Architektury a techniky DS

Efektivní navrhování schémat

Přednáška č. 2

RNDr. David Žák, Ph.D.
Fakulta elektrotechniky a informatiky
david.zak@upce.cz

Indexy

Bez indexů:

- záznamy v tabulkách ukládány neuspořádaně, nejčastěji v pořadí, v jakém byly do tabulky vkládány,
- záznamy hledáme od prvního záznamu postupně, dokud hledaný nenajdeme,
- v průměru vždy prohledáme asi polovinu záznamů,
- tyto problémy řeší přidání dalšího objektu – INDEXu, ten obsahuje informace o tom, kde se jaký záznam nachází.

Indexy

S indexy:

- při hledání podle indexového sloupce stačí najít údaj v indexu a odtud získat rowID (jedinečný identifikátor, který databázový server přiřadí každému záznamu v tabulce) a podle něho najít požadovaný záznam,

- moderní dtb. servery obvykle používají stromovou strukturu indexů, například B- stromy,

- průchod uspořádaným stromem je v průměru výrazně kratší než sekvenční hledání v neuspořádaném poli.

Indexy

Vlastnosti:

- zrychlení jen tehdy, pokud potřebujeme najít několik záznamů, pokud je cílem najít více než 5% záznamů, bude obvykle rychlejší hledání bez použití indexu,

- zvýšení režie při vkládání a změně dat, protože kromě zápisu do tabulky je třeba zapsat indexový údaj na přesně definované místo v indexu,

- nedoporučuje se používat indexy (kromě bitmapových indexů) nad malými tabulkami nebo nad sloupci s nízkou variabilitou (např. sloupec pohlaví žena/muž),

- odstraněním indexu zůstane tabulka nezměněná.

Syntaxe příkazu CREATE INDEX

Vytvoří index nad uvedeným sloupcem definované tabulky.

CREATE [BITMAP] INDEX <název indexu>
ON <název tabulky> (<název sloupce>)

BITMAP vytvoří bitmapový index - vhodné řešení pro sloupce s nízkou variabilitou. Lze si to představit tak, že každé hodnotě ve sloupci odpovídá jeden bit kódu,
tj. pro pohlaví jsou nutné 2 bity (muž, žena),
pro rodinný stav 4 bity (svobodný, ženatý, rozvedený, vdovec).

Příklad:

```
CREATE INDEX ix_jmeno ON ucitele(jmeno);
CREATE BITMAP INDEX ix_jmeno ON
ucitele(pohlavi);
```

Syntaxe příkazu DROP INDEX

Odstraní index, ale nikoli data v tabulce.

DROP INDEX <název indexu>

Příklad:

```
DROP INDEX ix_jmeno;
```

Základní principy návrhu schémat

Integrita dat vynucená databází

- Při nenastavení integrity na úrovni databáze může docházet i chybám v transakčním zpracování (je nutné ručně zamykat podřízené tabulky a podobně)

Použití správných datových typů

Integrita dat = fakt, že data věrně (tj. přesně a konzistentně) zobrazují reálný stav, který popisují.

Základním předpokladem je kvalitně navržená datová základna, která omezí duplicity dat (ty jsou vždy vysokým rizikem pro vznik nekonzistencí).

Použití správných datových typů

Nesprávně:

- Řetězec pro uchování dat nebo času,
- Řetězec pro uchování čísel,
- VARCHAR2(4000) pro všechny řetězce,
- CHAR(2000) k uchování všech řetězců,
- Vkládání textu do BLOB

Pravidla:

- Nikdy nepoužívat určitý datový typ pro ukládání jiného typu dat !
- Vždy použít ten nejpřesnější.
- Srovnávejte správné typy vůči sobě (data s daty, řetězce s řetězci, čísla s čísly).

Použití správných datových typů

Rizika při nedodržení pravidel:

- Při vkládání dat neproběhne jejich kontrola,
- Nižší výkon – používání konverzních funkcí, ošetřování chyb,
- Zvýšení požadavků na prostor,
- Snížení integrity dat – tj. existence neplatných dat,
- U datumu uchovávaných jako řetězec hrozí záměna dnů a měsíců, případně i let 01/02/03

Potíže s výkonem

Příklad

Tabulka (datum date, datum_str varchar2(8))

Sloupec datum_str má „formát" YYYYMMDD

Riziko: Bude špatně fungovat optimalizátor

Ten ví, že mezi daty 31.12.2000 a 1.1.2001 je jen jediný den, ale domnívá se, že mezi řetězcovými položkami '20001231' a '20010101' je velké množství hodnot. Mohutnost příkazů je posouzena nesprávně, což může mít vliv na výkon.
V určitém okamžiku bude u vyhledávání podle data použit rozsah hodnot indexu, ale u řetězce proběhne úplné prohledávání.

Navíc nikdo nezabrání vložení hodnot '20001299' do řetězce.

Správný datový typ

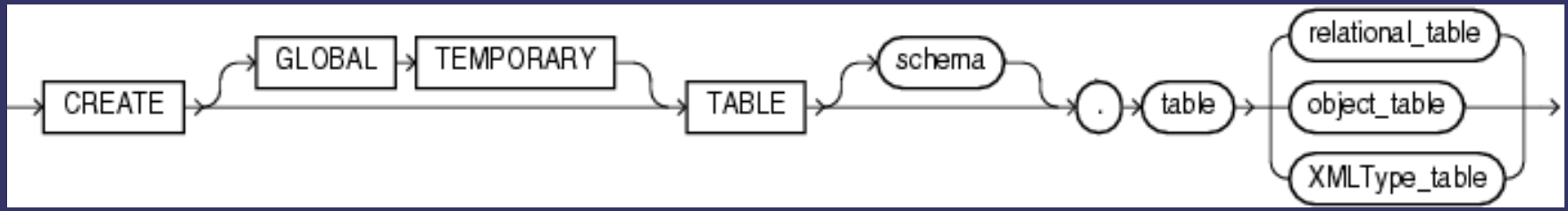
Diskuse VARCHAR2(50) a VARCHAR2(4000)

1) Co se stane, pokud uživatelé načtou data dotazovacím nástrojem, který formátuje každé pole na základě šířky sloupce v databázi ?

2) Pro vyhodnocení dotazu je třeba uložit data – tedy je nutná rezervace místa v paměti,

3) Velikost místa ve vstupním formuláři

Syntaxe CREATE TABLE



Použití příkazu CREATE TABLE pro vytvoření jednoho z následujících typů tabulek:

- Relační tabulky, které jsou základní strukturou pro uchovávání dat.
- Objektové tabulky, tabulky vytvořené nad datovým typem object
- XML tabulky, tabulky vytvořené nad datovým typem XMLType.

Poznámka – datové typy object či XMLType mohou být také použity pro definici sloupců.

http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/statements_7002.htm

Přehled typů relačních tabulek

Tabulka uspořádaná do haldy – standardní, při přidávání dat je použito první volné místo v segmentu, do něhož se data vejdou, po odstranění je toto místo opět k dispozici pro opětovné vložení či aktualizace. Halda je hromada prostoru, který je využíván náhodně.

Clustrově uspořádaná tabulka pomocí indexu B*Tree – mnoho tabulek může být fyzicky spojeno, ve stejném bloku data z mnoha tabulek, všechna data obsahující stejnou hodnotu klíče clusteru budou fyzicky uložena společně, klíč clusteru je vytvořen pomocí indexu B*Tree

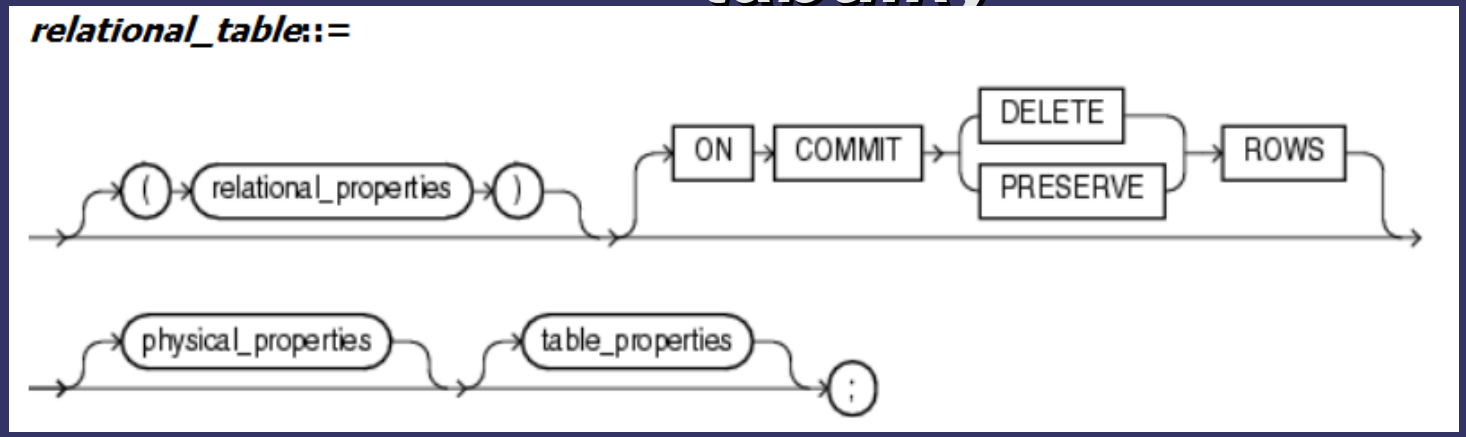
Clustrově uspořádaná tabulka pomocí algoritmu hash – obdoba výše uvedené, jen namísto použití indexu B*Tree zatřiďuje cluster hash klíč ke clusteru, čímž určuje blok, v němž budou data uložena. Obrazně řečeno, v clusteru typu hash jsou data indexem. Vhodné pro data, která jsou často čtena prostřednictvím srovnání rovnosti klíče.

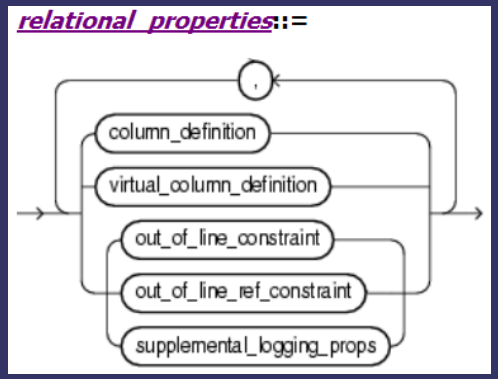
Přehled typů relačních tabulek

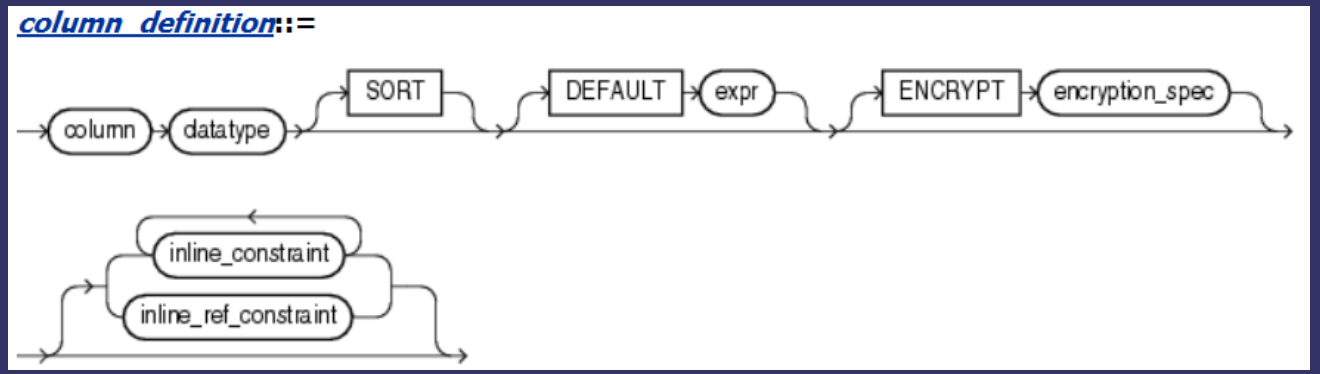
Indexově uspořádané tabulky – tabulka má strukturu indexu, čímž je určeno samotné řazení řádků. Uložená data jsou seřazena podle hodnoty primárního klíče.

Externí tabulka – data jsou uložena v souborech ve zjednodušeném formátu mimo databázi Oracle. Data nemohou být také konvenčně indexována.

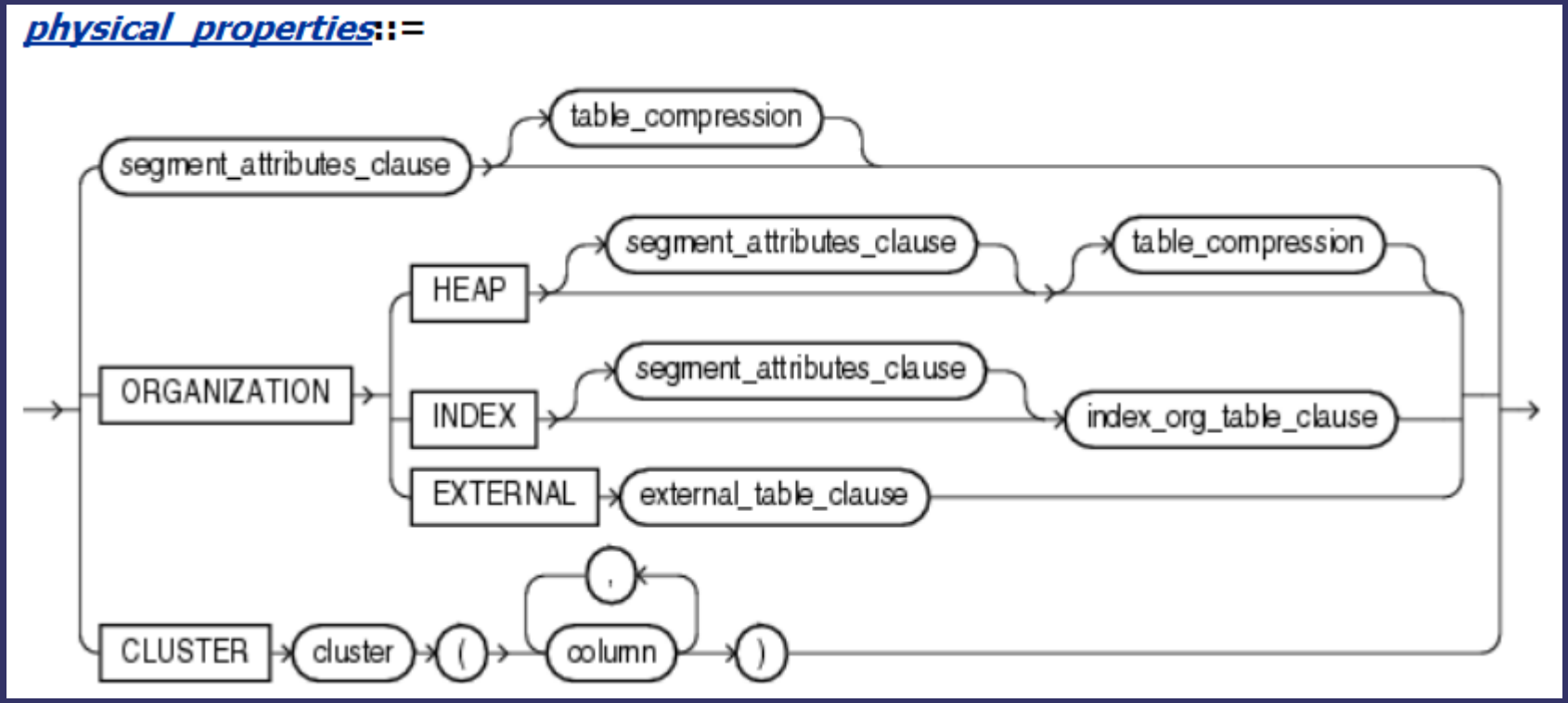
Syntaxe CREATE TABLE – relační tabulky







Syntaxe CREATE TABLE – relační tabulky



Clustrově uspořádaná tabulka pomocí indexu B*Tree

Clustery jsou segmenty v Oracle, které plní 2 účely:

- **Fyzicky spojují data dohromady pomocí běžného klíče (data nejsou tříděna, jsou pouze uložena společně podle klíče)**
  Pokud je cluster tvořen na základě numerického klíče, budou veškerá data např. s hodnotou klíče 10 fyzicky spojena dohromady ve stejném bloku databáze (pokud se do něho vejdou)

- **Umožňují uložit data z několika tabulek ve stejném fyzickém bloku (například řádek pro DEPTNO=10 z tabulky DEPT se může nacházet ve stejném bloku jako řádky z tabulky EMP pro stejnou hodnotu DEPTNO – jde tedy o metodu předběžného spojení dat), výsledkem je snížení požadavků na indexy**

- **Smysl – omezení počtu V/V operací pro odpověď na dotazy, zejména při vyhledávání a spojování tabulek dle hodnoty klíče.**

Tvorba clusterů

Oracle podporuje 2 typy clusterů:

- Cluster B*Tree – k uložení hodnoty klíče a adresy bloku, kde lze data najít (obdoba s Indexy)

- Cluster hash – převede hodnotu klíče na adresu bloku databáze pomocí algoritmu hash, vynechá tedy všechny V/V operace mimo vlastního čtení bloku (říkáme tedy, že data jsou sama indexem)

Tvorba clusterů

```
CREATE CLUSTER [schema.]cluster
    (column datatype [,column datatype] ... )
    [PCTUSED integer] [PCTFREE integer]
    [SIZE integer [K|M] ]
    [INITRANS integer] [MAXTRANS integer]
    [TABLESPACE tablespace]
    [STORAGE storage_clause]
    [  PARALLEL ( [ DEGREE { integer | DEFAULT } ]
                  [ INSTANCES { integer | DEFAULT } ]
                )
     | NOPARALLEL ]
    [  CACHE | NOCACHE  ]
    [INDEX
    | [HASH IS column] HASHKEYS integer]
```

http://www.stanford.edu/dept/itss/docs/oracle/10g/server.101/b10759/statements_5

Tvorba clusterů B*Tree - příklad

```
CREATE CLUSTER XYZ_btree (username VARCHAR2(30))
SIZE 512;
```

Vytvoří cluster, který je fyzickým úložištěm,
Cluster bude klíčován podle sloupce username typu VARCHAR2(30)
To znamená, že veškerá data v libovolné tabulce budou fyzicky uspořádána podle hodnoty username.

SIZE – každé hodnotě klíče by měl být přiřazen určitý počet Bajtů, údaj je použit pro vypočítání maximálního počtu klíčů, které by se měly vejít do jednoho bloku (např. o velikosti 4 nebo 8 kB)

To neznamená, že data budou seřazena podle hodnoty username, ale data vztahující se k jakémukoli zadanému jménu bude obvykle možné získat z jednoho bloku

Dále povinně vytvoříme INDEX

```
CREATE INDEX XYZ_idx on cluster XYZ_btree
```

Index bude obsahovat položku pro každou jedinečnou hodnotu klíče vloženou do clusteru a ukazatel na první blok obsahující data k této hodnotě klíče.

Tvorba clusterů hash - příklad

```
CREATE CLUSTER cluster_hash (username VARCHAR2(30))
SIZE 512
HASHKEYS 100 ;
```

Vytvoří cluster hash

HASHKEYS – počet očekávaných jedinečných hodnot klíče (pokud tuto hodnotu překročíme, snížíme efektivnost tabulky hash vzhledem ke kolizím)

SIZE – každé hodnotě klíče by měl být přiřazeno určitý počet Bajtů, údaj je použit pro rezervaci prostoru

Vytvoření tabulek v clusterech

CREATE TABLE XYZ_users
(username varchar2(30), User_id int, narozen date)
CLUSTER XYZ_btree (username);

CREATE TABLE XYZ_objects
(owner varchar2(30), name varchar(50), type char(1))
CLUSTER XYZ_btree (owner);

Vytvoří 2 prázdné tabulky.
První tabulka používá pro vlastní uspořádání svůj sloupec username,
Druhá sloupec owner.
V praxi to znamená, že veškerá data v tabulce XYZ_users, jejichž hodnota username bude X, budou fyzicky v databázi pravděpodobně ve stejném umístění, jako všechna data tabulky XYZ_objects, kde owner=X.

Použití clusterů

Clustery budou nejefektivnější v prostředí, kdy lze řídit způsob jejich načítání (tedy vkládání dat). Budou tedy vhodné například pro operace v datovém skladu, kdy jsou data čas od času načítána a je možné jejich načítání řídit. Jsou také vhodné pro transakční systémy (vkládán jeden nadřízený a poté mnoho nebo všechny podřízené záznamů současně).

Clustery B*Tree nejsou vhodné v případě, kdy jsou data vkládána náhodně, nikoli podle pořadí klíče clusteru.

Smysl – společné uložení souvisejících informací do stejných bloků.

Clustery hash jsou navíc ještě díky vyloučení bloků indexu je o krok dále, do vyrovnávací paměti lze vložit více důležitých dat (protože důležitých dat je méně).

Clustery hash s jedinou tabulkou

- Speciální případ
- Oracle ví, že bloky obsahují data z jediné tabulky – optimalizace
- smysl pro vyhledávání (zrychlení, jinak by probíhalo několik přístupů k indexu následované přístupem do tabulky podle hodnoty ROWID, zde může jít o jedinou V/V operaci)

Příklad

```
CREATE CLUSTER cust_orders
(customer_id NUMBER(6))
SIZE 512
SINGLE TABLE
HASHKEYS 100;
```

Co bychom získali

Například při 50000 řádcích a 150000 vyhledáváních při použití clusteru snížení logických V/V operací na 42%.

*Clustery B*Tree - shrnutí*

Výhody
- Fyzické spojení dat
- zvýšení efektivity vyrovnávací mezipaměti
- snížení počtu logických V/V operací
- nižší potřeba indexů, neboť všechny tabulky sdílí jeden index klíče clusteru

Omezení
- Správné nastavení velikosti clusteru (SIZE) vyžaduje přemýšlení,
- Vkládání musí být řízeno a musí být řiditelné,
- Vkládání je pomalejší, jelikož data musí být vložena do určitého umístění

Indexově uspořádané tabulky

Velice podobná clusteru B*Tree s těmito odchylkami:

- IOT má jednu datovou strukturu, jednu strukturu indexu (cluster B*Tree má indexový a datový segment)

- data jsou setříděná uložená podle primárního klíče (v clusteru B*Tree jsou organizována podle hodnoty klíče, ale samotné klíče nejsou setříděné)

- nastavení velikosti IOT (indexově uspořádané tabulky) je jednodušší (není třeba odhadovat maximální počet klíčů)

- tabulky lze přestavět

IOT jsou užitečné pro implementaci:

- Asociativní tabulky pro vztahy M:N (tedy úzké a vysoké tabulky)
- Důležité fyzické spojení dat, ale pořadí nepředvídatelné nebo by clusteru neumožňovalo udržovat data pohromadě

IOT pro asociativní tabulky

Obvykle (normálně) jsou tyto tabulky vytvářeny jako:

```
create table poznatky
    (Id_trpaslika NUMBER(3),
    Id_vlastnost NUMBER(3));
Create index poznatky_idx1 on poznatky(Id_trpaslika, Id_vlastnost);
Create index poznatky_idx2 on poznatky(Id_vlastnost, Id_trpaslika);
```

Jsou tedy vytvořeny 3 struktury.

Pomocí IOT

```
create table poznatky
    (Id_trpaslika NUMBER(3),
    Id_vlastnost NUMBER(3)
    primary key(Id_trpaslika, Id_vlastnost))
    organization index;
Create index poznatky_idx on poznatky(Id_vlastnost);
```

Externí tabulky

Umožňují použít příkaz SELECT na soubory s daty

a) oddělenými čárkami,
b) poziční soubory s pevnou šířkou

Ve starších verzích Oracle je nebylo možné upravovat, bylo možné se na ně jen dotazovat.

Nemohou být indexovány, neboť pro řádky neexistují žádné hodnoty ROWID.

Vhodné pro načítání a slučování dat – tedy nikoli jako náhrada tabulek.

Externí tabulky

Závisí na nastavení objektu DIRECTORY databáze Oracle

```
CREATE OR REPLACE DIRECTORY ext_dir as '/tmp/';
```

A práv uživatele k tomuto adresáři

```
GRANT read, write on directory ext_dir to username;
```

Soubory musí být na databázovém serveru, vlastní databázové procesy musí být schopny soubor zjistit, otevřít a přečíst. To lze řešit i připojením/mapováním disků.

Externí tabulky s oddělovačem

Vytvoření tabulky

```
create table ext_table_csv
    (  i   Number,
     n   Varchar2(20),
     m   Varchar2(20))
     organization external (
                type   oracle_loader
                default directory ext_dir
                access parameters (
                           records delimited  by newline
                           fields  terminated by ','
                           missing field values are null)
                           location ('file.csv') )
                reject limit unlimited;
```

Externí tabulky s pevnou šířkou sloupců

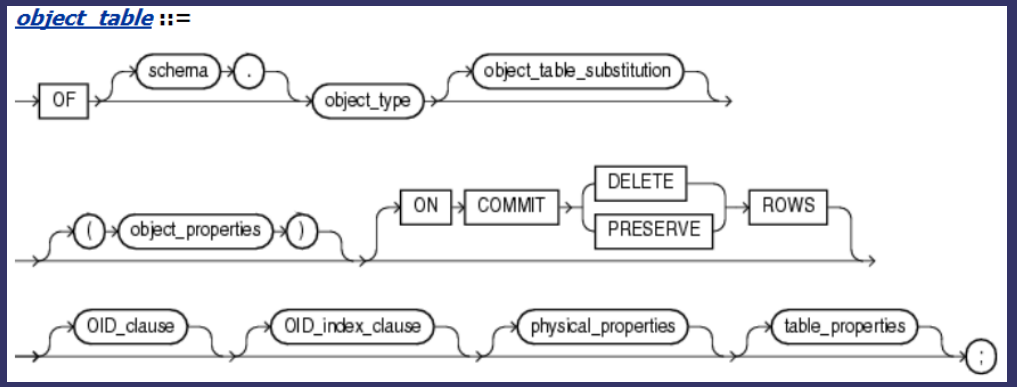
Vytvoření tabulky

```
create table ext_table_fixed
    (   field_1 char(4),
        field_2 char(30))
   organization external
   (   type        oracle_loader
       default directory ext_dir
       access parameters (
                   records delimited by newline
                   fields (
                               field_1 position(1: 4) char( 4),
                               field_2 position(5:30) char(30)))
     location ('file')
    )
    reject limit unlimited;
```

Použití typu object

Příklad použití typu object
v Oracle DB

```
CREATE TYPE t_adresa AS OBJECT (
             ulice     VARCHAR2(30),
            cislo     NUMBER(3),
            obec      VARCHAR2(30),
            psc       NUMBER(5));
```



Příklad – uložení v tabulce

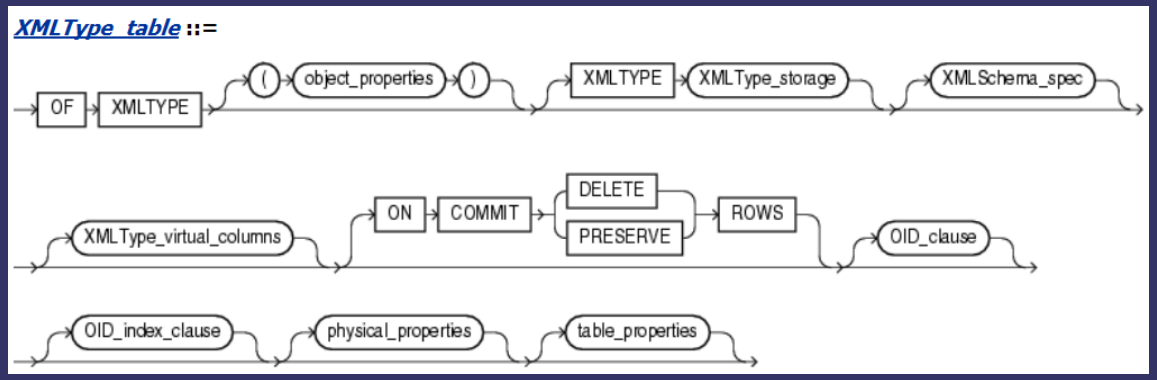
```
CREATE TABLE ADRESY OF t_adresa;
```

Příklad – uložení ve sloupci

```
CREATE TABLE obyvatele (
            prijmeni  VARCHAR2(30),
            jmeno     VARCHAR2(30),
            adresa    t_adresa);
```

Použití typu XMLType

Příklad uložení dokumentu XML v Oracle DB použitím datového typu XMLType



Příklad – uložení v tabulce

```
CREATE TABLE XMLTABLE OF XMLType;
```

Příklad – uložení ve sloupci

```
CREATE TABLE PURCHASE_ORDER_TABLE (
po_number number(16),
purchase_order xmltype );
```

Techniky indexování

a) B*Tree – standardní
b) index s obráceným pořadím bajtů
   (vhodný pro jediný blok s monotónně rostoucí hodnotou)
c) Sestupný index – jedno nebo více polí uloženo sestupně
d) tabulka IOL (indexově uspořádaná tabulka)
e) clusterový index B*Tree
f) funkční index (FBI) – index u složitého výpočtu nebo definované funkce
   např. upper(name)
g) doménový index – index, jenž si můžete vytvořit sami

Architektury a techniky DS

Tvorba efektivních příkazů I

Přednáška č. 3

RNDr. David Žák, Ph.D.
Fakulta elektrotechniky a informatiky
david.zak@upce.cz

Proč optimalizovat ?

- Snížit požadavky na systémové prostředky
- Zkrátit dobu zpracování

Předpoklady:

- Porozumění zpracování dotazů
- Znalost fyzického uspořádání dotazovaného objektu (schéma)
- Znalost všech složitostí jazyka SQL
- Vědět, co je k dispozici = znalost možností databáze
- Dobré porozumění cíle – jaké je vlastně zadání

Principy přístupových cest

Přístupové cesty jsou metodami, jejichž prostřednictvím může databáze Oracle získat data.

Přístupové cesty:

- Úplné prohledávání
- Indexové přístupy (mnoho typů)
- Přímý přístup pomocí algoritmu hash nebo adresy ROWID

Neexistuje jediná nejlepší cesta, kdyby existovala, používal by databázový systém jen tuto jedinou!

TABLE ACCESS FULL

Úplné prohledávání

- Databáze přečte segment od začátku do konce a zpracuje každý blok,
- Účinná metoda čtení velkého množství dat, neboť databáze použije souběžné čtení několika bloků (najednou),
- V rámci jedné V/V operace se přečte najednou N bloků,
- Tato metoda není nic špatného, často je i nejrychlejší metodou pro získání odpovědi (pokud je výsledkem více než 40% záznamů, pak jde i o vhodnou metodu).

TABLE ACCESS BY INDEX ROWID

Přístup pomocí adresy ROWID

- ROWID jsou fyzickými adresami dat

- ROWID má podobu 18 znakového řetězce ve formátu: OOOOOOFFFBBBBBBRRR, kde
  - O - reprezentuje data object number a identifikuje segment databáze
  - F - je datafile se záznamem
  - B - je data block, tedy datová stránka v rámci datafile, která obsahuje záznam
  - R - je row, číslo záznamu v datové stránce

- Jednotlivé informace lze extrahovat pomocí balíčku DBMS_ROWID
- Adresa ROWID je nejrychlejší cestou k určitému řádku, k většímu množství řádků však nebude nejrychlejší metodou

Přístup pomocí adresy ROWID

- Typické použití pro přístup k tabulce po jejím prohledání pomocí některého typu indexu

select * from table where indexovany_sloupec=hodnota;

Databáze

- Prohledá index u indexovaného sloupce
- Načte hodnotu ROWID řádku, na který odkazuje
- Podle hodnoty ROWID přistoupí k tabulce a načte konkrétní soubor a blok

Přístup pomocí adresy ROWID

- ROWID mohu použít při tvorbě vlastních tabulek, které budou plnit roli obdobnou indexům

- Paralelizace zpracování dotazu, přičemž jednotlivé části nebudou ve sporu o stejné sdílené diskové prostředky (rozdělení jednoho úplného prohledávání na více úplných prohledávání na částech tabulky)

```
Select * from table where ROWID between :a1 and :a2
Union all Select * from table where ROWID between :b1 and :b2
Union all Select * from table where ROWID between :c1 and :c2
Union all Select * from table where ROWID between :d1 and :d2;
```

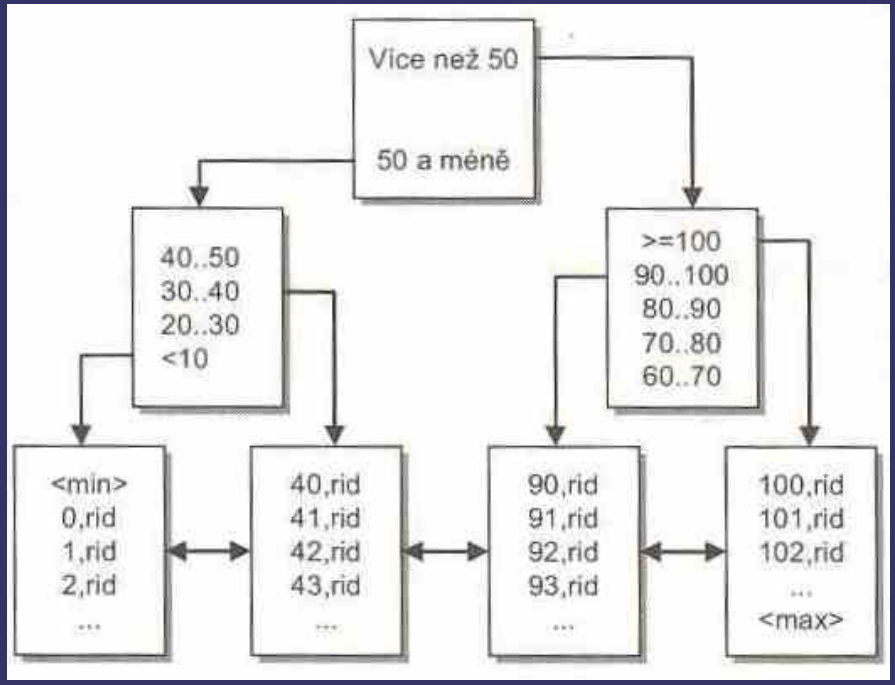
Indexové prohledávání

- Jde pravděpodobně o nejoblíbenější přístupovou metodu

- Bloky na nejnižší úrovni stromu (listy) obsahují všechny indexované klíče a adresu ROWID, odkazující na řádek, který indexuje
- Vnitřní bloky (větví) slouží k navigaci strukturou
- Listové uzly indexu jsou ve skutečnosti dvojitě propojeným seznamem. Jakmile zjistíme, kde začít, je prohledání seřazených hodnot (prohledání rozsahů indexů) snadné (lze pokračovat v dalších listech, dokud nenarazíme na hodnotu vyšší)
- Všechny listové bloky by měly být na stejné úrovni stromu,
- Většina indexů bude mít výšku 2 až 3, tj. pro vyhledání klíče bude třeba 2 nebo 3 V/V operací

Indexové prohledávání

Struktura indexu B*Tree u číselného sloupce

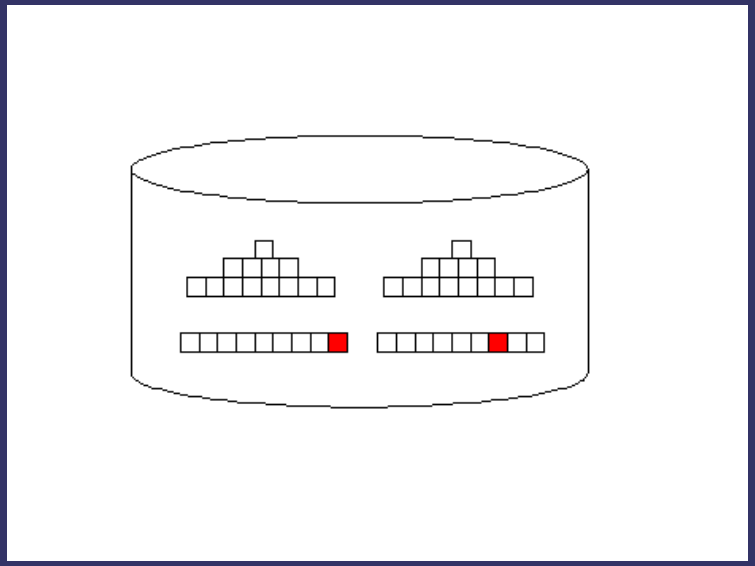


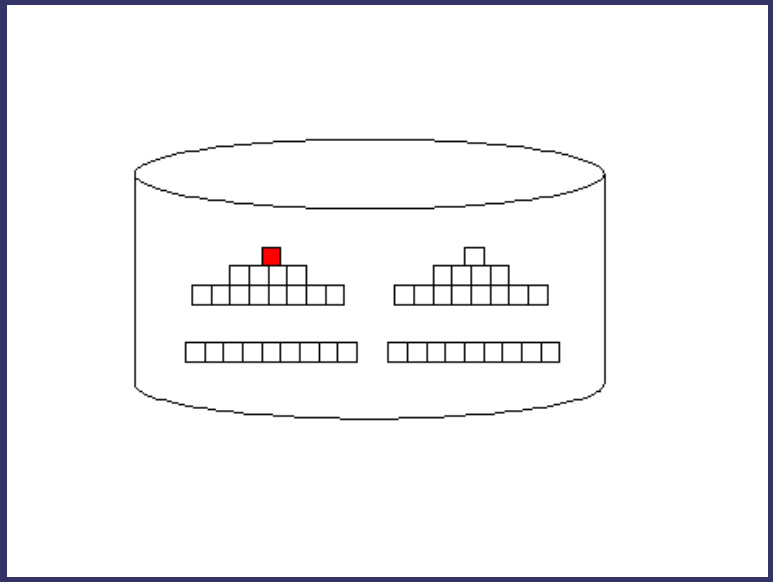
Prohledávání jedinečného indexu

- Optimalizátor pozná, že indexované sloupce v indexu jsou jedinečné
- Po prohledání indexu bude vrácen nejvýše jeden řádek

- u nejedinečného indexu jsou data v indexu seřazena podle hodnot klíče indexu a poté podle hodnoty ROWID

Porovnání úplného prohledávání a přístupu pomocí indexu





Prohledávání rozsahů indexu

- Optimalizátor očekává vrácení žádného, jednoho, dvou nebo více řádků
- Rozsahy indexu mohou být prohledány jedním nebo dvěma směry (obvykle vzestupně, ale může být i sestupně)

Příklad:

```
Select cislo_zamestnance from zamestnanci
Where cislo_zamestnance<5000
Order by cislo_zamestnance desc
```

Databáze rozpozná, že se může vyhnout řazení, pokud index bude číst obráceně

Prohledávání rozsahů indexu

- Příklad:

  Select max(cislo_zamestnance) from zamestnanci

  Databáze rozpozná, že použije úplné prohledávání indexu, začte z opačného konce a zastaví se na první (nejvyšší) nalezené hodnotě.

Vyhledávání s vynecháním sloupců

- Příklad:

```
Create table t (a, b, c, d, e, f);
Create index t_idx on t(a, b, c);

Select * from t where a=:a;
Select * from t where a=:a and b=:b;
Select * from t where b=:b and c=:c;
```

U posledního příkladu by v dřívějších verzích nedošlo k využití indexu. Od verze Oracle9i je uplatněn princip prohledávání s vynecháním sloupců, za podmínek, že:

- Vnitřní sloupce jsou uvedeny v predikátu (definici indexu)
- Optimalizátor rozpozná, že sloupce na počátku indexu obsahují jen několik diskrétních hodnot, pak považuje index za N malých indexových struktur uvnitř jedné velké struktury

Úplné prohledávání indexu

- Nejsou načteny všechny bloky indexu, ale jsou zpracovány všechny listové bloky indexu, ale jen tolik bloků větví, kolik je třeba k nalezení prvního listového bloku
- Mezi listovými bloky se pak prochází prostřednictvím ukazatele (možno se pohybovat oběma směry, tedy dvojitě propojený seznam)
- Ke čtení jsou používány jednoblokové v/v operace
- Data jsou čtena z tabulek v seřazeném pořadí, jak se nachází ve struktuře indexu

INDEX FAST FULL SCAN

Úplné rychlé prohledávání indexu

Od úplného prohledávání indexu se liší:

- Jsou čteny všechny bloky ve struktuře indexu, včetně všech vnitřních bloků větví
- Používá víceblokové čtení
- Data nejsou načítána v seřazeném pořadí

- Struktura indexu je tedy přečtena rychleji

- Index je použit jako štíhlejší verze tabulky v případě, kdy se dotaz odkazuje pouze na indexované sloupce a je možné se vyhnout úplnému prohledání tabulky (to může být i důvodem přidání zdánlivě nadbytečného sloupce do indexu nikoli pro vyhledávání podle hodnot, ale protože jsou uvedena ve výběru v dotazu SELECT)
- Úplné rychlé prohledávání nemůže nahradit řazení, protože data nejsou vrácena seřazená

Spojení indexů

Spojení indexů může být metodou, která bude zvolena k zodpovězení dotazu, pokud u tabulky existuje více indexů obsahujících veškeré sloupce v dotazu (není třeba tedy přistupovat k tabulce)

Tato metoda je nejužitečnější v případě typu:

Create table t (a, b, c, d, e, f);

Select a, b, c from where a=:a and b=:b ;

Zde existují indexy A, B, C (můžou být zřetězené nebo indexy u jednotlivých sloupců)

Clustrové prohledávání

Clustrový přístup je proces, kdy je použit klič B*Tree nebo hash pro získání adresy kořenového bloku databáze obsahující data klíče a poté je následován zřetězený seznam bloků připojených ke kořenovému bloku.

Principy spojení (join)

- Spojení vnořenými cykly (nested-loop join)
- Spojení hash (hash join)
- Spojení sloučením po seřazení (sort-merge join)
- Kartézské spojení (Cartesion join)
- Anti spojení (anti join)
- Úplné vnější spojení (full outer join)

Spojení vnořenými cykly

Pro každý jednotlivý řádek první tabulky je indexově prozkoumána druhá tabulka a výsledkem jsou shodné řádky.

Tato technika spojení je obecně vhodná, chceme-li rychle získat ze sady výsledků první řádek

- Tato technika je vhodná pro vnitřní i vnější spojení

Obecně: Jelikož je vnější spojení více náročné (a omezuje optimalizátoru možnosti použití dalších metod), je třeba vždy zvážit, zda je použití vnějšího spojení vždy nezbytné.

Spojení hash

Databáze Oracle za optimálních podmínek použije menší ze 2 tabulek (například po úplném prohledání na omezující rozsah hodnot uvedených klauzuli where) a v paměti z těchto výsledků vytvoří tabulku hash (kde indexem je klíč spojení).

Dále se provede úplné prohledání větší tabulky, pro každý řádek podle klíče nalezne nebo nenalezne v tabulce hash v paměti shodné řádky a vrátí spojený obraz.

Získáním prvního řádku trvá určitou dobu (musí počkat na úplné prohledání menší z tabulek a vytvoření hash tabulky, poté jsou další řádky vraceny rychle, dokonce rychleji, než je možné přijímat.

Pokud se celá tabulka nevejde do paměti, pak je rozdělena do oddílů ukládaných do prostoru TEMP.

Spojení sloučením po seřazení

Při tomto spojení je seřazena vstupní sada 1, seřazena vstupní sada 2 a poté jsou výsledky sloučeny.

- Obecně méně účinné než spojení hash, jelikož je nutné prohledat a seřadit obě vstupní sady dat

- Vhodné pro operace spojování nerovnicemi, nebo kde je porovnáván rozsah

- Někdy optimalizátor zvolí toto spojení i v případě, kdy bude moci vynechat řazení pro operaci GROUP BY, ORDER BY a podobně

Kartézské spojení

Kartézské spojení je uplatněno vždy, když jsou v dotazu uvedeny 2 tabulky bez podmínek spojení. Každý řádek tabulky T1 je spojen s každým řádkem tabulky T2.

Výsledný počet řádků je dán n x m.

Často je toto spojení výsledkem nesprávně vytvořeného dotazu.

Anti spojení

Anti-spojení je používáno k vrácení řádků z tabulky, které se v tabulce nenachází – nejsou v nějakém jiném zdroji řádků.

Např.

Select * from katedry

where cislo_katedry NOT IN (select cislo_katedry from zamestnanci)

Obě tabulky jsou pak spojeny pomocí vnějšího spojení a uchovány jsou pouze řádky, kde neexistoval řádek tabulky zamestnanci.

Úplné vnější spojení

- Syntaxe úplného vnějšího spojení je od Oracle 9i
- jde jen o syntaktické zjednodušení pro zápis SQL dotazů (tedy běžného vnějšího spojení a antispojení obou tabulek)
- úplné vnější spojení je u velkých sad záznamů velmi „nákladnou“ operací, proto je nezbytné jej vždy důkladně zvážit

Pseudo-sloupec ROWNUM

ROWNUM – magický sloupec, příčinou řady potíží, proto je nezbytné mu porozumět, pak může být velmi užitečný.

Lze jej použít:

- Pro ladění dotazů,
- Pro číslování v rámci dotazu,
- pro provádění nejvyšších N- zpracování

Sloupci ROWNUM budou přiřazena čísla 1, 2, 3, 4, .. N

Hodnota ROWNUM není přiřazena řádku, řádek v tabulce nemá číslo.

Hodnota ROWNUM je přiřazena řádku po jeho průchodu fází predikátu dotazu, ale před řazením nebo souhrnem.

Pseudo-sloupec ROWNUM

Hodnota ROWNUM je zvýšena pouze po jejím přiřazení, proto následující dotaz nikdy nevrátí žádný řádek

Select * from zamestnanci where ROWNUM<=5 order by mzda

Správný zápis je

Select * from (Select * from zamestnanci order by mzdy)
where ROWNUM<=5

Pseudo-sloupec ROWNUM

Zpracování nejvyšších N – položek

Chceme jen 10 řádků z velmi rozsáhlé tabulky, kterou požadujeme seřadit podle neindexovaného sloupce.

Standardně by

1. Úplné prohledání tabulky T
2. Úplné řazení podle neindexovaného sloupce,
3. Paměť nebude dostatečná, tedy nutno odkládat dočasné rozsahy na disk
4. Sloučení dočasných rozsahů za účelem získání prvních 10 záznamů
5. Odstranění (tedy uvolnění) dočasných rozsahů

Při správném zápisu s omezením pomocí ROWNUM

1. Úplné prohledání tabulky T
2. V poli N elementů se řadí pouze N řádků, toto pole se naplní N prvními řádky a poté pro každý další řádek se provede porovnání s posledním řádkem v poli, zda má být vřazen nebo zahozen – vystačí se s pamětí (výrazně urychlí operaci)

Skalární poddotazy

Od verze Oracle8i je možné použít poddotaz,
který vrátí nejvýše jeden řádek a sloupec, a to kdekoli,
kde bylo dříve možné používat znakový řetězcový literál.

```
SELECT (SELECT sloupec FROM … WHERE …)
FROM dual
WHERE
        (SELECT sloupec FROM … WHERE …) = (SELECT sloupec FROM … WHERE
        …);
```

Odstranění vnějšího spojení

Příklad

```
SELECT a.username, count(b.username)
FROM all_users a LEFT JOIN all_objects b
ON b.owner = a.username
GROUP BY a.username;
```

Možno přepsat se skalárním poddotazem jako

```
SELECT a.username, (SELECT count(*)
FROM all_objects b WHERE b.owner = a.username) pocet
FROM all_users a;
```

Tento dotaz je výkonnější.

Odstranění vnějšího spojení

Příklad – chceme ještě doplnit

```
SELECT a.username, count(*), avg(object_id)
FROM all_users a LEFT JOIN all_objects b
ON b.owner = a.username
GROUP BY a.username;
```

a) Použití 2 skalárních poddotazů

```
SELECT a.username,
(SELECT count(*) FROM all_objects b WHERE b.owner = a.username) počet,
(SELECT avg(object_id) FROM all_objects b WHERE b.owner = a.username) průměr,
FROM all_users a;
```

Tímto dojde ke zdvojnásobení práce.

Odstranění vnějšího spojení

Příklad – chceme ještě doplnit

```
SELECT a.username, count(*), avg(object_id)
FROM all_users a LEFT JOIN all_objects b
ON b.owner = a.username
GROUP BY a.username;
```

b) Použití 1 skalárního poddotazu s trikem

```
SELECT username,
        TO_NUMBER(SUBSTR(data,1,10)) počet,
        TO_NUMBER(SUBSTR(data,11)) průměr
FROM (SELECT a.username,
        (SELECT TO_CHAR(count(*), 'fm0000000009') || avg(object_id)
        FROM all_objects b WHERE b.owner = a.username) data,
FROM all_users a);
```

fm potlačí úvodní mezeru, kterou by číslo mohlo obsahovat, dále je číslo naformátováno na pevnou šířku. Pozor však u této techniky na případnou hodnotu NULL, pro jejíž obsluhu by bylo třeba použít funkci NVL.

Nicméně náročnost dotazu se opět dostala prakticky na polovinu.

Odstranění vnějšího spojení

Příklad – chceme ještě doplnit

```
SELECT a.username, count(*), avg(object_id)
FROM all_users a LEFT JOIN all_objects b
ON b.owner = a.username
GROUP BY a.username;
```

c) Použití objektového typu

```
CREATE OR REPLACE TYPE mtype AS OBJECT
        (pocet number,
         prumer number);

SELECT username, a.data.pocet, a.data.prumer
FROM (select username,
        (SELECT mtype (count(*), avg(object_id))
        FROM all_objects b WHERE b.owner = a.username) data,
FROM all_users a);
```

Přítomnost NULL nebude dotaz komplikovat a dotaz je zjednodušen, ale musel být vytvořen typ.

Agregace z několika tabulek

Příklad

```
SELECT a.username, nvl(b.cons_pocet, 0) počet_omezeni, nvl(c.tables_pocet) počet_tabulek
FROM all_users a
LEFT JOIN
        (SELECT owner, count(*) cons_pocet FROM all_constraints GROUP BY owner) b
        ON a.username = b. owner
LEFT JOIN
        (SELECT owner, count(*) tables_pocet FROM all_tables GROUP BY owner) c
        ON a.username = c. owner;
```

Použití skalárních poddotazů

```
SELECT a.username,
        (SELECT count(*) FROM all_constraints WHERE owner=username) počet_omezeni
        (SELECT count(*) FROM all_tables WHERE owner=username) počet_tabulek
FROM all_users;
```

Tento dotaz je nejen atraktivní pro velký nárůst výkonu, ale také pro svoji jednoduchost.

Výběr z různých tabulek

Smysl využití skalárních poddotazů pro výběr z různých tabulek je:

Při spojení řádků v tabulce nebo pohledu s určitou sadou jiných tabulek
– prostřednictvím dat v samotném dotazu vybereme tabulku pro spojení.

Použití skalárních poddotazů

```
SELECT object_type, object_name,
        decode(object_type,
        'TABLE', (select tablespace_name from user_tables
                  where table_name = object_name),
        'INDEX', (select tablespace_name from user_indexes
                  where index_name = object_name),
        …,
         'LOB', (select tablespace_name from user_segments
                  where segment_name = object_name),
        null) tablespace_name
        FROM user_objects a;
```

Jde o techniku spojení každého řádku v sadě výsledků s odlišnou tabulkou.

Architektury a techniky DS

Tvorba efektivních příkazů II

Přednáška č. 4 a 5

RNDr. David Žák, Ph.D.
Fakulta elektrotechniky a informatiky
david.zak@upce.cz

Motto:

- Databáze obsahuje data
- Uživatelé se na data ptají
- Hloupý systém přehrabuje data bez rozmyslu
- Chytrý systém obsahuje optimalizátor
- Optimalizátor optimalizuje – plánuje efektivní provedení příkazů
- Chytrý optimalizátor si něco pamatuje
- Někdy se snaží být moc chytrý a sestavuje mnoho plánů - to mu může trvat dlouho
- Můžeme mu ale poradit
- Proč tedy optimalizovat práci optimalizátoru?
- Protože jen hlupák si nenechá poradit…

Proč optimalizovat ?

- Snížit požadavky na systémové prostředky
- Zkrátit dobu zpracování

Předpoklady:

- Porozumění zpracování dotazů
- Znalost fyzického uspořádání dotazovaného objektu (schéma)
- Znalost všech složitostí jazyka SQL
- Vědět, co je k dispozici = znalost možností databáze
- Dobré porozumění cíle – jaké je vlastně zadání

Cíl

- vytvořit k původnímu SQL dotazu dotaz sémanticky ekvivalentní, který půjde efektivněji zpracovat

Trocha počtů

- Při 5 tabulkách existuje 5!=120 různých pořadí spojení tabulek
- pro každé spojení existuje mnoho různých plánů, používajících různé kombinace indexů, přístupových metod a metod spojování
- celkem pro 5 tabulek existuje několik tisíc plánů, což je dostatečně malé množství, aby bylo možné uvážit všechny
- pro 10 tabulek už však dostaneme přes 3,5 milionů možných pořadí a hodně přes 100 milionů možných plánů
- hrubá síla tedy není řešením

Jak moc je třeba optimalizovat?

- Oracle používá adaptivní vyhledávací strategii, která má zajistit to, že doba strávená hledáním optimálního řešení nepřesáhne zlomek doby potřebné na vyhodnocení
- pokud bude dotaz trvat 1 vteřinu, nemá smysl jej 10 vteřin optimalizovat
- pokud dotaz pravděpodobně poběží několik hodin, je vhodné věnovat několik vteřin nebo i minut na nalezení lepšího plánu

Optimalizace serveru

Cílem tvůrců IS využívajícího RDBMS je dosáhnout co nejlepšího výkonu při stávající hardwarové konfiguraci DB serveru. Za tímto cílem je nutné pečlivě provádět potřebné optimalizační kroky na následujících úrovních vývoje systému:

návrh

je nutno promyslet tvorbu datového modelu a

zvážit použití indexů pro urychlení SELECTů z tabulek

implementace

jedná se o optimalizaci použitých SQL příkazů

správa databáze

správné nastavení parametrů instance databáze (velikosti bufferů, redo-logu...)

Přehled architektury zpracování

Parser

Provádí syntaktickou a sémantickou analýzu dotazu.

Optimalizátor

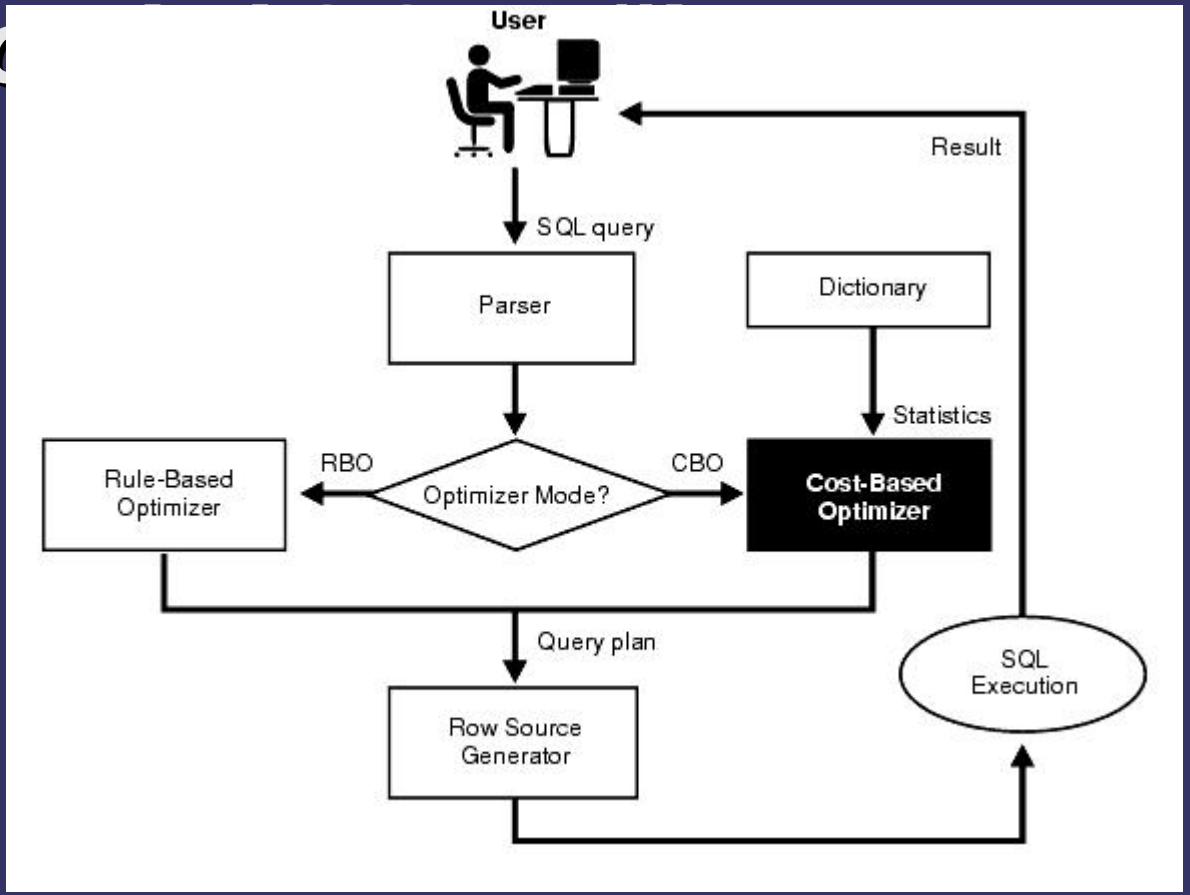
Navrhuje nejefektivnější způsob získání výsledku dotazu. K dispozici jsou dvě metody optimalizace (viz dále): cost-based (CBO) a rule-based (RBO).

Row Source Generator

Vstupem je optimální plán vytvořený opimalizátorem. Vytváří exekuční plán pro daný SQL dotaz ve formě stromu, jehož uzly jsou tvořeny jednotlivými *"row sources"* ("zdroje řádků" - vrací množinu řádků pro příslušný krok plánu).

SQL Execution Engine

Podle exekučního plánu vytvořeného pro daný SQL příkaz získá výsledek dotazu. "Spouští" postupně jednotlivé "row sources" vytvořené "Row Source" generátorem.



Optimalizátor

SQL příkaz lze zpracovat mnoha různými způsoby (např. tabulky mohou být spojovány v různém pořadí), což může velmi ovlivnit efektivitu a rychlost dotazů. Optimalizátor má za úkol určit nejefektivnější způsob zpracování.

Vývojář však může ovlivňovat rozhodování optimalizátoru pomocí "**hintů**" v SQL příkazech, čímž lze dosáhnout požadovaného chování na **specifických** datech.

Oracle podporuje dva druhy optimalizace:

1. rule-based - plán se sestavuje nezávisle na datech v tabulkách, pouze na základě tvaru dotazu. Zastaralý, pouze pro zpětnou kompatibilitu.

2. cost-based - optimalizace založena na vyhodnocení kvantitativního využívání zdrojů v průběhu zpracování dotazu (CPU, pamět, I/O...).
   Využívá statistiky o tabulkách a datech, které jsou uloženy v "Data Dictionary".

Kroky optimalizátoru

1. vyhodnocení výrazů a podmínek obsažených v příkazu

2. úprava příkazu - složitější dotazy mohou být transformovány do ekvivalentního tvaru výhodnějšího pro zpracování

3. výběr metody optimalizace (cost-based nebo rule-based)

4. výběr "přístupové metody" k získání dat z tabulek (pro každou tabulku zvlášť)

5. výběr pořadí spojení - v dotazech užívajících spojování tabulek se určí, v jakém pořadí budou tabulky spojovány

6. výběr "metod spojování"

Volba optimalizačního přístupu

Výběr optimalizačního přístupu je ovlivněn následujícími faktory:

- Inicializační parametr OPTIMIZER_MODE
- Parametr OPTIMIZER_GOAL - nabývá stejných hodnot jako OPTIMIZER_MODE a překrývá jeho hodnotu pro aktuální session
- CBO Statistiky v Data Dictionary
- SQL Hints

OPTIMIZER_MODE

Parametr OPTIMIZER_MODE definuje defaultní chování při výběru optimalizačního přístupu. Tato hodnota je platná pro instanci databáze. Jeho hodnoty mohou být následující:

CHOOSE

Defaultní hodnota. Optimalizátor vybírá mezi cost-based approach a rule-based přístupem podle toho, zda jsou dostupné statistiky v Data Dictionary.

ALL_ROWS

Užije se cost-based přístup pro všechny příkazy bez ohledu na existenci statistik (statistiky se "domyslí"). Optimalizuje se throughput.

FIRST_ROWS_n

Užije se cost-based přístup s optimalizací odezvy - tj. zpracování prvních n řádků. (n se může rovnat 1, 10, 100, nebo 1000).

FIRST_ROWS

Kombinuje se cost-based přístup s aplikací heuristik pro rychlé zpracování několika prvních řádků.

RULE

Užije se rule-based přístup.

Nastavení parametru se provádí příkazem ALTER SESSION SET OPTIMIZER_MODE = <hodnota>

Hinty

Hinty umožňují přinutit optimalizátor k použití určité přístupové metody, pořadí spojení... Tím lze ovlivnit exekuční plán, který Oracle zvolí. Zadávají se ve formě komentáře uvnitř SQL příkazu s následující syntaxí:

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

Pokud není zadání hintů korektní, Oracle nenahlásí chybu, ale ignoruje je.

U komplexních dotazů může nastat situace, že optimalizátor nepoužije zadané hinty v případě, že se neslučují s ostatními přístupovými metodami, které musel sám určit.

Hinty pro určení cíle optimalizace

- ALL_ROWS - optimalizuje "throughput"

- FIRST_ROWS(n) - optimalizuje odezvu - volí plán, který vrací nejrychleji prvních n řádků (nelze použít a je tedy ignorován v případech, že je nejprve nutno získat všechny řádky před výstupem prvního - např. dotazy s GROUP BY nebo ORDER BY...)

- CHOOSE - vybírá mezi RBO a CBO

Hinty pro výběr přístupové metody

- FULL(table_alias) - použít "full scan" na uvedenou tabulku
- INDEX(table_alias index_name) - použít "index-scan" s uvedeným indexem
- INDEX_FFS(table_alias index_name) - použít "fast full index-scan" s uvedeným indexem

Hinty pro pořadí spojování tabulek

- ORDERED - tabulky jsou spojovány ve stejném pořadí, v jakém jsou uvedeny v klauzuli FROM

- STAR - největší tabulka je připojena až jako poslední a to pomocí "nested loops" (aplikovatelné pouze, pokud dotaz přistupuje alespoň ke třem tabulkám)

Hinty pro operaci spojení

- USE_NL(table1 table2) - vynucuje spojení pomocí "nested loops"
- USE_MERGE(table1 table2) - vynucuje spojení pomocí "sort-merge join"
- USE_HASH(table1 table2) - vynucuje spojení pomocí "hash join"
- LEADING(table) - specifikovaná tabulka bude první v pořadí spojení

Hinty - příklad

```
SELECT jmeno, prijmeni, plat
  FROM ucitel
  WHERE pohlavi='M';
```

Optimalizátor by v takovémto případě zřejmě zvolil full table scan, protože pohlaví může obsahovat pouze dvě hodnoty, tedy vrácených řádků by měla být velká část ze všech možných. Pokud však víme, že učitelů - mužů je hodně málo (například ukládáme pouze učitele z mateřských škol), pak si můžeme pomocí hintu vynutit rychlejší přístup - index scan.

```
SELECT /*+ INDEX(ucitel pohlavi_index) */ jmeno, prijmeni, plat
  FROM ucitele
  WHERE pohlavi='M';
```

Hinty se zapisují přímo za klíčové slovo SELECT, pokud chceme použít více hintů pro jeden dotaz, oddělujeme je mezerami.

Např. `SELECT /*+ ORDERED USE_MERGE(tabulka1 tabulka2) */ ….`

Generování CBO statistik

Statistiky jsou užívány při cost-based optimalizaci k určení nejefektivnějšího plánu. Umožňují odhadnout "cenu" různých přístupových a spojovacích metod pro konkrétní tabulky a vybrat tak nejoptimálnější přístup.

Statistiky obsahují např. následující údaje:

* Table statistics
  - počet řádků
  - počet bloků
  - průměrná délka řádku
* Column statistics
  - počet různých hodnot
  - počet řádků s hodnotou NULL ve sloupci
* System statistics
  - využití I/O
  - využití CPU

Statistiky by měly být generovány pro každý objekt (tabulky, indexy, clustery), který je využíván v dotazech a měly by být přegenerovány po každé větší změně na objektu (např. při vložení velkého množství dat do tabulky). Statistiky lze generovat buď pomocí package DBMS_STATS nebo příkazem ANALYZE

Sběr statistik

- Při odhadu selektivity a dalších faktorů vedoucích ke stanovení ceny plánu optimalizátor využívá, vedle základních statistik a histogramů, posbíraných pro datové objekty (tabulky, jejich sloupce a indexy), i ***systémové statistiky*** dokumentující hospodaření s operační pamětí a výpočetní kapacitou procesoru.
- Sběr statistik představuje nezanedbatelnou režii a může tedy ovlivnit průchodnost systému pro další prováděné úlohy.
- Použití neaktuálních statistik může vést k volbě neadekvátního plánu provedení SQL příkazu, což také může vést ke snížení průchodnosti systému.
- Problém je jak rozhodnout, které statistiky jsou zastaralé a musí být tudíž aktualizovány.
- Zatímco dříve se spíše uvažovalo o tom, zda je nutné znovu analyzovat danou tabulku nebo index, zvyšující se výpočetní síla serverů nabízí extenzivní uvažování. Předmětem sběru statistik jsou pak spíše „všechny objekty daného schématu“ nebo „všechny objekty celé databáze“.

Sběr statistik

- Základní potřebou je průběžně udržovat statistiky aktuální, ale s co nejmenším vlivem na průchodnost ostatních úloh.
- Je třeba naplánovat aktualizaci statistik na období klidového provozu a aktualizovat pouze ty statistiky, které si to zaslouží, které od posledního sběru **zastaraly.**
- Pro potřebu vyhodnocení, zda statistiky nad danou tabulkou ztratily na aktuálnosti, lze zapnout automatické sledování změn dat v tabulce. Statistika nad tabulkou může být například prohlášena za neaktuální, když se data v tabulce od posledního sběru změní v rozsahu cca 10%.
- Proces pro periodickou aktualizaci zastaralých statistik nad objekty celé databáze je zařazen do fronty úloh při vytvoření databáze. Jeho spuštění je naplánováno na období, které je pomocí parametrů databáze prohlášeno za období s nízkým transakčním provozem. Implicitně to je noc a celé víkendové dny.

Osnovy

- Oracle poskytuje prostředky, pomocí nichž se buď jednorázově, nebo po zvolenou dobu, zaznamenávají zpracovávané SQL příkazy společně
s plány, které pro ně optimalizátor zvolil.

- Záznamy mají podobu **znovupoužitelných osnov (tzv. „outlines“) a zaznamenávají se do specializované části** datového slovníku, kterou tvoří samostatné schéma jménem **OUTLN.**

- Zaznamenané osnovy je možné seskupovat do skupin, které se nazývají **kategorie.**

- **Na databázovém stroji**, který slouží jako etalon pro skupinu zákaznických instalací, můžeme při čerstvých statistikách vytvořit různé sady osnov pro období s různým typem zátěže a pro databáze s různou robustností.

Osnovy

- V rutinním provozu je možné „zapnout“ používání dané kategorie osnov.
- Po příchodu nového příkazu optimalizátor vyhodnotí, zda je možné pro příkaz využít již existující kontext kurzoru.
- Jestliže ne, vytvoří se nový kontext kurzoru. Pak se prohledá zapnutá kategorie osnov, zda v ní je osnova zpracovávaného příkazu. Jestliže ano, použije se tato osnova jako velmi striktní vodítko při stavbě nového plánu.
- Může se tak ušetřit na režii údržby aktuálních statistik, protože optimalizátor, vedený návodem v osnově je nepotřebuje.
- Může se snížit výpočetní náročnost stavby plánu, protože při použití osnovy optimalizátor nemusí téměř nic rozhodovat.

Profily

- Verze Oracle 10g přinesla nový typ databázového objektu – **profil SQL příkazu.**

- statistiky o zpracování SQL příkazů se ukládají do persistentního úložiště označovaného AWR (Automatic Workload Repository). Zde jsou uchovávány po dobu *N dnů (implicitně 7).*

- Vedle statistik vytvořených pro tabulky, na které příkaz odkazuje, je profil daného příkazu další informací, kterou optimalizátor využije ke zpřesnění svých odhadů cen jednotlivých plánů přicházejících v úvahu.

- SQL profil nenahrazuje osnovu plánu. Osnova vede optimalizátor tvrdě k jedné volbě. Profil „jen“ poskytuje optimalizátoru informaci o historii zpracování daného příkazu - s jakými plány byl proveden, jaké byly časové charakteristiky provedení příkazu apod. Optimalizátor tuto informaci použije při stavbě nového plánu.

Ladění exekučního plánu

Exekuční plán je posloupnost kroků, které Oracle provádí při zpracování SQL příkazu. Obsahuje také zvolené přístupové metody k tabulkám a pořadí a metody spojování. Plán má formu stromu, jehož vrcholy jsou tvořeny jednotlivými "row source". Jsou v něm uvedeny následující informace:

* pořadí, v jakém dotaz přistupuje k tabulkám
* přístupová metoda pro každou tabulku
* spojovací metoda pro každou operaci spojení
* operace na datech (např. filter, agregační funkce...)

Zobrazení exekučního plánu

Existuje několik možností zobrazení exekučního plánu:

1. příkaz EXPLAIN PLAN
2. SQL Trace
3. autotrace v SQL*Plus
4. isqlplus WWW konzola (nebo WWW konzola APEX)
5. SQL Developer (tlačítko)

EXPLAIN PLAN

Před použitím EXPLAIN PLAN je nutno vytvořit tabulku, do které se plán uloží.
K tomu je možné použít skript UTLXPLAN.SQL
(umístění: $ORACLE_HOME/rdbms/admin/).
Skript vytvoří tabulku nazvanou PLAN_TABLE v aktuálním schématu.

Syntaxe příkazu:

```
EXPLAIN PLAN
  [INTO <different_table>]
  [SET STATEMENT_ID = <identifier>]
   FOR <SQL_Statement>
```

* klauzulí INTO je možné specifikovat jinou cílovou tabulku než PLAN_TABLE
* SET STATEMENT_ID umožňuje určit ID, které budou mít řádky tabulky PLAN_TABLE patřící k vytvořenému plánu

Tabulka PLAN_TABLE

Výčet některých sloupců tabulky PLAN_TABLE :

* ID - číslo přiřazeno kroku plánu
* STATEMENT_ID - hodnota volitelného parametru STATEMENT_ID v příkazu EXPLAIN PLAN
* OPERATION - jméno operace prováděné v daném kroku (SELECT STATEMENT, FILTER, MERGE JOIN, NESTED LOOPS, ...)
* OPTIONS - druh operace popsané ve sloupci OPERATION (FULL SCAN,...)
* OBJECT_NAME - jméno tabulky / indexu
* OPTIMIZER - aktuální mód optimalizátoru
* COST - cena operace odhadnutá optimalizátorem při cost-based přístupu
* CARDINALITY - odhad počtu řádků zpracovaných v průběhu operace
* CPU_COST - využití CPU v průběhu operace (pouze cost-based přístup)

Zobrazení exekučního plánu

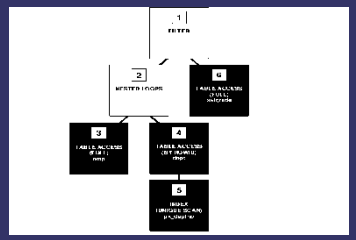
Příklad

Následující SQL příkaz vybírá job, salary a department name pro všechny zaměstnance, jejichž plat nepadne do doporučeného rozmenzí.

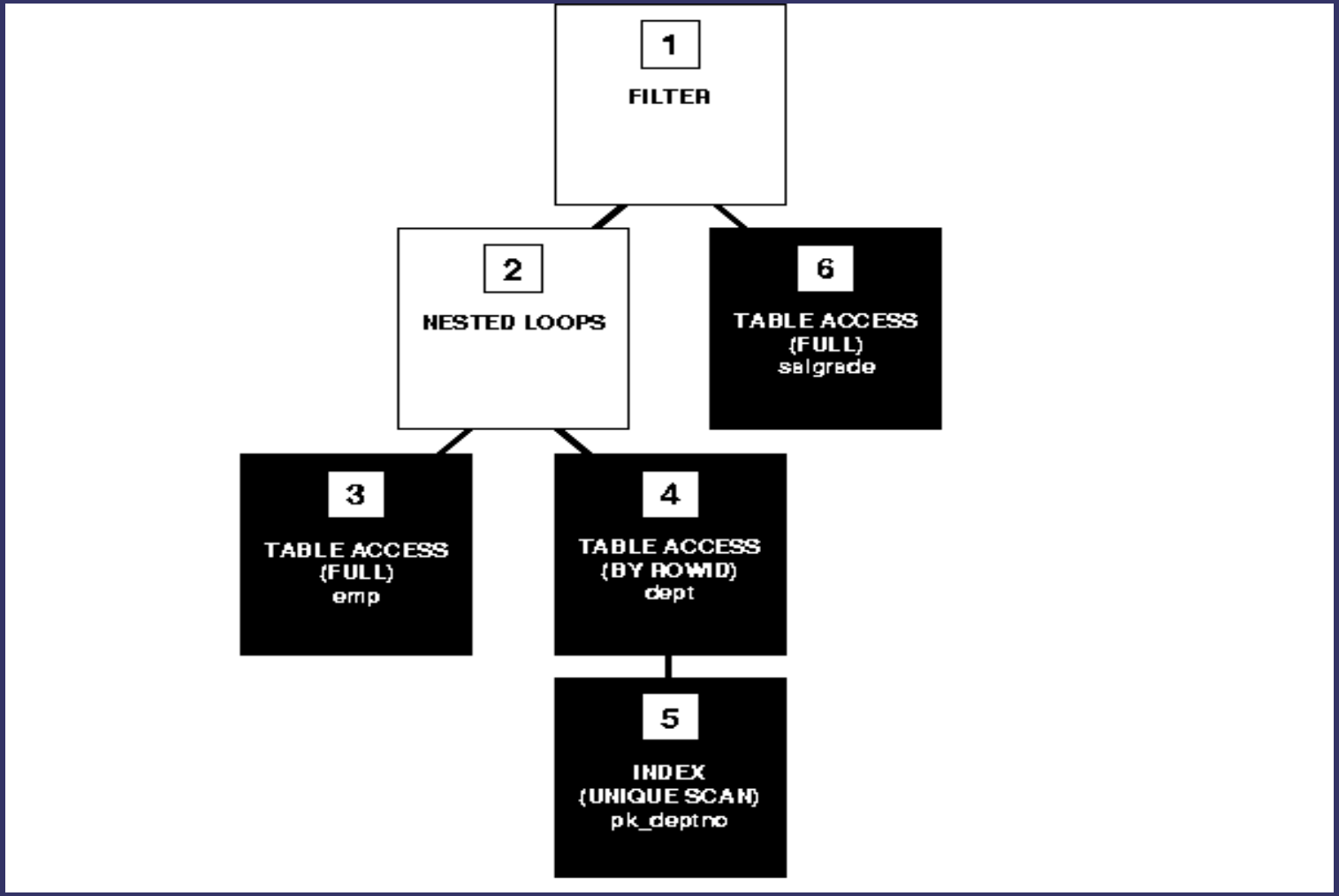
```
EXPLAIN PLAN FOR
   SELECT employee_id, job_id, salary, department_name
    FROM employees, departments
    WHERE employees.department_id = departments.department_id
        AND NOT EXISTS
          (SELECT *
            FROM jobs
           WHERE employees.salary BETWEEN min_salary AND max_salary);
```

Výsledek může vypadat takto:

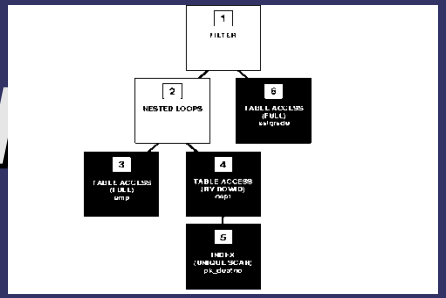
```
ID    OPERATION           OPTIONS        OBJECT_NAME
-------------------------------------------------------------
0     SELECT STATEMENT
1       FILTER
2         NESTED LOOPS
3           TABLE ACCESS  FULL           EMPLOYEES
4           TABLE ACCESS  BY ROWID       DEPARTMENTS
5             INDEX       UNIQUE SCAN    PK_DEPARTMENT_ID
6         TABLE ACCESS    FULL           JOBS
```



Zobrazení exekučního plánu



Rozbor exekučního pl



Každý z kroků plánu vrací množinu záznamů (row set) a jako svůj vstup bere buď záznamy získané přímo z databáze (na diagramu jsou tyto kroky vyznačeny černě), nebo přebírá a dále zpracovává výstup jiného kroku.

- kroky 3 a 6 čtou všechny řádky tabulek EMPLOYEES a JOBS
- krok 5 vyhledá v indexu pk_department_id každou hodnotu department_id vrácenou krokem 3. Vrací ROWID příslušných řádků v tabulce DEPARTMENTS
- krok 4 získá z tabulky DEPARTMENTS záznamy, které přísluší ROWID vráceným krokem 5
- krok 2 provádí operaci spojení vnořenými cykly, kdy spojuje řádky z kroků 3 a 4 a výsledky předává kroku 1
- krok 1 provádí filter - přebírá řádky z kroku 2 a filtruje ty, které nemají "protějšek" ve výsledku kroku 6

Nástroj EXPLAIN PLAN FOR

Zjištění plánu dotazu pro dotaz SQL, kdyby k jeho spuštění došlo nyní v aktuálním prostředí.

Dříve i v budoucnu mohly(mohou) být dotazy provedeny odlišně – s jiným plánem.

```
SELECT trpaslici.jmeno, count(*) FROM trpaslik.trpaslici
JOIN trpaslik.tezby ON trpaslici.id=tezby.id_trpaslika
JOIN trpaslik.sachty ON sachty.id=tezby.id_sachty
JOIN trpaslik.rudy ON sachty.id_rudy=rudy.id
WHERE tezby.plan<rudy.norma*8
GROUP BY trpaslici.jmeno;
```

```
Execution Plan
----------------------------------------------------------
0          SELECT STATEMENT Optimizer=ALL_ROWS (Cost=15 Card=6 Bytes=25 2)
1       0      SORT (GROUP BY) (Cost=15 Card=6 Bytes=252)
2       1          HASH JOIN (Cost=14 Card=6 Bytes= 252)
3       2              HASH JOIN (Cost=10 Card=6 Bytes=150)
4       3                  HASH JOIN (Cost=7 Card=5 Bytes=75)
5       4              TABLE ACCESS (FULL) OF 'SACHTY' (TABLE) (Cost=3 Ca rd=5 Bytes=40)
6       4             TABLE ACCESS (FULL) OF 'RUDY' (TABLE) (Cost=3 Card =7 Bytes=49)
7       3                  TABLE ACCESS (FULL) OF 'TEZBY' (TABLE) (Cost=3 Card=127 Bytes=1270)
8       2              TABLE ACCESS (FULL ) OF 'TRPASLICI' (TABLE) (Cost=3 Card=7 Bytes=119)
```

Nástroj EXPLAIN PLAN FOR

Význam hodnot v sestavě plánu dotazu v případě využití optimalizátoru CBO (Cost Based Optimization)

Cost (náklady) – pro jeden dotaz generuje mnoho různých průběhů/plánů a každému přiřadí určitou hodnotu představující náklady na konkrétní průběh, plán dotazu s nejnižšími náklady je vyhodnocen jako nejúspěšnější

Card (Mohutnost) – představuje očekávaný počet výstupních řádků z daného kroku plánu dotazu

Bytes (Bajtů) – předpokládaná velikost vracených dat v B, hodnota závisí na počtu řádků a očekávané šířce řádku

Pokud tyto hodnoty nejsou uvedeny, byl dotaz spuštěn optimalizátorem RBO

Autotrace v SQL*Plus

SQL*Plus nabízí možnost automatického vygenerování a zobrazení exekučního plánu po každém provedeném DML příkazu. Tato funkčnost se aktivuje příkazem **SET AUTOTRACE ON** Výsledek každého dotazu bude potom přibližně následující:

```
SQL> SELECT D.DNAME, E.ENAME, E.SAL, E.JOB FROM EMP E, DEPT D  WHERE E.DEPTNO = D.DEPTNO

DNAME          ENAME             SAL JOB
-------------- ---------- ---------- ---------
ACCOUNTING     CLARK            2450 MANAGER
ACCOUNTING     KING             5000 PRESIDENT
RESEARCH       SMITH             800 CLERK
SALES          ALLEN            1600 SALESMAN
....

14 rows selected.

Execution Plan
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=CHOOSE
1    0   MERGE JOIN
2    1     SORT (JOIN)
3    2       TABLE ACCESS (FULL) OF 'DEPT'
4    1     SORT (JOIN)
5    4       TABLE ACCESS (FULL) OF 'EMP'

Statistics
----------------------------------------------------------
148  recursive calls
  4  db block gets
 24  consistent gets
  6  physical reads
 43  redo size
591  bytes sent via Oracle Net to client
256  bytes received via Oracle Net from client
  3  Oracle Net roundtrips to/from client
  2  sort (memory)
  0  sort (disk)
 14  rows processed
```

Pozn.: Pokud chceme zobrazit pouze exekuční plán se statistikami bez případného výsledku dotazu, lze použít pro zapnutí funkce AUTOTRACE příkaz SET AUTOTRACE TRACEONLY

SQL trace

Utilita SQL trace společně umožňuje sledovat efektivitu jednotlivých SQL příkazů, které jsou spouštěny aplikací. Je možné ji aktivovat pro session nebo celou instanci (doporučuje se pouze pro session kvůli zátěži navíc).

Pro každý příkaz generuje následující statistiky:

- počet spuštění příkazu
- počet "physical reads" a "logical reads"
- počet zpracovaných řádek
- počet výpadků na library cachi (sdílené PL/SQL a SQL příkazy)
- každý commit a rollback

Všechny generované statistiky se ukládají do trace souborů. Pro čitelné formátování těchto souborů slouží program TKPROF. Ten také navíc umožňuje zobrazení exekučního plánu pro jednotlivé SQL dotazy.

Informace získané z trace souborů umožňují lokalizovat dotazy, které spotřebují nejvíce prostředků a je proto třeba je vyladit.

Aktivace SQL trace

- umístění trace souborů je dáno parametrem USER_DUMP_DEST, který lze dynamicky nastavit příkazem

```
ALTER SYSTEM SET USER_DUMP_DEST= newdir;
```

- SQL trace pro aktuální session se aktivuje příkazem

```
ALTER SESSION SET SQL_TRACE = true;
```

- SQL trace pro celou instanci se aktivuje příkazem

```
ALTER SYSTEM SET SQL_TRACE = true;
```

Tipy pro efektivní dotazy

- Používejte množinový operátor **MINUS** místo vložených dotazů s operátorem **EXISTS**
- Využívejte **SQL analytických funkcí** - Oracle analytické funkce jsou optimalizovány pro různé agregace (např. při práci s datovými sklady)
- **Indexujte hodnoty NULL** – máte-li dotaz s podmínkou na hodnotu NULL, může být vhodné indexovat daný sloupec s hodnotami NULL. Řešením může být function-based index, který indexuje pouze hodnoty ve sloupci s NULL hodnotami.
- **Omezte používání NOT IN or HAVING –** použití dotazů s operátorem NOT EXISTS může být rychlejší.
- Namísto operátoru **LIKE** se pokuste používat porovnání na rovnost v případech, kdy je to možné.

Tipy pro efektivní dotazy

- **přepište NOT EXISTS a NOT IN vnořené dotazy jako outer joins (vnější spojení)**

```
SELECT book_key FROM book
    WHERE book_key NOT IN (select book_key from sales);
```

Tento dotaz přepíšeme s využitím vnějšího spojení, přičemž nás zajímají ty řádky tabulky book, pro které nebyl nalezen vhodný řádek v tabulce sales.

Nepoužívá se vnořený dotaz a výsledkem je rychlejší exekuční plán.

```
SELECT b.book_key FROM book b RIGHT JOIN sales s
    ON    b.book_key = s.book_key
    AND   s.book_key IS NULL;
```

Tipy pro efektivní dotazy

- **Ponechte označení sloupců samostatně**

– neprovádějte kalkulace na indexovaných sloupcích, **pokud nepoužíváte function-based indexy**

- Nevhodné jsou následující podmínky:

```
where salary*5          > :myvalue

where substr(ssn,7,4)    = '1234'


where to_char(mydate,mon) = 'january'
```

lépe extract(month from mydate)=1

Tipy pro efektivní dotazy

- **Nikdy nemixujte datové typy** – v případech, kdy používáte sloupce numerického typu, nepoužívejte uvozovky, naopak pro znakové typy char uvozovky používejte.

- Nevhodné příklady (porovnání mixovaných datových typů vyžaduje implicitní konverze datových typů při zpracování podmínky na každém řádku:

Správně:

```
where cust_nbr = ’123’          where cust_nbr = 123

where substr(ssn,7,4) = 1234    where substr(ssn,7,4) = ’1234’
```

Tipy pro efektivní dotazy

- **Používejte decode a case** – agregace s "decode" nebo "case" funkcemi mohou minimalizovat čas a opakované výběry tabulek

- Používejte aliasy pro označení tabulek vždy, když se odkazujete na sloupce

Tipy pro efektivní dotazy

Představme si následující příklad opakovaně provádějící shodné vnořené dotazy:

```
select  store_name,  sum(quantity) store_sales,
        (select sum(quantity) from sales)/
                (select count(*) from store) avg_sales
        from    store  s,    sales  sl
where    s.store_key = sl.store_key
having  sum(quantity) > (select sum(quantity) from sales)/(select count(*)
        from store) group by store_name;
```

Řešení:

1) Příkaz SELECT s klauzulí WITH pro jednotlivé vnořené dotazy, které by jinak byly opakovaně volány

2) Oracle zavedl technologii **Global Temporary Tables (GTT)**, princip je obdobný

Tipy pro efektivní dotazy

Ad 1) SELECT s klauzulí WITH:

```
WITH t1 as
        select sum(quantity) all_sales from stores,
 t2 as
        select count(*) nbr_stores  from stores,
 t3 as
        select store_name, sum(quantity) store_sales
        from store natural join sales
 SELECT    store_name  from    t1,    t2,    t3
     where    store_sales > (all_sales / nbr_stores);
```

ad 2) pomocí **Global Temporary Tables (GTT)**

```
create global temporary table t1 as select sum(quantity) all_sales from stores;
create global temporary table t2 as select count(*) nbr_stores  from stores;
create global temporary table t3 as select store_name, sum(quantity) store_sales
    from store natural join sales;

select    store_name  from    t1,    t2,    t3
    where    store_sales > (all_sales / nbr_stores);
```

Materializované pohledy

- vlastnosti:
  - bývají menší
  - mohou obsahovat předpočítané hodnoty (vyhodnocené agregační funkce)
  - definují se podmínky pro přepočtení pohledu
- transformace:
  - spočívá v nahrazování čtení z několika tabulek čtením z materializovaných pohledů
  - může výrazně urychlit vyhodnocení dotazu
  - nemusí být vždy urychlením (např. základní tabulka je vhodně indexovaná, ale materializovaný pohled ne)

Materializované pohledy (příklad)

- Máme 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
```

- Dotaz

```
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
```

- Po přepsání

```
SELECT CUSTOMER.CUST_NAME, SALES_SUMMARY.MONTH, SALES_SUMMARY.AMT
    FROM        CUSTOMER, SALES_SUMMARY
    WHERE       CUSTOMER.CUST_ID = SALES_SUMMARY.CUST_ID
```

Architektury a techniky DS

Efektivní programování v jazyce PL/SQL

Přednáška č. 6

RNDr. David Žák, Ph.D.
Fakulta elektrotechniky a informatiky
david.zak@upce.cz

Jazyk PL/SQL

- Hlavním omezením jazyka SQL je, že se jedná o neprocedurální jazyk

- V praxi to znamená, že se příkazy jazyka SQL provádějí sekvenčně bez možnosti klasických programátorských konstrukcí (cykly, podmínky, procedury, funkce, případně objektové programování)

- Říkáme CO, nikoli JAK

- Proto většina databázových platforem nabízí rozšíření umožňující naprogramovat i ty nejsložitější algoritmy pro práci s daty

- PL/SQL (Transaction Procesing Language)

Jazyk PL/SQL

- Umožňuje deklarovat konstanty, proměnné, kurzory
- Nabízí podporu dynamických deklarací
- Podpora transakčního zpracování
- Chybové stavy procesu je možné ošetřit pomocí výjimek
- Podpora modularity (vkládání modulů i do sebe)
- Podporuje dědičnost

- Existují různé vývojové nástroje
  - Oracle JDeveloper
  - Oracle SQL Developer
  - …
  - PL/SQL Developer
  - Rapid SQL
  - SQL Programmer
  - SQL Navigator a TOAD

Struktura jazyka PL/SQL

Proměnné v PL/SQL

1. Je nutné před prvním použitím vždy deklarovat

2. Během deklarace je možné proměnnou inicializovat, případně omezit, že nesmí nabývat hodnoty NULL

Příklad

```
DECLARE
    v_promenna1 NUMBER(3);
    v_promenna2 NUMBER NOT NULL DEFAULT 88;
    v_promenna3 NUMBER := 77;

BEGIN
    v_promenna1 := 33;
    DBMS_OUTPUT.PUT_LINE(v_promenna1);
    DBMS_OUTPUT.PUT_LINE('Hodnota proměnné 2 je' || v_promenna2);

END;
```

Proměnné v PL/SQL

Typ proměnné podle jiné proměnné

Příklad

```
DECLARE
        v_text VARCHAR2;
        v_text2 v_text%TYPE;
```

Typ proměnné podle sloupce tabulky

Příklad

```
DECLARE
        v_text3 ucitel.jmeno%TYPE;
```

Vnořené bloky PL/SQL

```
DECLARE
    … deklarační sekce
    ……

BEGIN
    … výkonná sekce

        DECLARE
            … deklarační sekce
        BEGIN
            … výkonná sekce
        EXCEPTION
            … sekce pro zpracování výjimek
        END;

EXCEPTION
    … sekce pro zpracování výjimek
    ……

END;
```

Programový kód ve vnitřním bloku může používat proměnné deklarované ve vnějším bloku

Komentáře v PL/SQL

Příklad

```
DECLARE
-- jednořádkový komentář
    v_promenna1 NUMBER(3);
    v_promenna2 NUMBER NOT NULL DEFAULT 88;
    v_promenna3 NUMBER := 77;

BEGIN
/* víceřádkový
komentář
*/
    v_promenna1 := 33;
    DBMS_OUTPUT.PUT_LINE(v_promenna1);
    DBMS_OUTPUT.PUT_LINE('Hodnota proměnné 2 je ' || v_promenna2);

END;
```

Práce s daty v tabulkách v PL/SQL

Dotaz pro získávání dat

```
SELECT [* | seznam_položek]
    INTO [seznam_položek nebo proměnná typu záznam]
    FROM název_tabulky
    WHERE podmínky_výběru
```

Podmínkou úspěšnosti takto zadaného dotazu je, aby byl vrácen vždy jen jeden řádek

Práce s daty v tabulkách v PL/SQL

Příklad

```
DECLARE
    v_jmeno ucitel.jmeno%TYPE;
    v_Id ucitel.Id%TYPE;

BEGIN
    SELECT jmeno, Id INTO v_jmeno, v_Id
    FROM ucitel WHERE Id=2;

    -- výpis hodnot proměnných
    DBMS_OUTPUT.PUT_LINE('Jméno ' || v_jmeno);
    DBMS_OUTPUT.PUT_LINE('Id ' || v_Id);

END;
```

Samozřejmě možno s daty i manipulovat příkazy INSERT, UPDATE, DELETE

Příklad ošetření výjimek v PL/SQL

Příklad

```
DECLARE
    v_jmeno ucitel.jmeno%TYPE;
    v_Id ucitel.Id%TYPE;

BEGIN
    SELECT jmeno, Id INTO v_jmeno, v_Id
    FROM ucitel WHERE Id=2;

    DBMS_OUTPUT.PUT_LINE('Jméno' || v_jmeno);
    DBMS_OUTPUT.PUT_LINE('Id' || v_Id);

EXCEPTION
     -- ošetření výjimky při nenalezení dat
     WHEN NO_DATA_FOUND THEN DBMS_OUTPUT.PUT_LINE('Data nenalezena');

     -- ošetření výjimky při nalezení více řádků splňujících podmínku
     WHEN TOO_MANY_ROWS THEN DBMS_OUTPUT.PUT_LINE('Mnoho řádků');

END;
```

Řízení toku programu - podmínky

Zápis podmínky

```
IF podmínka
THEN
  posloupnost_příkazů
END IF;
```

nebo

```
IF podmínka
THEN
  posloupnost_příkazů1
ELSE
  posloupnost_příkazů2
END IF;
```

nebo

```
IF podmínka1
THEN
  posloupnost_příkazů1
ELSIF podmínka2
THEN
  posloupnost_příkazů2
ELSE
  posloupnost_příkazů3
END IF;
```

Řízení toku programu

Příkaz CASE pro vícenásobné větvení programu

```
CASE
WHEN podmínka1 THEN  posloupnost_příkazů1;
WHEN podmínka2 THEN  posloupnost_příkazů2;
..
WHEN podmínkaN THEN  posloupnost_příkazůN;
[ ELSE posloupnost_příkazůN+1; ]
END CASE;
```

Podmínka může být i například v_promenna BETWEEN 1 AND 5

Řízení toku programu - cykly

Jednoduchý cyklus LOOP

```
LOOP
        posloupnost_příkazů
        IF podmínka THEN
        .. ukončuje se příkazem EXIT
        END IF;
END LOOP;
```

nebo

```
LOOP
        posloupnost_příkazů
        EXIT WHEN podmínka;
END LOOP;
```

Příklad

```
DECLARE
V_pocet NUMBER := 0;
BEGIN
LOOP
        v_pocet:=v_pocet+1;
        IF v_pocet >=100 THEN
        EXIT;
        END IF;
END LOOP;
        DBMS_OUTPUT.PUT_LINE(' v_pocet ' || v_pocet );
END;
```

Řízení toku programu - cykly

Cyklus FOR s čítačem

```
FOR počítadlo IN [REVERSE] Nejnižší_hodnota .. Nejvyšší_hodnota
LOOP
        posloupnost_příkazů1
END LOOP;
```

Příklad

```
BEGIN
FOR v_citac IN 1..3
LOOP
        DBMS_OUTPUT.PUT_LINE('v_citac ' || v_citac );
END LOOP;
END;
```

Řízení toku programu - cykly

Cyklus WHILE s podmínkou na začátku

```
WHILE podmínka
      LOOP
          posloupnost_příkazů
      END LOOP;
```

Příklad

```
DECLARE
V_pocet NUMBER := 0;
BEGIN
WHILE v_pocet < 100
      LOOP
          v_pocet:=v_pocet+1;
      END LOOP;
      DBMS_OUTPUT.PUT_LINE(' v_pocet ' || v_pocet );
END;
```

Kurzory

- Privátní pracovní oblasti, které jsou databázovým serverem vytvořeny pro každý příkaz SQL
  - Implicitní kurzory jsou vytvářeny automaticky databázovým serverem, není nutné je otevírat, zavírat, deklarovat nebo z něj načítat data,
  - Explicitní – deklarované programátorem

Základní kroky pro práci s explicitními kurzory

- Deklarace kurzoru
- Otevření kurzoru
- Výběr dat prostřednictvím kurzoru
- Uzavření kurzoru

Kurzory - syntaxe

- Deklarace kurzoru

  CURSOR <název kurzoru> IS <příkaz SELECT>;

- Otevření kurzoru

  OPEN <název kurzoru>;

- Výběr dat prostřednictvím kurzoru (opakovat v cyklu)

  FETCH <název kurzoru> INTO <seznam proměnných>;

- Uzavření kurzoru

  CLOSE <název kurzoru>;

Kurzory – testování stavu

Pro testování stavu kurzoru jsou k dispozici atributy

%ROWCOUNT

Zjištění pořadového čísla aktuálního záznamu
(pokud nebyl vybrán žádný, je hodnota 0)

%FOUND

Pokud poslední příkaz FETCH načetl nějaký záznam, má atribut hodnotu TRUE
Používá se pro zjišťování konce cyklu

%NOTFOUND

Používá se pro zjišťování konce cyklu

%ISOPEN

Pokud je kurzor otevřen, má hodnotu TRUE

Použití: **<název kurzoru>%ROWCOUNT**

Práce s kurzory

Příklad s využitím explicitního kurzoru

```
DECLARE
    v_jmeno ucitel.jmeno%TYPE;
    v_Id ucitel.Id%TYPE;

    CURSOR k1 IS
        SELECT jmeno, Id FROM ucitel;

BEGIN
    OPEN k1;
    LOOP
        FETCH k1 INTO v_jmeno, v_Id;
        EXIT WHEN k1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Jméno ' || v_jmeno || ', Id ' || v_Id);
        END LOOP;
    CLOSE k1;
END;
```

Záznamy

Struktura typu záznam zapouzdřuje více položek i rozdílných datových typů.

Deklarace záznamu

```
DECLARE
    TYPE <název proměnné typu záznam> IS RECORD
        (           <název atributu>    <datový typ>
                    [, <název atributu>    <datový typ> …]
        );
```

Příklad

```
DECLARE
    TYPE rec_ucitel IS RECORD
        ( jmeno     ucitel.jmeno%TYPE;
          Id        ucitel.Id%TYPE;
        );
```

Nebo po zjednodušení jen

```
DECLARE
     rec_ucitel          ucitel%ROWTYPE;
```

Práce s kurzory a záznamy

S využitím záznamů můžeme s kurzory pracovat mnohem efektivněji

Cyklus FOR s explicitním kurzorem

(kurzor v tomto případě nemusíme ani otevírat ani zavírat, dokonce ani cyklicky vybírat data pomocí příkazu FETCH, všechny tyto úkony za nás provede server standardně)

Příklad

```
DECLARE
        rec_ucitel    ucitel%ROWTYPE;

        CURSOR k1 IS
                SELECT jmeno, Id FROM ucitel;

BEGIN
        FOR rec_ucitel IN k1
        LOOP
                DBMS_OUTPUT.PUT_LINE('Jméno ' || rec_ucitel .jmeno || ', Id ' || rec_ucitel.Id);
        END LOOP;

END;
```

Práce s kurzory

Příkaz SELECT … INTO … musí vrátit alespoň jeden a nejvýše jeden řádek

Následující příklad ukazuje využití implicitního kurzoru pro sady výsledků s omezeným počtem řádků (řekněme méně než 100)

```
For x in (select … from … where …)
Loop
    Process …
End loop;
```

```
BEGIN
        FOR x IN (SELECT jmeno, Id FROM trpaslici)
        loop
        DBMS_OUTPUT.PUT_LINE('Jméno ' || x.jmeno || ', Id ' || x.Id);
        END LOOP;
END;
```

Kurzory s parametry

Kurzor můžeme rozšířit o parametry, které budou dosazeny do dotazu až během otevření kurzoru

Deklarace kurzoru

CURSOR <název kurzoru> [<název parametru> <datový typ>, …]

IS <příkaz SELECT>;

Příklad

```
DECLARE
    rec_ucitel      ucitel%ROWTYPE;

    CURSOR k1 (v_jmeno VARCHAR2) IS
      SELECT jmeno, Id FROM ucitel WHERE jmeno LIKE (v_jmeno || '%');

BEGIN
    FOR rec_ucitel IN k1 (‘Za’)
    LOOP
      DBMS_OUTPUT.PUT_LINE('Jméno ' || rec_ucitel .jmeno || ', Id ' || rec_ucitel.Id);
    END LOOP;

    FOR rec_ucitel IN k1 (‘Sm’)
    LOOP
      DBMS_OUTPUT.PUT_LINE('Jméno ' || rec_ucitel .jmeno || ', Id ' || rec_ucitel.Id);
    END LOOP;
END;
```

Kurzory shrnutí

- Pokud můžeme použít implicitní kurzory, tak je použijeme:
  - Pro výběr jednoho řádku

    SELECT <sloupce> INTO <proměnné> FROM <tabulky>
  - U sady výsledků s omezeným množstvím řádků

    For x in (SELECT <sloupce> FROM <tabulky>)

    Loop

    Process …

    End loop;

  Výhody implicitních kurzorů:
  - Kratší kód
  - Jsou rychlejší (tedy efektivnější)
  - Dělají kód bezpečnější
- Pro stovky a více řádků zvažte kurzory s klauzulí BULK COLLECT

Ošetření chyb

V zásadě se mohou v PL/SQL vyskytnout 2 druhy chyb:
Syntaktické – projeví se ještě v procesu kompilace (upozorní nás na ně překladač)
Logické – projeví se až za běhu programu

Nejčastěji se vyskytují následující výjimky:

DUP_VAL_ON_INDEX	výskyt duplicitní hodnoty ve sloupci, který připouští jen jedinečné hodnoty
INVALID_NUMBER	neplatné číslo nebo data nemohou být převedena na číslo
NO_DATA_FOUND	nebyly nalezeny žádné záznamy
TOO_MANY_ROWS	dotaz vrátil více než jeden záznam
VALUE_ERROR	problém s matematickou funkcí
ZERO_DIVIDE	dělení nulou

Ošetření chyb

Všeobecná syntaxe pro zpracování výjimek:

```
EXCEPTION
    WHEN <název výjimky> THEN <příkazy>;

    [WHEN <název výjimky> THEN <příkazy>; …]

    OTHERS THEN <příkazy>;

END;
```

Výjimku můžeme navodit nebo simulovat příkazem

```
RAISE <název výjimky>;
```

například

```
RAISE NO_DATA_FOUND;
```

Ošetření chyb

Aktuální kód chyby vrací systémová funkce **SQLCODE**
a její textový popis systémová funkce **SQLERRM**,
takže při zpracování výjimky máme k dispozici tyto údaje.

Příklad

```
DECLARE
    v_vysledek NUMBER(9,2);

BEGIN
     v_vysledek := 5/0;

EXCEPTION

      WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE(' Chyba ');
                DBMS_OUTPUT.PUT_LINE('Kód chyby:' || SQLCODE);
                DBMS_OUTPUT.PUT_LINE('Popis chyby:' || SQLERRM);

END;
```

Definování vlastních výjimek

Máme možnost definovat i vlastní výjimky.

Pro vlastní výjimky je SQLCODE rovno 1 a SQLERRM vrací Text User-Defined Exception

Syntaxe

```
DECLARE
    <název výjimky> EXCEPTION;

BEGIN
    <příkazy>;
    RAISE <název výjimky>;

EXCEPTION

     WHEN <název výjimky> THEN <příkazy>;

END;
```

Definování vlastních výjimek

Příklad definice vlastní výjimky pro kontrolu počtu trpaslíků.

```
DECLARE
     PRILIS_MNOHO_TRPASLIKU EXCEPTION;
    v_pocet_trpasliku NUMBER;

BEGIN
     v_pocet_trpasliku:=7;
     IF v_pocet_trpasliku > 7 THEN
                    RAISE PRILIS_MNOHO_TRPASLIKU;
    END IF;

EXCEPTION

       WHEN PRILIS_MNOHO_TRPASLIKU
         THEN DBMS_OUTPUT.PUT_LINE('Trpaslíků může být maximálně sedm');

END;
```

Procedury

Procedura je posloupnost příkazů, které se provedou v okamžiku spuštění procedury.
Na základě vstupních parametrů jsou vráceny výsledky v podobě výstupních parametrů.

Syntaxe

```
CREATE [OR REPLACE] PROCEDURE <název procedury> [(<seznam parametrů>)] AS

    … deklarační sekce
    ……

BEGIN
    … výkonná sekce
    ……

EXCEPTION
    … sekce pro zpracování výjimek
    ……

END;
```

Procedury

Příklad definice procedury

```
CREATE [OR REPLACE] PROCEDURE zvyseni_mzdy (procento IN NUMBER) AS

 BEGIN
      UPDATE pracovnici SET mzda = mzda * (1+procento/100);

 END;
```

Příklad spuštění procedury

```
      EXECUTE zvyseni_mzdy(6);
```

nebo

```
      EXEC zvyseni_mzdy(6);
```

nebo

```
       BEGIN
                zvyseni_mzdy(6);
       END;
```

Funkce

Funkce na rozdíl od procedur dokáží vrátit nějakou hodnotu,
která je ve většině případů vypočítána v těle funkce.

Syntaxe

```
CREATE [OR REPLACE] FUNCTION <název funkce> [(<seznam parametrů>)]
RETURN <datový typ výsledku>
AS

    … deklarační sekce
    ……….

BEGIN
    … výkonná sekce
    ……….
    RETURN <hodnota>;

EXCEPTION
    … sekce pro zpracování výjimek
    ……….

END;
```

Funkce

Příklad

```
CREATE [OR REPLACE] FUNCTION pocet_smen (Id_trp IN NUMBER)
RETURN NUMBER
AS
    v_pocet NUMBER;

 BEGIN
      SELECT count(*) INTO v_pocet FROM tezby
      WHERE Id_trpaslika=Id_trp AND skutecnost>0;
      RETURN v_pocet ;

 END;
```

Použití funkce

```
SELECT Jmeno, pocet_smen(Id) Pocet_smen FROM trpaslici;
```

Proč používat PL/SQL

- Obecně je PL/SQL málo používán a zřídka jsou využity všechny jeho možnosti
- Oracle podporuje kromě PL/SQL také jazyky Java a C

Fakta pro PL/SQL - Nejvýkonnější jazyk pro zpracování dat:

- Datové typy PL/SQL jsou datovými typy jazyka SQL (tj. není nutný převod mezi typy)
- Těsná vazba – např. cyklus FOR s explicitním kurzorem
- Není třeba provádět akce jako otevření a zavření kurzoru – to je prováděno automaticky
- Jsme chráněni před velkým počtem změn v databázi (přidání či odstranění sloupce často nevyžaduje změnu procedury)
- 1 analýza dotazu – mnoho provedení
- implicitní ukládání kurzoru do mezipaměti

Proč používat PL/SQL

- Změny ve schématu databáze
  - Například změna sloupce COMMENTS z VARCHAR(80) na VARCHAR(255) při správně navržených aplikacích v PL/SQL nebude znamenat žádný zásah do kódu

- V ostatních jazycích může být potíž v tom, že schází informace o používání objektu jiným vývojářem (tj. nedostatečné sledování závislostí), první vývojář provede změnu objektu a může vzniknout problém, u PL/SQL je vazba mezi uživateli objektů a místy uložení uložena přímo v datovém slovníku

- Použití příkazu SELECT * FROM table je v PL/SQL bezpečné, v ostatních aplikacích může přehození pořadí sloupců vyvolat problémy

Tvorba minimálního množství kódu

Tvorba minimálního množství kódu

- Procedurální jazyk by měl být použit až v případě, kdy množinový přístup aplikovaný v SQL jazyce je nepoužitelný

Příklad - nevhodně

```
Begin
    For x in (select * from table1)
    Loop
        Insert into table2 (c1, c2, c3) values (x.c1, x.c2, x.c3);
    End loop;
End;
```

Příklad – správné řešení

```
Insert into table2 (c1, c2, c3) SELECT c1, c2, c3 FROM table1;
```

Tvorba minimálního množství kódu

- Často se naprosto nevhodně používá hledání v několika samostatných tabulkách dle výsledku předchozího dotazu namísto spojení tabulek
- Výsledkem je pak samozřejmě několikanásobné zpomalení
- Nebuďme líní hledat řešení v jazyce SQL dříve než přistoupíme k používání PL/SQL

Umístění celého kódu na obrazovku

- To je spíše „dobrá“ rada
- Zajistit, aby se rutiny (procedury, funkce, ….) vešly na obrazovku
- Pokud tomu tak není, je dobré rozdělit kód na menší úseky

Balíčky - výhody

- Zvětšují obor názvů – může být použit stejný název procedury v různých balíčcích

- V jednom balíčku může být mnoho procedur, ale v datovém slovníku bude existovat pouze jeden objekt – balíček, namísto jednoho objektu slovníku pro každou proceduru nebo funkci bez použití balíčků

- Podporují zapouzdření, části kódu (podřízené rutiny), které nemají využití mimo balíček, jsou ukryty v balíčku a mimo něj nejsou viditelné a jsem jediným, kdo je může zobrazit

Balíčky - výhody

- Podporují proměnné uchovávané po celou dobu relace - můžete mít proměnné, které si udrží své hodnoty mezi jednotlivými voláními v databázi

- Podporují spouštěcí kód – tj. úsek kódu, který se provede při prvním odkazu na balíček v relaci, tj. umožňuje automatické provedení složitého inicializačního kódu

- Umožňují seskupení souvisejících funkcí

- Porušují řetězec závislostí – tj. odstraňují nebo omezují vliv kaskádování neplatných objektů

Balíčky

Balíček má 2 části

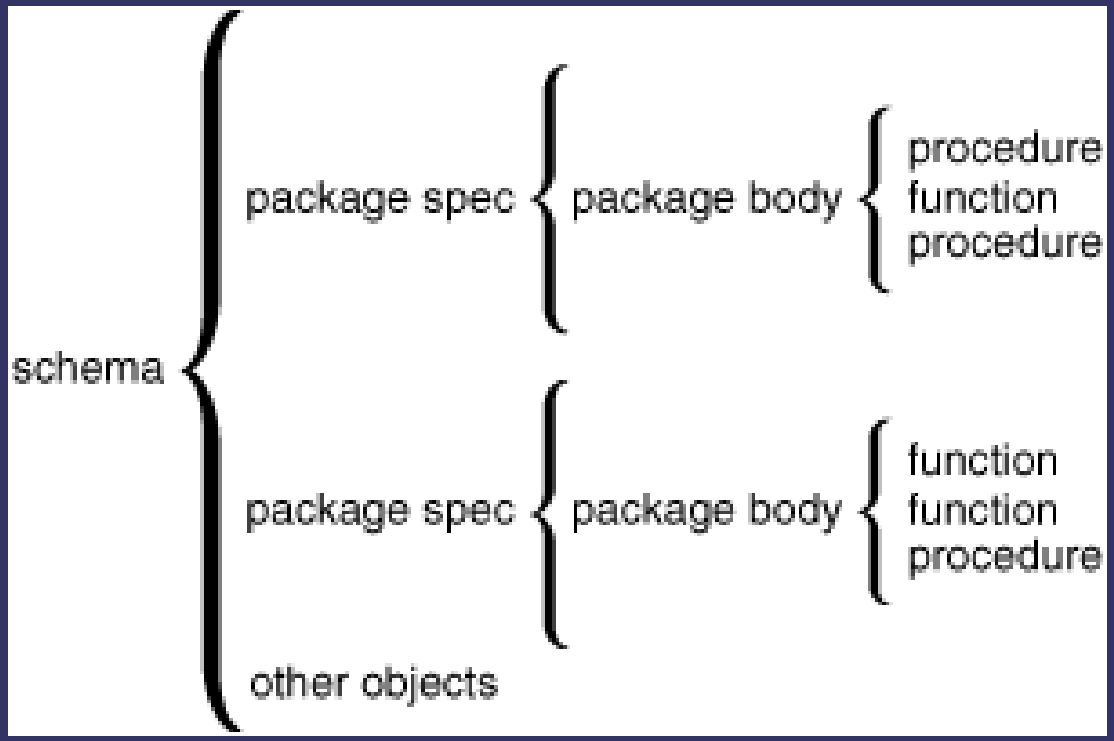
- specifikaci balíčku (interface k aplikacím)
- tělo balíčku

Ve specifikaci jsou deklarovány typy, proměnné, konstanty, výjimky, kurzory a podprogramy pro použití.

Tělo úplně definuje kurzory a subprogramy – implementační detaily a privátní deklarace, které jsou neviditelné z aplikace.

Je možné změnit tělo balíčku bez změny specifikace a tím vlastně neovlivnit vazbu na další aplikace. Programy volající balíček nemusí být rekompilovány při změně těla balíčku (tzv. balíčky přerušují řetězec závislostí).

Struktura balíčků

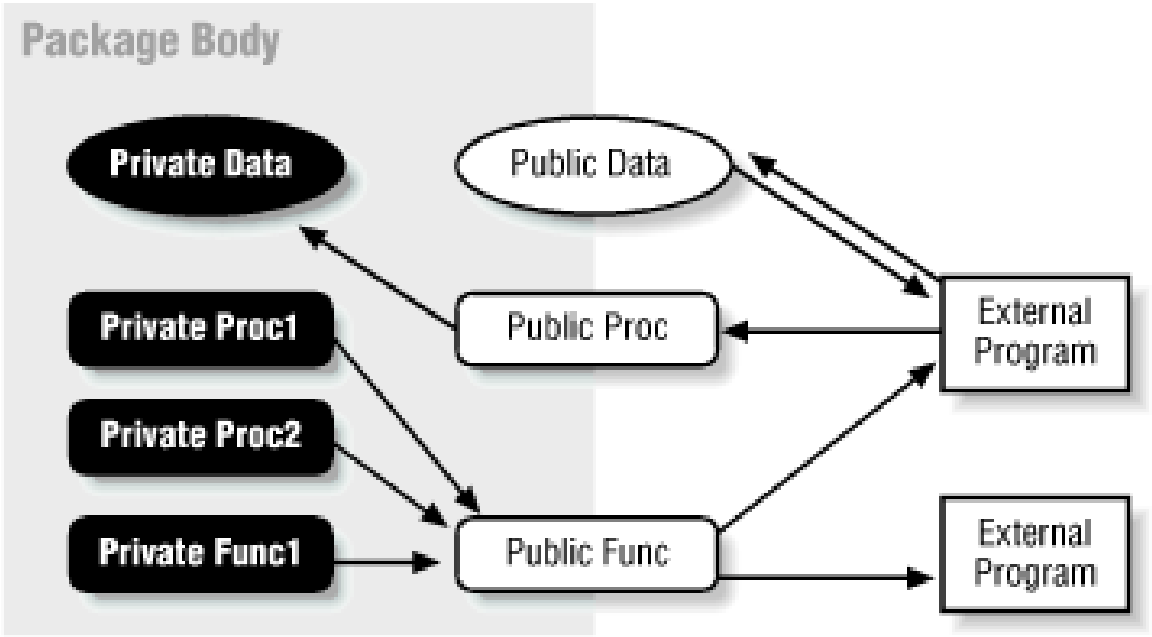


Balíčky - syntaxe

```
CREATE PACKAGE name AS  -- specification (visible part)
   -- public type and item declarations
   -- subprogram specifications
END [name];


CREATE PACKAGE BODY name AS  -- body (hidden part)
   -- private type and item declarations
   -- subprogram bodies
[BEGIN
   -- initialization statements]
END [name];
```

Veřejné a privátní elementy balíčků



Balíčky - odkazování

Referencing Package Contents

package_name.type_name

package_name.item_name

package_name.subprogram_name

Balíčky

Příklady například

http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/intro.htm#sthref18

http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14261/packages.htm#sthref1849

Porušení řetězce závislostí

Příklad řetězce závislostí bez použití balíčku

```
Create table  t (x int);

Create view v as select * from t;

Create procedure p as
    x v%rowtype;
    begin
    for x in (select * from v)
          loop
                null;
          end loop;
    end;
```

```
Create function f
    return number
          as pocet number;
          begin select count(*)
                into pocet from t;
          return pocet;
end;
```

Porušení řetězce závislostí

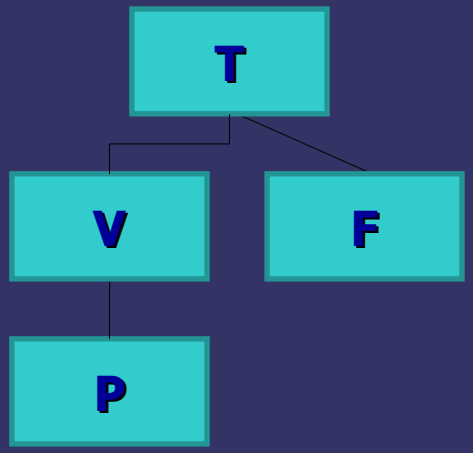
Zjištění řetězce závislostí a platnosti objektů

```
SELECT name, type, referenced_name, referenced_type from user_dependencies
WHERE referenced_owner=user order by name;
```

NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
F	FUNCTION	T	TABLE
P	PROCEDURE	V	VIEW
V	VIEW	T	TABLE

```
SELECT object_name, object_type, status from user_objects;
```

OBJECT_NAME	OBJECT_TYPE	STATUS
T	TABLE	VALID
V	VIEW	VALID
P	PROCEDURE	VALID
F	FUNCTION	VALID



Porušení řetězce závislostí

Provedení změny

Alter table t add y number;

Zjištění platnosti objektů

SELECT object_name, object_type, status from user_objects;

OBJECT_NAME	OBJECT_TYPE	STATUS
T	TABLE	VALID
V	VIEW	**INVALID**
P	PROCEDURE	**INVALID**
F	FUNCTION	**INVALID**

Objekty přestaly být platné, protože musí být všechny znovu zkompilovány. Kdyby proceduru P používaly desítky nebo stovky rutin v systému – všechny by se staly neplatnými.

Objekty budou automaticky zkompilovány při jejich prvním použití, ale to vyžaduje vykonání většího objemu práce (například zavoláním procedury p se automaticky zkompiluje i pohled v, ale nikoli funkce f).

Porušení řetězce závislostí

Použití balíčku a jejich vliv na řetězec závislostí

```
Create package p1 as
    procedure p;
end;

Create package body p1 as
      procedure p as
          x v%rowtype;
          begin
          for x in (select * from v)
            loop
                        null;
            end loop;
          end;
      end p1;
```

```
Create package p2 as
        procedure p;
end;

Create package body p2 as
      procedure p as
            begin
                        p1.p;
            end;
      end p2;
```

Porušení řetězce závislostí

Zjištění řetězce závislostí a platnosti objektů

```
SELECT name, type, referenced_name, referenced_type from user_dependencies
WHERE referenced_owner=user order by name;
```

NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
P1	**PACKAGE BODY**	**V**	**VIEW**
P1	**PACKAGE BODY**	**P1**	**PACKAGE**
P2	PACKAGE BODY	P2	PACKAGE
P2	PACKAGE BODY	P1	PACKAGE
V	VIEW	T	TABLE

Objekty jsou závislé na specifikaci balíčku – nikoli na těle balíčku.

```
SELECT object_name, object_type, status from user_objects;
```

OBJECT_NAME	OBJECT_TYPE	STATUS
T	TABLE	VALID
V	VIEW	VALID
P1	PACKAGE	VALID
P1	PACKAGE BODY	VALID
P2	PACKAGE	VALID
P2	PACKAGE BODY	VALID

Porušení řetězce závislostí

Provedení změny

Alter table t add z number;

Zjištění platnosti objektů

SELECT object_name, object_type, status from user_objects;

OBJECT_NAME	OBJECT_TYPE	STATUS
T	TABLE	VALID
V	VIEW	**INVALID**
P1	PACKAGE	VALID
P1	PACKAGE BODY	**INVALID**
P2	PACKAGE	VALID
P2	PACKAGE BODY	VALID

Platnost ztratil pouze pohled V a tělo balíčku P1.

Procedura P2.P volající P1. P už zůstala platnou, neboť se porušil řetězec závislostí – tj. databáze bude kompilovat pouze neplatné objekty (tedy méně než v minulém příkladu bez použití balíčků).

Provedením exec p2.p se tyto neplatné objekty automaticky zkompilují.

Klauzule returning into

V některých případech potřebuji návrat hodnoty v určitém sloupci pro DML příkazem delete/update ovlivněný řádek.

V kódu PL/SQL pak můžeme použít tento zápis:

```
UPDATE trpaslici SET
jmeno='Brůča'
WHERE jmeno='Bručoun'
returning id into v_ID;
```

Statické a dynamické příkazy SQL

- Většina databázových aplikací dělá velmi konkrétní úkony. Například na základě vstupního čísla zaměstnance a nového platu upraví tabulku s informacemi o zaměstnancích dle daných požadavků.

- Nicméně existuje i třída aplikací, které musí provádět velmi různé SQL dotazy o jejichž konkrétním tvaru se rozhoduje až při běhu programu. Například obecný generátor výstupních sestav musí sestavovat různé SELECTy pro různé výstupy, jež jsou po něm požadovány.

- V takovém případě ještě není v době kompilace přesné znění dotazu známo a dotazy se budou pravděpodobně spuštění od spuštění lišit - nazýváme je dynamické SQL.

Použití statických příkazů SQL

Výhody

- Je kontrolován při kompilaci
- Jsou ověřeny datové typy a velikosti
  (není nutné definovat záznamy a množství proměnných)
- Závislosti jsou nastaveny a udržovány v datovém slovníku
- Je 1x analyzován a mnohokrát proveden
- Je rychlejší

Použití dynamických příkazů SQL

Dynamické příkazy se naopak mohou jevit „univerzálnější“ a umožňují tvorbu kratšího kódu.

- Využití je možno doporučit v případě, kdy nestačí statické SQL příkazy - například pokud není v době kompilace známo:
  - text SQL dotazu
  - počet hostitelských proměnných
  - datové typy hostitelských proměnných
  - odkazy na databázové objekty, jako jsou sloupečky či tabulky nebo schémata
  - Konstrukce podmínek

Použití dynamických příkazů SQL

- V typickém případě je text SQL dotazu vyžádán na vstupu od uživatele spolu s potřebnými hodnotami hostitelských proměnných.

- Oracle pak rozparsuje SQL dotaz, aby ověřil dodržení syntaktických pravidel

- Dále pak jsou hostitelské proměnné připojeny (bind) k SQL dotazu. Což pro Oracle znamená získání jejich adres tak, aby mohly být načteny jejich hodnoty.

- Poté již je "spuštěn" samotný dotaz a jsou provedeny odpovídající akce nad databází.

- Takovéto dynamické dotazy samozřejmě mohou být spouštěny opakovaně, s měnícími se hodnotami hostitelských proměnných.

Dynamické příkazy – nativní

Základní, nejjednodušší metoda.

Příkaz se spouští voláním příkazu EXECUTE IMMEDIATE, kde jako parametr uvedeme řetězcovou proměnnou nebo textový řetězec:

EXECUTE IMMEDIATE {string};'

Dynamický příkaz je při použití Metody 1 parsován při každém spuštění. Proto se hodí zejména pro příkazy, které se spouštějí pouze jednou - typicky příkazy typu CREATE TABLE, CREATE INDEX apod.

Dynamické příkazy – nativní

V PL/SQL možno realizovat i operace přímo nepodporované v PL/SQL – příklad:

```
BEGIN
  EXECUTE IMMEDIATE 'TRUNCATE TABLE my_table;';
END;
```

Dynamické příkazy – nativní

V PL/SQL EXECUTE IMMEDIATE umí i proměnné – příklad:

```
CREATE PROCEDURE vyhodit_studenta
      (podm VARCHAR, st_id NUMBER) AS

  BEGIN
     EXECUTE IMMEDIATE
        'DELETE FROM student
                WHERE student_id = :id AND '
                || podm USING st_id;
  END;
```

Dynamické příkazy – nativní

Pro zvýšení výkonu je vhodné užívat vázaných proměnných (bind variables) v případech, kdy to má smysl (např. hodnoty, nikoli nazvy objektů). V následujícím příkladu je vidět porovnání, kdy bude vytvořen vždy nový kurzor pro každou hodnotu emp_id nebo použit kurzor jediný:

```
CREATE PROCEDURE fire_employee (emp_id NUMBER) AS
BEGIN
   EXECUTE IMMEDIATE
      'DELETE FROM employees WHERE employee_id = '
  || TO_CHAR(emp_id);

   EXECUTE IMMEDIATE
      'DELETE FROM employees WHERE employee_id
  = :id' USING emp_id;
END;
```

Dynamické příkazy – nativní

Volání procedury různých názvů:

```
CREATE PROCEDURE run_proc
  (proc_name IN VARCHAR2, table_name IN VARCHAR2)
  AS BEGIN
   EXECUTE IMMEDIATE 'CALL "' || proc_name || '" (
  :tab_name )' using table_name;
END;



BEGIN
  run_proc('DROP_TABLE', 'employees_temp');
END;
```

Dynamické příkazy – nativní

Atributy kurzoru %FOUND, %ISOPEN, %NOTFOUND a %ROWCOUNT pro jednořádkové SELECT příkazy můžeme využít i pro dynamické SQL:

```
BEGIN
  EXECUTE IMMEDIATE 'DELETE FROM employees WHERE
  employee_id > 1000';
  DBMS_OUTPUT.PUT_LINE('Number of employees de-
  leted: ' || TO_CHAR(SQL%ROWCOUNT));
END;
```

Dynamické kurzory

Někdy není známá definice kurzoru až do doby spuštění, pak přijde vhod tato kontrukce ..

```
CREATE OR REPLACE
PROCEDURE DynamicCursor (p_parameter IN VARCHAR2) IS
  TYPE cur_typ IS REF CURSOR;
  c_cursor cur_typ;
BEGIN
  v_query := 'SELECT first_name || '' '' || last_name FROM
   users WHERE user_type = :parameter';
  OPEN c_cursor FOR v_query USING p_parameter;
  LOOP
    FETCH c_cursor INTO v_text;
    EXIT WHEN c_cursor%NOTFOUND;
    -- process row here
  END LOOP;
  CLOSE c_cursor;
END;
```

Native Dynamic SQL vs DBMS_SQL

Další variantou dynamických dotazů je použití balíčku DBMS_SQL, ten :

- Parsuje dotaz pouze jednou, provádí vícekrát
  (nativní SQL parsuje při každém spuštění)
- Je pomalejší než nativní dotazy
- Nepodporuje uživatelem definované typy (nativní SQL ano)
- Nepodporuje FETCH INTO record types (nativní SQL ano)
- Podporuje Multiple row Updates/Deletes with returning clause
  (nativní SQL jen sigle row Updates/Deletes with returning clause)
- Podporuje dotazy delší než 32 KB (nativní SQL jen do 32 KB)

Hromadné zpracování

Význam – snížení počtu V/V operací, až několikanásobné zvýšení rychlosti provedení požadavku
(pozor – více současně zpracovávaných řádků nad určitou mez nemusí vždy znamenat zvýšení výkonu)

Často je prováděn následující proces

```
For x in (select * from ….)
Loop
    Zpracování dat DANÉHO ŘÁDKU;
    Insert into tabulka hodnoty (…);
End loop;
```

Hromadné zpracování

Hromadné zpracování by mohlo vypadat

```
DECLARE
    type array is table of t%rowtype index by binary_integer;
    data  array;
      cursor c is select * from t;
BEGIN
  open c;
  loop
    fetch c BULK COLLECT INTO data LIMIT 100;
    /* some processing */
        begin
      FORALL i IN 1 .. data.count
            insert into t2 values data(i);
        exit when c%notfound;
      end loop;
      close c;
END;
```

Otázky

Děkuji za pozornost.

Zajímavé odkazy:
http://download-east.oracle.com/docs/cd/B14117_01/server.101/b10759/statements_6006.htm#i2088318

Přehled systémových balíčků najdete například na adrese:
http://download-uk.oracle.com/docs/cd/B19306_01/appdev.102/b14258/toc.htm

Oracle PL/SQL Programming - kniha
http://www.unix.org.ua/orelly/oracle/prog2/index.htm

Dynamické dotazy:
http://oracle.chesio.com/dynamicke_sql.html#zpusoby_vyuziti_dynamickeho_sql
http://youngcow.net/doc/oracle10g/appdev.102/b14261/dynamic.htm

Architektury a techniky DS

Přednáška č. 8

RNDr. David Žák, Ph.D.
Fakulta elektrotechniky a informatiky
david.zak@upce.cz

Obsah

- Transakce, transakční protokoly, žurnály
- Technologie Flashback

Transakce

Transakce je logická část, která obsahuje jeden nebo více příkazů SQL. Transakce je atomickou jednotkou.

Transakce je ukončena buď:

- jejím dokončením (COMMIT)
- vrácením zpět/odvoláním transakce (ROLLBACK bez uvedení názvu SAVEPOINTU)
- Uživatel ukončí spojení se serverem Oracle (transakce je potvrzena automaticky)
- Spojení uživatele se serverem se ukončí abnormálně – transakce se odroluje zpět
- explicitně, když je proveden příkaz DDL (CREATE, DROP, RENAME, ALTER), ukončí se předchozí, v samostatné transakci se provede příkaz DDL

Transakce buď proběhne jako celek nebo se jako celek odvolá.

Cíl: zajištění konzistence dat v databázových tabulkách a nedělitelnost provedených změn, tj. princip VŠE NEBO NIC.

Transakce

Příklady:

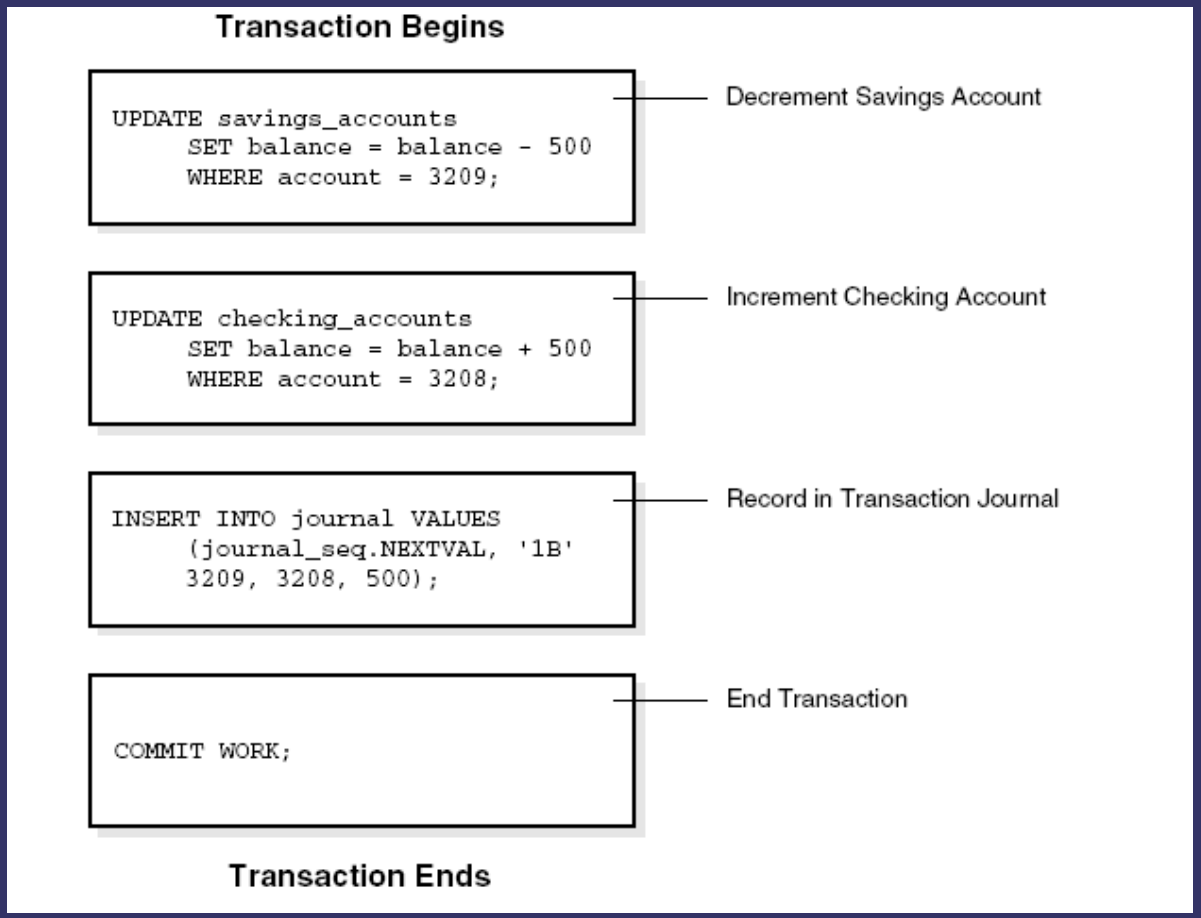
- není možné připustit provedení úpravy mzdy (inflační navýšení) jen u části zaměstnanců (než bylo požadováno),

  (při vrácení transakce se mzda nikomu nenavýší)

- není možné strhnout platbu z účtu plátce a nezaúčtovat ji na účet příjemce (z důvodů např. zrušení účtu příjemce, nesprávného účtu čísla příjemce, selhání techniky atd.).

  (tj. při odvolání se peníze nestrhnou ani z účtu plátce)

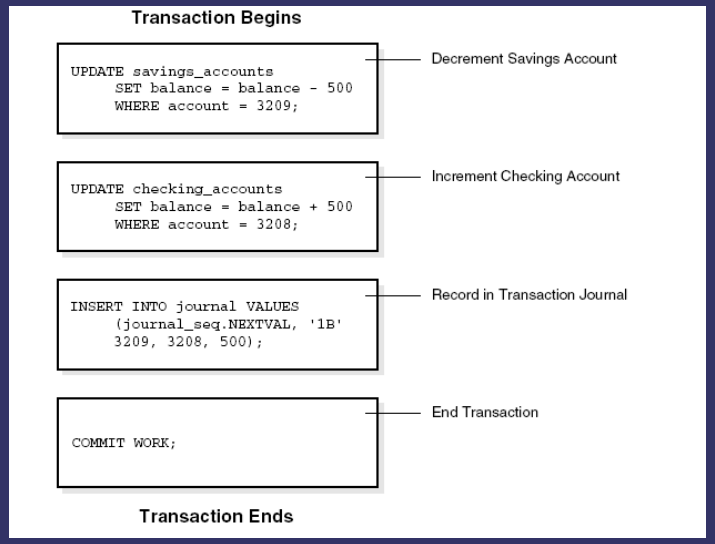
Příklad bankovní transakce



Příklad bankovní transakce

Při zpracování transakce mohou nastat problémy:

- Nedostatek finančních prostředků na účtu odesílatele
- Nesprávné číslo účtu (odesílatele nebo příjemce)
- Jiné omezení na účtu (exekuce)
- HW problém
- Špatně sestavený SQL dotaz



Transakce - potvrzení

Potvrzení transakce (committing) znamená, že změny provedené transakcí se stávají trvalými.

Explicitní potvrzení – příkazem COMMIT

Implicitní potvrzení – po normálním ukončení nějaké aplikace nebo provedením DDL operace

Jakékoli příkazy DDL (např. create table či alter index) způsobí tedy ukončení aktivní transakce a implicitní vytvoření nové transakce.

Změny provedené příkazy obsaženými v transakci jsou viditelné pro ostatní uživatele až od okamžiku potvrzení transakce.

Transakce - vytvoření

Transakce je inicializována implicitně. Pokud je po dokončení transakce příkazem COMMIT následně vložen, aktualizován či odstraněn alespoň jeden řádek, je tím implicitně vytvořena nová transakce.

Po zahájení transakce je jí přiřazen dostupný undo tablespace, kam se ukládají změny pro možnost provedení rollbacku. Undo informace obsahují staré hodnoty dat, které byly SQL příkazem v rámci transakce změněny.

Statement-Level Rollback

Pokud během provádění SQL příkazu nastane nějaká chyba, všechny změny provedené příkazem jsou odrolovány zpět. Toto se nazývá Statement-Level Rollback (rollback na úrovni příkazu).

Příklady takových chyb:

- Pokus o vložení řádku s duplicitní hodnotou primárního klíče
- Narušení referenční integrity
- Deadlock (pokus o současnou změnu shodných dat dvěma transakcemi)

Syntaktická chyba při parsingu neumožní spustit provádění příkazu, proto se nejedná o Statement-Level Rollback.

Chyba při provádění příkazu nezpůsobí změny provedené předchozími příkazy v rámci dané transakce.

Příkazy pro řízení transakcí

- COMMIT – potvrzení transakce, zafixování stavu.

- ROLLBACK – odvolání celé transakce,
  návrat do původního/zafixovaného stavu,

  odvolat transakci je možné pouze uživatelem, který operaci provedl a jen do okamžiku jejich potvrzení příkazem COMMIT
  (takto můžeme například obnovit záznamy, které jsme omylem vymazali, protože jsme například zapomněli na podmínku v klauzuli WHERE)

Odvolání ROLLBACK

- Statement level rollback (rollback příkazu)
- Rollback to savepoint
- Rollback celé transakce na žádost uživatele
- Rollback celé transakce z důvodu abnormálního ukončení procesu
- Rollback nedokončených transakcí, když dojde k ukončení instance
- Rollback nekompletní transakce z důvodu zotavení systému

Příkazy pro řízení transakcí

Použití návratových bodů:

- **SAVEPOINT <bod návratu>** - slouží pro označení místa (například po provedení některých operací), kam bychom se v případě potřeby mohli navrátit, slouží pro rozdělení velké transakce na menší části

- **ROLLBACK TO <bod návratu>** – odvolání transakce, návrat do bodu návratu

Příklad:

```
SAVEPOINT plosne_zvyseni_mzdy;
UPDATE pracovnici SET mzda=mzda*1.04;
SAVEPOINT navyseni_manazeri;
UPDATE pracovnici SET mzda=mzda+1000 WHERE pozice LIKE ‘manažer’;
ROLLBACK TO navyseni_manazeri;
```

Příkazy pro řízení transakcí

Po odrolování transakce k návratovému bodu:

- jsou vráceny pouze změny provedené SQL příkazy po nastavení návratového bodu
- Všechny návratové body nastavené po daném savepointu jsou ztraceny
- Oracle uvolní všechny uzamčené tabulky a řádky, které byly uzamčeny po nastavení návratového bodu (ty nastavené dříve zůstanou zachovány)
- Transakce zůstává aktivní a může dále pokračovat

Pojmenování transakcí

Pomocí příkazu set transaction name <název> je možné pojmenovat transakci. Tento příkaz asociuje jméno transakce s jejím ID.

Aplikaci, která ji vytvořila to nepřinese žádné výhody, ale název transakce bude dostupný v dynamickém výkonovém pohledu V$TRANSACTION a umožní správci databáze sledovat déle trvající transakce.

Dále je tento název viditelný například v Enteprise Manageru při monitorování aktivity systému.

Příkaz set transaction musí být prvním příkazem v rámci transakce.

Dvoufázový potvrzovací mechanismus

V distribuovaných databázích je třeba zajistit přes síťové prostředí kontrolu nad konzistencí dat.

Distribuovanou transakcí je transakce, která mění data nad 2 a více uzly distribuované databáze.

Dvoufázový mechanismus potvrzování transakcí garantuje, že všechny databázové servery participující na distribuované transakci buď všechny potvrdily nebo odvolali změny provedené transakcí.

Tento mechanismus zajišťuje také integritní omezení, volání vzdálených procedur a triggerů.

Použití tohoto mechanismu nevyžaduje žádné změny v databázové aplikaci, neboť ta nemusí ani vědět, že pracuje s distribuovanou databází.

Základy návratové technologie

Návratové tabulkové prostory zajišťují vrácení změn provedených transakcemi.

Kromě toho zajišťují další funkce, například

- zajištění konzistence dat s ohledem na čtení,
- provádění operací souvisejících s obnovou dat databáze,
- zajištěním funkcionality technologie Flashback.

Správa návratových tabulkových prostorů

Tabulkové prostory se vytváří:

- Jako součást příkazu create database

```
CREATE DATABASE rbdb1
  CONTROLFILE REUSE
  ...
  UNDO TABLESPACE undotbs_01
    DATAFILE 'C:\Oracle\Ordata\TSH1\undo0101.dbf'
    SIZE 100M REUSE AUTOEXTEND ON;
```

- Kdykoli později příkazem create tablespace

```
CREATE UNDO TABLESPACE undotbs_02
  DATAFILE 'C:\Oracle\Ordata\TSH1\undo0201.dbf'
  SIZE 100M REUSE AUTOEXTEND ON;
```

Parametry návratových tabulkových prostorů

Parametr UNDO_RETENTION udává minimální dobu, pro kterou jsou udržovány návratové informace pro dotazy. V automatickém režimu je výchozí hodnota 900 s. Může však být i několik hodin …

Tato hodnota je však platná pouze v případě, kdy je v návratovém tabulkovém prostoru dostatek místa pro zajištění konzistence pro čtení.

Pokud aktivní transakce vyžaduje dodatečné místo, dojde k přepsání jiných platných návratových informací tak, aby se uspokojily požadavky aktuální transakce.

Data v návratových tabulkových prostorech

Data v návratovém tabulkovém prostoru mohou být ve 3 stavech:

- Aktivní (active) – vyžadována pro dokončení transakce nebo zachování konzistence pro probíhající operace čtení (i po dokončení transakce)
- Neexpirovaná (unexpired) - nebylo dosaženo doby platnosti návratových dat
- Neplatná (expired) – po dokončení všech dotazů, které vyžadovaly platná návratová data a pokud bylo dosaženo doby platnosti návratových dat (možno využít např. pro podporu technologie Flashback
- Nepoužitá – veškeré volné místo v návratovém tabulkovém prostoru

Pro správnou volbu velikosti návratového tabulkového prostoru se používají nástroje databáze Oracle na principu analýzy trendů.

Konzistentní čtení

- Konzistentní čtení na úrovni
  - SQL příkazů
  - transakcí

Konzistentní čtení

- Oracle zabraňuje destruktivní interakci mezi transakcemi, které přistupují ke stejným zdrojům dat, tento proces je automatický a nevyžaduje od uživatelů žádnou součinnost.

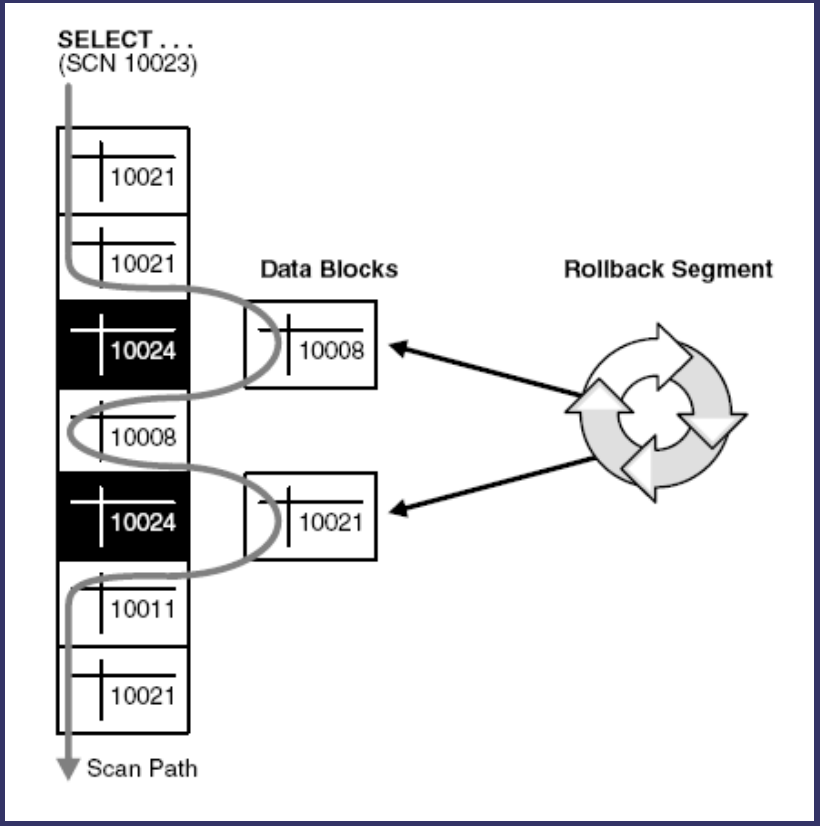
- Doporučuje se, aby jedna transakce obsahovala co nejméně operací. Pokud obsahuje mnoho kroků, zpomaluje práci databázového serveru, protože musí mnoho dat ukládat do transakčního žurnálu. Server do nich neukládá příkazy SQL, ale informace o všech změnách, které provedl.

- Představme si situaci, kdy probíhá SELECT, jehož vyhodnocení trvá několik hodin a mezitím někteří uživatelé provedou určité změny dat a tyto potvrdí příkazem COMMIT. Výsledek bude takový, že:
  - ve výsledku dříve spuštěného dotazu nebudou provedené změny během doby zpracování zahrnuty,
  - ostatní uživatelé budou normálně pracovat s aktuální verzí dat (tedy po potvrzení změn).

Konzistentní čtení

Obrázek ukazuje příklad pro čtení dat z tabulky. Příkaz SELECT je spuštěn v okamžiku SCN (system change number) = 10023. Pokud se při čtení narazí na bloky dat změněné po tomto „čase“ (SCN je vyšší), budou pro výsledek dotazu vstupní data rekonstruována s využitím informací uložených v návratovém segmentu. Tím je zajištěna konzistence vstupních dat pro zpracování dotazu a tím platnost výsledku.

SCN se při každé transakci zvyšuje o 1.



Konzistentní čtení – na úrovni SQL dotazu

Oracle vždy zajišťuje konzistenci pro čtení na úrovni dotazu. Ani potvrzení jiné transakce příkazem COMMIT během zpracování dotazu neovlivní jeho výsledek – vždy je důležitý stav při zahájení dotazu. Tato konzistence je zajištěna automaticky bez účasti uživatele. Vztahuje se i na vnořené dotazy.

Toto ovšem také znamená, že daný dotaz nevidí změny vyvolané jím samotným, ale vždy pracuje s daty ve stavu při jeho spuštění!

Poznámka: Problém by mohl nastat, pokud by příkaz SELECT spouštěl funkci, která by opět obsahovali jiný SELECT (ta by ovšem měla jiné SCN začátku) a uvažovala by již změněná data.

Konzistentní čtení – na úrovni transakce

Oracle nabízí možnost zajištění konzistence pro čtení na úrovni transakcí. Pokud transakce běží v seriazible módu, všechny přístupná data reflektují stav databáze v čase zahájení transakce. To platí pro všechny příkazy v rámci transakce kromě změn provedených transakcí samotnou.

Izolace transakcí

Oracle nabízí možnost zajištění konzistence pro čtení na úrovni transakcí. Pokud transakce běží v seriazible módu, všechny přístupná data reflektují stav databáze v čase zahájení transakce. To platí pro všechny příkazy v rámci transakce kromě změn provedených transakcí samotnou.

Izolace transakcí

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Izolace transakcí

Standard SQL92 definuje čtyři úrovně izolace transakcí s různým ovlivňováním transakcí a různou propustností.

Smyslem izolací je chránit data zpracovávaná transakcí před 3 vlivy:

- Dirty reads: Transakce čte data zapsaná jinou transakcí, která ještě nebyla potvrzena
- Nonrepeatable (fuzzy - zmatené) reads: Transakce čte data již jednou čtená a shledává, že jsou ovlivněna (upravena či smazána) jinou transakcí, jejíž potvrzení nastalo během probíhání dané transakce.
- Phantom reads (or phantoms - zjevení): Transakce opětovně spouští dotazy vracející množinu řádků vyhovující vyhledávacím podmínkám a nachází, že jiná potvrzená transakce přidala další řádky, které splňují podmínky vyhledávání.

Oracle izolační úrovně

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

Toto je výchozí (default) izolační úroveň. Každý dotaz vykonávaný transakcí „vidí“ pouze data, která byla potvrzena (COMMIT) před zahájením dotazu (dotazu – nikoli transakce).

Jelikož Oracle nezamezuje ostatním transakcím modifikovat data, mohou být data modifikována jinou transakcí mezi 2 spuštěními stejného dotazu. Pokud tedy transakce spustí stejný dotaz dvakrát, může dostat rozdílné výsledky.

Read committed transakce neřeší problematiku „nonrepeateble reads“ a „phantoms“.

Oracle izolační úrovně

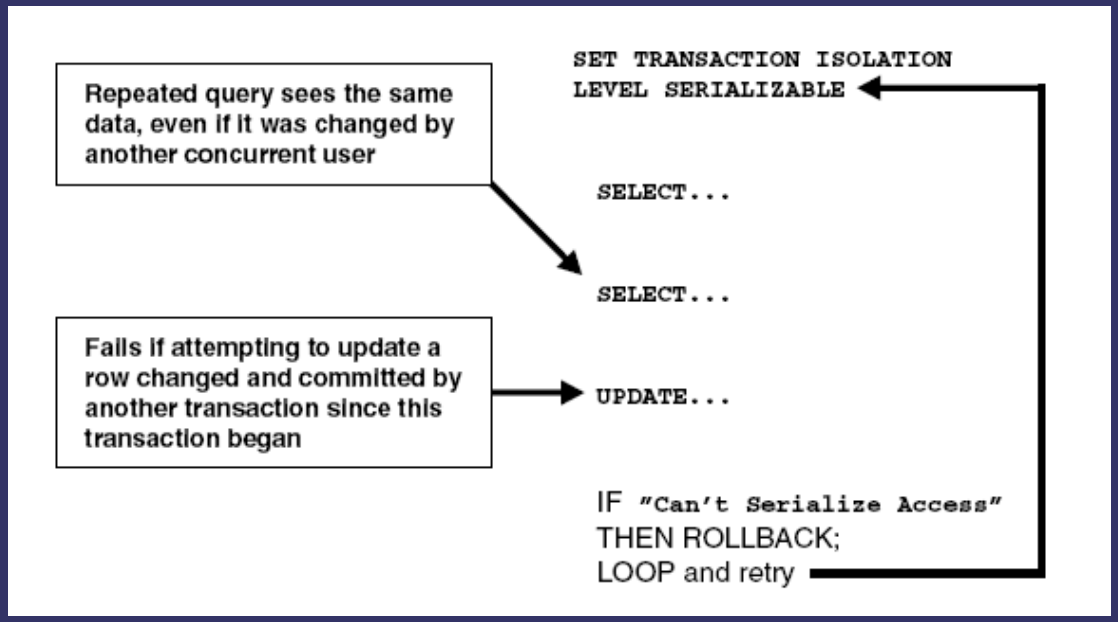
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Serializovaná transakce „vidí“ pouze změny, které byly potvrzeny před zahájením transakce a dále ty změny, které transakce sama provedla pomocí příkazů INSERT, UPDATE, DELETE.

Serializovaná transakce řeší problematiku „nonrepeateble reads“ a „phantoms“.

Oracle izolační úrovně

Obrázek ukazuje příklad, kdy není možné serializovat přístup, neboť jiná transakce upravila data od doby, kdy tato transakce začala. Výsledkem je chybová hláška při průběhu serializované transakce.



Oracle izolační úrovně

Pokud při běhu serializované transakce je výsledkem chybová hláška *„ORA-08177: Cannot serialize access for this transaction“*, je možné:

- potvrdit práci vykonanou do této doby,
- vykonat další (ale odlišný) příkaz (případně odrolovat zpět na nějaký savepoint v transakci a poté zadat další příkaz),
- odrolovat zpět celou transakci.

Oracle izolační úrovně

SET TRANSACTION READ ONLY;

Read-only transakce „vidí“ pouze změny, které byly potvrzeny v době, kdy transakce začínala a v rámci transakce nelze použít příkazy INSERT, UPDATE, DELETE.

Oracle izolační úrovně - porovnání

Read commited izolace je vhodná v případě, kdy se několik transakcí může ovlivňovat. Zajišťuje dobrou propustnost/výkon databázového systému.

Serializovaná izolace je vhodná pro prostředí, kde:

- jsou velké databáze a krátké transakce ovlivňující pouze pár řádků,
- je šance, že 2 konkurenční transakce budou modifikovat stejný řádek je relativně malá,
- kde dlouhodobé transakce jsou primárně read-only transakce.

Oby uvedené typy izolací používají uzamykání na úrovni řádků a obě budou čekat, když se budou pokoušet změnit řádek ovlivněný nepotvrzenou konkurenční transakcí. Druhá transakce čeká, jestli první potvrdí nebo odvolá provedenou změnu a odstraní zámek. Pokud odvolá, tak druhá transakce může pokračovat, jako by první transakce neexistovala.

Pokud blokující transakce (první) potvrdí změnu a odstraní zámek, read commited transakce pokračuje v zamýšlené změně. Serializovaná transakce bohužel vygeneruje chybovou hlášku.

Automatické zamykání

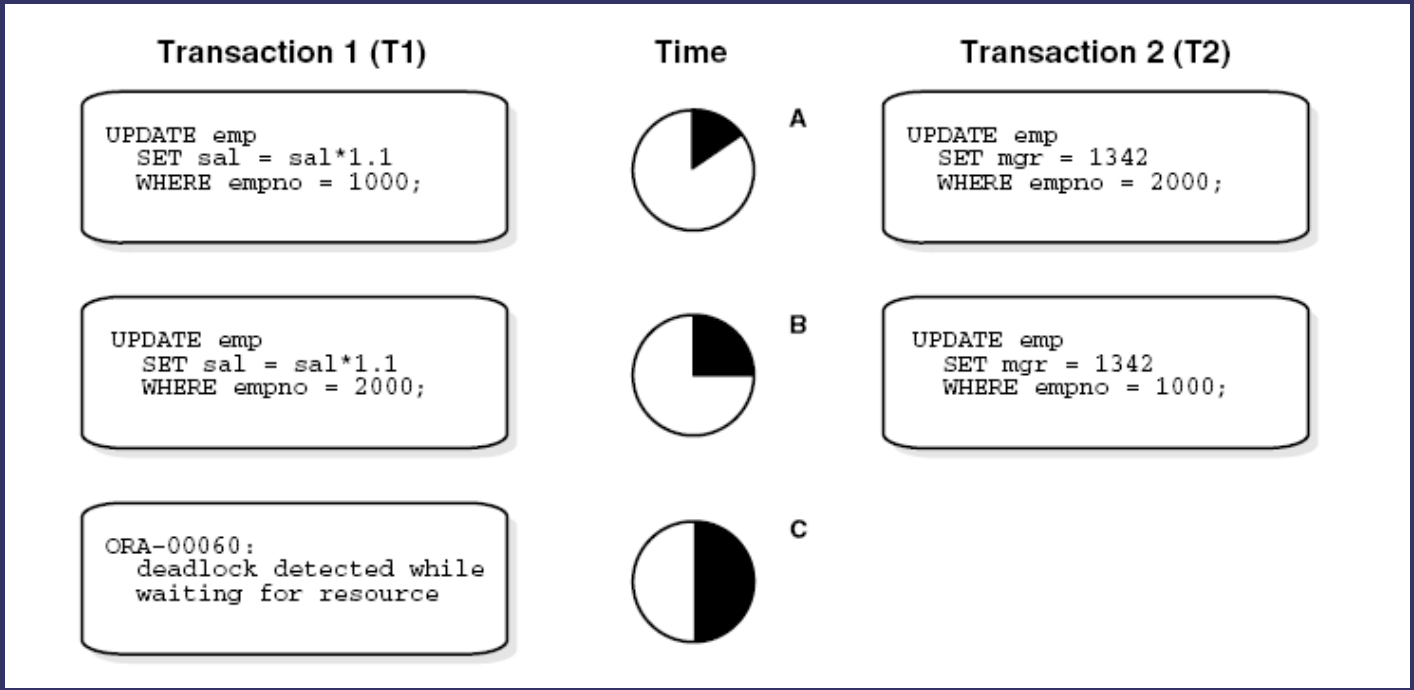
- Slouží pro řešení sdíleného (konkurenčního) přístupu více klientů k datům v tabulkách.

- Je třeba zajistit situace, kdy by například příkaz SELECT byl spuštěn v době, kdy právě probíhá navyšování mezd zaměstnanců - nutno zajistit konzistentní odpověď.

Řešení: k nepotvrzeným změnám, které provedl jeden uživatel ostatní uživatelé nemají přístup,
tj. databázový systém zajišťuje konzistentní pohled na data v každém okamžiku
(ostatní uživatelé kromě autora změn vidí data ve stavu před provedením změn až do okamžiku potvrzení)

Upozornění: u APEX (WWW SQL konzole) dochází k potvrzení každého dotazu a není možný návrat zpět;

Deadlock



Oracle automaticky detekuje deadlocky a řeší je tím, že odroluje zpět jeden z příkazů, který deadlock způsobil. Transakce dostává zprávu, že proběhl rollback na úrovni příkazu. Transakce se poté může odrolovat zpět explicitně nebo se pokusí odrolovaný příkaz znovu provést.

Módy zamykání a typy zámků

Oracle používá 2 módy zamykání:

- Exklusívní (uzamčený zdroj nemůže být sdílen)
- Sdílený – zamčený zdroj může být sdílen, což umožňuje zejména více přístupů pro čtení, ale zabraňuje konkurenčním zápisům

Kategorie zámků

- DML zámky – např. uzamykání celých tabulek či vybraných řádků
- DDL zámky – chrání např. strukturu tabulek či pohledů
- Interní zámky – chrání interní databázové struktury, jako například datové soubory

Uzamykání a odemykání probíhá automaticky dle požadavků transakcí.

Obnova databáze

Návratové tabulkové prostory jsou klíčovou součástí procesu obnovy instance.

Online soubory protokolu umožní provést všechny potvrzené i nepotvrzené transakce až do bodu, kdy došlo k selhání instance.

Návratová data se následně použijí pro vrácení změn provedených transakcemi, které v okamžiku selhání instance ještě nebyly potvrzeny operací commit.

Nástroj pro konfiguraci a analýzu

http://uei.upceuei.cz:5500/em/console/database/instance/undoAdvisory?target=oracle10&type=or

Configuration

Auto-tuned Undo Retention (minutes)	**15**	Undo Tablespace	UNDOTBS1 Change Tablespace
Low Threshold Undo Retention (minutes)	15	Size (MB)	**60**
Undo Retention Guarantee	**No**	Auto-Extensible	**Yes**

Recommendations

Choose the time period that best represents the system activity to get the recommendations for undo retention length and undo tablespace size. Edit Undo Tablespace

Analysis Time Period: Last Seven Days — Update Analysis

Selected Analysis Time Period: **11.4.07 18:00 - 18.4.07 18:00**

Potential Problems: **No Problem Found**

Recommendations: **No Recommendation**

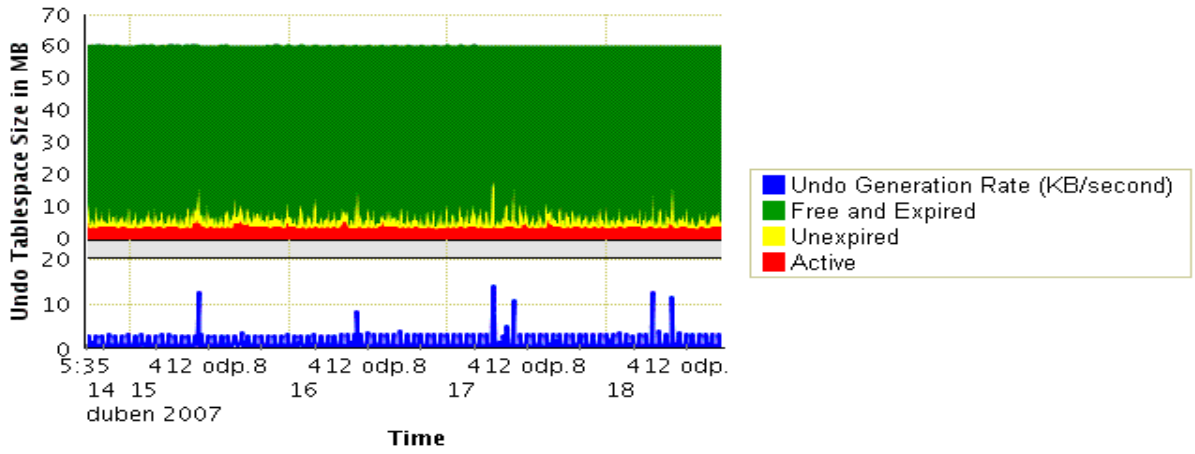
System Activity and Tablespace Usage

The recommendations are based on system activity and undo tablespace usage for the selected analysis time period.

Longest Running Query (minutes)	**25**
Average Undo Generation Rate (KB/minute)	**89.0**
Maximum Undo Generation Rate (KB/minute)	**859.0**

Hide Graph

Undo Generation Rate and Tablespace Usage



Technologie Flashback

Technologie Flashback využívá dat z návratového prostoru pro:

- Flashback table (umožní obnovit data z tabulky k danému časovému okamžiku v minulosti),

- Flashback Query (umožní zobrazit obsah tabulky k danému systémovému bodu změny SCN nebo danému časovému okamžiku v minulosti),

- Podporu procedur z balíčku DBMS_Flashback (rozhraní pro provádění operací typu Flashback).

Technologie Flashback query

Od verze Oracle 9i2 je součástí dotazů SELECT … klauzule AS OF

Tento příklad Flashback Query zobrazí stav tabulky v určité době.

- Předpokládejme například, že databázový administrátor (DBA) zjistí ve 12:30, že data o zaměstnanci JOHN byla vymazána v tabulce employee, a zároveň DBA ví, že v 9:30 tato data ještě byla v pořádku.

- DBA může použít Flashback Query pro zjištění stavu tabulky v 9:30, k nalezení dat , která byla smazána. Je-li to třeba, DBA může opětovně ztracená data vložit do tabulky.

```
SELECT * FROM employee AS OF TIMESTAMP
  TO_TIMESTAMP('2003-04-04 09:30:00', 'YYYY-MM-DD HH:MI:SS')
  WHERE name = 'JOHN';
```

- Tento insert vloží data o zaměstnanci JOHN do tabulky employee:

```
INSERT INTO employee
   (SELECT * FROM employee AS OF TIMESTAMP
    TO_TIMESTAMP('2003-04-04 09:30:00', 'YYYY-MM-DD HH:MI:SS')
    WHERE name = 'JOHN');
```

Technologie Flashback Query

- Klauzuli AS OF je možné použít pro každou tabulku a specifikovat různé časy pro různé tabulky.

- Klauzuli AS OF je možné využit uvnitř příkazů

```
INSERT INTO TABLE … (SELECT …. AS OF …)
CREATE TABLE AS SELECT … AS OF …
```

- Obdobně je možné vytvořit pohled vyjadřující stav v minulosti

Příklad:

```
CREATE VIEW hour_ago AS
        SELECT * FROM employee
        AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '60' MINUTE);
```

- Dále je možné použít tuto technologii pro spojení tabulek (každá může být v jiném čase) a k množinovým operacím, viz například následující příklad:

```
INSERT INTO employee
        (SELECT * FROM employee
                AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '60' MINUTE)
        MINUS SELECT * FROM employee);
```

Balíček DBMS_Flashback

- Balíček DBMS_FLASHBACK obecně poskytuje stejnou funkcionalitu jako Flashback Query
- DBMS_FLASHBACK balíček funguje jako stroj času.
- Můžete vrátit zpátky čas, vykonat dotazy, jako kdybyste byli v minulosti a poté se opět vrátit do současnosti. Protože je možné využít balíček DBMS_FLASHBACK k vykonání příkazů v minulosti bez speciálních klauzulí jako AS OF nebo VERSIONS BETWEEN, můžete použít existující kód PL/SQL beze změn k dotazům do databáze v minulosti.
- Uživatel musí mít práva EXECUTE pro balíček DBMS_FLASHBACK, aby jej mohl použít.

Balíček DBMS_Flashback

Pro použití balíčku DBMS_FLASHBACK v kódu PL/SQL:

1. volejte DBMS_FLASHBACK.ENABLE_AT_TIME nebo DBMS_FLASHBACK.ENABLE_AT_SYSTEM_CHANGE_NUMBER k přechodu do minulosti
   Od té doby vrací platná data z minulosti.
2. Proveďte normální dotazy (tj. bez speciální syntaxe pro Flashback jako AS OF).
   Databáze automaticky pracuje s daty v minulosti
   Vykonávejte pouze dotazy; nezkoušejte DDL nebo DML operace.
3. volejte DBMS_FLASHBACK.DISABLE k návratu do současnosti (vždy je třeba nezapomenout na tento příkaz ENABLE a DISABLE musí být vždy v párech)

Také je možné použít kurzory pro ukládání výsledků dotazů do minulosti.
Otevřete kurzor před voláním DBMS_FLASHBACK.DISABLE.
Po volání DBMS_FLASHBACK.DISABLE můžete udělat následující:

- INSERT nebo UPDATE operace, k modifikaci současné databáze užitím výsledků z minulosti
- porovnat současná data s daty v minulosti (po volání DBMS_FLASHBACK.DISABLE otevřete druhý kursor pro stejný dotaz nad aktuálními daty).
- také je možné data z minulosti dát do temporary table (dočasné tabulky) a použít množinové operace jako MINUS nebo UNION ke zjištění rozdílu atd.

Technologie Flashback Table

Technologie Flashback Table umožní obnovit stav řádků tabulky do daného časového okamžiku v minulosti, ale také umožní obnovit stav indexů, triggerů a integritních omezení, to vše při spuštěné databázi, čímž se zvyšuje celková dostupnost databáze.

Tabulku je možné obnovit:

- K danému časovému okamžiku
- K dané systémové změně popsané identifikátorem SCN (systém change number)
- Odstraněnou tabulku

Snadno se používá v případě, kdy:

- Rozsah chyb je malý a zahrnuje pouze málo tabulek
- Stačí provést ničím nepodmíněnou obnovu dat tabulky k danému časovému okamžiku
- Nebyl použit příkaz pro definici dat (DDL) nad danou tabulkou

Pro obnovu většího objemu dat se lépe hodí technologie Flashback Database

Pro fungování technologie Flashback table je nutné aktivovat režim přesunu řádků, použití tohoto režimu může změnit identifikátor ROWID řádku!

Do jednoho příkazu je možné uvést více tabulek, které jsou například provázány cizími klíči.

Technologie Flashback Table

FLASHBACK TABLE <original_table_name> TO BEFORE DROP
{RENAME TO <new_table_name>};

```
FLASHBACK TABLE test2 TO BEFORE DROP
    RENAME TO recovered_test;
```

FLASHBACK TABLE <table_name> TO SCN <scn number>;

```
FLASHBACK TABLE test TO SCN 1833265;
```

FLASHBACK TABLE <table_name> TO TIMESTAMP <timestamp>;

```
ALTER TABLE employee ENABLE ROW MOVEMENT;

FLASHBACK TABLE employee TO TIMESTAMP
  TO_TIMESTAMP('2003-04-04 09:30:00', 'YYYY-MM-DD HH:MI:SS')
```

System Change Number (SCN)

- *System Change Number (SCN)* je neustále se zvyšující číslo, které jednoznačně identifikuje potvrzenou (commited) verzi databáze. Pokaždé, když uživatele zadá příkaz COMMIT nebo je tento realizován implicitně, zvýší se SCN.

- Oracle používá SCNs v řídících souborech, návratových souborech, ...

- Každá návratový log soubor obsahuje log sequence number a *low* a*high* SCN.
  - low SCN znamená nejnižší SCN uložené v log souboru
  - high SCN znamená nejvyšší SCN v log souboru

Zjištění SCN:

```
select dbms_flashback.get_system_change_number from dual;
```

Odkazy

Technologie a licenční podmínky:
http://www.oracle.com/global/cz/database/edice.html

K technologii Flashback:
http://www.psoug.org/reference/tab_flashback.html
http://www.psoug.org/reference/dbms_flashback.html
http://download-west.oracle.com/docs/cd/B13789_01/appdev.101/b10795/adfns_fl.htm

Architektury a techniky DS

Přednáška č. 9

RNDr. David Žák, Ph.D.
Fakulta elektrotechniky a informatiky
david.zak@upce.cz

Obsah

- Architektura databáze Oracle
- Logické a fyzické úložné struktury
- Paměťové struktury Oracle

Pojmy - databáze

Databáze je kolekce dat na disku, která je fyzicky uložena v jednom nebo více souborech na databázovém serveru a která obsahuje souvztažné informace.

Databáze sestává z různých fyzických a logických struktur.

Nejdůležitější logickou strukturou databáze je tabulka. Tabulka je složena z řádků a sloupců, které obsahují souvztažná data. Aby databáze mohla uchovávat data, musí obsahovat alespoň tabulky.

Data jsou v každém řádku tabulky souvztažná: každý řádek obsahuje informace o konkrétním zaměstnanci firmy.

Pojmy - databáze

Databáze také poskytuje zabezpečení, která brání neautorizovanému přístupu k datům.

Soubory, ze kterých je databáze složena, se dělí do dvou kategorií:

- databázové soubory a
- nedatabázové soubory.

Rozdíl je v datech, která jsou v souborech uložena. Databázové soubory obsahují databázová data a metadata, nedatabázové soubory obsahují inicializační parametry, protokolovací informace atd.

Pojmy - instance

Vlastní databáze Oracle je uložena na pevném disku serveru a v operační paměti serveru existuje instance databáze Oracle.

Instance se skládá z rozsáhlého bloku paměti, který je vyhrazený v systémové globální oblasti System Global Area (SGA), a z procesů, které běží na pozadí a komunikují s SGA a databázovými soubory na disku.

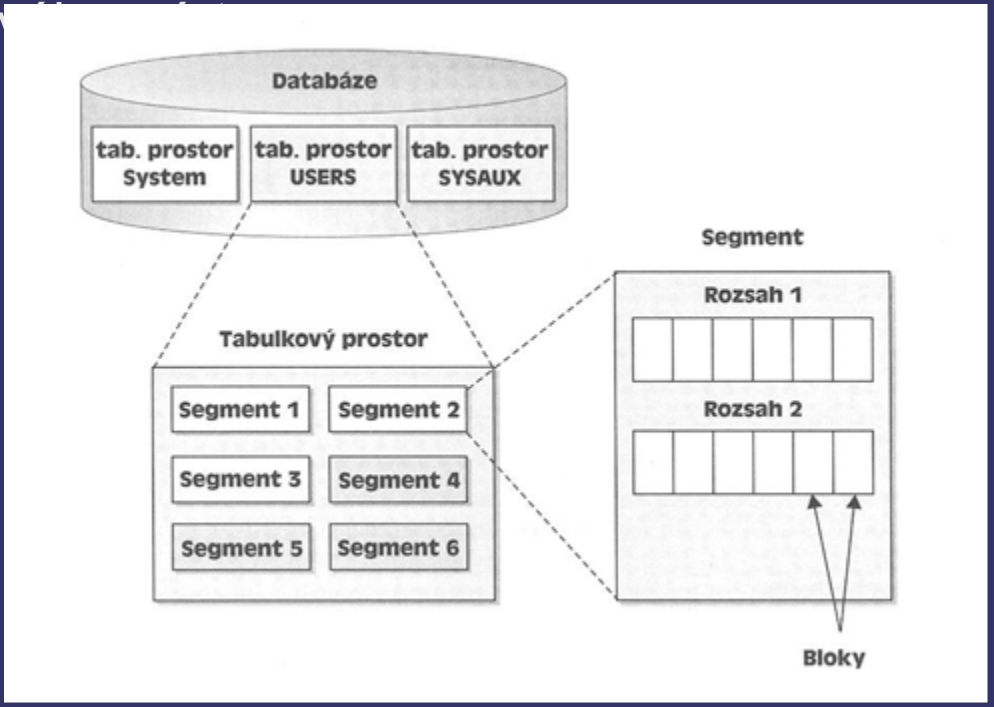
Pojmy - instance

Při použití technologie Oracle Real Application Cluster (RAC) využívá stejnou databázi více instancí.

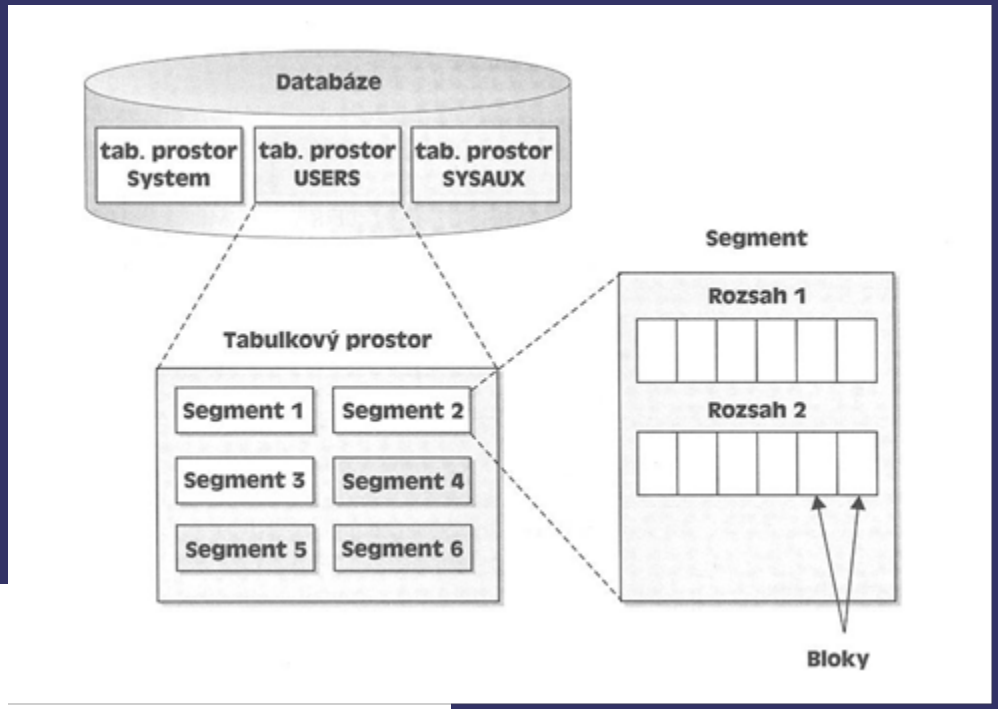
Jednotlivé instance, které sdílí databázi, mohou být umístěny na stejném serveru, ale bývá obvyklé, že se tyto instance nacházejí na oddělených serverech, které jsou vysokorychlostně propojeny a přistupují k databázi, která je uložena na speciálním diskovém subsystému, který využívá technologii RAID.

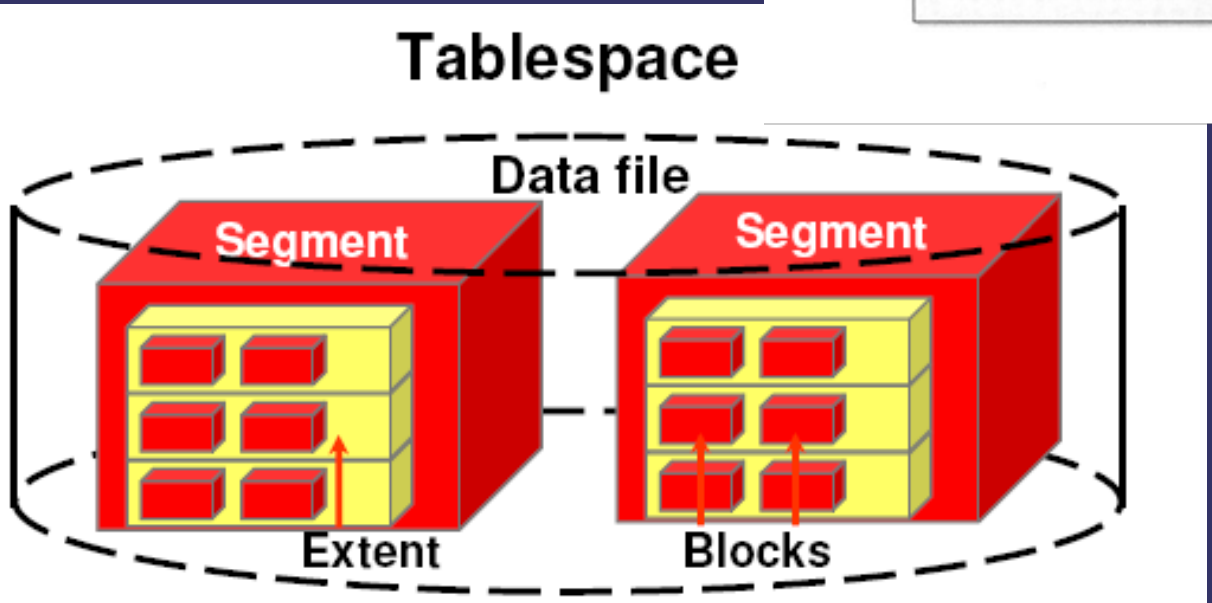
Logické úložné struktury

Datové soubory databáze Oracle jsou sdruženy do jednoho nebo více tabulkových prostorů. V rámci každého tabulkového prostoru jsou logické úložné struktury (např. tabulky nebo indexy) reprezentovány segmenty, které se dále dělí do rozsahů a bloků. Toto logické rozdělení úložného prostoru umožňuje databázi Oracle účinnější kontrolu nad využitím disko

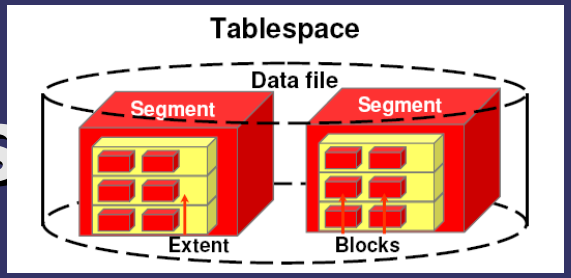


Logické úložné struktury





Tabulkové pros

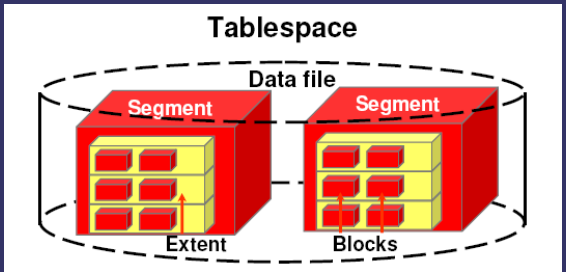


Tabulkový prostor databáze Oracle sestává z jednoho nebo více datových souborů. Jeden datový soubor může být součástí jediného tabulkového prostoru. Při instalaci Oracle 10g jsou vytvořeny dva tabulkové prostory: SYSTEM a SYSAUX.

Oracle 10g umožňuje vytvořit speciální typ tabulkového prostoru s názvem bigfile tablespace, jehož maximální velikost může být až 8EB (exabajtů , neboli miliónu terabajtů, tj. 10^{18} B).

Využitím tohoto typu tabulkového prostoru se stává správa databáze pro správce naprosto transparentní, jinými slovy to znamená, že správce může tabulkový prostor spravovat jako samostatnou jednotku bez nutnosti starat se o velikost a strukturu datových souborů.

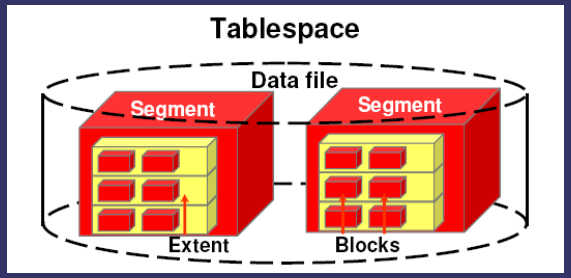
Bloky



Blok je nejmenší úložnou jednotkou databáze Oracle. Velikost bloku je číslo udávající počet bajtů, které blok zabírá v daném tabulkovém prostoru.

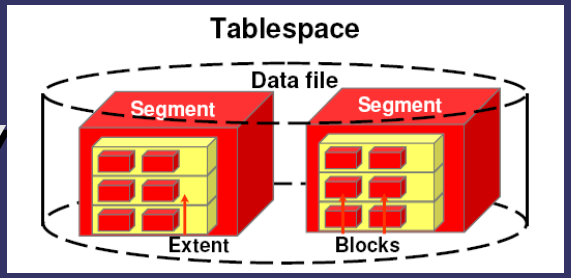
Aby byla zaručena efektivita provádění operací, bývá velikost bloku obvykle násobkem velikosti bloku, definované operačním systémem.

Rozsahy



Rozsah je další úrovní logického seskupování elementů v rámci databáze. Rozsah sestává z jednoho nebo několika bloků. Při zvětšování velikosti databázového objektu je nově přidaný prostor alokovaný právě jako rozsah.

Segmenty



Další vyšší úrovní logického seskupování elementů databáze je segment. Segment je skupina rozsahů, které dohromady tvoří databázový objekt, považovaný za jednotku, například tabulku nebo index.

V databázi Oracle rozlišujeme čtyři typy segmentů:

- datové segmenty,
- indexové segmenty,
- dočasné segmenty a
- návratové segmenty.

Segmenty

Datové segmenty

- Každá tabulka v databázi je uložena v datovém segmentu, který sestává z jednoho nebo více rozsahů. Pokud je tabulka rozdělená (partitioned) nebo clusterovaná (clustered), je pro ni alokováno více segmentů.

Indexové segmenty

- Každý index je uložen ve svém vlastním indexovém segmentu.

Dočasné segmenty

- V případě, kdy provedení příkazu jazyka SQL vyžaduje pro své dokončení diskový prostor (může to být například operace řazení, kterou není možné provést v operační paměti, je alokován dočasný segment. Dočasné segmenty existují pouze po dobu trvání příkazu jazyka SQL.

Segmenty

Návratové (rollback) segmenty

- v databázi Oracle 10g existují návratové segmenty pouze v tabulkovém prostoru SYSTEM a správce obvykle nemusí provádět jejich údržbu. V předchozích verzích databáze Oracle byl návratový segment využíván pro ukládání původních hodnot při manipulacích s daty a pro udržování konzistentních pohledů na data tabulek pro ostatní uživatele, kteří k tabulkám přistupovali.
- Návratové segmenty byly také využívány v průběhu obnovy dat pro vrácení změn transakcí, které byly aktivní v okamžik, kdy byla databázová instance neočekávaně ukončena.
- Ve verzi Oracle 10g se o automatickou alokaci a správu návratových segmentů v rámci návratového (undo) tabulkového prostoru stará technologie Automatic Undo Management. V návratovém tabulkovém prostoru jsou segmenty strukturovány podobně jako návratové segmenty s tím, že detaily správy těchto segmentů jsou pod kontrolou databáze Oracle a nespravuje je (ve většině případů neefektivně) správce databáze.

Schéma

- A schema is a named collection of objects
- A user is created, and a corresponding schema can be created
- User can be associated only with one schema
- Username and schema are often used interchangeably
- Users are not necessarily associated with a schema

Schema Objects

Tables

Triggers

Constraints

Indexes

Views

Sequences

Stored program units

Synonyms

User-defined data types

Database links

Fyzické úložné struktury

Databáze Oracle využívá na discích velké množství fyzických úložných struktur pro uložení a správu dat vzniklých transakcemi uživatelů.

Některé z těchto struktur obsahují aktuální uživatelská data:

- datové soubory,
- soubory protokolu a
- archivované soubory protokolu,

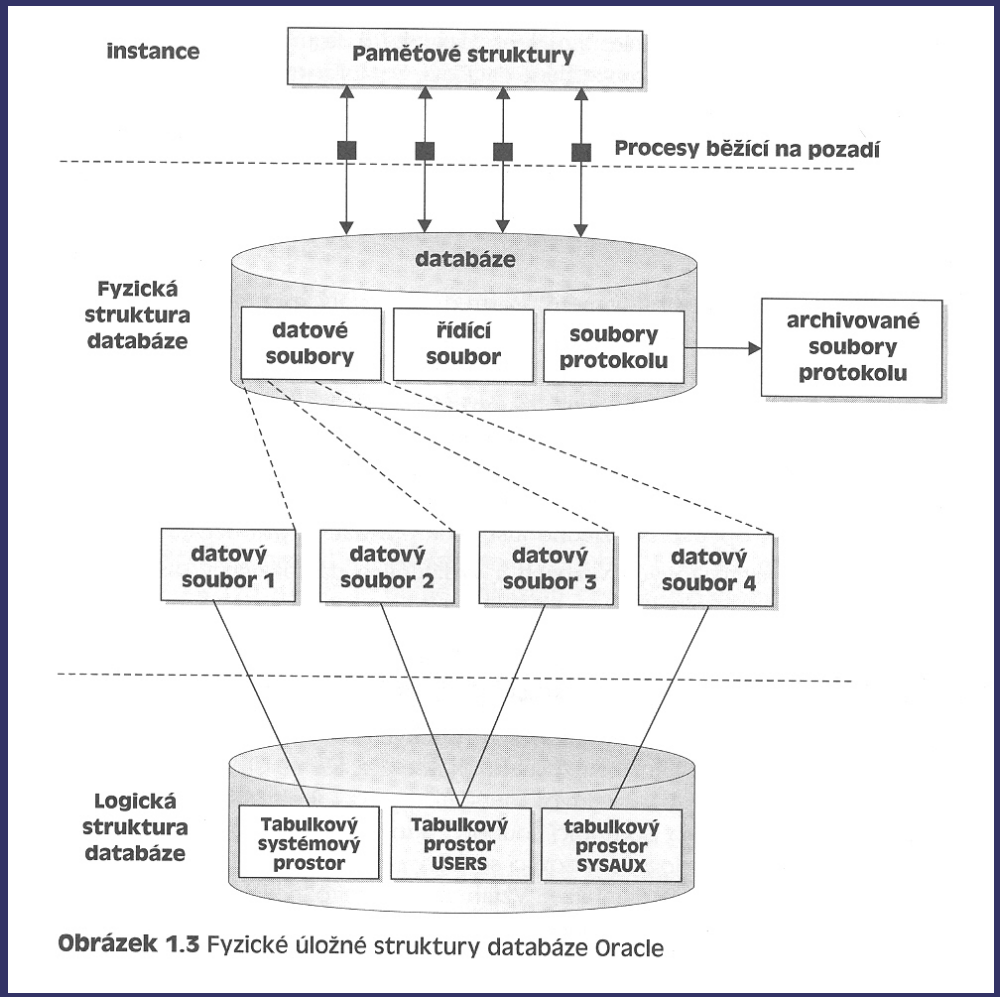
jiné struktury, jako jsou

- řídící soubory,

udržují stav databázových objektů a textové soubory s poplachy (alert 10gs) a trasovací soubory obsahují protokol s informacemi o událostech a chybových stavech databáze.

Fyzické úložné struktury

Vztah mezi fyzickými strukturami a logickými úložnými strukturami je na obrázku.



Obrázek 1.3 Fyzické úložné struktury databáze Oracle

Datové soubory

- Každá databáze Oracle musí obsahovat alespoň jeden datový soubor.
- Jeden datový soubor odpovídá jednomu fyzickému souboru operačního systému na disku.
- Každý datový soubor patří jednomu tabulkovému prostoru, tabulkový prostor však může sestávat z mnoha datových souborů.
- Datový soubor je místo, kde jsou uložena všechna data databáze. Bloky dat, ke kterým se přistupuje nejčastěji, jsou uloženy ve vyrovnávací paměti. Nové datové bloky nejsou okamžitě zapisovány do datového souboru, ale až v závislosti na aktivitě procesu, který data zapisuje. Před dokončením každé uživatelské transakce jsou změny, které transakce provede, vždy zapsány do souborů protokolu (redo log files).

Soubory protokolu (redo log)

- Při každém přidání, odstranění nebo změně dat v tabulce, indexu nebo jiném objektu databáze Oracle je zapsán záznam do aktuálního souboru protokolu.
- Každá databáze Oracle musí mít alespoň dva soubory protokolu, protože Oracle využívá tyto soubory kruhovým způsobem. Když je jeden soubor protokolu zaplněn záznamy souboru protokolu, je tento soubor označen jako ACTIVE v případě, kdy je potřebný pro případnou obnovu instance nebo jako INACTIVE, pokud není potřebný pro obnovu instance. Záznamy se pak začnou zapisovat do dalšího souboru protokolu ze seznamu od začátku souboru a tento soubor je označen jako CURRENT.

Soubory protokolu (redo log)

- V ideálním případě nejsou informace ze souboru protokolu nikdy použity. Pokud ovšem dojde k výpadku napájení nebo jakákoliv jiná příčina způsobí selhání instance Oracle, může se stát, že nové nebo aktualizované bloky dat z vyrovnávací paměti databáze nebyly zapsány do datových souborů. Při spuštění instance Oracle jsou jednotlivé záznamy ze souboru protokolu aplikovány na datové soubory operací roll forward, pomocí které se obnoví stav databáze až do bodu, ve kterém došlo k selhání.

- Aby bylo možné provést obnovu i v případě, kdy dojde ke ztrátě jednoho souboru protokolu ze skupiny souborů protokolu, je vhodné, aby existovaly vícenásobné kopie souborů protokolu na různých fyzických discích.

Řídící soubory

- Každá databáze Oracle má alespoň jeden řídící soubor, který obsahuje metadata databáze (data o fyzické struktuře databáze). Řídící soubor obsahuje kromě jiného informace o názvu databáze, době vytvoření databáze, názvech a umístěních všech datových souborů a souborů protokolu.

- Při změnách struktury databáze jsou informace okamžitě zapsány do řídícího souboru.

- Protože je řídící soubor tak důležitý pro provozuschopnost databáze, je vhodné jej udržovat ve více kopiích.

Archivované soubory protokolu

- Databáze Oracle může pracovat ve dvou režimech: v režimu archive10g nebo v režimu noarchivelog. Pokud je databáze v režimu noarchivelog, znamená to, že díky kruhovému využití souborů protokolu (online soubory protokolu) jednotlivé záznamy o transakcích nejsou v případě selhání disku dále dostupné.
- Práce v režimu noarchive10g ochrání integritu databáze v případě selhání instance nebo operačního systému, protože všechny transakce potvrzené operací commit, ale ještě nezapsané do datových souborů, jsou dostupné v online souborech protokolu.

Archivované soubory protokolu

- Při práci v režimu archive10g se zaplněné soubory protokolu kopírují na jedno nebo více umístění a je možné je použít pro rekonstrukci databázových dat v jakémkoliv časovém okamžiku v případě, kdy dojde k selhání média, na kterém je databáze uložená. Pokud například dojde k poruše na disku, který obsahuje datové soubory, obsah databáze je možné obnovit až do okamžiku vzniku poruchy v případě, kdy je k dispozici poslední záloha datových souborů a soubory protokolu, které byly vygenerovány od okamžiku vytvoření této zálohy až do vzniku poruchy.

Inicializační soubory parametrů

- Při spuštění instance databáze Oracle je nejprve alokována paměť instance databáze a otevře se jeden ze dvou typů inicializačního souboru parametrů. Je to buď textový soubor s názvem init<SID>.ora (známý i pod označením init.ora nebo PFILE) nebo soubor parametrů serveru (známý pod označením SPFILE).
- Inicializační soubor parametrů, bez ohledu na jeho formát, obsahuje umístění trasovacích souborů (trace files), řídících souborů, archivovaných souborů protokolu atd.
- V souboru jsou také uložena omezení velikostí různých struktur v paměťové oblasti SGA (System Global Area) a také informace o tom, kolik souběžně pracujících uživatelů se může připojit k databázi.

Soubory s protokoly poplachů a trasování

- Pokud dojde ke vzniku chyby při běhu databáze, Oracle obvykle zapisuje chybové zprávy do protokolu poplachů (alert log) nebo v případě procesů běžících na pozadí do trasovacího protokolu (trace log).
- Při spuštění nebo zastavení databáze je do protokolu poplachů zaznamenána informace, která obsahuje kromě jiného i seznam inicializačních parametrů, které mají jiné než výchozí hodnoty.
- Jsou zde zaznamenány i všechny příkazy alter database i alter system, které provedl správce databáze.
- Dále zde naleznete i operace, které se provádějí nad tabulkovými prostory nebo nad datovými soubory tabulkových prostorů, tzn. přidání tabulkového prostoru, odstranění tabulkového prostoru nebo přidání datového souboru do tabulkového prostoru. Jsou zde zaznamenány i všechny chyby vzniklé při běhu databáze, například informace o vyčerpání volného místa v tabulkovém prostoru, informace o poškozených souborech protokolu atd.

Paměťové struktury

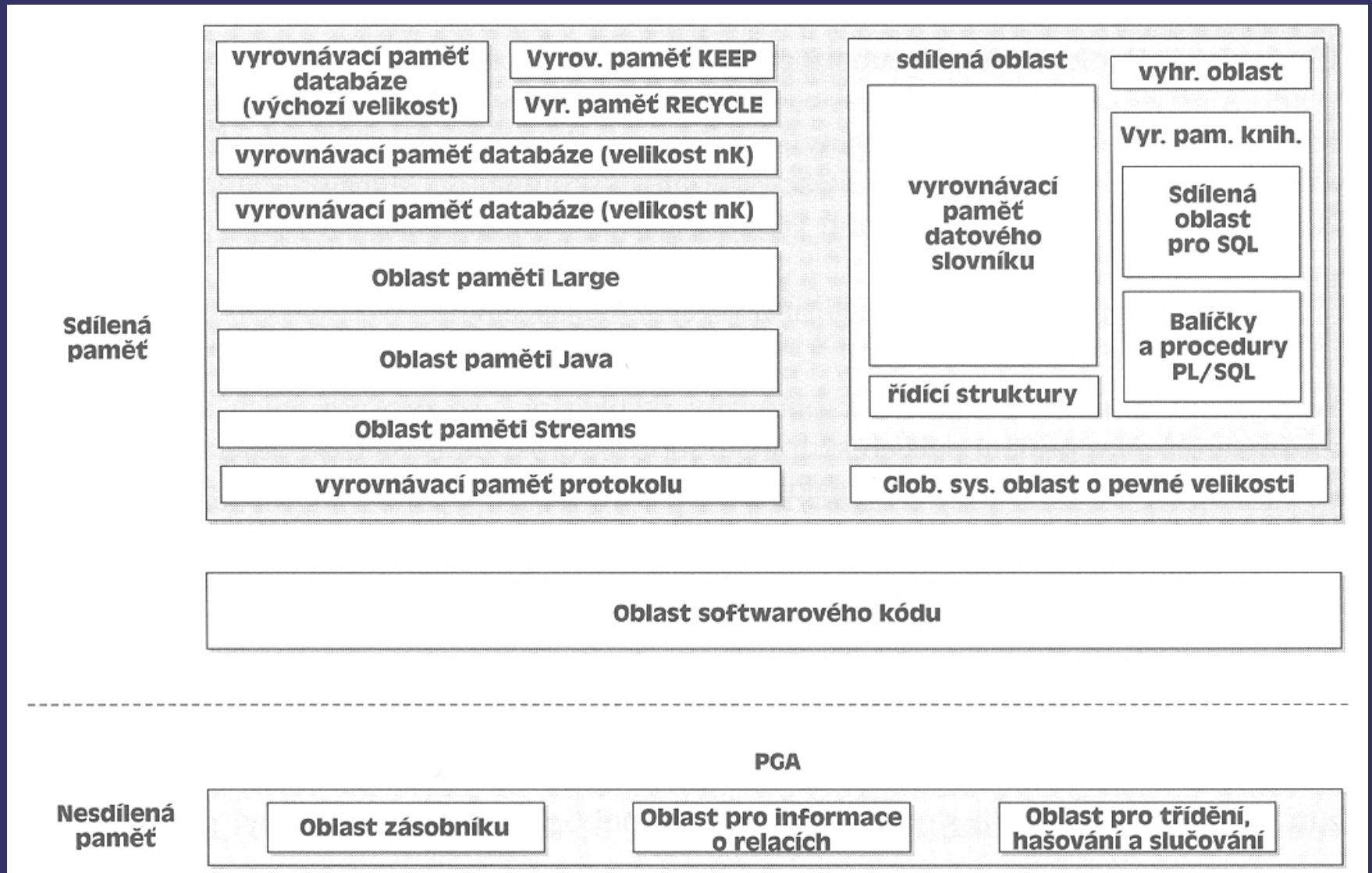
Oracle využívá fyzickou paměť serveru pro uložení různých komponent instance. V paměti je spustitelný kód, informace o relacích, jednotlivé procesy databáze a informace sdílené mezi procesy (třeba informace o uzamčení jednotlivých databázových objektů).

Paměťové struktury kromě jiného obsahují i uživatelské příkazy jazyka SQL i příkazy datového slovníku a také vyrovnávací paměť, jejíž obsah je dle potřeby ukládán na disk a která obsahuje datové bloky databázových segmentů a informace o dokončených databázových transakcích.

Datová oblast vyhrazená pro instanci Oracle se nazývá globální systémová oblast SGA (System Global Area). Spustitelný kód Oracle je uložen v oblasti pro softwarový kód.

Pro každý server a proces běžící na pozadí v paměti existuje oblast s názvem globální programová oblast PGA (Program Global Area).

Paměťové struktury



Globální systémová oblast SGA

Globální systémová oblast SGA (System Global Area) je skupina paměťových struktur instance Oracle sdílená uživateli databázové instance.

Při spuštění instance Oracle je pro oblast SGA vyhrazena paměť v závislosti na hodnotách nastavených v inicializačním souboru parametrů nebo napevno zakódovaných v softwaru Oracle. Některé parametry, které řídí velikost různých parametrů SGA, jsou dynamické.

Pokud je zadán parametr SGA_MAX_SIZE, celková velikost všech o oblastí SGA nesmí překročit hodnotu SGA_MAX_SIZE.

Paměť v oblasti SGA je alokována po jednotkách, nazývaných granule. Granule může mít velikost 4 MB nebo 16 MB v závislosti na celkové velikosti SGA. Pokud je velikost oblasti SGA menší nebo rovna 128 MB, granule má velikost 4MB, v opačném případě má velikost 16 MB.

Vyrovnávací paměť

Vyrovnávací paměť obsahuje bloky dat načtené z disku, které byly načteny z důvodů provádění příkazu select nebo které obsahují modifikované (dirty) bloky se změněnými nebo přidanými daty z příkazů pro manipulaci s daty.

Sdílená oblast

Sdílená oblast obsahuje dvě hlavní vyrovnávací paměti: vyrovnávací paměť knihoven a vyrovnávací paměť datového slovníku. Velikost sdílené oblasti je možné nastavit inicializačním parametrem SHARED_POOL_SIZE

Vyrovnávací paměť knihoven

Vyrovnávací paměť knihoven obsahuje informace o příkazech SQL a PL/SQL spuštěných v databázi. Díky tomu, že je tato vyrovnávací paměť sdílená pro všechny uživatele, může více databázových uživatelů sdílet stejný příkaz SQL.

Kromě vlastních příkazů jazyka SQL jsou v této vyrovnávací paměti uloženy také prováděcí plány a stromy rozkladu příkazů jazyka SQL. Pokud je stejný dotaz spuštěn podruhé, bez ohledu na to, jestli stejným nebo jiným uživatelem, je prováděcí plán a strom rozkladu dotazu již k dispozici, což zvyšuje rychlost provádění dotazů nebo příkazů pro manipulaci s daty.

Pokud je vyrovnávací paměť knihoven příliš malá, jsou prováděcí plány a stromy rozkladu odstraňovány z vyrovnávací paměti, ale to pak vyžaduje časté nahrávání příkazů SQL do vyrovnávací paměti.

Vyrovnávací paměť datového slovníku

Datový slovník je kolekce databázových tabulek, vlastněných schématy SYS a SYSTEM, která obsahuje metadata databáze, struktur databáze a oprávnění a rolí databázových uživatelů. Vyrovnávací paměť datového slovníku obsahuje bloky datového slovníku. Datové bloky z tabulek v datovém slovníku jsou využívány při provádění uživatelských dotazů a příkazů pro manipulaci s daty nepřetržitě.

Pokud je velikost vyrovnávací paměti datového slovníku příliš malá, požadavky na informace z datového slovníku způsobí dodatečné V/V operace. Tyto požadavky na data z datového slovníku se nazývají rekurzivní volání a z pohledu výkonu databáze by se jim mělo zabránit správným nastavením velikosti vyrovnávací paměti datového slovníku.

Vyrovnávací paměť protokolu

Vyrovnávací paměť protokolu obsahuje poslední prováděné změny datových bloků v datových souborech. Pokud je tato vyrovnávací paměť zaplněna z jedné třetiny nebo uplynuly tři sekundy, jsou záznamy zapsány do souborů protokolu.

Záznamy v souborech protokolu po zapsání do souborů protokolu mají důležitý význam při provádění obnovy dat databáze v případě, kdy instance z nějakého důvodu selže předtím, než jsou datové bloky zapsány z vyrovnávací paměti protokolu do datových souborů.

Transakce ukončená operací commit se nepovažuje za dokončenou, dokud nejsou všechny záznamy protokolu úspěšně zapsány do souborů protokolu.

Oblast paměti Large

Oblast paměti Large je volitelná paměťová oblast SGA. Používá se pro transakce, které probíhají nad více databázemi, pro buffery zpráv při provádění paralelních dotazů a pro paralelně prováděné operace zálohování nebo obnovy dat nástrojem RMAN. Jak název této paměti implikuje Large=velký), tato oblast poskytuje datové bloky pro operace, které vyžadují alokaci velkých datových bloků paměti.

Velikost této oblasti paměti je možné nastavit hodnotou inicializačního parametru LARGE_POOL_SIZE a od verze Oralce9i Release 2 je to dynamický parametr.

Oblast paměti Java

Oblast paměti Java používá stroj Oracle JVM (Java Virtual Machine) pro javovský kód a data v rámci uživatelské relace. Ukládání kódu a dat v této paměťové oblasti je analogické ukládání kódu SQL a PL/SQL do sdílené paměťové oblasti.

Oblast paměti Streams

Od verze Oracle 10g je možné nastavit velikost této paměťové oblasti hodnotou inicializačního parametru STREAMS_POOL_SIZE. Oblast paměti Streams obsahuje data a řídící struktury pro podporu vlastnosti Oracle Streams verze Oracle Enterprise Edition.

Oracle Streams spravuje sdílení dat a událostí v distribuovaném prostředí. Pokud inicializační parametr STREAMS_POOL_SIZE není nastaven nebo je nastaven na hodnotu 0, je paměť potřebná pro operace Streams alokována ze sdílené oblasti a tato oblast může využít až 10 procent velikosti sdílené oblasti.

Globální programová oblast

Globální programová oblast PGA (Program Global Area) je oddíl paměti alokovaný pro privátní použití jedním procesem. Konfigurace PGA závisí na konfiguraci připojení databáze Oracle, může to být buď sdílený server (shared server) nebo vyhrazený server (dedicated server).

U konfigurace se sdíleným serverem uživatelé sdílejí připojení k databázi, čímž se minimalizuje využití paměti na serveru, ale teoreticky se může prodloužit doba odezvy na uživatelské požadavky.

V prostředí se sdíleným serverem jsou informace o uživatelských relacích uloženy v oblasti SGA místo v oblasti PGA. Prostředí se sdíleným serverem jsou ideální pro velký počet současných připojení k databázi s malým množstvím krátce trvajících požadavků.

Globální programová oblast

V prostředí s vyhrazeným serverem má každý uživatelský proces vlastní připojení k databázi a paměť vyhrazená pro relace je v oblasti PGA.

Oblast PGA také obsahuje oblast pro řazení. Oblast pro řazení se použije vždy, když uživatelský požadavek vyžaduje provést řazení, rastrové třídění nebo operace spojení tabulek s operací hašování.

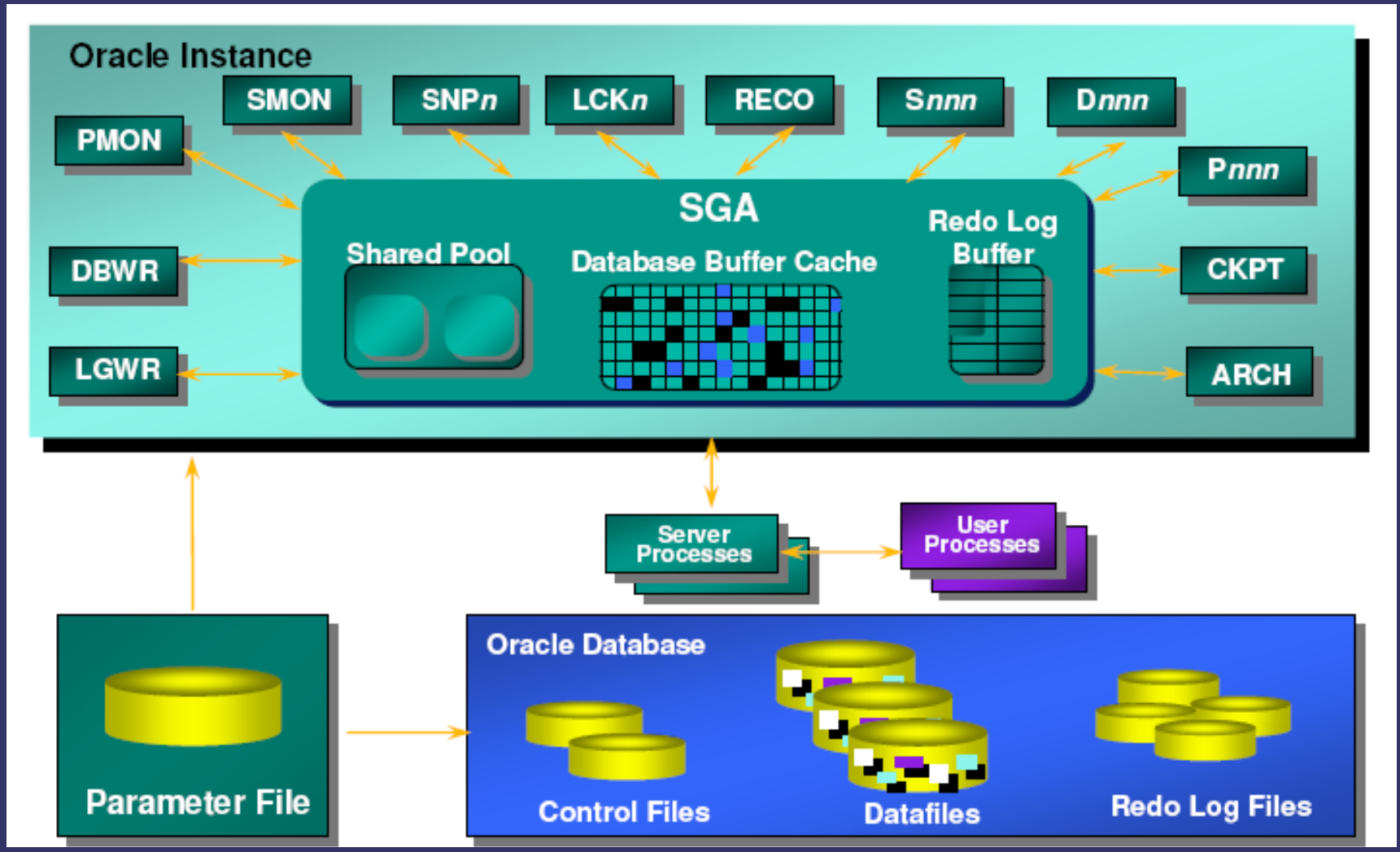
Oblast softwarového kódu

Oblast softwarového kódu obsahuje spustitelné soubory Oracle, které jsou spuštěny jako součást instance Oracle. Tyto oblasti kódu jsou svou povahou statické a mění se pouze s instalací nové softwarové verze.

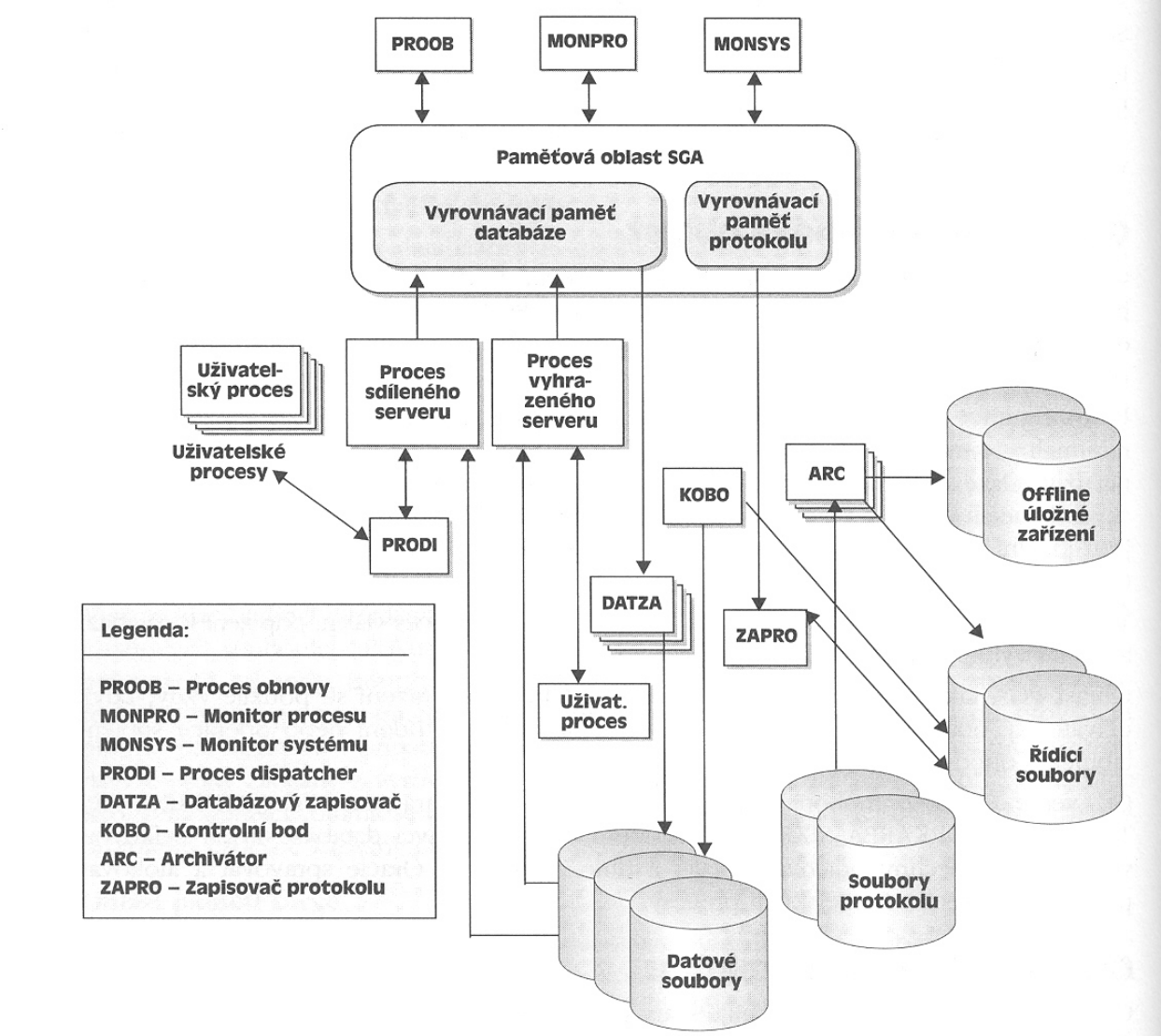
Oblasti softwarového kódu Oracle jsou umístěny v privilegované paměťové oblasti odděleně od ostatních uživatelských programů.

Softwarový kód Oracle je určen striktně jen pro čtení a může být instalován jako sdílený nebo nesdílený. Sdílená instalace softwarového kódu Oracle může ušetřit paměť v případě, kdy je na stejném serveru spuštěno více instancí Oracle stejné softwarové verze.

Oracle architektura - procesy



Oracle architektura - procesy



Jiný obrázek – pozor, zkratky procesů jsou počeštěny!

Proces SMON

Proces SMON je tzv. System Monitor.

V případě pádu systému nebo selhání instance díky výpadku napájení nebo selhání procesoru proces SMON provede obnovu instance aplikováním záznamů z online souboru protokolu na datové soubory.

Kromě toho při restartu systému zabezpečuje tento proces čištění dočasných segmentů ve všech tabulkových prostorech.

Jednou z úloh procesu SMON je i pravidelné slučování volného místa v tabulkových prostorech u tabulkových prostorů řízených slovníkem.

Online transakční žurnál

K dispozici jsou 2 režimy:

- ARCHIVELOG – všechny změny databáze jsou trvale uloženy v archivovaném transakčním žurnálu, tento režim umožňuje zotavení nejen po selhání instance databázového serveru, ale také po selhání diskového media – tedy při poškození datových souborů

- NOARCHIVELOG – pouze zotavení při selhání instance, zotavení bude provedené jen z aktivního transakčního žurnálu

- Stav zjistíme dotazem ARCHIVE LOG LIST;
- Start archivace ARCHIVE LOG START;

Online transakční žurnál

Aby se mohl režim změnit, je třeba instanci restartovat (to není pro databázový server a jeho administrátora obvykle moc žádoucí, ale v tomto případě nevyhnutelné).

Proces zastavení může proběhnout ve 4 režimech

- Normal – se zastavením se čeká na odpojení všech aktuálně připojených uživatelů
- Immediate – v tomto režimu jsou před zastavením odvolány všechny aktivní transakce a následně odpojeni všichni uživatelé
- Abort – v tomto režimu bude zastavení provedeno okamžitě a „bezohledně“
- Transactional – všichni aktivní uživatelé budou odpojeni po ukončení svých transakcí a až poté dojde k zastavení

Samotný průběh restartu může trvat dlouhou dobu.

Proces PMON

Pokud je uživatelské připojení přerušeno nebo uživatelský proces selže z jiného důvodu, provede proces PMON (Process Monitor), potřebné úklidové práce.

Vyčistí vyrovnávací paměť a ostatní prostředky, které uživatelské připojení používalo. Uživatelská relace například mohla provádět aktualizaci některých řádků v tabulce a tím tyto řádky uzamknout.

Proces PMON

Pokud díky bouřce vypadne napájení klientského počítače a relace SQL*Plus zmizí s výpadkem proudu, proces PMON zjistí, že dané připojení bylo přerušeno a již neexistuje a provede následující:

- Vrátí zpět změny provedené transakcemi, které probíhaly při výpadku napájení.
- Označí ve vyrovnávací paměti bloky, použité transakcemi jako volné.
- Odstraní uzamčení na odpovídajících řádcích tabulky.
- Odstraní identifikátor odpojeného procesu ze seznamu aktivních procesů.

Proces PMON také komunikuje s listenery a poskytuje jim informace o stavu instance pro požadavky na příchozí připojení.

Proces DBWn

Proces DBWn, nazývaný databázový zapisovač, ve starších verzích systému Oracle pod názvem DBRW zapisuje nové nebo změněné datové bloky (dirty blocks) z vyrovnávací paměti do datových souborů.

S využitím algoritmu LRU proces DBWn zapisuje jako první nejstarší, nejméně aktivní bloky. Výsledkem je, že v paměti zůstávají nejčastěji používané bloky i v případě, kdy jsou označeny jako změněné (dirty).

V databázi může být spuštěno až 20 procesů, DBWO až DBW9 a DBWa až DBWj. Počet procesů DBWn je řízen hodnotou parametru DB_WRITER_PROCESSES.

Proces LGWR

Proces LGWR, neboli zapisovač protokolu řídí správu vyrovnávací paměti protokolu. Proces LGWR je nejaktivnější proces v instanci s velkou aktivitou příkazů pro manipulaci s daty.

Transakce není považována za dokončenou, dokud proces LGWR nezapíše úspěšně všechny záznamy, včetně záznamu o operaci commit, do souborů protokolu.

Bloky z vyrovnávací paměti označené pro zápis není možné zapsat do datových souborů procesem DBWn dokud proces LGWR nedokončí zápis všech záznamů do protokolu.

Proces LGWR

Pokud jsou soubory protokolu sdruženy do skupin a jedna z kopií souboru protokolu ve skupině je poškozená, LGWR zapíše informace do zbývajících souborů a zaznamená chybovou zprávu v souboru protokolu poplachů.

Pokud jsou nepoužitelné všechny soubory ze skupiny, proces LGWR selže a celá instance se zastaví, dokud není problém odstraněn.

Proces ARCn

Pokud je databáze v režimu ARCHIVELOG, proces archivátor neboli ARCn provádí kopírování souborů protokolu na ostatní nadefinovaná umístění (složky, zařízení nebo síťové umístění) vždy, když se soubor protokolu zaplní a informace se začnou zapisovat do dalšího souboru protokolu v řadě.

Proces ARCn

V optimálním případě proces archivátoru dokončí kopírování předtím, než je kopírovaný soubor opět vyžádán pro zápis záznamů. V opačném případě může dojít k vážnému problému s výkonem - uživatelé nemohou dokončit transakce, dokud nejsou všechny záznamy o transakci zapsány do souborů protokolu a soubor protokolu nemůže přijímat nové záznamy, protože se pořád kopíruje na archivní umístění.

Existují minimálně tři řešení tohoto problému: zvětšit soubory protokolu, zvětšit počet skupin souborů protokolu nebo zvětšit počet procesů ARCn. Pro každou instanci může být spuštěno až 10 procesů ARCn, to se nastaví hodnotou inicializačního parametru

Proces CKPT

Proces kontrolní bod (checkpoint) pomáhá snižovat množství času, potřebného pro obnovu instance. Při provádění kontrolního bodu proces CKPT aktualizuje záhlaví řídícího souboru a datových souborů tak, aby aktualizovaná data obsahovala poslední úspěšně provedenou změnu SCN (System Change Number).

Kontrolní bod nastává automaticky pokaždé, když dojde k přepnutí mezi soubory protokolu. Procesy DBWn zapisují změněné datové bloky tak, aby se posunul bod, ze kterého může začít obnova instance, čímž snižuje střední dobu do obnovy MTTR (Mean Time to Recovery).

Proces RECO

Proces RECO, proces obnovy, ošetřuje selhání distribuovaných transakcí (transakcí, které zahrnují změny v tabulkách ve více databázích).

Pokud je tabulka v databázi DB1 změněna současně s tabulkou v databázi DB2 a síťové připojení se přeruší předtím, než je možné aktualizovat tabulku v databázi DB2, proces RECO provede vrácení změn selhavší transakce.

Průběh zkoušky

1. Podmínka zápočet ze cvičení

2. Na začátku rychlý test obsahující
   - Teorii databází, relací
   - Logický a fyzický návrh databáze, normalizace tabulek
   - Veškerou teorii probíranou na přednáškách

   Na otázky bude 0 až N správných odpovědí. Předpokládaný počet otázek 20, čas 15 minut.

1. Praktický test
   - Realizace objektů (všech probíraných) dle zadání
     (z důvodů automatické kontroly bude vyžadováno naprosté naplnění zadání počínaje jmény objektů i funkcí objektů, přiřazení práv)
   - Sestavení optimálního exekučního plánu

   V této části je povoleno použití literatury či donesených elektronických dokumentů. Internet bude zakázán.
   Předpokládaná doba řešení 60-90 minut.

1. Ústní část spočívající
   - V diskusi prací řešených na cvičeních
   - V otázce z teorie