Česká zemědělská univerzita v Praze

Fakulta provozně ekonomická

Optimalizace dotazů

v DBS Oracle

Praha 2008 **Stanislav Trýzna**

Česká zemědělská univerzita v Praze | Fakulta provozně ekonomická

Katedra informačního inženýrství | Akademický rok 2007/2008

ZADÁNÍ DIPLOMOVÉ PRÁCE

Stanislav Trýzna

obor Informatika

Vedoucí katedry Vám ve smyslu Studijního a zkušebního řádu ČZU v Praze čl. 17 odst. 2 určuje tuto diplomovou práci.

Název tématu: **Optimalizace dotazů v DBS Oracle**

Struktura diplomové práce:

1. Úvod
2. Cíl práce a metodika
3. Popište principy fungování optimalizátoru včetně metod přístupů k tabulkám
4. Vymezte a popište metody optimalizace SQL dotazů
5. Aplikujte různé metody otimalizace dotazů na konkrétním zadání a srovnejte výsledky
6. Zobecněte dosažené výsledky pro další možná uplatnění
7. Závěr
8. Seznam literatury
9. Přílohy

Rozsah původní zprávy: 50 - 60 stran

Seznam odborné literatury:

1. Greenberg N. - Vennapusa P. Oracle 9i: SQL tuning workshop. Oracle corporation, 2001
2. Sitansu S. M.Database performance tuning and optimization. Springer-Verlag New York. Inc., 2003
3. Gulutzan P. - Pelzer T. SQL performance tuning.
Addison-Wesley, 2002

Vedoucí diplomové práce: **Ing. Václav Vostrovský, Ph.D.**

Termín odevzdání diplomové práce: duben 2008

L.S.

Vedoucí katedry

Děkan

V Praze dne: 9.1.2008

Evidováno děkanátem PEF pod č.j.: KII-003-08D

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně s použitím uvedené literatury.

Praha 18. 4. 2008 **Stanislav Trýzna**

Poděkování

Tímto bych chtěl poděkovat Ing. Václavu Vostrovskému Ph. D. za cenné připomínky, odborné vedení a trpělivost při zpracovávání diplomové práce.

Souhrn

Tato diplomová práce se zaměřuje na optimalizaci dotazů v databázovém systému Oracle. Konkrétně se v teoretické části věnuje obecnému popisu optimalizátoru, způsobu jeho fungovaní a jednotlivým krokům jeho činnosti. Dále je v teoretické části popsáno jakým způsobem optimalizátor přistupuje k tabulkám, jakým způsobem je možné provádět spojování tabulek mezi sebou, jakými pravidly se optimalizátor při volbě spojení řídí a na závěr je popsáno ovlivnění činnosti optimalizátoru pomocí tzv. hintů. Druhá část práce se zabývá praktickými testy nad vybranou bází dat. Je zde charakterizována báze dat, nad kterou jsou prováděny praktické testy. Praktické testování je rozděleno do několika testovaných oblastí, a to optimalizace konstrukce dotazu, porovnání typů indexů, změny transakční uspořádání a volba vhodného režimu zpracování.

Klíčová slova

Optimaalizátor, rule-based optimalizace, cost-based optimalizace, nested loop, hash, merge, full table, index, hint, spojení, dotaz, optimalizace, cena, transakce, režim zpracování

Abstract

This diploma thesis is focused on queries optimization in Oracle DBS. Concretely, in theoretical part, it devotes to general description of optimizer, the way it works and particulat steps of it's behavior. Further, there is described, how optimizer accesses tables, how it's possible to join the table mutually a what rules the optimizer is governed by when joining tables. At the end of theoretical part, there is described, how thebehavior of optimizer is possible to be influenced by use of hints. The second part of diploma thesis covers the practical tests over the selected database. In this part, there is described the selected database schema and practical testing is devided into several areas, respectively query optimization, compare index type, transaction scope chase and choose proper process environment.

Key words

Optimizer, rule-based optimization, cost-based optimization, nested loop, hash, merge, full table, index, hint, join, query, optimization, cost, transaction, process environment

OBSAH

1 Úvod

V dnešní době není pro databázového vývojáře dostačující napsat dotaz, kterým získá správná data, tak jak skutečně zamýšlel. SQL je deklarativní a umožňuje mnoho možných způsobů jak formulovat dotaz tak, že výsledek běhu dotazu bude stejný, respektive získaná data budou shodná, ale každý z dotazů bude mít jinou dobu běhu. Je tedy i rozhodující sestavit takový dotaz, který bude dostatečně optimalizován tak, aby jeho doba běhu byla v rámci možností co nejmenší.

Na téma optimalizace dotazů bylo v uplynulých desítkách let napsáno nespočet článků a knih. Je to široká oblast, kterou je možno zkoumat jak teoreticky, tak i na reálných databázových systémech v praxi. Pokud by se diplomová práce měla věnovat optimalizaci všeobecně, bylo by potřeba být, buď velmi stručný, nebo popsat stovky stran. Proto se práce zaměřuje především na optimalizaci dotazu prostřednictvím tzv. hintů v prostředí databázového systému Oracle.

Pomocí SQL dotazu zakomponovaného v aplikaci formuluje programátor požadavek vůči databázovému serveru. Říká mu, jaké informace, ze kterých databázových tabulek, jsou požadovány, co klient potřebuje vědět. Neříká již, jak má databázový systém dotaz zpracovat. Otázkou, jakým způsobem má být databázovým systémem dotaz efektivně zpracován, se zabývá tzv. optimalizátor SQL dotazů (query optimizer).

Tematicky patří problematika optimalizátoru SQL dotazů do rozsáhlé oblasti ladění výkonnosti databázového systému. Víme, že na výkonnost databázového systému má rozhodující vliv aplikační a databázový design. Tato diplomová práce se ale vzhledem k rozsáhlosti problému optimalizace databází zabývá především optimalizací z hlediska dotazu.

2 Cíl práce a metodika

Cíle této diplomové práce, zabývající se optimalizací dotazu v prostředí DBS Oracle, lze formulovat do následujících bodů:

1) Popis zpracování dotazu optimalizátorem
2) Charakteristika metod přístupu, k tabulkám a práce s nimi
3) Definice obecných aspektů optimalizace zpracování úloh
4) Ovlivnění činnosti optimalizátoru změnou konstrukce dotazu
5) Praktické testování ostatních aspektů optimalizace zpracování dotazu

V části věnované popisu zpracování dotazu optimalizátorem se práce konkrétně věnuje obecnému popisu optimalizátoru, způsobu jeho fungovaní a jednotlivým krokům jeho činnosti. V druhé části, zabývající se teoretickou charakteristikou metod přístupu k tabulkám a prací s nimi je popsáno, jakým způsobem optimalizátor přistupuje k tabulkám s daty, respektive jakou volí metodu přístupu, jakým způsobem je možné provádět spojování tabulek mezi sebou a jakými pravidly se optimalizátor při volbě spojení řídí.

Ve svém závěru se teoretická část zabývá metodou ovlivnění činnosti optimalizátoru prostřednictvím tzv. hintů. Praktická část práce se zabývá několika praktickými testy nad vybranou bází dat. Je zde poukázáno na různé aspekty optimalizace, charakterizována báze dat, nad kterou jsou prováděny praktické testy. Praktické testování je rozděleno do několika testovaných oblastí.

Je zde ukázáno praktické použití hintů za účelem ovlivnění činnosti optimalizátoru a optimalizování daného dotazu. Jsou zde uvedeny čtyři varianty dotazu, které jsou vzájemně porovnány na základě výkonu. Z uvedených variant je na závěr vybrána varianta optimální z výkonového hlediska.

Dále jsou testovány další vybrané významné aspekty optimalizace, jako je volba vhodného typu indexu respektive návrh typ indexu na základě povahy úlohy a charakteru dat (optimalizace struktury báze dat), změna transakčního uspořádání úloh s ohledem na výkon

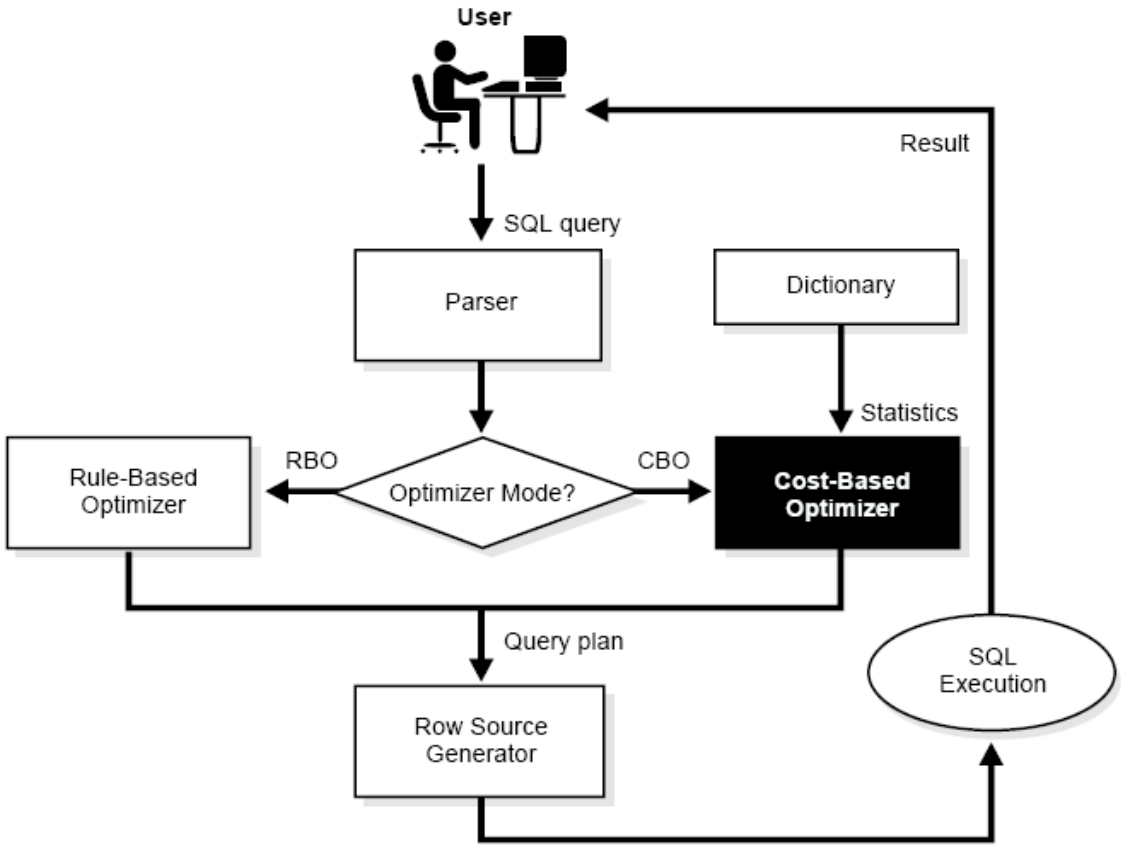
(optimalizace využití systémových zdrojů databázového stroje) a volba vhodného režimu zpracování úloh (optimalizace využití provozní infrastruktury respektive aplikační architektury).

3 Teoretická část

3.1 SQL optimalizátor

Součástí DBS Oracle 9i je komponenta nazývaná optimalizátor. Úkolem optimalizátoru je nalézt nejlepší způsob vykonání uživatelských SQL dotazů, což v podstatě znamená nelézt cestu nejvýkonnějšího provedení dotazu. Optimalizátor například určuje, jestli bude nebo nebude pro daný dotaz použit index a jaké techniky spojení budou použity v případě spojovaní více tabulek.[11] Tato rozhodnutí mají obrovský efekt na výkon SQL dotazu a optimalizace dotazu je klíčovým prvkem pro každou databázovou aplikaci.

Optimalizátor, při rozhodování o nejlepším způsobu vykonání SQL dotazu, vytvoří tzv. exekuční plán, který představuje po sobě následující sérii kroků, které jsou vykonávány při zpracovávání dotazu. Na obrázku 3.1 je zobrazena architektura zpracování SQL dotazu.



Obr.3.1 – Architektura zpracování SQL dotazu [5]

3.1.1 Zpracování dotazu

Základní kroky zpracování dotazu jsou rozložení (parsing), spojení (binding), vykonání (executing) a výběr záznamů (fetching).

Parser

Parser provadí dvě základní operace:

- Analýza syntaxe – kontroluje SQL dotaz z hlediska správnosti syntaxe
- Analýza sémantiky – například kontrola správnosti odkazovaných databázových objektů a jejich atributů

Generátor řádek (Row source generator)

Tato komponenta, nazývaná generátor řádek, připraví plán vykonání SQL dotazu na základě optimálního plánu vyprodukovaného optimalizátorem.

SQL execution engine

SQL execution engine je komponenta, která operuje nad exekučním plánem a na jeho základě generuje výsledky SQL dotazu.

3.1.2 Kroky činnosti optimalizátoru

Pro jakýkoliv dotaz zpracovávaný databázovým systémem oracle platí, že optimalizátor při své činnosti vykoná následující kroky:

1) Vyhodnocení výrazů a podmínek obsažených v SQL dotazu
2) Transformace dotazu – složitější dotazy, obsahující například souvztažné poddotazy (correlated subqueries) nebo pohledy, mohou být optimalizátorem transformovány do ekvivalentního tvaru, výhodnějšího pro zpracování.
3) Volba optimalizačního přístupu – optimalizátor zvolí vhodný optimalizační přístup, buď tzv. Rule-based nebo Cost-based přístup.

4) Volba přístupové metody – pro každou tabulku, ke které je přistupováno v rámci SQL dotazu, optimalizátor zvolí vhodnou metodu přístupu k datům těchto tabulek.
5) Výběr pořadí spojení tabulek – v případě dotazu, ve kterém je spojováno více než dvě tabulky, vybere optimalizátor jaké dvě tabulky budou spojeny nejdřív, a posléze výsledek spojení propojí s další tabulkou.
6) Výběr spojovací metody – je-li potřeba provést spojení dvou a více tabulek, optimalizátor vybere vhodnou spojovací metodu.

Databázový systém Oracle má k dispozici různé metody a přístupové cesty, které může použít pro vykonání SQL příkazu. Pro nalezení nejvýkonnějšího způsobu provedení dotazu má optimalizátor k dispozici dvě metody, a to metodu založenou na ceně dotazu (cost-based optimalizátor CBO) a metodu založenou na přednostních pravidlech (rule–based optimalizátor RBO). [5]

Rule-based optimalizace

Rule-based optimalizátor využívá k určení, která cesta bude použita k přístupu k databázi, sadu přednostních pravidel. Přednostní pravidla představují předdefinované přístupové metody, z nich každá je číselně ohodnocena a dle ohodnocení jsou tyto metody seřazeny od nejlepší po nejhorší. Nejlepší metodou je v tomto žebříčku prohledávání podle unikátního identifikátoru záznamu v tabulce tzv. rowid. Naopak metodou s nejhorším ohodnocením je prohledávání všech záznamů v tabulce tzv. full table scan. Optimalizátor zvažuje všechny možné přístupové cesty na základě syntaxe daného SQL dotazu a vybírá tu, která má nejnižší ohodnocení.[5] To znamená, že jakákoliv změna syntaxe SQL dotazu může mít vliv na výkon zpracování dotazu.

Další charakteristickou vlastností rule-based optimalizace je to, že optimalizátor nemá k dispozici žádné informace o tabulkách a datech v nich. Nevyužívá tedy informace o počtu řádek či hodnotách ve sloupcích tabulek k nimž bude přistupováno. Tyto informace, které jsou označovány jako statistiky, jsou naopak podstatné při použití cost-based optimalizace.

Cost-based optimalizace

Možnost optimalizace na základě ceny či nákladů na provedení SQL dotazu byla prvně implementována v sedmé verzi databázového systému Oracle a je založena na odhadu náročnosti na systémové prostředky. Oproti RBO je CBO mnohem sofistikovanějším prostředkem. Pro určení nejlepšího způsobu vykonání dotazu využívá, jak již bylo zmíněno, statistik uložených v tzv. datovém slovníku (data dictionary), v němž jsou uchovávány informace o objektech databáze. Je zřejmé, že tento přístup představuje flexibilnější a efektivnější cestu k nalezení nejhodnějšího exekučního plánu, než je řízení se pevně danými pravidly jak je tomu v případě RBO.[11]

Při kalkulaci ceny dotazu optimalizátor zvažuje následující:

- Počet logických čtení (čtení z datové paměti) – nejdůležitější faktor
- Využití CPU
- Propustnost sítě

Při své činnosti vykonává CBO 3 základní kroky:

1. V prvním kroku optimalizátor vygeneruje na základě přístupových cest k tabulkám a hintů sadu potenciálních exekučních plánů pro daný SQL dotaz.
2. Podle statistik v data dictionary optimalizátor určí cenu jednotlivých exekučních plánů, při čemž zohledňuje i volnou paměť serveru, zatížení CPU a složitost I/O operací.
3. V další části své činnosti optimalizátor porovná ceny jednotlivých plánů a vybere ten s cenou nejnižší.

3.1.3 Konfigurační parametr OPTIMIZER_MODE

Databázový systém Oracle rozlišuje mezi čtyřmi možnými základními nastaveními přístupu optimalizátoru. Výběr přístupu optimalizátoru se určuje parametrem OPTIMIZER_MODE, ten může být nastaven na:

- CHOOSE
- RULE

- FIRST_ROWS
- FIRST_ROWS_n
- ALL_ROWS

CHOOSE

Hodnota CHOOSE je standardním nastavením přístupu optimalizátoru. V tomto případě optimalizátor zvolí mezi cost-based optimalizací a rule-based optimalizací podle toho, zda jsou k dispozici statistiky či nikoliv. Jsou-li k dispozici statisky alespoň k jedné z tabulek, k nimž se přistupuje, zvolí optimalizátor cost-based přístup. V případě existence pouze částečných statistik je opět zvolen cost-based přístup a chybějící statistiky jsou odhadnuté. Následkem toho může být zvolení neefektivního plánu vykonání SQL dotazu, což se negativně projeví na výkonně zpracování. Když se v datovém slovníku (data dictionary) nenachází žádné statistiky k objektům obsaženým v SQL dotazu, optimalizátor vybere rule-based přístup.[5]

FIRST_ROWS

V případě nastavení parametru optimizer_mode na FIRST_ROWS bude použit cost-based přístup s aplikací heuristik pro rychlé zpracování několika prvních řádků. Bude navracen určitý počet řádků v co nejkratší době, třebaže doba běhu celého dotazu je delší. Použití je vhodné v případě že chceme získat alespoň pár záznamů co nejrychleji.[5]

FIRST_ROWS_n

Tento mód byl zaveden prvně ve verzi Oracle9i, je shodný s FIRST_ROWS, navíc umožňuje definování počtu navrácených záznamů (1,10,100 nebo 1000).

RULE

Hodnota RULE v parametru OPTIMIZER_MODE udává použití rule based optimalizace

ALL_ROWS

ALL_ROWS je protikladem FIRST_ROWS a přinutí optimalizátor použít cost-based přístup i v případě, že neexistují statistiky k objektům k nimž SQL dotaz přistupuje. Tento mód je nejvhodnější pro dávkově orientované dotazy.

Nastavení parametru OPTIMIZER_MODE je možné na třech úrovních:

- Instance
- Session
- SQL příkazu

Nastavení optimalizačního přístupu na databázové instanční úrovní je platné i pro ostatní dvě úrovně. Je ale možné toto nastavení změnit on-line pro danou session pomocí příkazu ALTER SESSION.

Na úrovni SQL dotazu je možno ovlivnit nastavení optimalizačního módu tzv. hinty, těm bude věnována pozornost později.

3.2 CBO statistiky v data dictionary

Optimalizační přístup cost-based využívá statistiky uložené v data dictionary. Tyto informace o objektech databáze jsou optimalizátoru k dispozici jakmile je daný objekt (tabulka) analyzována pomocí příkazu ANALYSE nebo prostřednictvím balíku DBMS_STATS.

3.2.1 Příkaz ANALYZE

Jak již bylo zmíněno, příkazem ANALYZE lze získat statistiky o objektech databáze, tedy statistiky o tabulkách, clusterech, indexech. Získané statistiky jsou posléze uloženy do data dictionary. Analyzováním tabulky příkazem ANALYZE je možné nechat Oracle, aby prošel všechny záznamy v tabulce (ANALYZE COMPUTE) nebo, aby prošel pouze vzorek dat (ANALYZE ESTIMATE). Obecně je vhodnější použití ANALYZE ESTIMATE pro zisk statistik nad tabulkami většími (nad 1 milion záznamů) a ANALYZE

COMPUTE pro tabulky malé a střední. Následující příkazy představují použití příkazu ANALYZE pro výpočet statistik konkrétní tabulky a indexu.[9]

```
ANALYZE TABLE EMP ESTIMATE STATISTICS SAMPLE 5 PERCENT FOR ALL INDEXED
COLUMNS;
ANALYZE TABLE EMP COMPUTE STATISTICS FOR ALL INDEXED COLUMNS;
ANALYZE INDEX EMP_NDX1 ESTIMATE STATISTICS SAMPLE 5 PERCENT FOR ALL
INDEXED COLUMNS;
```

3.2.2 Balík DBMS_STATS

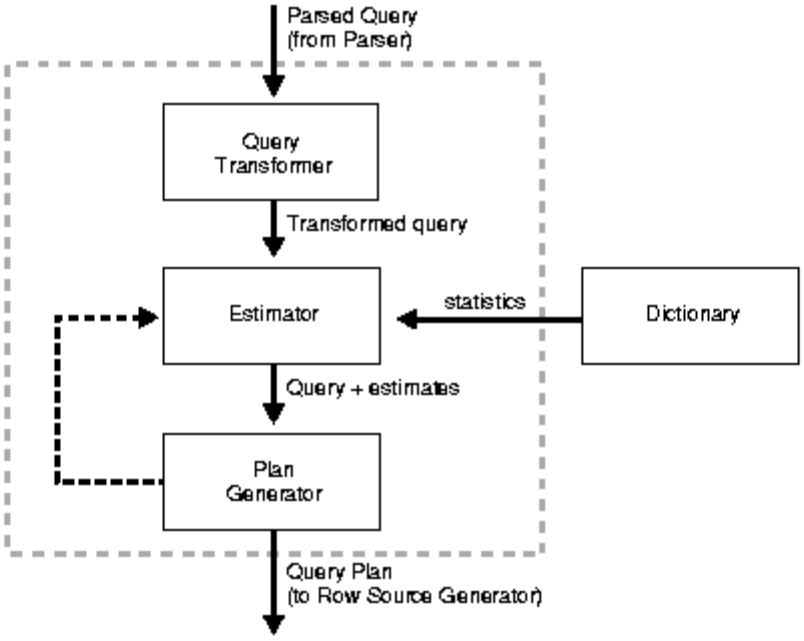
Další, komplexnější možností získání statistik je balík DBMS_STATS. Tento balík obsahuje procedury, které umožňují:

- Shromažďovat statistiky nad tabulkami a indexy
- Modifikovat statistiky nad tabulkami
- Navrácení k původním statistikám, dříve vypočítaným
- Kopírování statistik z jednoho schéma do druhého
- Import statistik z jedné databáze do druhé

3.2.3 Architektura CBO

CBO se skládá ze třech hlavních komponent, jsou jimi:

- Transformátor dotazu
- Ohodnocovač dotazu
- Generátor plánu



Obr. 3.2- Architektura CBO[5]

Transformátor dotazu

Poté, co je parserem zkontrolována syntaktická správnost dotazu, je dotaz postoupen optimalizátoru a v činnost je uveden transformátor dotazu. Jeho hlavní úlohou je rozhodnout, zda je možné změnit formu dotazu tak, aby bylo možné vygenerovat lepší plán vykonání dotazu. Transformovaný dotaz je sémanticky ekvivalentní s původním, ale jeho vykonání je efektivnější. Transformátor dotazu při své činnosti využívá čtyři transformační techniky:

- Spojování pohledů
- Protlačování predikátů
- Rozložení poddotazů
- Přepsání dotazu s materializovaným pohledem

Spojování pohledů

Technika spojování pohledů je pravděpodobně nejjednodušší formou transformace SQL dotazu. Tato technika umožňuje z dotazu, který obsahuje pohled, odkazování

na pohled zcela vyjmout. A to tak, že definice pohledu je spojena s dotazem. Následující příklad zobrazuje definici pohledu a SQL dotaz s odkazem na pohled.

```
CREATE VIEW test_view AS
SELECT ename, dname, sal FROM emp e, dept d
WHERE e.deptno = d.deptno;
SELECT ename, dname FROM test_view WHERE sal > 10000;
```

Bez transformace je jedinou možností vykonání dotazu z příkladu 1.1 spojení všech řádek tabulky emp se všemi řádky tabulky dept a poté vyfiltrování patřičných záznamů splňujících podmínku sal > 1000.

Technikou spojení pohledů může být dotaz z příkladu 1.1 transformován do následující podoby:

```
SELECT ename, dname FROM emp e, dept d
WHERE e.deptno = d.deptno
AND e.sal > 10000;
```

Při zpracování uvedeného transformovaného dotazu může být predikát e.sal > 1000 aplikován ještě před spojením tabulek. Tím dojde ke zvýšení výkonu, jelikož je redukováno množství spojovaných dat. Transformace znamenají zjevné zvýšení výkonu i v případě takovéhoto velmi jednoduchého případu.

Spojování komplexních pohledů

Předchozí příklad zobrazoval velmi přímou operaci spojení pohledu, nicméně komplexnější pohledy, jako jsou pohledy obsahující klauzuli GROUP BY nebo DISTINCT není až tak jednoduché spojit. K těmto účelům Oracle poskytuje několik sofistikovaných technik.[7]

Jako příklad poslouží následující definice pohledu obsahující klauzuli GROUP BY. Je zde vypočítána průměrná mzda pro každé oddělení.

```
CREATE VIEW avg_sal_view AS
SELECT deptno, AVG(sal) avg_sal_dept FROM emp
GROUP BY deptno;
```

Dotaz, který vyhledá průměrnou mzdu pro každé oddělení v určité oblasti, vypadá takto:

```
SELECT dept.name, avg_sal_dept
FROM dept, avg_sal_view
WHERE dept.deptno = avg_sal_view.deptno
AND dept.loc = 'OAKLAND';
```

Pohled a dotaz mohou být transformovány do této podoby:

```
SELECT dept.name, AVG(sal)
FROM dept, emp
WHERE dept.deptno = emp.deptno
AND dept.loc = 'OAKLAND'
GROUP BY dept.rowid, dept.name;
```

Pozitivní vliv této transformace na výkon zpracování dotazu je zjevný. Namísto sloučení všech dat v tabulce emp podle klauzule GROUP BY, je pomocí transformace dosaženo toho, že data tabulky emp jsou spojeny a vyfiltrovány ještě před sloučením klauzulí GROUP BY.

Protlačování predikátů

Složité dotazy mohou obsahovat několik pohledů a poddotazů s velkým množstvím predikátů, na ně aplikovaných. Za účelem generování dotazu s lepším výkonem umožňuje Oracle přesun predikátů dovnitř nebo mimo definici pohledu. Tato technika je ilustrována následujícím příkladě pohledu.

```
CREATE VIEW emp_agg AS
SELECT
 deptno,
 AVG(sal) avg_sal,
FROM emp
GROUP BY deptno;
```

Při vykonání dotazu

```
SELECT deptno, avg_sal FROM emp_agg WHERE deptno = 10
```

je predikát deptno = 10 přesunut do pohledu a dotaz je transformován do takovéto podoby:

```
SELECT deptno, AVG(sal) FROM emp
WHERE deptno = 10
GROUP BY deptno;
```

Výhodou takto transformovaného dotazu je, že predikát deptno = 10 je aplikován před operací GROUP BY což umožňuje značnou redukci dat která mají být agregována.

V důsledku předání podmínky z WHERE klauzule se objevují možnosti filtrace záznamů a tím se sníží velikost dat, nad kterými jsou později prováděny operace jako je spojování ne slučovaní GROUP BY klauzulí. Navíc tato technika umožňuje vyšší výkon tím, že zpřístupní nové cesty přístupu k datům, které nemusely být předtím k dispozici.[3]

Přepsání dotazu s materializovaným pohledem

Rychlost zpracování SQL dotazu je možno výrazně ovlivnit tím, že běžně užívaná data jsou přepočítáním a uložena ve formě materializovaného pohledu. Oracle umožňuje transformaci SQL dotazu takovým způsobem, že jedna nebo více tabulek odkazovaných v dotazu jsou nahrazeny odkazem na materializovaný pohled. Když je materializovaný pohled menší než původní tabulka, nebo má umožněnou lepší přístupovou cestu k datům, pak by transformovaný dotaz mohl být vykonán podstatně rychleji než ten původní.[3]

Následující příklad ukazuje materializovaný pohled, který může sloužit k optimalizaci daného dotazu a transformovaný dotaz.

Materializovaný pohled:

```
CREATE MATERIALIZED VIEW sales_summary
AS SELECT sales.cust_id, time.month, SUM(sales_amount) amt
FROM sales, time
WHERE sales.time_id = time.time_id
GROUP BY sales.cust_id, time.month;
```

Dotaz před transformací:

```
SELECT customer.cust_name, time.month,
     SUM(sales.sales_amount)
FROM sales, customer, time
WHERE sales.cust_id = cust.cust_id
AND sales.time_id = time.time_id
GROUP BY customer.cust_name, time.month;
```

Transformovaný dotaz:

```
SELECT customer.cust_name,
     sales_summary.month,sales_summary.amt
FROM customer, sales_summary
WHERE customer.cust_id = sales_summary.cust_id;
```

Vykonání transformovaného dotazu bude pravděpodobně mnohem rychlejší oproti dotazu původnímu, a to z důvodu menší velikosti tabulky sales_summary oproti tabulce sales, žádné agregaci dat a jednomu spojení méně.

Neplatí vždy, že transformovaný dotaz, který používá materializovaný pohled je efektivnější oproti dotazu původnímu. A to ani když je materializovaný pohled menší než tabulka na, které je založen. Základní tabulka nebo tabulky, mohou být rozsáhleji indexovány, a tudíž poskytují rychlejší přístup k datům. Jedinou cestou jak určit optimální exekuční plán je tedy spočítání ceny dotazu s materializovaným pohledem a bez něj, tyto porovnat a vybrat ten s nižší cenou.

Ohodnocovač dotazu (Estimator)

Další komponentou CBO je oceňovač dotazu. Jeho úkolem je odhad celkové ceny daného exekučního plánu. Za tímto účelem generuje tři typy měr, které jsou mezi sebou navzájem spojené a jedna je odvozena z druhé. Jsou jimy:

- Selektivita
- Kardinalita
- Cena

Selektivita

Selektivita je měrou, která představuje určitou část řádků z nějaké celkové množiny řádků. Je spojena s určitým predikátem nebo kombinací predikátů dotazu. Predikát zde tedy finguje jako filtr, který filtruje určitý počet řádek z celkové množiny záznamů. Je tedy zřejmé, že selektivita ukazuje kolik řádek z celkové množiny projde danou podmínkou.

Selektivita nabývá hodnot od 0 do 1. Hodnota 0 značí, že z celkové množiny nebudou vybrány žádné záznamy, naopak 1 znamená výběr všech záznamů množiny. Pro určení selektivity jsou využívány statistiky, v případě že nejsou k dispozici, tak je přiřazena standardní hodnota, která je implicitně nastavena. Například pro predikát rovnosti je tato standardní hodnota nižší než pro predikát >. To je dáno obecným předpokladem, že predikátem = projde méně záznamů než predikátem >.[5]

Kardinalita

Další mírou je kardinalita. Ta představuje celkový počet řádek v určité množině. Kardinalita může být:

- *Základní kardinalita* - představuje počet řádek v základní tabulce.
- *Efektivní kardinalita* - počet řádek, vybraných ze základní tabulky. Je výsledkem celkové kardinality zkombinované se selektivitou všech predikátů specifikovaných na dané tabulce. Když tedy není na tabulce definován žádný predikát, tak se efektivní kardinalita rovná základní.
- *Kardinalita spojení* - počet řádek vzniklých spojením dvou množin záznamů. Představuje kartézský součin mezi těmito dvěmi množinami s aplikováním predikátu spojení mezi nimi. Kardinalita spojení je tedy produktem kardinalit obou množin záznamů vynásobený selektivitou predikátu spojení.
- *Distinct kardinalita* - počet odlišných hodnot ve sloupci množiny řádek.
- *Group kardinalita* - počet řádek množiny po provedení operace GROUP BY. Závisí na distinct kardinalitě každého sloupce v klauzuli GROUP BY.

Cena

Cena představuje množství jednotek práce nebo využitých zdrojů. Je tedy odhadem vstupně/ výstupních operací disku, vytížení procesoru a množství paměti využité pro provedení daných operací. Operacemi se rozumí prohledávání tabulky, přistupování k řádkům tabulky za použití indexu, spojování dvou tabulek dohromady nebo třídění množiny řádků.

Generátor plánu

Úkolem komponenty nazývané generátor plánu je prověřit možné plány vykonání dotazu a z nich vybrat ten s nejnižší předpokládanou cenou. K dispozici má mnoho různých plánů, jelikož pro daný dotaz existují různé kombinace přístupových cest, metod a pořadí spojení, které mohou být použity pro dosažení stejného výsledku.[11]

Pří výběru optimálního plánu jsou nejprve vybrány optimální plány pro existující poddotazy. Nejprve jsou optimalizovány nejvnitřnější poddotazy a až posléze je optimalizován nejzevnější blok dotazu, kterým je dotaz jako celek.

3.3 Indexy

Pojmem index se v oblasti databází rozumí databázový objekt, který je logicky a fyzicky nezávislý na datech v tabulce. Oracle používá indexy pro přístup k datům nebo pro účely zajištění integritních omezení. Takovéto indexy mohou nabývat buď unikátních nebo neunikátních hodnot. Je zřejmé, že unikátní index zaručuje, že žádné dva prvky indexu nebudou mít identické hodnoty. Krom indexu nad jedním sloupcem tabulky existují i tzv. složené indexy, které jsou vytvářeny na více sloupcích tabulky.[11]

Oracle využívá tyto typy indexů:

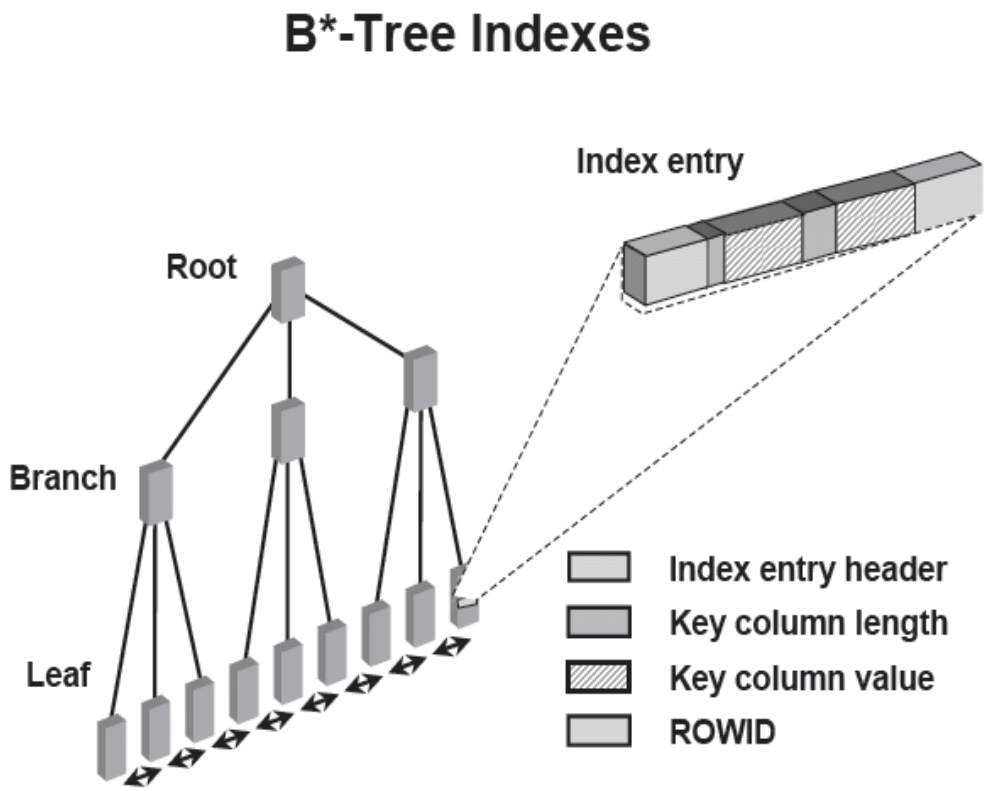
- B-tree index - standardní indexy pro vyrovnání doby přístupu
- Reverzní index - obrací byty každého indexovaného sloupce při zachování jejich pořadí
- Funkční indexy
- Bitmapové indexy
- Doménové a aplikačně specifické indexy

3.3.1 B *- tree indexy

B - tree index je nejstarším a nejznámějším typem indexování systému. Zlepšuje výkon především u dotazů, které z tabulky vybírají malé procento záznamů.

Struktura B*tree indexu je zobrazena na obrázku 2.1. Základním bodem B*-tree indexu je kořen. V závislosti na počtu prvků, může obsahovat jeden nebo více uzlů. Listy B*-tree indexu obsahují samotné hodnoty a ROWID, které určuje řádek v příslušném datovém segmentě.

B*-tree indexy systému Oracle jsou vždy vyvážené, a mohou se rozšiřovat. Může se stát, že výška B*-stromu zbytečně naroste, v takovém případě je potřeba index reorganizovat.[5]



Obr. 3.3.1 - Struktura B*-tree indexu[11]

Koncept prvku indexu je následující:

- vstupní hlavička, která obsahuje počet sloupců a informace pro uzamknutí
- délka a hodnota klíčového sloupce
- ROWID záznamu

Vnitřní uzly stromu slouží na vyhledávání a obsahují:

- minimální prefix klíče, aby bylo možné rozlišit dvě hodnoty
- ukazatel na blok, který obsahuje klíč

Když mají bloky *n* klíčů, pak obsahují *n+1* ukazatelů, přičemž počet klíčů a ukazatelů je omezen velikostí bloku.

Listy B*-stromu složí pro ukládání hodnot a obsahují:

- kompletní klíčovou hodnotu pro každou řádku
- ROWID ukazatele na záznamy v tabulce

Jakmile Oracle přečte z listu indexu informaci o ROWID, může přistupovat fyzicky přímo k řádkům v tabulce. V případě, že se optimalizátor rozhodne použít pro přístup k datům v tabulce index, provede následující kroky:

- prohledá index na výskyt atributů, které obsahuje SQL dotaz
- když jsou všechny atributy obsaženy v indexu, jsou data získaná přímo z indexu, bez použití datového segmentu.
- v případě existence i jiných atributů, je z indexu zjištěna adresa příslušných řádků a datové bloky jsou vyhledané prostřednictvím ROWID.

Výhody B*-tree indexu:

- všechny listy stromu mají stejnou hloubku, tudíž získaní jakéhokoliv záznamu odkudkoliv z indexu zabere zhruba stejnou dobu.
- B*-tree indexy zůstávají automaticky vyvážené.
- je vhodný jak pro malé tak i velké tabulky a není znehodnocován růstem záznamů v tabulce.

3.3.2 Bitmapové indexy

Základní úlohou indexu je poskytnout ukazatel na řádky v tabulce, které obsahují nějakou požadovanou hodnotu. Oproti B*-tree indexu, který k tomuto účelu uchovává ROWID, v bitmapovém indexu se namísto ROWID používá bitmapa pro každou hodnotu sloupce. Každý bit v bitmapě tak odpovídá jednomu řádku v tabulce. Jakmile je bit nastavený na 1, znamená to, že daný řádek nabývá určité hodnoty v bitmapově indexovaném sloupci.[5]

Bitmapové indexy mohou významně zvýšit výkon dotazů, které mají tyto charakteristiky:

- WHERE klauzule obsahuje více predikátů na sloupcích s nízkou nebo střední kardinalitou, existuje jen několik málo možných hodnot pro daný sloupec.
- Jednotlivé predikáty na těchto sloupcích vybírají velké množství záznamů.
- Existuje bitmapový index na některých nebo na všech těchto sloupcích dotazu.
- Dotazované tabulky obsahují velké množství záznamů.

Bitmapové indexy jsou extrémně výkonné při vykonávání dotazu s několika predikáty, které jsou kombinovány s AND nebo OR podmínkami. Optimalizátor pomocí jednoduchých bitových operací AND, OR, NOT skombinuje bitové mapy a získá jednu výslednou, ta je posléze mapována na ROWID. Pro mapování bitmapy na ROWID používá Oracle interní algoritmy.[11]

Oproti B*-tree indexům, které mají často větší velikost než samotná data v tabulce, bitmapové indexy mají díky dobré kompresy zlomek velikosti dat v tabulce, představují tedy značnou úsporu paměti.

Optimalizátor má při své činnosti možnost kombinovat různé typy indexů pro získaní dat z tabulek. Může takto například použít kombinaci indexu bitmapového, doménového a B*tree indexu.

3.4 Základní přístupové metody

Přístupové metody, představují způsob, jakým jsou získávána data z databáze. Základní metody, kterými lze získat jakýkoliv záznam z databáze jsou tyto:

- *Full table prohledávání* - při tomto způsobu prohledávání jsou procházeny všechny záznamy v tabulce a odfiltrovávány ty, které nesplňují vyhledávací kriteria.
- *Rowid* - nejrychlejší metoda získání požadovaných záznamů, jelikož je specifikováno přesné umístění záznamu v databázi.
- *Indexové prohledávání* - touto metodou jsou požadované záznamy získány pomocí indexu, využitím hodnot indexovaného sloupce.

Obecně je vhodnější použití indexového prohledávání v situacích, kdy SQL dotaz vrací malou podmnožinu záznamů tabulky, zatímco full table prohledávání je efektivnější v případě, že se přistupuje k velké části tabulky. Například OLTP (On Line Transaction Processing) databázové aplikace, které se skládají z SQL příkazů s krátkou dobou běhu, jsou charakteristické použitím indexové metody prohledávání. Oproti tomu, systémy pro podporu rozhodování inklinují k používání rozdělených tabulek a full table prohledávání relevantních částí.[5]

3.4.1 Full table prohledávání

Během full table prohledávání jsou prohledávány všechny bloky, které jsou pod tzv. High water mark (značka určující hranici mezi používaným a nepoužívaným místem segmentu). Každý záznam je zkoumán, jestli splňuje podmínky WHERE klauzule.

Full table prohledávání tabulky znamená sekvenční čtení tabulky bez použití indexů nebo jiných prostředků. Je to nejzákladnější metoda přístupu k tabulce, avšak ani zdaleka ne nejpomalejší. Oracle umožňuje čtení více bloků z databázového souboru během jedné vstupně/výstupní (I/O) operace, to je označováno jako multiblokové čtení. Při full table prohledávání je rovněž možné využít paralelizmus a rozdělit prohledávání na více podprocesů.[5] Ve většině zdrojů je jako hranice použití či nepoužití full table prohledávání uváděna hodnota 5%. Tedy pokud se očekává, že dotaz vybere víc než 5% řádků tabulky, pak bude vhodné použít full table prohledávání. V opačném případě je lepší přistupovat k tabulce pomocí indexu. Obecně optimalizátor použije full table prohledávání v následujících případech:

- Dotaz není schopen použít žádný z existujících indexů
- Když optimalizátor myslí, že bude přistupováno k většině bloků v tabulce.
- V případě že se jedná o malou tabulku
- V případě že tabulka nebyla dlouho analyzována, statistiky jsou tedy zastaralé a optimalizátor myslí, že se jedná o malou tabulku

Bylo zmíněno, že full table prohledávání je vhodnější pro přístup k velkému množství dat než indexové prohledávání, a to z důvodu, že full table prohledávání používá rozsáhlejší

I/O operace a provedení méně rozsáhlých I/O operací je méně nákladné než provedení mnoha menších takových operací.

3.4.2 Rowid prohledávání

Rowid záznamu specifikuje datový soubor a blok obsahující konkrétní řádku a umístění této řádky v bloku. Lokalizování řádky na základě jejího rowid je nejrychlejší možný způsob jakým v Oracle lze najít jednu řádku. K tomu aby bylo možné přistupovat k tabulce přes rowid záznamů, musí nejprve Oracle tato rowid získat. To je možné, buď přímo z WHERE klauzule dotazu, nebo skrze jeden nebo více indexů tabulky.

Rowid představuje interní interpretaci Oracle kde jsou data uložena. Jelikož umístění záznamů se může různě měnit, například při migraci, exportu nebo importu, není přistupování k záznamům přes rowid doporučováno.[11]

3.4.3 Indexové prohledávání

Indexovým prohledáváním se data získávají z indexu na základě hodnot jednoho nebo více sloupců v indexu. Pakliže dotaz přistupuje pouze ke sloupcům indexu, Oracle získává data přímo z indexu spíše než z tabulky.

Kromě indexovaných hodnot obsahuje index také rowid záznamu. Proto, v případě, že dotaz přistupuje i k jiným sloupcům než indexovaným, tak Oracle může najít záznamy v tabulce indexovým prohledáváním. Indexové prohledávání může být těchto druhů:

- *Unikátní indexové prohledávání (Index unique scan)* - Toto prohledávání vrací nanejvýš jednu řádku. Je prováděno, pokud SQL dotaz obsahuje omezení na unikátní nebo primární klíč, které zaručí, že je přistupováno pouze k jediné řádce. A zároveň musí být specifikovány všechny sloupce unikátního indexu podmínkou rovnosti.
- *Range indexové prohledávání (Index range scan)* - Používá se pro nalezení výběru dat s vysokou selektivitou. Může být ohraničen (na obou stranách) nebo neohraničen (na jedné či obou stranách), to znamená že může být použita podmínka =, >, < atd.. Data jsou vracena setříděna vzestupně dle sloupců v indexu. Jsou-li hodnoty shodné, vrací se dle rostoucího rowid.

- *Skip indexové prohledávání (Index skip scan)* - Skip indexové prohledávání se vyznačuje tím, že v případě spojeného indexu, složeného z více sloupců, není potřeba v dotaze uvádět prefixový sloupec indexu, je tzv. přeskočen. Optimalizátor použije algoritmus skip indexového prohledávání na získání rowid řádků, které nepoužívají prefixový sloupec indexu. Skip indexový algoritmus tak redukuje potřebu vytvářet index, který se používá sporadicky.
- *Fast full indexové prohledávání* - Je alternativou k full table prohledávání, v případě, že index obsahuje všechny sloupce k nimž dotaz přistupuje a alespoň jeden sloupec indexu má omezení NOT NULL. Fast full indexové prohledávání přistupuje přímo k datům v indexu, nikoliv k tabulce. Čte celý index použitím multiblokového čtení, tudíž je rychlejší než klasický indexový přístup, a může využívat paralelizmu.

Použití indexu je velmi výhodné jako metoda jak se vyhnout třídění. Jazyk SQL obsahuje více komponent, které implicitně používají třídění. Např. operace DISTINCT, GROUP BY, UNION, MINUS a INTERSECT vždy vyvolají třídění. Jelikož třídění hodnot jakéhokoliv sloupce tabulky může být velmi nákladná operace, tak pokud je to možné, je lepší se třídění vyhýbat. Především spojené indexy můžou při redukování třídění velmi pomoct. Protože se index musí číst sekvenčně, ztrácí se při tom schopnost číst víc bloků v jedné I/O operaci. Prohledávání tabulky použitím fast full indexového prohledávání zaručuje, že řádky budou vybrány ve správném pořadí.[11]

3.5 Spojení

Operace spojení představuje příkazy, které získávají data z více než jedné tabulky. Spojení je charakteristické existencí více tabulek v klauzuli FROM a vztah mezi těmito tabulkami je definován spojovací podmínkou v klauzuli WHERE.

Aby mohl být vybrán exekuční plán příkazu spojení, optimalizátor musí rozhodnout o:

- Metodě přístupu - stejně jako v případě jednoduchého dotazu, optimalizátor musí vybrat vhodnou cestu získání dat z každé tabulky.

- Metodě spojení - pro spojení každého páru řádek musí Oracle vykonat jednu z následujících operací:
    - Nested loop spojení
    - Sort merge spojení
    - Hash spojení
    - Clusterové spojení
- Pořadí spojení - při vykonání příkazu, který spojuje více než dvě tabulky, jsou spojeny nejprve řádky dvou tabulek a výsledek je poté spojen s další tabulkou. Tento proces trvá dokud nejsou spojeny všechny tabulky. Je tedy potřeba rozhodnout v jakém pořadí budou tabulky spojovány.[7]

3.5.1 Výběr metody spojení

Výběr vhodné metody spojení probíhá tak, že optimalizátor určí cenu každé spojovací metody a vybere tu s cenou nejnižší. V případě, že spojení vrací mnoho řádek, optimalizátor zvažuje tyto faktory:

- Nested loop spojení je neefektivní v situaci, kdy spojení vrací velké množství řádek (10000 řádek je považováno za velké množství) a optimalizátor se může rozhodnout ho nepoužít.[5] Cena nested loop spojení se vypočte následujícím vzorcem:

  *Cena = cena přístupu k tab. A + (cena přístupu k tab. B * počet řádek z tab. A)*

- Když je vráceno spojením velké množství řádek a je použita CBO, je nejefektivnější metodou spojeni hash spojení. Cena toho spojení se spočte následovně:

  *Cena = (cena přístupu k tab. A * počet hash částí tab. B) + ena přístupu k tab. B*

- V případě vrácení velkého množství řádek po spojení a je-li metou optimalizace RBO, pak nejefektivnější metodou spojení je merge.

  *Cena = cena přístupu k tab.A + cena přístupu k tab. B * (cena setřídění tab.A + cena setřídění tab. B)*

Oracle umožňuje tyto metody spojení:

- Nested loop spojení
- Sort merge spojení
- Hash spojení
- Cluster spojení
- Full outer spojení

Nested loop spojení

Spojení nested loop je užitečné v případě, že jsou spojovány malé podmnožiny dat a spojovací podmínka je účinným způsobem přístupu k druhé tabulce. Je podstatné aby spojovací podmínka byla dobře nastavena.[7]

Důležité je, aby bylo zajištěno, že vnitřní tabulka je závislá na vnější tabulce. Jedna tabulka je zde tedy označena jako vnější (řídící tabulka) a další jako vnitřní (závislá tabulka). Jestliže je přistupování k vnitřní tabulce nezávislé na tabulce vnější, pak jsou při každé iteraci vnějšího cyklu(prohledávání vnější tabulky) získány stejné záznamy, což značně snižuje výkon. [5]

Nested loop spojení zahrnuje tyto kroky:

- Optimalizátor určí řídící tabulku a označí ji jako vnější
- Ostatní tabulky jsou označeny jako vnitřní
- Pro každou řádku vnější tabulky Oracle přistupuje ke všem řádkám vnitřní tabulky

Standardní plán vykonání nested loop spojení:

```
NESTED LOOPS
TABLE ACCESS (...) OF vnější_tabulka
       TABLE ACCESS (...) OF vnitřní_tabulka
```

Nested loop spojení se často používá v kombinaci s přístupem přes indexy. V takovém případě plán vykonání vypadá následovně:

```
NESTED LOOPS
       TABLE ACCESS (BY ROWID) OF vnější_tabulka
              INDEX (... SCAN) OF index_vnější_tabulky
       TABLE ACCESS (BY ROWID) OF vnitřní_tabulka
              INDEX (RANGE SCAN) OF index_vnitřní_tabulky (NON-UNIQUE)
```

V exekučním plánu se vnější cyklus zobrazuje před vnitřním. Tato skutečnost odpovídá jak výše zmíněnému standardnímu plánu vykonávání, tak dále uvedenému příkladu s odpovídajícím exekučním plánem.

Dotaz na tomto příkladu je zpracován tak, že ve vnějším cyklu jsou vybrány v řídící tabulce objednávky daného zákazníka a pro každou objednávku získanou ve vnějším cyklu jsou vybrány záznamy z vnitřní tabulky v průběhu vnitřního cyklu. Identifikace zákazníka je zpracovávajícímu SQL dotazu předána ve vazební proměnné zakaznik.

```
SELECT pol.unit_price * pol.quantiy
FROM order_items pol, orders obj
WHERE obj.customer_id = :zakaznik AND
pol.order_id = obj.order_id
```

Exekuční plán:

```
SELECT STATEMENT
 NESTED LOOPS
  TABLE ACCESS BY INDEX ROWID ORDERS
   INDEX RANGE SCAN CUSOMER_ID
  TABLE ACCESS BY INDEX ROWID ORDER_ITEMS
   INDEX RANGE SCAN ORDER_ID
```

Nested loop spojení je možné použít prakticky při jakémkoliv spojení dvou tabulek bez ohledu na predikát spojení. Kvůli jeho slabé výkonnosti při spojování větších tabulek se ale používá jen tehdy, když není možné použít jinou, efektivnější metodu spojení.[11]

Hash spojení

Hash spojení se používá při spojování velkých datových sad a za podmínky že se jedná o spojení na rovnost. Při použití tohoto typu spojení jsou spojované tabulky prohledávány metodou full prohledávání. Menší ze spojovaných tabulek je použita optimalizátorem

k vytvoření hashovací tabulky v paměti na základě spojovacího klíče. Tato metoda je nejlépe využitelná, když se menší tabulka vejde paměti.

Pakliže hashovaní tabulka příliš naroste, a do paměti se nevejde, pak ji optimalizátor rozdělí na různé části (partitions). Počet partitions závisí na množství přístupné paměti. Když i jednotlivé části přesáhnou alokovanou paměť, jsou potom uloženy do dočasných segmentů na disk. [5]

Po vytvoření hashovaní tabulky dochází k následujícím procesům:

- Druhá, větší tabulka je prozkoumána
- Stejně jako menší tabulka je rozdělena do partitions
- Partitions jsou zapsány na disk

Následující příklad obsahuje ukázku dotazu s hash spojením, kde tabulka orders je použita jako první na vytvoření hashovaní tabulky a druhá větší tabulka je prohledávána posléze.

```
SELECT pol.unit_price * pol.quantiy
FROM order_items pol, orders obj
WHERE pol.order_id = obj.order_id
```

Exekuční plán:

```
SELECT STATEMENT
 HASH JOIN
  TABLE ACCESS FULL ORDERS
  TABLE ACCESS FULL ORDER_ITEMS
```

Sort merge spojení

Metoda spojení sort merge může být použita při spojování dvou nezávislých zdrojů dat. Obecně je tato metoda výkonnostně horší než hash join. Na druhé straně, je ale sort merge spojení výhodnější než hash spojení platí-li obě tyto podmínky:

- Zdroje dat jsou setříděné
- Operace třídění nemusí být vykonána – byla vykonána jinou operací

Nutnost třídění je u tohoto spojení často nejnáročnnější operací, proto bylo-li třídění provedeno během některé z předešlých operací, je toto velkou výhodou.

Výhoda použití sort merge spojení může být ztracena v případě že je při použití tohoto spojení využita pomalejší přístupová metoda k datům (např. indexové prohledávání namísto full table prohledávání).[7] Užitečnost sort merge spojení se projeví v případě, že spojovací podmínka mezi tabulkami je typu <, <=, >, >=. Nevyužívá se zde žádný koncept řídící tabulky, spojení sestává v podstatě ze dvou kroků:

- Operace třídění – vstupní data jsou setříděna dle spojovacího klíče (sloupce přes které se spojuje).
- Operace sloučení (merge) – setříděná data jsou sloučena.

Optimalizátor se rozhoduje, zda použije sort merge a nebo hash spojení na základě již zmíněného, a to:

- Spojovací podmínka mezi tabulkami není typu rovnosti.
- Setřídění tabulek již bylo vykonáno jinou operací, tak optimalizátor shledá sort merge spojení výkonnější.
- Na základě nastavení různých parametrů se optimalizátor domnívá že cena hash spojení bude vyšší.

3.6 Hinty

Databázový systém Oracle umožňuje manuální ovlivnění zpracování SQL dotazu. Je tedy umožněno dělat rozhodnutí, která obvykle provádí právě optimalizátor. K účelům ovlivnění způsobu zpracování dotazu slouží tzv. hinty.

Jako architekt aplikace můžeme častokrát mít větší přehled o datech a znát o nich informace, které samotný optimalizátor nezná. Například můžeme vědět, že určitý index je selektivnější pro určité dotazy a na základě takových informací jsme schopni zvolit efektivnější plán zpracování dotazu než optimalizátor.[5] V takovém, případě použitím hintů, jsme schopni donutit optimalizátor k použití optimálního exekučního plánu.

Můžeme použít hinty pro specifikaci následujícího:

- Optimalizační přístup pro daný SQL dotaz
- Cíl cost-based optimalizace daného dotazu
- Metody přístupu k tabulkám
- Pořadí spojení tabulek
- Metoda operace spojení tabulek

Všechny hinty, kromě hintu RULE, mají za následek použití cost-based optimalizace. Nemá li optimalizátor k dispozici statistiky, budou použity standardní hodnoty.

Hinty ovlivňují zpracování bloku dotazu, v kterém jsou uvedené. Blok dotazu může být například:

- Jednoduchý příkaz SELECT, UPDATE nebo DELETE
- Poddotaz komplexního dotazu
- Část složeného dotazu (například pomocí příkazu UNION)

Hinty jsou definované jako součást komentáře se znakem “+”, který informuje optimalizátor, že následující text v komentáři je hint. Syntaxe definování hintu může mít dvě podoby, a to:

```
{DELETE|INSERT|SELECT|UPDATE} /*+ hint [text] [hint[text]]... */

{DELETE|INSERT|SELECT|UPDATE} --+ hint [text] [hint[text]]...
```

Klíčová slova DELETE, INSERT, SELECT nebo UPDATE začínají blok dotazu a komentáře, které obsahují hinty mohou následovat jen po těchto příkazech. V případě, že hint je definovaný nesprávně (nenásleduje jedno z uvedených klíčových slov, chyba syntaxe nebo uvedení hinty jsou konfliktní), pak Oracle hint ignoruje, ale nevrací žádnou chybu.

3.6.1 Definování kompletní množiny hintů v dotazu

V některých případech je potřeba definovat každou operaci v dotaze pomocí hintu, aby bylo dosaženo optimálního zpracování dotazu. V případě optimalizace komplexního dotazu, který obsahuje více tabulek a je uveden jen hint INDEX pro některou z tabulek, optimalizátor doplní ostatní operace podle svého. Může se stát, že hintem definovaný index vůbec nebude zohledněn, jelikož může být nepoužitelný v kombinaci s metodou spojení dané tabulky již zvolil optimalizátor. [5]

V následujícím dotaze je pomocí hintu definováno pořadí tabulek, v kterém se budou spojovat, použité metody spojení i přístupové metody k tabulkám:

```
SELECT /*+ ORDERED INDEX (b, jl_br_balances_n1) USE_NL (j b)
           USE_NL (glcc glf) USE_MERGE (gp gsb) */
 b.application_id ,
 b.set_of_books_id ,
 b.personnel_id,
 p.vendor_id Personnel,
 p.segment1 PersonnelNumber,
 p.vendor_name Name
FROM jl_br_journals j,
     jl_br_balances b,
     gl_code_combinations glcc,
     fnd_flex_values_vl glf,
     gl_periods gp,
     gl_sets_of_books gsb,
     po_vendors p
WHERE ...
```

3.6.2 Rozdělení hintů

Hinty pro ovlivnění optimalizátoru lze rozdělit do těchto skupin:

- Hinty pro definování přístupu a cíle optimalizace
- Hinty pro definování metody přístupu k tabulce
- Hinty pro transformaci dotazu
- Hinty pro definování pořadí spojení tabulek
- Hinty pro definování metod spojení tabulek
- Hinty pro paralelní vykonání dotazu

3.6.3 Hinty pro definování přístupu a cíle optimalizace

Hinty z této kategorie umožňují zvolit mezi použitím rule-based optimalizací a cost-based optimalizací. Při použití cost-based optimalizace dále umožňují výběr způsobu optimalizace dotazu pro největší propustnost nebo nejrychlejší dobu odezvy. Konkrétní hinty v této kategorii jsou tyto:

- ALL_ROWS
- FIRST_ROWS (n)
- CHOOSE
- RULE

Jakmile je v SQL dotazu uvedený konkrétní přístup a cíl optimalizace, tak ho optimalizátor použije i v případě, že k daným objektům neexistují statistiky. Chybějící statistiky se nahradí předdefinovanými standardními hodnotami, což může mít za následek neefektivní optimalizaci.

Příklad použití hintu pro definování přístupu a cíle optimalizace:

```
SELECT /*+ FIRST_ROWS(10) */ empno, ename, sal, job
FROM emp
WHERE deptno = 20;
```

V tomto příkladě má každé oddělení více zaměstnanců. Uživatel pomocí hintu definuje požadavek, že chce prvních 10 zaměstnanců získat co nejrychleji.

3.6.4 Hinty pro definování metody přístupu k tabulce

Každý z hintů v této kategorii definuje metodu přístupu k tabulce a mohou být následující:

- FULL - přinutí optimalizátor použit full table prohledávání
- ROWID - prohledávání tabulky přes rowid
- INDEX - prohledávání tabulky prostřednictvím indexu
- INDEX_ASC - indexové prohledávání, v případě index range prohledávání jsou indexové hodnoty prohledávány vzestupně
- INDEX_COMBINE - určuje přístup k tabulce použitím bitmapových indexů.
- INDEX_DESC - obdoba INDEX_ASC, prohledávání indexových hodnot je však sestupné
- INDEX_FFS - určuje fast full indexové prohledávání
- NO_INDEX - vylučuje použití jednoho nebo více indexů

Je-li definována, pomocí hintu, konkrétní metoda přístupu k tabulce, tak optimalizátor tuto metodu použije jen v případě, že je tento způsob přistupu možné zvolit (v závislosti např. na existenci indexu nebo na syntaktických konstrukcích SQL dotazu). V opačném případě je hint ignorován.

Následující příklady ukazují použití hintů FULL a INDEX:

```
SELECT /*+ FULL(A) nepoužije index na sloupci accno */ accno, bal
FROM accounts a
WHERE accno = 7086854;
```

V příkladě použití hintu FULL optimalizátor vykoná full table prohledávání tabulky accounts i v případě, že existuje index na sloupci accno, který by mohl být použitý.

```
SELECT /*+ INDEX(patients sex_index) použij index sex_index protože
        v tabulce patients je jen málo mužských pacientù*/
     name, height, weight
```

```
FROM patients
WHERE sex = 'm';
```

V příkladě použití hintu INDEX je optimalizátoru určeno prohledávání přes index, jelikož je známo, že v tabulce je jen malé množství pacientů mužského pohlaví.

3.6.5 Hinty pro transformaci dotazu

Následující hinty určují použití různých metod transformací dotazu:

- USE_CONCAT - obsahuje-li dotaz podmínky spojené operátorem OR, pak je dotaz přepsán na složený dotaz s použitím UNION ALL operátoru.
- NO_EXPAND - zabraňuje optimalizátoru provést přepsání dotazu s OR podmínkami.
- REWRITE - optimalizátor se pokusí přepsat dotaz pomocí materializovaných pohledů
- MERGE - tento hint dovoluje spojení pohledu s dotazem
- STAR_TRANSFORMATION
- FACT
- NO_FACT

Následující příkaz ukazuje použití hintu USE_CONCAT:

```
SELECT /*+USE_CONCAT*/ *
FROM emp
WHERE empno > 50 OR sal < 50000;
```

Tento dotaz je použitím hintu USE_CONCAT transformován do následující podoby:

```
SELECT * FROM emp
WHERE empno > 50
UNION ALL
SELECT * FROM emp
WHERE sal < 50000;
```

3.6.6 Hinty pro definování pořadí spojení tabulek

Tyto hinty ej možné použít na definování pořadí, ve kterém se budou spojovat tabulky k nimž je přistupováno v SQL dotazu:

- ORDERED - tento hint přinutí optimalizátor, aby spojoval tabulky v takovém pořadí, v kterém jsou uvedeny v klauzuli FROM.
- STAR - definuje použití star plánu vykonání. Největší tabulka je zpracována jako poslední.

3.6.7 Hinty pro definování metod spojení tabulek

Následující hinty se používají na výběr konkrétní metody spojení dvou tabulek v dotazu:

- USE_NL - tabulky budou spojeny metodou nested loop
- USE_MERGE - tabulky budou spojeny metodou sort merge
- USE_HASH - určuje spojení metodou hash spojení
- DRIVING_SITE
- LEADING - definuje vedoucí tabulku při spojování (vnější tabulku)
- HASH_AJ, MERGE_AJ a NL_AJ - tyto hinty definují operaci anti-join
- HASH_SJ, MERGE_SJ a NL_SJ - tyto hinty definují operaci semi-join

Příklad dotazu s hintem USE_NL a HASH_SJ:

```
SELECT /*+ ORDERED USE_NL(customers) získa rychleji první řádky*/
     accounts.balance, customers.last_name, customers.first_name
FROM accounts, customers
WHERE accounts.custno = customers.custno;
```

V tomto příkladě se tabulka customers spojuje jako vnitřní tabulka spojení. Metoda nested loop umí vrátit první řádky už po načtení jednoho řádku z vnější tabulky a nalezení ekvivalentního řádku ve vnitřní tabulce. oproti tomu např. při sort-merge spojení se nejprve prohledávají a třídí obě spojované tabulky a až potom je možné vygenerovat první řádky výsledku.

```
SELECT * FROM dept
WHERE EXISTS (SELECT /*+HASH_SJ*/ *
              FROM emp
              WHERE emp.deptno = dept.deptno
              AND sal > 200000);
```

Použití hintu HASH_SJ přinutí optimalizátor přepsat poddotaz na speciální spojení mezi oběma tabulkami v dotaze. Toto speciální spojení funguje tak, že i když existuje v poddotaze více řádků, které jsou ekvivalentní s řádkem hlavního dotazu, do výsledku se generuje jen jeden řádek.

Poddotaz je možné řepsat na semi-join jen v některých případech a existují jistá omezení:

- v poddotaze může být jen jedna tabulka
- vnější blok dotazu nesmí být sám poddotazem
- poddotaz musí být spojený s vnějším dotazem pomocí rovnosti
- v poddotaze nesmí být použit příkaz GROUP BY, CONNECT BY nebo ROWNUM

3.6.8 Hinty pro paralelní vykonání dotazu

Pro ovlivnění paralelního vykonání dotazu slouží tyto hinty:

- PARALLEL - definuje paralelizmus a počet souběžných procesů, které budou dotaz vykonávat
- NOPARALEL - zakazuje paralelní operace nad tabulkou

4 Aplikace optimalizačních metod

Obecně lze konstatovat, že optimalizace v obecném pojetí není jen a pouze dosažení rychlejšího zpracování dané úlohy, ale dosažení nejlepšího řešení ze všech možných z pohledu celého komplexu nároků na danou úlohu a její vlastnosti. Z pohledu definic oboru operačního výzkumu má tato úloha charakter hledání optima ve vícerozměrné účelové funkci.

4.1 Aspekty optimalizace

Dále jsou uvedeny vybrané aspekty, které lze při optimalizaci řešení dané úlohy uvažovat. Naprostá většina reálných úloh podléhá jejich kombinaci.

Pro praktickou část byly vybrány některé z nich s cílem demonstrovat, že nejen úprava SQL dotazu samotného, ale i kontextu daného dotazu a režim zpracování může mít dramatický dopad na dosahované výsledky.

4.1.1 Rychlost

Typickým parametrem řešení je doba potřebná na dokončení zpracování. V případě, že máme možnost podřídit veškeré další parametry zpracování tomuto kritérii, tak se lze soustředit jen a pouze na zkrácení času zpracování. To ale v reálném prostředí lze jen zčásti, neb prakticky nikdy nemůžeme opomenout například čerpání zdrojů systému, jeho limity a důsledky jejich překročení.

Jako vhodným příkladem může být úvaha nad hash joinem, který bude proveden plně v operační paměti databázového serveru. Je zřejmé, že pro úlohy nad entitami v řádech do jednotek GB lze takovou metodu řešení uvažovat s naprosto vynikajícími výsledky z pohledu doby zpracování. Nicméně pro entity, jejichž velikost bude už jen o dva řády větší, to nebude reálné nebo za extrémní cenu.

Pokud se vydáme při řešení v praktickém prostředí touto cestou, je třeba mít odpovědi na otázky týkající se vývoje velikosti zpracovávaných entit v čase a případné kolize se stejně masivními úlohami zpracovávanými ve stejné době, případně jejich řízení, přidělování

priorit. Cena za toto řešení v podobě pracnosti, které má být skutečně robustní bude v běžném prostředí konvenční aplikace velmi vysoká.

Tento aspekt bude hlavní oblastí praktických testů. Bude demonstrován různý přístup k datům za použití indexů resp. indexů různého typu a dopady hintování, kde například nested loop bude zaměněn hash joinem nebo obráceně.

4.1.2 Charakter zpracování

Na celou řadu úloh jsou kladeny nároky nejen co do zpracování definované úlohy nad definovaným vstupem, ale je požadován také určitý způsob zpracování například s ohledem na možný průběžný monitoring ze strany uživatelů nebo operátorů.

Tyto nároky bývají menší u dávkových úloh, kde zde připadá v úvahu například zápis po každých n-iteracích do protokolu zpracování. U interaktivních úloh bývají nároky větší například v podobě požadavku na zobrazení průběhu zpracování do přírůstkového pruhu, který se měnit n-krát za definovaný časový úsek s doplňujícím požadavkem na zobrazení procenta dokončenosti úlohy, spotřebovaného času a odhadu doby zbývající. Dále bývá definován požadavek na možnost uživatelského přerušení úlohy a dle povahy úlohy možnosti jejího dodatečného odloženého dokončení nebo navrácení zpracování do výchozího stavu opět s možností průběžného monitorování.

Z výše uvedeného jasně vyplývá, že jakkoli může být zpracování realizováno jedním SQL dotazem, není dosažení takových parametrů zpracování triviální otázkou jeho optimalizace. Naopak může být výhodné nahradit optimální hash join podstatně horším nested loopem (s vyšší cenou exekučního plánu) který ale umožní lepší iterační monitoring.

Protože těžiště tohoto aspektu leží především v uživatelsky definované části zadání úlohy nebude v teoretické části předmětem žádné demonstrace.

4.1.3 Heuristická znalost o datech

Přes sofistikované metody plánování metody zpracování dotazu může mít analytik nebo vývojář určité heuristické znalosti o datech, které uplatní v podobě hintů upravujících metody přístupu. Může se jednat o volby vhodnějších indexů než těch, které určí optimalizátor nebo obecně jiný způsob zpracování dat především s cílem minimalizovat

počet iterací a objem analyzovaných bloků databáze. Typicky vývojář předchází plnému prohledávání tabulek tam, kde lze využít indexů.

4.1.4 Transakční vlastnosti úloh

Tento aspekt je zpravidla určen požadavkem na dosažení konzistence dat a vychází z definice relací v datovém modelu. Hovoříme zde o zajištění konzistence na elementární úrovni z pohledu obchodního významu zpracovávaných dat s tím, že úloha může být optimalizována v tom smyslu, že z pohledu zpracování může být výhodné sdružení elementárních obchodních transakcí do jedné větší technické.

Tento pohled na optimalizaci bude také předmětem základního testu v praktické části, kdy úloha pro vytvoření testovací báze dat bude testována v režimu elementární transakce a v režimu úplné transakce. Z pohledu optimalizace kódu se jedná o optimalizaci PL/SQL kódu, který obsahuje různý kontext pro základní SQL operace.

Z transakčního pohledu lze úlohy dělit do těchto kategorií:

Transakční režimy

- Bez transakcí
- Transakce na úrovni iterace
- Plně transakční zpracování
- Hypertransakce (řízené iterační zpracování na úrovni aplikace, nikoli DB stroje, aplikace si udržuje informaci o stavu zpracování úlohy, úlohu lze vždy opakovaně spustit a ta sama rozhodne jak a z kterého místa pokračovat – pouze velmi omezené možnosti využití)

4.1.5 Režim zpracování úlohy

Prostředí, ve kterém bude úloha zpracovávána není určen žádnou konstrukcí PL/SQL nebo samotného SQL příkazu, ale rozhodnutím, zda bude úloha zpracovávána na klientské straně (režim Klient/Server) nebo pouze na straně serveru (režim Host Based).

Lze uvažovat i kombinovaný režim, kdy úloha je realizována jako procedurální, nicméně je umístěna do databáze jako vložená procedura, funkce nebo celý balík (package) a na klienta zasílá pouze informace pro monitoring.

Pro zpracování samotného SQL příkazu s malým objemem dat na výstupu je to prakticky jedno. V takovém případě představuje přenos samotného SQL dotazu na server prakticky neměřitelné čerpání systémových zdrojů, a pokud je výsledkem malá množina dat, bude tomu tak při zaslání výsledku zpracování zpět.

Zásadně se ale situace změní, když je výsledkem zpracování dotazu velká množina dat, kterou je třeba ze serveru přenést na klienta nebo se nejedná o samostatný SQL dotaz, ale o strukturovaný PL/SQL kód, který generuje velké množství elementárních SQL dotazů a masivně se serverem komunikuje.

V praktické části využijeme úlohy z předchozího příkladu v optimalizované podobě, kterou budeme provozovat jak v režimu Client/Server, tak v režimu Host Based a očekáváme, že výsledky optimalizace dosažené úpravou transakčního zpracování budou dále významně posunuty volbou vhodného režimu, kterým je v tomto případě režim na serveru.

4.2 Vybraná báze dat

Pro následující praktickou část bylo zvoleno konvenční relační schéma typické pro informační systémy ekonomického a organizačního charakteru. Cílem bylo zajištění možnosti maximálního zobecnění závěrů a zajištění jejich široké platnosti a použitelnost pro nejpoužívanější typ uspořádání dat.

Záměrně nebyla zvolena báze dat se specifickými vlastnosti jako například star schéma používané při implementaci datových skladů nebo jiné uspořádání specifické pro informační systémy ryze technického charakteru nebo jiné IS se speciálními vlastnostmi a zvláštním určením.

Pro tyto systémy platí, že lze jít sice při optimalizaci k samé hranici možnosti systému řízení správy dat, použít extrémní metody řešení úloh, ale zároveň také platí, že jakkoli mají takové konstrukce vysokou účinnost, bývá jejich platnost omezena na velmi speciální podmínky a obecné použití takových závěrů je prakticky nemožné. Často platí, že za těchto

podmínek optimalizované dotazy se v jiném prostředí nejen že nechovají optimálně, ale mohou v krajním případě zcela selhat nebo poskytnout výrazně horší výsledky než konvenční konstrukce.

Vybrané konvenční relační schéma má jednu entitu, která je tvořena třemi tabulkami. Pro názornost byla vybrána hypotetická entita ***EFaktury*** s tím, že ji tvoří tabulka ***ZP_DOKUMENT***, ***ZP_DOKLAD*** a ***ZP_RADEK***. Tabulka ***ZP_DOKLAD*** obsahuje hlavičky faktur a tabulka ***ZP_RADEK*** jejich řádky. Tabulka ***ZP_DOKUMENT*** rozšiřuje informace hlavičky faktury a je k dispozici jen pro některé záznamy tabulky ***ZP_DOKLAD.*** Znamená to, že ne všechny faktury mají tuto rozšiřující informaci, jako doprovodný text, odkaz na fotokopii originálu, atd.

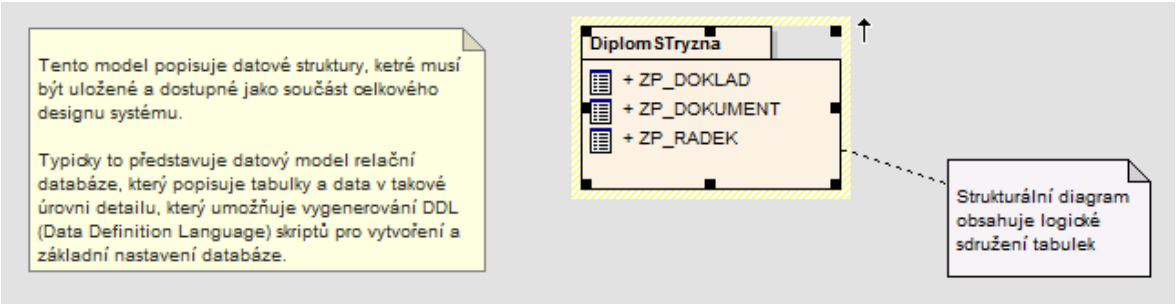
Pro účely testů byly záznamy těchto tabulek generovány speciálním PL/SQL skriptem s náhodným algoritmem. Samotný tento skript byl předmětem optimalizace z pohledu transakčního zpracování (Single Task Transaction) a zároveň z pohledu režimu zpracování (Host Based). Po této optimalizaci byla doba generování celkově cca 26 miliónů záznamů zkrácena z původních 8,56 hodin na 1,84 hodiny.

Počty záznamů v jednotlivých tabulkách jsou následující:

```
ZP_DOKUMENT:         69 245
ZP_DOKLAD:        4 000 000
ZP_RADEK:        21 996 994
```

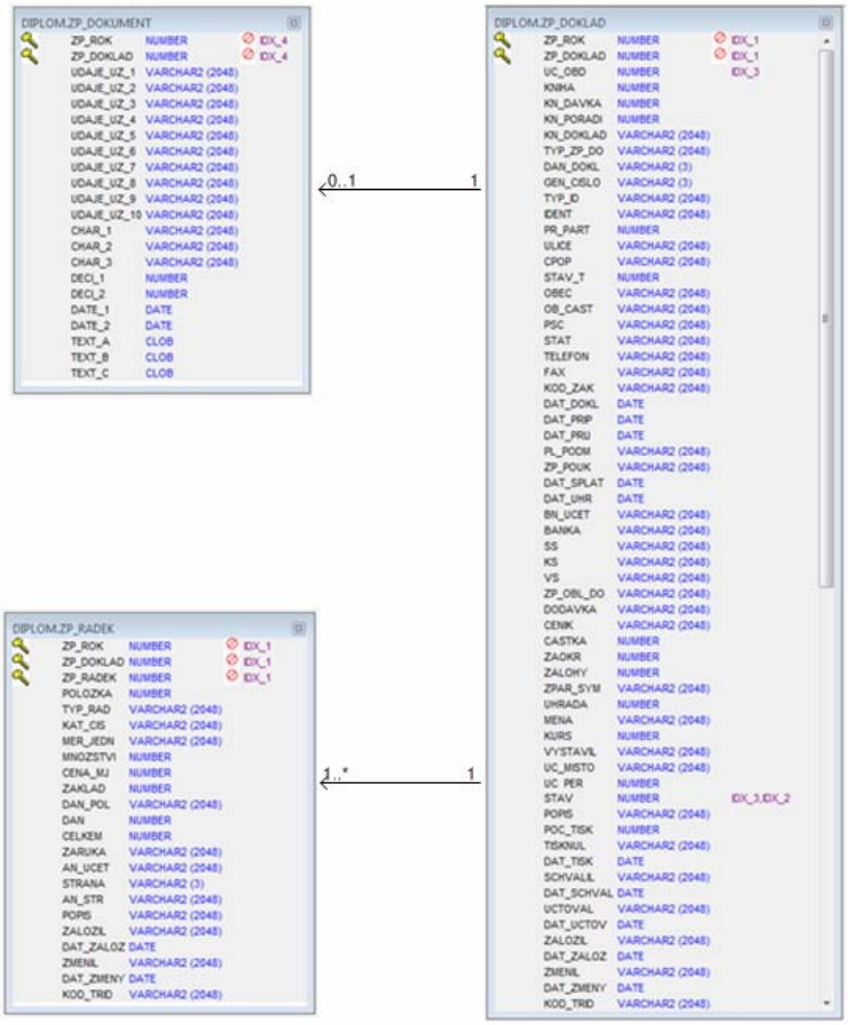
Předpokládáme tedy, že se jedná o informační systém velké nadnárodní společnosti, který obsahuje milióny faktur a desítky miliónů řádků těchto faktur. Data nejsou ale plně konzistentní, část nekonzistencí budeme úmyslně vytvářet v rámci testů a stejně tak budeme vytvářet různé indexy různých typů za účelem demonstrace dopadů do výsledků za použití různých SQL konstrukcí.

V tabulkách byly použity standardní datové typy bez typů binárních, uživatelských nebo jinak pro DBS Oracle specifických. Následuje obrázek 4.1 se základní definicí uvedených tabulek dle UML standardů:



Obr.4.1 – Základní UML schéma zvoleného modelu

Na následujícím obrázku 4.2 je uvedeno relační schéma vybraných tabulek.



Obr.4.2 – Relační schéma datového modelu

Relace zde představují šipky s vyznačením typu relace. V uvedeném schématu jsou tyto typy relací

0..1 Vztah mezi dokladem a dokumentem vyjadřující, že k dokladu neexistuje žádný nebo právě jeden dokument.

1..* Vztah mezi dokladem a řádky dokladu vyjadřující, že k dokladu vždy existuje jeden nebo více řádků. Jedná se o typickou vazbu 1:N

1 Vztah vyjadřující, že musí existovat právě jeden záznam, zpravidla nadřazené tabulky.

4.3 Materializované pohledy

Databáze Oracle obsahuje speciální objekt, který za určitých okolností může optimalizátor využít při vytváření exekučního plánu. Jedná se o materializované pohledy na fyzicky existující data, které na rozdíl od standardních pohledů, které představují pouze logické uspořádání fyzicky existujících dat a jejich interpretaci, skutečně obsahují záznamy v databázi, které takový pohled reprezentují. Důsledkem toho je, že takové pohledy lze indexovat a považovat je ze standardní zdroj dat tak jako běžné tabulky.

Optimalizátor může v rámci analýzy zpracování SQL dotazu usoudit, že výsledek komplexního zpracování je zcela nebo z části již předmětem materializovaného pohledu a místo zpracování dat primárních tabulek může použít obsah materializovaných pohledů a SQL dotaz v krajním případě zcela přepracovat. Jakkoli mocná může být tato technika má své úskalí a lze ji použít pouze za určitých okolností. Jak teoretické, tak praktické zkoumání této techniky však přesahuje rámec diplomové práce a tak zůstaneme pouze o tohoto obecného konstatování dalších možností optimalizace zpracování.

4.4 Praktické testy

4.4.1 Úpravy SQL dotazu

Hlavní část praktických testů představuje obecná úprava SQL dotazu. Prakticky máme dvě možnosti, jak SQL dotaz upravovat. První možností je změnit samotnou konstrukci SQL dotazu, což má výrazně obecnější platnost a poznatky lze zpravidla aplikovat i pro jiné databázové stroje než pouze pro DBS Oracle.

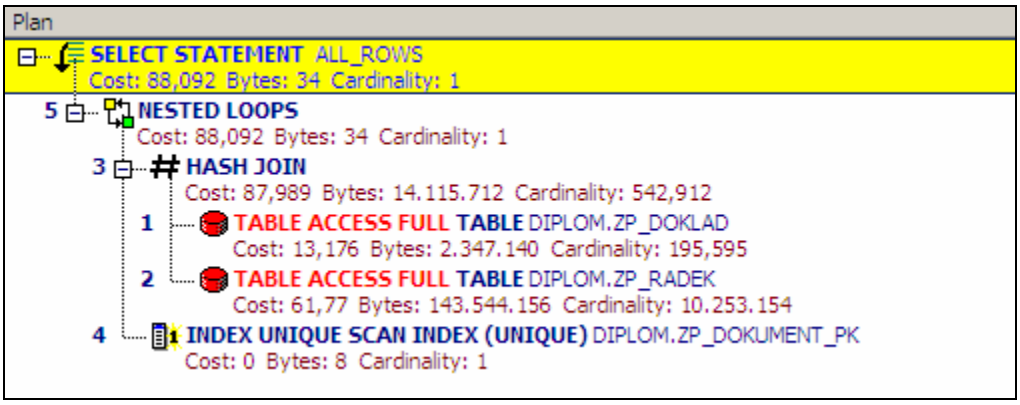
Druhou možností je rozšíření dotazu o speciální konstrukce upravující chování optimalizátoru, jejichž použití vede na změnu exekučního plánu. Tyto jazykové konstrukce nejsou předmětem žádné verze SQL normy a jakkoli jejich použití umožňuje prakticky každý vyspělý databázový stroj, je jejich zpracování závislé na konkrétní platformě. Tyto speciální konstrukce jsou označovány jako hinty.

Vzhledem k tomu, že se cíleně pohybujeme v prostředí DBS Oracle, bude demonstrace obsahovat příklady užití typických hintů tak jak jsou svou četností zastoupeny v typické komerční aplikaci. Nejprve si ukážeme zpracování dotazu bez použití hintů, kdy optimalizátor vybere konvenční metodu přístupu k datům. Konstrukci pak upravíme o použití vybraného indexu, který vzhledem k neaktuálním statistikám optimalizátor neuvažoval, v dalších variantách budeme upravovat typy spojení.

Základní dotaz bez hintů

```
SELECT zpd.zp_rok,
       zpd.zp_doklad,
       zpr.cena_mj
  FROM zp_radek zpr,
       zp_doklad zpd,
       zp_dokument doc
 WHERE zpr.zp_doklad = zpd.zp_doklad
   AND zpr.zp_rok = zpd.zp_rok
   AND 5000 < zpr.celkem
   AND 800 > zpr.cena_mj
   AND zpd.dat_prip IS NOT NULL
   AND zpd.dat_prip > TO_DATE('1.1.2004', 'dd.mm.yyyy')
   AND zpd.zp_rok = doc.zp_rok
```

```
   AND zpd.zp_doklad = doc.zp_doklad
   AND doc.zp_doklad = zpr.zp_doklad
   AND doc.zp_rok = zpr.zp_rok
```



Obr.5.1 – Exekuční plán základního dotazu

Z exekučního plánu je zřejmé, že optimalizátor se rozhodl přistupovat k tabulkám zp_doklad a zp_radek metodou full table prohledávání a spojil je pomocí hash spojení. V dalším kroku provedl indexové prohledávání tabulky zp_dokument a získané řádky spojil s řádky získanými v předchozích krocích.

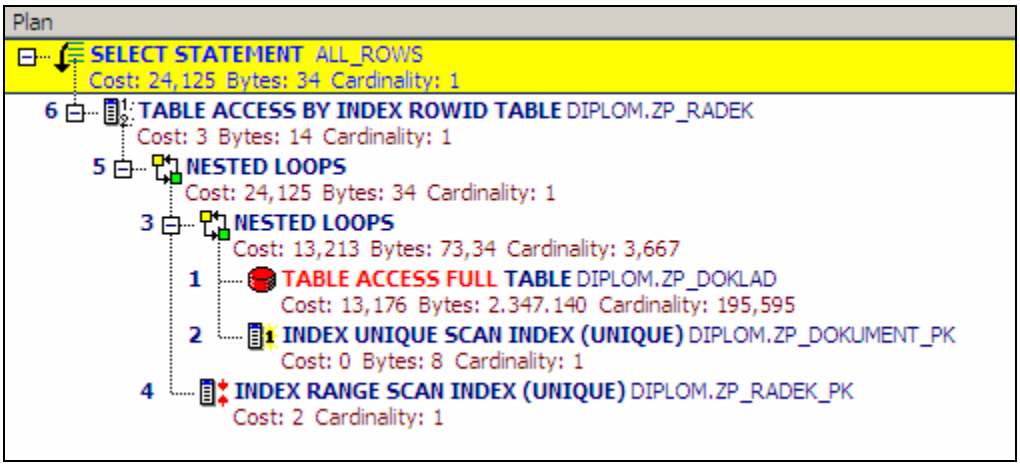
První alternativa (doplnění indexu, použití hintu INDEX)

První alternativou je dotaz rozšířený o hint INDEX, čímž je dosaženo toho, že tabulka zp_dokument bude prohledávána pomocí stanoveného indexu.

```
SELECT /*+INDEX(doc zp_dokument_pk)*/
       zpd.zp_rok,
       zpd.zp_doklad,
       zpr.cena_mj
  FROM zp_radek zpr,
       zp_doklad zpd,
       zp_dokument doc
 WHERE zpr.zp_doklad = zpd.zp_doklad
   AND zpr.zp_rok = zpd.zp_rok
   AND 5000 < zpr.celkem
   AND 800 > zpr.cena_mj
   AND zpd.dat_prip IS NOT NULL
   AND zpd.dat_prip > TO_DATE('1.1.2004', 'dd.mm.yyyy')
```

```
   AND zpd.zp_rok = doc.zp_rok
   AND zpd.zp_doklad = doc.zp_doklad
   AND doc.zp_doklad = zpr.zp_doklad
   AND doc.zp_rok = zpr.zp_rok
```

Plán vykonání dotazu, na obrázku 5.2, znázorňuje průběh vykonání s použitím hintu INDEX. Oproti první variantě, je zde vidět rozdíl v přístupu k tabulce zp_dokument, kdy optimalizátor byl prostřednictvím hintu přinucen k indexovému prohledávání a rozdíl v metodě spojení tabulek zp_doklad a zp_dokument, kdy při použití indexu se optimalizátor rozhodl spojovat tabulky metodou nested loop.



Obr. 5.2 – Exekuční plán dotazu s hintem INDEX

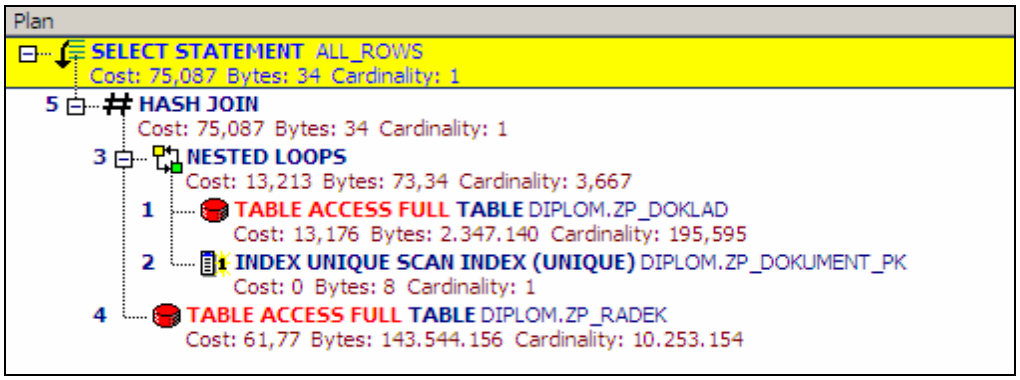
Druhá alternativa (použití kombinace hintu INDEX a USE_HASH)

Druhá alternativa dotazu, rozšiřuje základní dotaz o kombinaci hintů INDEX a USE_HASH. Zde je stejně jako v předchozí alternativě optimalizátor donucen k indexovému prohledávání tabulky zp_dokument. Navíc je zde ale určena metoda spojení řádek získaných spojením tabulek zp_dokument a zp_doklad s tabulkou zp_radek.

```
SELECT /*+INDEX(doc zp_dokument_pk) USE_HASH (zpr)*/
       zpd.zp_rok,
       zpd.zp_doklad,
       zpr.cena_mj
  FROM zp_radek zpr,
       zp_doklad zpd,
       zp_dokument doc
```

```
WHERE zpr.zp_doklad = zpd.zp_doklad
  AND zpr.zp_rok = zpd.zp_rok
  AND 5000 < zpr.celkem
  AND 800 > zpr.cena_mj
  AND zpd.dat_prip IS NOT NULL
  AND zpd.dat_prip > TO_DATE('1.1.2004', 'dd.mm.yyyy')
  AND zpd.zp_rok = doc.zp_rok
  AND zpd.zp_doklad = doc.zp_doklad
  AND doc.zp_doklad = zpr.zp_doklad
  AND doc.zp_rok = zpr.zp_rok
```

Plán vykonání výše uvedeného dotazu je znázorněn na obrázku 5.3. Je zde videt zřejmý rozdíl oproti plánu vykonání dotazu z předešlé varianty, kdy byl použit pouze hint INDEX. Hint USE_HASH skutečně donutil optimalizátor k použití hash spojení nad výslednými tabulkami a zároveň došlo ke změně v metodě přístupu k tabulce zp_radek. Metodou přístupu k této tabulce je nyní full table prohledávání.



Obr. 5.3 – Exekuční plán dotazu s kombinací hintů INDEX a USE_HASH

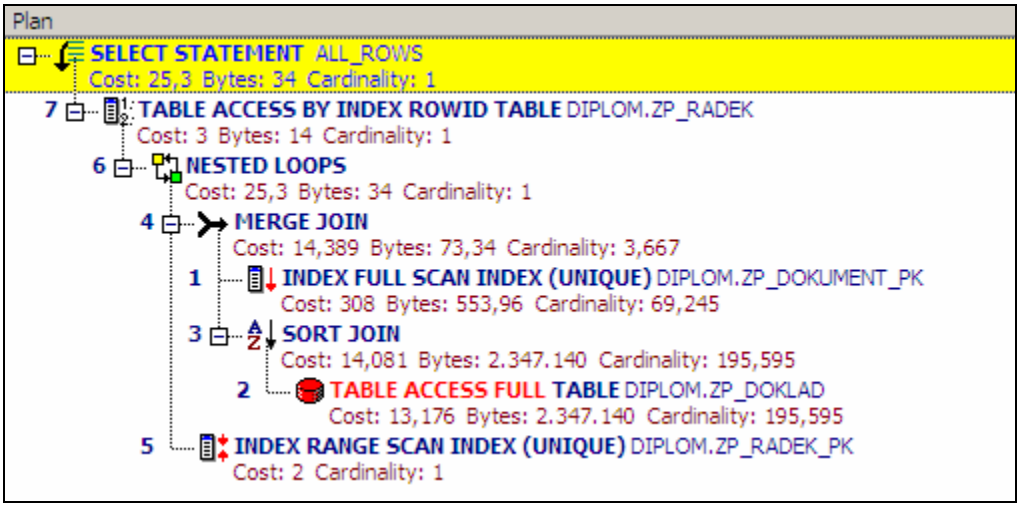
Třetí alternativa (změna metody spojení, použití hintu USE_MERGE)

Poslední třetí zkoumanou alternativou ovlivnění optimalizátoru je jeho instruování k použití SORT_MERGE metody spojení tabulek zp_doklad a zp_dokument.

```
SELECT /*+ USE_MERGE(doc,zpd) */ zpd.zp_rok,
       zpd.zp_doklad,
       zpr.cena_mj
  FROM zp_radek zpr,
       zp_doklad zpd,
```

```
      zp_dokument doc
WHERE zpr.zp_doklad = zpd.zp_doklad
  AND zpr.zp_rok = zpd.zp_rok
  AND 5000 < zpr.celkem
  AND 800 > zpr.cena_mj
  AND zpd.dat_prip IS NOT NULL
  AND zpd.dat_prip > TO_DATE('1.1.2004', 'dd.mm.yyyy')
  AND zpd.zp_rok = doc.zp_rok
  AND zpd.zp_doklad = doc.zp_doklad
  AND doc.zp_doklad = zpr.zp_doklad
  AND doc.zp_rok = zpr.zp_rok
```

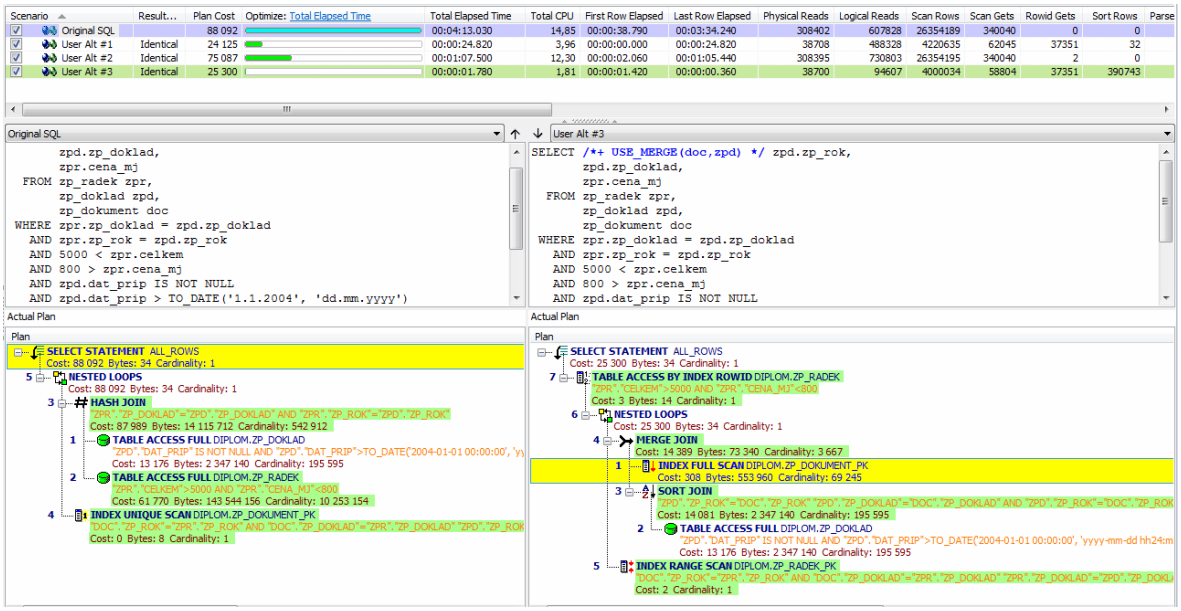
Na obr. 5.4, je vidět, že došlo ke změně v metodě spojení tabulek zp_doklad a zp_dokument a metodou spojení je nyní SORT MERGE spojení.



Obr. 5.4 – Exekuční plán dotazu s hintem USE_MERGE

Souhrnné výsledky jsou uvedeny v následující tabulce, která byla vytvořena speciálním optimalizačním nástrojem ***Quest SQL Optimizer for Oracle***. V tomto nástroji lze mimo jiné přehledně porovnat exekuční plány alternativních konstrukcí, ale i celou řadu dalších charakteristik, které jdou daleko za rámec pouhého porovnání ceny vybraného exekučního plánu.

Z uvedeného přehledu na obrázku 5.5 je patrné, že nejvýkonnější je třetí alternativa a to přesto, že cena exekučního plánu není nejnižší. Tento fakt zároveň dokládá, že cena exekučního plánu je pouze teoretickou hodnotou, která má sice vysokou korelaci k reálným výsledkům, ale přesné exaktní měření zcela nenahrazuje.



Obr. 5.5 – Vyhodnocení alternativ SQL dotazu

4.4.2 Demonstrace použití bitmapového indexu a analýza hranice užitku

Další optimalizační techniku, kterou budeme uvažovat je nikoli určení indexu jako takového, ale výběr vhodného typu indexu. Předmětem zkoumání bude bitmapový index a srovnání jeho výkonnosti s konvenčním B-tree indexem.

Pro bitmapové indexy je ideálním kandidátem pole s nízkou kardinalitou. V našem případě je vhodným kandidátem stav faktury, který nabývá pouze několika málo vybraných hodnot, které reprezentují uzly stavového diagramu. V rámci analýzy se také zaměříme na zkoumání hranice užitku, tj. na konkrétním případě zkusíme experimentálně najít hranici, kdy výhody bitmapového indexu díky vysoké kardinalitě převáží výkon indexu konvenčního.

Při testech bylo nezbytné eliminovat vliv technologických technik optimalizace na úrovni diskového pole, operačního systému a přístupu databázového stroje ke zdrojům systému. Testy bylo nutné provádět ve stabilním a iteračně nezávislém prostředí. Ověření tohoto

předpokladu jsme provedli tak, že stejné konstrukce vykazovali stejnou výkonnost bez ohledu na počtu opakování.

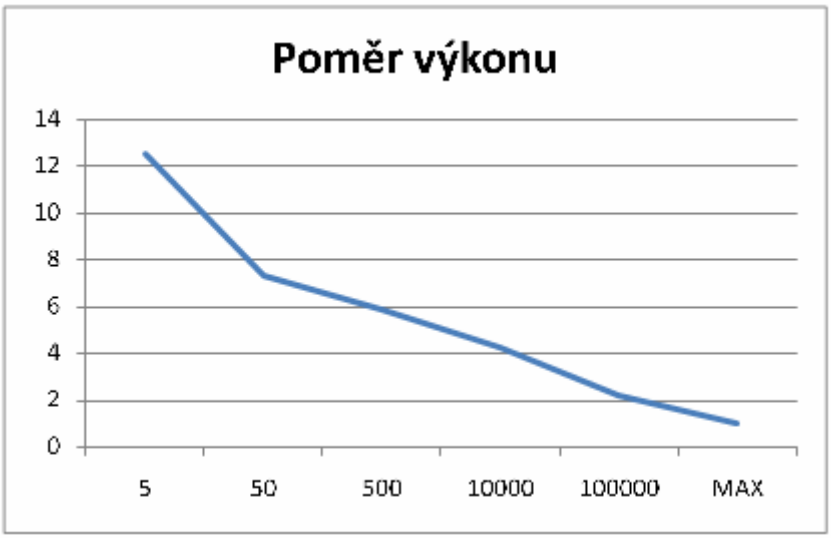
Kardinalita pole byla upravena v pásmech 5, 50, 500, 10000, 100000 a maximální počet hodnot. Časy zpracování v následující tabulce 4.1 představují průměrné hodnoty z šesti za sebou jdoucích posledních měření z celkového počtu 12 měření dané kardinality.

Kardinalita pole	BitMapIndex čas zpracování	BTreeIndex čas zpracování	Poměr výkonu
5	61 171	765 525	12,51450851
50	132 705	971 383	7,319867375
500	111 509	656 494	5,887364862
10000	78 629	336 344	4,277607499
100000	238 223	521 615	2,189603874
MAX	442 376	440 478	0,995709532

Tab.4.1 – časy zpracování v závislosti na typu indexu

S ohledem na úpravy testovacího skriptu má hlavní vypovídací schopnost poslední sloupec, který obsahuje poměr výkonu zpracování při použití bitmapového a konvenčního indexu. Dle očekávání se zvyšující se kardinalitou klesá výkon bitmapového indexu až k výkonu indexu konvenčního a pokud je kardinalita vysoká, je konvenční index výhodnější.

Je ale pravdou, že konvenční index převáží až s kardinalitou blíží se počtu záznamů a výsledky měření tedy korespondují s teoretickým doporučení využít konvenčních B-tree indexů u unikátních klíčů. Graf 4.1 vyjadřuje klesající poměr užitku ve vztahu ke vzrůstající kardinalitě.



Graf.4.1 – poměr užitku indexu ve vztahu ke kardinalitě

Testy byly prováděny za použití následujícího PL/SQL skriptu, kde hranice smyčky pro proměnnou stavy byla upravena pro každé zkoumané pásmo. To je také důvodem, proč jsou jednotlivá měření přímo nesouměřitelná a pro grafické vyjádření je bylo třeba indexovat.

```
DECLARE
   stavy PLS_INTEGER;
   num PLS_INTEGER;
BEGIN
   -- Sekce pro bitmapový index
   DBMS_OUTPUT.put_line ('BEG/BTIX:' || SYSTIMESTAMP);

   FOR stavy IN 100000 .. 110000
   LOOP
      SELECT /*+ index(zp_doklad btix_stav) */
             COUNT (*)
        INTO num
        FROM zp_doklad
       WHERE stav = stavy;
   END LOOP;

   DBMS_OUTPUT.put_line ('END/BTIX:' || SYSTIMESTAMP);

   -- Sekce pro konvenční index
   DBMS_OUTPUT.put_line ('BEG/IDX:' || SYSTIMESTAMP);
```

```
    FOR stavy IN 200000 .. 210000
    LOOP
       SELECT /*+ index(zp_doklad idx_stav) */
              COUNT (*)
         INTO num
         FROM zp_doklad
        WHERE stav = stavy;
    END LOOP;

    DBMS_OUTPUT.put_line ('END/IDX:' || SYSTIMESTAMP);

END;
```

4.4.3 Transakční zpracování

Samostatnou oblastí optimalizace je úprava kódu z transakčního pohledu. Je třeba odlišit elementární transakce, tak jak je definuje datový model a povaha dané úlohy, což představuje zpravidla atomární úroveň, pod kterou jít konstrukce kódu nemůže, a úroveň technologickou, která je určena konkrétní implementací a může obsahovat sdružení elementárních transakcí do větších celků z důvodu optimalizace zápisu změn do databáze.

Zcela samostatnou oblastí, kterou zmíníme pouze okrajově, je řešení inverzní úlohy, kdy elementární transakce definovaná úlohou zahrnuje velký objem dat, typicky v dávkových úlohách, kdy se například vytváří souhrnné předpisy pro účtování, které vychází z většího počtu provozních změn. Pokud souhrnné zpracování všech těchto změn má být předmětem jedné entity, tak mohou být při zpracování konvenčními konstrukcemi zahrnuty všechny dílčí záznamy pod jednu transakci a jsou to technologická hlediska, která na rozdíl od předchozího případu vyžadují rozdělení na menší počet databázových transakcí jakkoli aplikačně musí úloha zůstat transakcí jedinou. Při řešení této úlohy se uplatňuje řada technik, ale jejich bližší rozbor přesahuje rámec diplomové práce a nejde ani tak o techniky optimalizace, jako o techniky proveditelnosti, konzistence a robustnosti řešení.

Dále se budeme soustředit na optimalizaci prvního typu. Princip této optimalizace vychází z jednoduché úvahy. Jakkoli je totiž možné v komplexní úloze zapsat do databáze každou jednotlivou elementární transakci zvlášť, bude asi výhodnější sdružit zápis do určitých bloků a provést zápis do databáze s menší četností. Důvodem je fakt, že zápis transakce sám o sobě představuje samostatný, dále nedělitelný počet operací databázového stroje a jako takový představuje jistou formu čerpání určitého zdroje.

Testy byly provedeny na kódu, který generuje fiktivní záznamy do vybraného datového modelu s tím, že zápis jedné faktury, jejích řádků a případně i rozšiřujícího dokumentu představuje zmiňovanou elementární transakci. Úplný výpis tohoto PL/SQL kódu je uveden v příloze. Byla zavedena konstanta, která určuje, po kolika elementárních transakcích bude blok dat zapsán do databáze, tj. proveden COMMIT.

Úloha byla nastavena tak, aby vytvořila 2x 100 000 faktur, ke každé náhodná počet řádků (od jednoho do deseti) a náhodně také záznam o dokumentu (také náhodně s pravděpodobností 1:30). Objem dat je dost velký nato, aby i s náhodnými prvky generujícího algoritmu byly naměřené hodnoty dostatečně reprezentativní pro potvrzení teoretické úvah a jejích finální zobecnění.

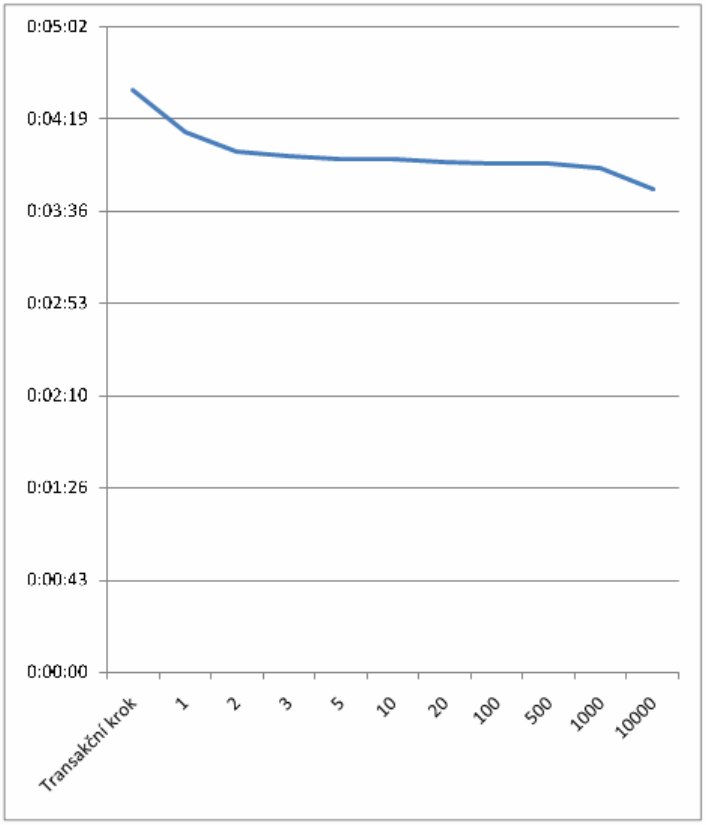
Následující tabulka 4.2 uvádí naměřené doby generování dat v závislosti na rozsahu sdružení elementárních transakcí do transakce technické.

Transakční krok	Začátek zpracování	Konec zpracování	Celkový čas
1	1:31:57	1:36:30	0:04:33
2	1:39:24	1:43:37	0:04:13
3	1:47:33	1:51:37	0:04:04
5	1:54:04	1:58:06	0:04:02
10	2:00:47	2:04:47	0:04:00
20	2:07:49	2:11:49	0:04:00
100	2:16:58	2:20:57	0:03:59
500	2:24:31	2:28:29	0:03:58
1000	2:31:45	2:35:43	0:03:58
10000	2:39:26	2:43:22	0:03:56
SingleTask	4:30:35	4:34:21	0:03:46

Tab.4.2 – časy zpracování v závislosti na transakčním kroku

Poslední řádek představuje uzavření celého zpracování do jediné transakce, kde očekáváme výsledek nejlepší. V případě našeho testovacího kódu mimo jiné i proto, že není v průběhu zpracování vyhodnocován transakční krok a je tedy uspořena i část algoritmu. Jedná se tedy o úsporu, byť minimální, plynoucí nikoli z optimalizace zápisu do databáze, ale ze změny algoritmu zpracování. S ohledem na velký počet iterací lze tedy v grafu očekávat dílčí zlom výkonnostní křivky.

Následující graf 4.2 uvádí vývoj časů zpracování v závislosti na transakčním kroku. Jak je patrné z výkonnostní křivky, je potvrzena teoretická úvaha včetně očekávaného zlomu v režimu Single Task Transaction (STT).



Graf.4.2 – vývoj časů zpracování v závislosti na transakčním kroku

Je evidentní, že očekávaný efekt se výrazně projeví již při malém počtu sdružení elementárních transakcí do jedné technické a tento poznatek lze považovat za dobré doporučení. To i přesto, že zlom na konci grafu by mohl naznačovat, že obecné optimální řešení je STT, ale z důvodu kritického čerpání systémových zdrojů to jako obecně platný závěr určit nelze.

Jako doložení této teorie byla testována úloha spuštěná v režimu Single Task Transaction v modifikované podobě, kdy vytvořila 2x 5 000 000 dokladů, 54 239 511 řádků a 174 820 dokumentů. Takto velkou transakci sice systém zvládnou, ale z monitoringu systémových zdrojů a zdrojů databázového stroje bylo zcela patrné, že na samé hranici zvládnutelnosti. Transakční oblasti databáze byly naplněny na 87,9% a jen o málo větší úloha nebo souběh

s úlohou jinou by vedl k chybě. Je zřejmé, že takový přístup doporučit nelze a v reálné aplikaci představuje riziko nestability.

4.4.4 Režim zpracování

Posledním prakticky zkoumaným aspektem bude režim běhu úlohy. Prakticky jsou možné tři modely uspořádání. Jedná se o klasický model **Client/Server**, kdy klientská aplikace komunikuje po síťové infrastruktuře se serverem, předává mu požadavky a dostává výsledky.

Druhým možným režimem je **HostBased**, kdy je úloha vykonávána přímo na serveru, kde běží také databázový stroj. Přesto, že po technické stránce, stránce protokolů a dalších je způsob komunikace úlohy a serverové části řešení shodný s režimem Client/Server, nevyužívá se síťové komunikační infrastruktury a vše se odehrává v rámci zdrojů jednoho stroje a úloha má podobu přímé komunikace procesů.

Třetím teoreticky uvažovaným modelem je **n-vrstvá** aplikační architektura, kdy klientská část komunikuje s aplikační vrstvou, s kterou má neprivilegované běžné spojení a ta zase spojení s vrstvou databázovou. Spojení aplikační vrstvy a vrstvy databázové je zpravidla realizování na páteřní komunikační infrastruktuře s privilegovaným zpracováním.

Předmětem testu byla úloha z předchozího transakčního testu, kdy optimalizovaný kód v režimu Client/Server byl přenesen přímo na server a porovnán čas zpracování zde, v režimu HostBased. Zpracování v n-vrstvé architektuře testováno nebylo. Její vybudování nebylo z kapacitních důvodů možné a především by bylo nutné úlohu jako takovou významným způsobem přepracovat.

Test samotný lze tedy považovat za doložení možnosti optimalizace využití infrastruktury respektive aplikační architektury bez změny kódu samotného a je dobrým příkladem, jak zvýšit efektivitu zpracování relativně jednoduchým zásahem který může v řadě případů předcházet riskantním a pracným změnám kódu samotného. Efekt přechodu z režimu Client/Server na režim HostBased bude tím větší, čím intenzivněji komunikuje úloha se serverem a čím větší je objem obdržených dat (tzv. result set size).

Pro test samotný byly vybrány dvě úlohy. První je označena jako komplexní SQL dotaz s analytickou funkcí poskytující jednoduchý statistický result set. Jedná se sice o masivní úlohu, která je ale jedním komplexním dotazem definována, předáme serveru a pak již jen očekáván výsledek. V tomto případě nečekáme z přechodu na režim HostBased prakticky žádný bendit. Druhou úlohou je samotné vytvoření testovací báze dat, kdy PL/SQL kód v mnoha iteracích vytváří záznamy tabulek a komunikuje se serverem velmi intenzivně. Přesto, že v tomto druhém případě není zpět ze serveru přenášen prakticky žádný result set, očekáváme naopak velký efekt přechodu do režimu HostBased. Následující tabulka 4.3 názorně dokládá teoretické předpoklady.

Typ úlohy	Režim běhu	Čas zpracování
Komplexní dotaz	Client/Server	2:33:12
	HostBased	2:28:56
Iterační PL/SQL úloha	Client/Server	4:20:11
	HostBased	1:12:32

Tab.4.3 – časy zpracování v závislosti na typu úlohy a režimu zpracování

4.5 Zobecnění

Zobecňující závěry lze rozdělit do dvou hlavních oblastí. První se týká samotné optimalizace konstrukcí SQL dotazů. Práce na optimalizaci by měly být primárně zaměřené na úpravy kódu samotného SQL dotazu a teprve v druhé fázi na specifické instrukce předávané optimalizátoru. Největší počet a maximální užitek úprav řešení je v určení správného indexu, případně indexu správného typu a na volbě nejlepšího možného typu spojení. Pokud to úloha nevyžaduje, je třeba se vyhnout plnému prohledávání tabulek a věnovat zvláštní pozornost nested loopům v exekučních plánech.

Druhá oblast zahrnuje především optimalizace datového modelu, transakčního zpracování a efektivní využití zdrojů databázového systému a provozního prostředí respektive architektury aplikace obecně. Obecně lze konstatovat, že zdroje, které nejsou nezbytně nutné, jako technické zpracování elementární úrovně transakcí nebo využití síťové komunikační infrastruktury, je vhodné nahradit jinou technikou nebo tyto zdroje čerpat efektivně.

5 Závěr

Závěrem lze konstatovat, že optimalizace úloh zpřístupnění a zpracování dat je výrazně multi-kriteriální disciplína a velmi záleží na očekávání ohledně výsledků optimalizace.

Jakkoli bylo diplomovou prací teoreticky i prakticky prokázáno, že aspektů optimalizace a metod řešení existuje celá řada, některé z nich jsou velmi efektivní a lze vůči těmto metodám uplatnit určité zjevné preference, je třeba si položit otázku, za jakých okolností je třeba k optimalizačním metodám přistupovat.

Pokud se jedná o aspekty optimalizace mimo rámec samotných úprav kódu SQL dotazu, lze konstatovat, že kritéria kladená na danou úlohu vcelku jednoznačně vymezí rámec pro možné úpravy a optimalizační proces je bez větších alternativ definován.

Pokud se ale zaměříme na úpravy kódu SQL dotazu, vyvstává výše zmíněná otázka, proč je zpracování tak jak jej určí optimalizátor a plánovač databázového stroje třeba upravovat. DBS Oracle disponuje natolik sofistikovanými metodami vytvoření nejlepšího možného exekučního plánu, že pokud nejsou porušeny základní předpoklady (jako například chybné nebo zastaralé statistiky, histogramy hodnot atributů, chybějící nebo neaktivní indexy, atd.) určí optimální způsob zpracování správně a co víc, bez vnějšího zásahu může plán zpracování za určitých okolností dokonce průběžně upravovat.

Neoptimální způsob přístupu k datům a jejich zpracování je v drtivé většině způsoben nikoli chybou optimalizátoru a plánovače, porušením základních předpokladů jak byly zmíněny výše nebo technickými důvody, ale výhradně chybným datovým modelem který obsahuje strukturální nedostatky nebo absentují základní podpůrné prostředky jako například indexy. Při řešení nedostatečného výkony by se měli tedy řešitelé soustředit především na tyto příčiny než nekoncepčním způsobem upravovat kód aplikace.

Z širšího obecného obchodně politického pohledu lze konstatovat, že příliš sofistikované úpravy kódu řešení v informačním systému nejsou ani žádoucí to minimálně z těchto důvodů:

1) Stabilita programového kódu řešení informačního systému při provozu a rozvoji

2) Čitelnost, srozumitelnost kódu má pro vlastníka systému větší cenu než vyšší výkon ve smyslu rychlosti zpracování. V krajním případě, pokud nelze zlepšit datový model a podmínky pro optimalizátor obecně, je rozumnější použít větší hrubou výpočetní sílu, než nepřiměřeně komplikovat vnitřní strukturu řešení. Menší cenu za výkonný hardware pak vlastník a provozovatel informačního systému mnohonásobně zaplatí za špičkové specialisty, kteří jediní pak mohou složité řešení ovládnout.

6 Seznam literatury

[1] Donald K. Burleson: Donald Burleson Articles
http://www.dba-oracle.com/articles.htm

[2] Donald K. Burleson: Cost Control: Inside the Oracle Optimizer
http://www.dba-oracle.com/art_otn_cbo.htm

[3] Oracle Corp.: Oracle Database 11g Release 1 (11.1) Documentation
Oracle Technology Network, 2008

[4] Oracle Corp.: Oracle Database Performance Tuning Guide 11g Release 1 (11.1)
http://download.oracle.com/docs/cd/B28359_01/server.111/b28274.pdf

[5] Oracle Corp.: Oracle9i Database Performance Tuning Guide and Reference Release 2
http://download.oracle.com/docs/cd/B10500_01/server.920/a96533/toc.htm

[6] Peter Gulutzan, Trudy Pelzer: SQL Performace Tuning
Addison Wesley 2002. ISBN 0-201-79169-2.

[7] Gavin Powell: Oracle High Performance Tuning for 9i and 10g
Digital Press, 2004. ISBN 1-55558-305-9.

[8] C.J. Date: Database in Depth
O'Reilly, 2005. ISBN 0-596-10012-4.

[9] Mark Gurry: Oracle SQL Tuning Pocket Reference
O'Reilly, 2002. ISBN 0-596-00268-8.

[10] Dan Tow: SQL Tuning
O'Reilly, 2003. ISBN 0-596-00573-3.

[11] Oracle Corp.: Oracle9i: SQL Tuning Workshop
Oracle Technology Network, 2008

7 Přílohy

7.1 Úplný výpis skriptu pro generování testovací báze dat

```
DECLARE

   -- Konstanty
   con_commit_modulo CONSTANT PLS_INTEGER := 1;
   con_min_gen_rok CONSTANT PLS_INTEGER := 2007;
   con_max_gen_rok CONSTANT PLS_INTEGER := 2008;
   con_max_doklad_rok CONSTANT PLS_INTEGER := 100000;

   -- Proměnné
   var_rok PLS_INTEGER;
   var_doklad PLS_INTEGER;
   var_radek PLS_INTEGER;
   var_dan_dokl PLS_INTEGER;
   var_dat_prip DATE;
   var_mnozstvi PLS_INTEGER;
   var_cena PLS_INTEGER;
   var_doklad_count PLS_INTEGER;

BEGIN
   DBMS_OUTPUT.put_line ('BEG: ' ||
                         TO_CHAR (SYSDATE, 'dd.mm.yyyy hh24:mi:ss'));

   -- Zrušení aktuální báze dat --
   DELETE FROM diplom.zp_radek;
   DELETE FROM diplom.zp_doklad;
   DELETE FROM diplom.zp_dokument;

   COMMIT;
   DBMS_OUTPUT.put_line ('DEL: ' ||
                         TO_CHAR (SYSDATE, 'dd.mm.yyyy hh24:mi:ss'));

   -- Vygenerování fiktivního obsahu do reálnách datových struktur --
   FOR var_rok IN con_min_gen_rok .. con_max_gen_rok
   LOOP
      FOR var_doklad IN 1 .. con_max_doklad_rok
      LOOP
         -- Hlavní záznam, hlavička faktury
         var_dan_dokl := ROUND (DBMS_RANDOM.VALUE (0, 1));
         var_dat_prip := CASE
                           WHEN var_dan_dokl = 1
                              THEN TO_DATE ('1.1.2003', 'dd.mm.yyyy') +
                                   ROUND (DBMS_RANDOM.VALUE (-600, 600))
                           ELSE NULL
                         END;

         INSERT INTO diplom.zp_doklad (
zp_rok,
/* součást intetrní technické identifikace */

zp_doklad,
/* součást intetrní technické identifikace */

kn_doklad,
```

```
/* text, externí identifikace, vysoká kardinalita */

castka,
/* reálné číslo, celková částka k úhradě, vysoká kardinalita */

stav,
/* celočíselná hodnota, informace o zpracování v IS, nízká kardinalita */

dat_prij,
/* datum, datum přijetí dokladu, střední kardinalita */

dan_dokl,
/* logická hodnota, příznak daňového dokladu, minimální kardinalita */

dat_prip
/* datum, datum uskutečnění zdanitelného plnění, střední kardinalita */
)

         VALUES (
var_rok,
var_doklad,
ROUND (DBMS_RANDOM.VALUE (100000000, 999999999)),
ROUND (DBMS_RANDOM.VALUE (500, 150000), 2),
5 * ROUND (DBMS_RANDOM.VALUE (1, 5)),
TO_DATE('1.1.2006', 'dd.mm.yyyy') + ROUND(DBMS_RANDOM.VALUE (-600, 600)),
var_dan_dokl,
var_dat_prip);

         -- Záznam s rozšiřující informací k faktuře, relace 1:1
         IF ROUND (DBMS_RANDOM.VALUE (1, 30)) = 1
         THEN
            INSERT INTO diplom.zp_dokument
                        (zp_rok,
                         zp_doklad,
                         char_1,
                         date_1,
                         deci_1,
                         text_b)
                 VALUES (var_rok,
                         var_doklad,
                         'ABC-ABC-ABC-ABC',
                         SYSDATE,
                         123,
                         'XXX');
         END IF;

         -- Řádky faktury, náhodný počet, relace 1:N
         FOR var_radek IN 1 .. ROUND (DBMS_RANDOM.VALUE (1, 10))
         LOOP
            var_mnozstvi := ROUND (DBMS_RANDOM.VALUE (1, 12));
            var_cena := ROUND (DBMS_RANDOM.VALUE (7, 1000));

            INSERT INTO diplom.zp_radek
                        (zp_rok, zp_doklad, zp_radek,
                         polozka, kat_cis, mer_jedn,
                         mnozstvi, cena_mj, celkem,
                         an_ucet, strana, popis)
                 VALUES (
var_rok, var_doklad, var_radek, 10 * var_radek,
```

```
'KT' || ROUND (DBMS_RANDOM.VALUE (100, 999)),
'ks', var_mnozstvi, var_cena, var_mnozstvi * var_cena,
'501' || TO_CHAR (ROUND (DBMS_RANDOM.VALUE (100, 999))), 'MD',
'Text Text '
 || TO_CHAR (ROUND (DBMS_RANDOM.VALUE (5, 8)))
 || ' Text Text '
 || TO_CHAR (ROUND (DBMS_RANDOM.VALUE (2, 4)))
 || ' Text Text');
         END LOOP;

         -- Řízení transakčního zpracování
         IF NOT con_commit_modulo = NULL
         THEN
            var_doklad_count := var_doklad_count + 1;

            IF var_doklad_count MOD con_commit_modulo = 0
            THEN
               COMMIT;
            END IF;
         END IF;
      END LOOP;
   END LOOP;

   -- Finální uzavření transakce
   COMMIT;
   DBMS_OUTPUT.put_line ('END: ' ||
                         TO_CHAR (SYSDATE, 'dd.mm.yyyy hh24:mi:ss'));

END;
```