mohou zapříčinit naplánované aktivity, reporty, sběry statistik, ale také nějaká specifická aplikace v čase svého spuštění. Ovšem mnohdy je výkon ovlivněn nepravidelně a je třeba použít nástroje, které databáze nabízí a bez nichž je nalezení problematických SQL dotazů velmi časově náročné a tím i finančně nákladné.

V drtivém případě do databáze přistupují velké počty uživatelů, kteří chtějí načíst, případně měnit data. Tyto operace potřebují nějaký čas na jejich dokončení. V databázi Oracle probíhají všechny dotazy v transakcích. Transakce začne spuštěním dotazu a při standardním ukončení skončí příkazem COMMIT, v nestandardním se provede ROLLBACK. V případě transakcí požadovaný čas narůstá, neboť transakce jsou skupinou operací, které splňují podmínky ACID.

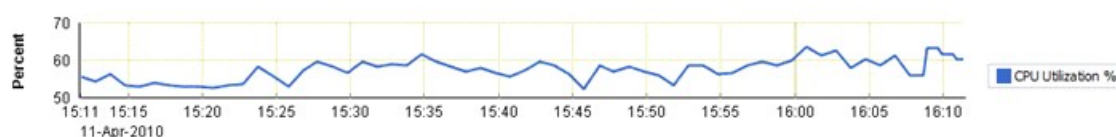
Protože v databázi probíhá mnoho dotazů najednou, je potřeba použít nástroje, abychom dokázali dotazy způsobující deadlocky nebo snížení výkonu snadněji identifikovat.

10.1 Nástroje na ladění

Databáze nabízí několik nástrojů, které pomáhají identifikovat dotazy ovlivňující její výkon. Tyto nástroje jsou vytvořeny tak, aby fungovaly shodně jak nad OLTP tak OLAP databázemi.

- ADDM
- SQL Tuning Advisor
- SQL Tuning Sets
- SQL Access Advisor

Oracle Enterprise Manager nabízí řadu možností. V sekci „Performance“ se nachází mnoho užitečných informací zpracovaných do přehledných grafů.



Obrázek 12, Enterprise manager – Využití CPU

10.1.1 ADDM

Automatic Database Diagnostic Monitor identifikuje možná slabá místa pomocí analýzy dat v Automatic Workload Repository (AWR). Cílem ADDM analýzy je poskytnout přehledný a agregovaný přehled dění v databázi po stránce výkonu i případných problémů. ADDM neslouží pouze k identifikaci slabých míst, ale může také navrhnout možná řešení na odstranění problému. ADDM monitoruje data uložená v (AWR) a generuje doporučení pro vyladění databáze. ADDM analýza zahrnuje například následující oblasti:

- Problémy s konfigurací databáze
- PL/SQL a SQL dotazy náročné na systémové prostředky
- CPU analýzu
- Analýzu použití paměti
- I/O analýzu

10.2 Trace nástroje

Databáze nabízí několik nástrojů vytvořených pro monitorování a analýzu aplikací, které se připojují k databázi Oracle. Mezi tyto nástroje patří například Oracle Trace.

10.2.1 Oracle TRACE

Databáze využívá Oracle *Trace* ke sběru informací o výkonu, využití dat apod. Oracle Trace se skládá ze tří částí

- Oracle Trace Manager
- Oracle Trace Collection Services
- Oracle Trace Application Programming Interface (API).

OTM je klient/server aplikace, běžící v konzoli OEM. OTM poskytuje grafické prostředí ke sběru informací z předdefinovaných událostí. Tyto události jsou důležité při analýze a ladění databáze. Existují dva typy událostí (events) – „point event“ a „duration events“. Point (bod) events představuje aktuální stav, příkladem může být výskyt chyby. „Duration events“ má začátek a konec. Vhodným příkladem této události je například transakce. Kompletní a podrobný výpis událostí naleznete zde:

http://download.oracle.com/docs/cd/A57673_01/DOC/sysman/doc/A55904_01/evt.htm#424970

10.2.2 Oracle AUTOTRACE

Během přípravy/testování dotazu je možné zapnout/vypnout automatické zobrazování exekučního plánu a statistik pomocí příkazu `SET AUTOTRACE ON/OFF`.

V automaticky zobrazeném exekučním plánu je vidět, zda byl či nebyl použit index, či jaká byla použita přístupová metoda a jiné detaily.

Tabulka 3: Oracle AUTOTRACE – Statistiky

physical read	Celkový počet bloků dat přečtených z disku.
Bytes sent via SQL*Net to klient	Celkový počet bajtů odeslaných klientovi
bytes received via SQL*Net from klient	Celkový počet bajtů přijatých klientem pomocí Oracle Net.
SQL*Net roundtrips to/from klient	Celkový počet Oracle NET zpráv odeslaných a přijatých klientem
rows processed	Počet zpracovaných řádků během operace
Sorts	Počet operací třídění

Zdroj: ORACLE

```
SQL> select jmeno, prijmeni, rodnecislo from tblobcan where jmeno='Radoslav';
```

```
-----
Plan hash value: 2427482928
```

```
-----
| Id | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-----
|  0 | SELECT STATEMENT   |           | 37028 | 759K | 7468  (1)| 00:01:30 |
|* 1 | TABLE ACCESS FULL| TBLOBCAN  | 37028 | 759K | 7468  (1)| 00:01:30 |
-----
```

```
Predicate Information (identified by operation id):
-----
```

```
1 - filter("JMENO"='Radoslav')
```

```
Statistiky
-----
```

```
0 recursive calls
0 db block gets
29525 consistent gets
27053 physical reads
0 redo size
1156189 bytes sent via SQL*Net to client
27712 bytes received via SQL*Net from client
2474 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
37095 rows processed
```

Obrázek 13: AutoTrace

10.2.3 End to End Application tracing

End to End Application Tracing zjednodušuje proces diagnostiky výkonnostních problémů ve vícevrstevném prostředí, kde je žádost z koncového klienta přesměrována do dalších databázových relací. Sledování takového klienta je pak velmi obtížné. End to End Application Tracing používá unikátní identifikátor k vysledování koncového klienta přes všechny databázové relace. Trace informace jsou zapsány do souboru a mohou být analyzovány pomocí utility TKPROF. Trace soubor se dá číst přímo, bez zformátování pomocí TKPROF, nevýhodou totiž je, že TKPROF některé informace nepřenáší. Například pořadí spuštěných SQL dotazů ve výstupu TKPROF již není viditelné. TKPROF spuštěná dohromady s `EXPLAIN PLAN` poskytuje cenné informace, důležité pro ladění výkonu.

10.3 Exekuční plán

Jakmile databáze obdrží SQL dotaz, sestaví *exekuční plán*. Tento plán pomůže databázi rozhodnout, jak hledat nebo zapisovat data. Při práci s daty musí databáze učinit některá rozhodnutí, nejdůležitějším z nich je, zda použije indexy či nikoliv. V případě více indexů musí navíc rozhodnout, které indexy budou použity. Dalším důležitým rozhodnutím, je bezesporu typ spojení, které bude použito (Nested join, Hash join, apod.) Zobrazení exekučního plánu napoví, kterou část dotazu je potřeba upravit, protože zabírá velké množství systémových prostředků, případně nevyužívá potencionálních možností. `EXPLAIN PLAN` (EP) názorně ukazuje, jak databáze spouští aktuální SQL dotaz.

Například následující dotaz:

```
SELECT allUrednikFunkce.Nazev, tblUrad.NazevUradu
FROM (allUrednikFunkce
INNER JOIN tblFunkceSMistem
ON Funkce_ID = ID) INNER JOIN tblUrad
ON Ma_mis_KodUradu = tblUrad.KodUradu
```

bude mít následující exekuční plán:



Obrázek 14, Exekuční plán

Exekuční plány je možné zobrazit také pomocí pohledu `V$SQL_PLAN`, který je zobrazí pro každý dotaz uložený v paměti (cache). Další možností je uložení výsledku *exekučního plánu* do nadefinované tabulky. V adresáři `$ORACLE_HOME/rdbms/admin` se nachází SQL skript `utlxplan.sql`, který obsahuje definici na vytvoření tabulky `PLAN_TABLE` (název tabulky je možno změnit) pro uložení výsledku exekučního plánu pomocí `EXPLAIN PLAN INTO PLAN_TABLE FOR SQL dotaz`.

V následujícím příkladě se spustí exekuční plán a výsledek bude uložen do nadefinované tabulky.

```
EXPLAIN PLAN INTO PLAN_TABLE FOR

SELECT allUrednikFunkce.Nazev, tblUrad.NazevUradu

FROM (allUrednikFunkce

INNER JOIN tblFunkceSMistem

ON Funkce_ID = ID) INNER JOIN tblUrad

ON Ma_mis_KodUradu = tblUrad.KodUradu
```

Do tabulky „PLAN_TABLE“ budou vloženy informace o spuštěném exekučním plánu. V následující tabulce uvádím několik důležitých sloupců, které si zaslouží pozornost. Kompletní a detailnější popis lze nalézt na:

http://download.oracle.com/docs/cd/B10500_01/server.920/a96533/ex_plan.htm#22727

Tabulka 4: Důležité sloupce tabulky PLAN_TABLE

Sloupec	Typ	Popis
OPERATION	Varchar2(30)	Obsahuje jméno operace provedené v daném kroku exekučního plánu.
COST	NUMERIC	Předpokládaná „cena“ operace. Tato hodnota je pouze orientační, nemá žádnou měřicí jednotku. Je to pouze jednotka sloužící k porovnání nákladů na jednotlivé operace
CARDINALITY	NUMERIC	Udává předpokládané množství řádků, ke kterým operace přistoupí
BYTES	NUMERIC	Udává předpokládané množství bytů, které operace přečte
CPU_COST	NUMERIC	Předpokládané využití procesoru. Tato hodnota je úměrná počtu procesorů
IO_COST	NUMERIC	Předpokládané náklady na I/O operace. Tato hodnota je úměrná přečtených operací
PROJECTION	VARCHAR2(4000)	Výrazy vytvořené operací

Zdroj: ORACLE

Následující tabulka obsahuje důležité hodnoty pro sloupec OPERATION. Kompletní a detailnější rozpis lze nalézt na:

http://www.lorentzcenter.nl/awcourse/oracle/server.920/a96533/ex_plan.htm#23465

Tabulka 5: Důležité hodnoty pro sloupec OPERATION

Sloupec	Typ	Popis
TABLE ACCESS	FULL	Získání všechny řádky z tabulky
TABLE ACCESS	SAMPLE	Získání části řádků z tabulky
TABLE ACCESS	SAMPLE BY ROWID RANGE	Získání části řádků z tabulky na základě rozsahu RowID
TABLE ACCESS	BY INDEX ROWID	Pro nonpartitioned (nerozdělené) tabulky, řádky nalezeny s použitím indexu.
INDEX	UNIQUE SCAN	Získání jednoho řádku z indexu. Indexové hodnoty jsou hledány vzestupně.
INDEX	FULL SCAN	Získání všech řádků z indexu
INDEX	SKIP SCAN	Získání hodnot RowID ze složeného indexu
HASH JOIN		Operace spojující dvě skupiny řádků a návrat výsledku
COUNT		Operace počítající počet řádků ve vybrané tabulce
SORT	JOIN	Operace třídící skupinu řádků přes MERGE-JOIN
MERGE JOIN		Operace přijetí dvou skupin řádků, každá skupina je seříděna dle specifických hodnot, a spojení každého řádku z jedné skupiny s odpovídajícími řádky druhé skupiny.

Zdroj: ORACLE

Exekuční plány nejsou univerzální. Databáze se rozhoduje, jak spustí dotaz na základě aktuální situace (případně aktuálního stavu databáze/tabulek), která se může měnit v závislosti na čase, nebo může být různá na jiných systémech.

Chceme-li ovlivnit běh exekučního plánu, můžeme použít *Hint*. Databáze obsahuje přes 100 hintů, které se dají využít při ladění dotazu. *Hint* je příkaz vložený do SQL dotazu, který ovlivní jeho spuštění. Lze jej například využít v případě znalosti informací o datech v databázi, které databáze nemusí mít, může se jednat například o zastaralé statistiky. V takovém případě by databáze mohla bez použití *hintu* vybrat horší exekuční plán. Pomocí *hintů* můžeme:

- specifikovat přístupové cesty pro tabulku
- určit pořadí JOIN
- určit metody, které budou použity pro JOIN
- nastavit optimalizační parametry

Hinty vkládáme do SQL dotazů za SELECT, UPDATE či DELETE do bloku, například `/*+ALL_ROWS */`, řekne optimalizátoru, aby vrátil všechny řádky co nejdříve. Můžeme mu také říct, aby provedl Full Table Scan pomocí hintu `/*+FULL */` nebo říct, že chceme pospojovat tabulky v pořadí, jaké je nastavené v podmínce WHERE `/*+ORDERED */`. Není-li hint správně syntakticky zapsán, bude ignorován a dotaz poběží, jako by tam nebyl. Výpis nejčastěji používaných hintů je dostupný v dokumentaci databáze: <http://www.oracle.com/hints.jsp>

Testujeme-li SQL dotaz na jiném prostředí, než je samotná produkce, tak se výsledek exekučního plánu v tomto prostředí může lišit od skutečného spuštění na produkci. Tento rozdíl může být zapříčiněn některou z následujících možností:

- EP běží na jiné databázi
- EP běží pod jiným uživatelem
- Schéma DB se liší (například rozdílné indexy)
- EP běží na jiném množství dat
- Inicializační parametry jsou rozdílně nastaveny

Optimalizátor sám o sobě nedokáže rozhodnout, zda je spuštěný dotaz efektivní či nikoliv. V případě, že EP ukáže použití index scanu místo Full Table Scanu, neznamená to, že dotaz běží efektivně. Některé EP mohou být neefektivní. Proto je vhodné použít EP ke zjištění přístupu k tabulkám a objektům a později ověřit, že použité metody a objekty jsou opravdu optimální.

11. BULK OPERACE

Tato kapitola se netýká přímo optimalizace, nicméně mi přišlo důležité se zmínit o možnostech bulk operací, které šetří jak systémové prostředky tak i čas. Jak již bylo zmíněno v úvodu, dotazy je potřeba testovat na vhodných datech, důležitá není jenom struktura, ale i jejich množství. Pokud se na produkci očekávají miliony záznamů, tak test dotazů na deseti řádcích nebude nejpřesnější. Pomocí PL/SQL lze vytvořit vhodná testovací data v potřebném množství. V následujících dvou tabulkách je porovnání operací `SELECT` a `INSERT`. Oba příklady je možno vyzkoušet prakticky (více informací na přiloženém DVD /SQL/bulk_test.txt)

Tabulka 6: Operace `INSERT`

Počet operací	INSERT non-BULK (ms)	INSERT BULK (ms)	Rozdíl (ms)
10	23	19	4
100	24	19	5
1000	94	37	57
10000	604	119	485
100000	5598	911	4687
1000000	72170	15526	56644

Tabulka 7: Operace `SELECT`

Počet operací	SELECT non-BULK (ms)	SELECT BULK (ms)	Rozdíl (ms)
10	7	1	6
100	12	2	10
1000	33	7	26
10000	179	22	157
100000	1736	146	1590
1000000	17084	1475	15609

Z posledního sloupce je zřejmé, že bulk operace je podstatně rychlejší než klasický *for* cyklus. Při uložení milionu řádků do tabulky je rozdíl téměř jedna minuta. V obrovských databázích přitom může být řádově více dat.

12. ZÁVĚR

Tato bakalářské práce měla dva základní cíle. Jednak upozornit na problematiku optimalizace SQL dotazů, které je mnohdy podceňovaná. Zároveň měla ukázat možnosti, jak nalezneme problematické dotazy, které ovlivňují výkonnost databáze Oracle. Představuje možnosti analytických nástrojů usnadňující proces optimalizace. Optimalizace SQL dotazů je náročný proces, který vyžaduje pochopení alespoň základních principů. Proto jsem práci rozdělil na dvě části. V první části popisuji základní objekty nezbytně nutné k pochopení optimalizace. V druhé části jsem se zaměřil na popis ladících a analytických nástrojů.

Člověk může přečíst tuny knih o psaní SQL dotazů, nastavování databází, konfiguracích, apod., nicméně tyto znalosti bez praxe jsou jen pouhou teorií.

Potřebujete-li ladit výkon, zaměřte se na:

1. Nalezení dotazů, které mají dopad na výkonnost
2. Zjistěte jaké Exekuční plány mají testované SQL dotazy
3. Ladění problematických dotazů

Během psaní této práce ale i během mé praxe, jsem narazil na několik pravidel, které se vyplatí dodržovat.

1. Ověřte použití nejvhodnějšího indexu

Někdy má databáze na výběr z více indexů, které jsou k dispozici. Je nutné se ujistit, že databáze použije ten nejvhodnější index. Doporučuji použít nabízených Hintů.

2. Používejte `EXPLAIN PLAN`

Exekuční plán Vám prozradí, zda a které indexy, typy spojení tabulek (pokud dotaz obsahuje více tabulek) databáze použije při spuštění dotazu a ukáže jakou metodou přistoupí k datům.

3. Používejte materializované pohledy

V případě ladění výkonu nás bude především zajímat možnost snížení zátěže a rozdělení tabulek na menší celky.

4. Nepoužívejte * za příkazem SELECT

* za příkazem `SELECT` znamená, že SQL dotaz ve výsledku vrátí všechny sloupce tabulky (takže i takové, které nutně nepotřebujeme, což zvýší objem přenesených dat), ale také musí najít jména sloupců v datového slovníku a nahradit je v SQL dotazu.

5. Používejte TRUNCATE místo DELETE

Při spuštění `TRUNCATE` nejsou na rozdíl od příkazu `DELETE` vytvářeny UNDO data. Nevýhodou však může být, že data po spuštění `TRUNCATE` již nejdou zpět obnovit (pomineme-li možnost zálohy), zato je operace rychlejší a potřebuje méně prostředků.

6. Neprovádějte početní operace a funkce nad indexovým sloupcem ve WHERE klauzuli

Jak bylo názorně ukázáno v kapitole Indexy, použití funkcí nad sloupcem s indexem způsobí spuštění Full Table Scan. Výjimkou je použití funkcí `MIN` a `MAX`.

7. Snižte počet zbytečných FTS nad velkými tabulkami, kde to není potřeba

Zbytečné FTS nad velkými tabulkami vyžadují obrovské množství diskových operací a zatěžují databázi. Jednou z nejběžnějších činností je přidání indexu (databáze jich obsahuje celou řadu), který zabrání FTS. I v tomto případě lze využít existujících hintů a přimět databázi k jejich aplikaci.

8. Nebojte se FTS

Ač se může zdát, že tato rada popírá předchozí, není tomu tak. Pokud dotaz vrací větší procento řádků celé tabulky, může být FTS rychlejší než index scan. Tato skutečnost je ovšem ovlivněna mnoha faktory.

9. Používejte aliasy

Aliasy používejte vždy, kdykoliv má dotaz víc než jednu tabulku. Aplikace aliasů přispěje ke snížení času na parsování, ale také zabrání syntaktickým chybám, při přidání jména sloupce, které se již vyskytuje v jiné tabulce, definované v SQL dotazu.

10. Udržujte statistiky

Statistika je nezbytná k správnému určení Exekučního plánu

A jedno zlaté pravidlo na konec:

11. Nevymýšlejte vymyšlené

Pokud máte možnost použít nástroje a skripty, které jsou k dispozici, využijte je.

Příkladem může být sada ladících skriptů:

Advanced Oracle Monitoring and Tuning Script Collection:

http://www.rampant-books.com/download_adv_mon_tuning.htm

Podobných rad by se našla celé řada a jak již bylo několikrát řečeno, každý projekt je jiný a vyžaduje různé úpravy a modifikace. Nicméně základní pravidla popsane v této práci se nemění a jsou všeobecně platná.

Téma této bakalářské práce se mi skutečně zdálo velmi zajímavé a poučné. Optimalizaci SQL dotazů jsem jako vývojář věnoval dostatečnou pozornost, nicméně si troufám říct, že se nyní na tuto problematiku dokážu podívat i z jiných úhlů a připravit ještě efektivnější dotazy než v minulosti. Domnívám se, že informace zpracované v této práci mi pomohly lépe se zorientovat v dané oblasti, neboť mnoho uvedených informací se nevztahuje pouze na databázi Oracle, ale i na jiné databázové systémy.

13. CONCLUSION

This thesis had two main goals. First, to highlight the problems of SQL query optimization, which are often underestimated. Secondly to show options for finding problematic queries that affect the performance of the Oracle database.

The thesis shows analytical tools that make the optimization process easier. The SQL query optimization is a difficult process that requires at least an understanding of basic principles. Therefore, I divided the work into two parts. The first part describes the basic structures necessary to understand the optimization. The second part focuses on a description of debugging and analysis tools.

A man can read many books about how to write proper SQL queries, how to configure databases, etc., but this knowledge without practice is just only theory.

If you need to tune performance, focus on below:

1. Find queries having an impact on performance
2. Find what Execution plans belong to SQL queries
3. Tune problematic queries

During both writing of this thesis and my praxis I found a few rules we should follow.

1. Ensure, you use the most suitable index

Sometimes, the database needs to choose from more than one available index. It is important to ensure the database uses the most suitable index. I recommend to use Hints.

2. Use `EXPLAIN PLAN`

An Execution plan will tell you whether there was an index, what connection type (if the query contains multiple tables), the database used to run the query and show how they will access to the data.

3. Use Materialized views

In the case of performance tuning we will primarily be interested in the possibility of reducing the workload and the splitting of tables into smaller units.

4. Do not use * behind clause `SELECT`

* behind `SELECT` clause means the SQL query returns all columns of the table (even those, we do not need. This increases amount of received data). The database has to find all column names in the data dictionary and replace them in the SQL query.

5. Use `TRUNCATE` instead of `DELETE`

With `TRUNCATE` no Undo information is generated. The `TRUNCATE` is faster than `DELETE` and needs less system resources. However, the data cannot be recovered back (if we do not have a backup) when `TRUNCATE` runs.

6. Do not use operations over indexed columns in `WHERE` clause

As it was graphically shown in the chapter Indexes, using functions over indexed columns leads to FTS. There is an exception in using function `MIN` and `MAX`

7. Reduce amount of unnecessary FTS over big tables when it is not needed

Unnecessary FTS over large tables requires a huge amount of disk operations and has an impact on database performance. One of the most common activities to avoid FTS is index creation (database contains a number of them). Even in this case, you can use existing hints to force the database to apply it.

8. Do not be afraid of FTS

It may seem that the previous tip is in conflict with this one, but it is not true. If a query returns bigger percent of rows in the table, FTS can be faster than Index scan. This

fact is influenced by many factors.

9. Use aliases

Use aliases every time a query contains more than one table. Using the aliases will reduce time needed for parsing and will avoid of errors in case some other table is added to the SQL query.

10. Maintain Statistics

Statistics are important for correct Execution plan.

And finally, a golden rule:

11. Don't reinvent the wheel.

If you have possibility to use already existing tools and scripts, use them.

A good example may be Tuning tools:

Advanced Oracle Monitoring and Tuning Script Collection:

http://www.rampant-books.com/download_adv_mon_tuning.htm

There may be similar advice. As previously mentioned, every project is different and requires many modifications and changes. However, the basic rules described in this thesis stay the same and are generally valid.

I found the topic of this thesis very interesting. I was used to optimizing every query I wrote during my time as a developer. However, I can say that I can see the problems of SQL performance tuning from different angles and now I can prepare more effective queries than ever before. Information in this thesis helped me to orientate in this area, because many information from this area is not connected to Oracle only, but also exists in other database systems.

14. SEZNAM POUŽITÉ LITERATURY

1. Sajal Dam (2004), *SQL Server Query Performance Tuning Distilled*, Druhé vydání, Apress
2. Bob Bryla, Kevin Loney (2009), *Mistrovství v Oracle Database 11g*, První vydání, Computer press, a.s.
3. Richard Niemiec (2007), *Oracle Database 10g Performance Tuning Tips & Techniques*, Oracle Press
4. Immanuel Chan (2005), *Oracle Database Performance Tuning Guide, 10g Release 2*, Oracle
5. Louis Davidson, Kevin Kline, Kurt Windisch (2006), *Pro SQL Server 2005 Database Design and Optimization*, Apress
6. Prashant S. Sarode, *How To Write Efficient SQL Queries With Tips N Tricks*
7. Dokumentace Oracle <http://www.oracle.com/technology/documentation/index.html>
8. Burleson Consulting <http://www.dba-oracle.com/>
9. Oracle FAQ <http://www.orafaq.com/>
10. Wikipedia http://en.wikipedia.org/wiki/Main_Page

15. SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

Zkratka	Popisek
CBO	Cost Based Optimizer
RBO	Rule Based Optimizer
ADDM	Automatic Database Diagnostic Monitor
AWR	Automatic Workload Repository
EP	EXPLAIN PLAN
OEM	Oracle Enterprise Manager
OTM	Oracle Trace Manager
FTS	Full Table Scan
IRS	Index Range Scan
IUS	Index Unique Scan
FFIS	Fast Full Index Scan
UIO	Index Unique Scan
MV	Materialized view

16. SEZNAM OBRÁZKŮ

1. Výstup profileru
2. Nazorný příklad – Function based index
3. Nazorný příklad – Function based index
4. Nazorný příklad – Function based index
5. Nazorný příklad – Function based index
6. B-Tree index - http://www.dba-oracle.com/art_9i_indexing.htm
7. Bimap index - http://images.devshed.com/ds/stories/Oracle_Data_Structures/image_2.JPG
8. R-Tree index - <http://en.wikipedia.org/wiki/File:R-tree.svg>
9. R-Tree index - 3D vizualizace - <http://en.wikipedia.org/wiki/File:RTree-Visualization-3D.svg>
10. Nested loop join – CD – images/nestedloop.gif
11. Sort Merge join – CD – images/sortmerge.gif
12. Sort Merge join – CD – images/hashjoin.gif
13. AutoTrace
14. Exekuční plán

17. SEZNAM TABULEK

1. Schéma tabulky
2. Statistiky
3. Oracle AUTOTRACE – Statistiky
4. Důležité sloupce tabulky PLAN_TABLE
5. Důležité hodnoty pro sloupec OPERATION
6. Důležité hodnoty pro sloupec OPERATION
7. Operace SELECT

18. SEZNAM PŘÍLOH

1. DVD - Optimalizace SQL dotazů v databázi Oracle