FAKULTA INFORMATIKY, MASARYKOVA UNIVERSITA V BRNĚ

Slide k přednášce

PV003 – Architektura relačních databází

Milan Drášil, 2004 - 2011

1 DATOVÉ MODELOVÁNÍ, ENTITNĚ-RELAČNÍ MODEL, OD KONCEPTUÁLNÍHO DATOVÉHO MODELU K FYZICKÉMU

Slouží k popisu datových struktur potřebných pro infomarční systém na konceptuální (abstraktní) úrovni.

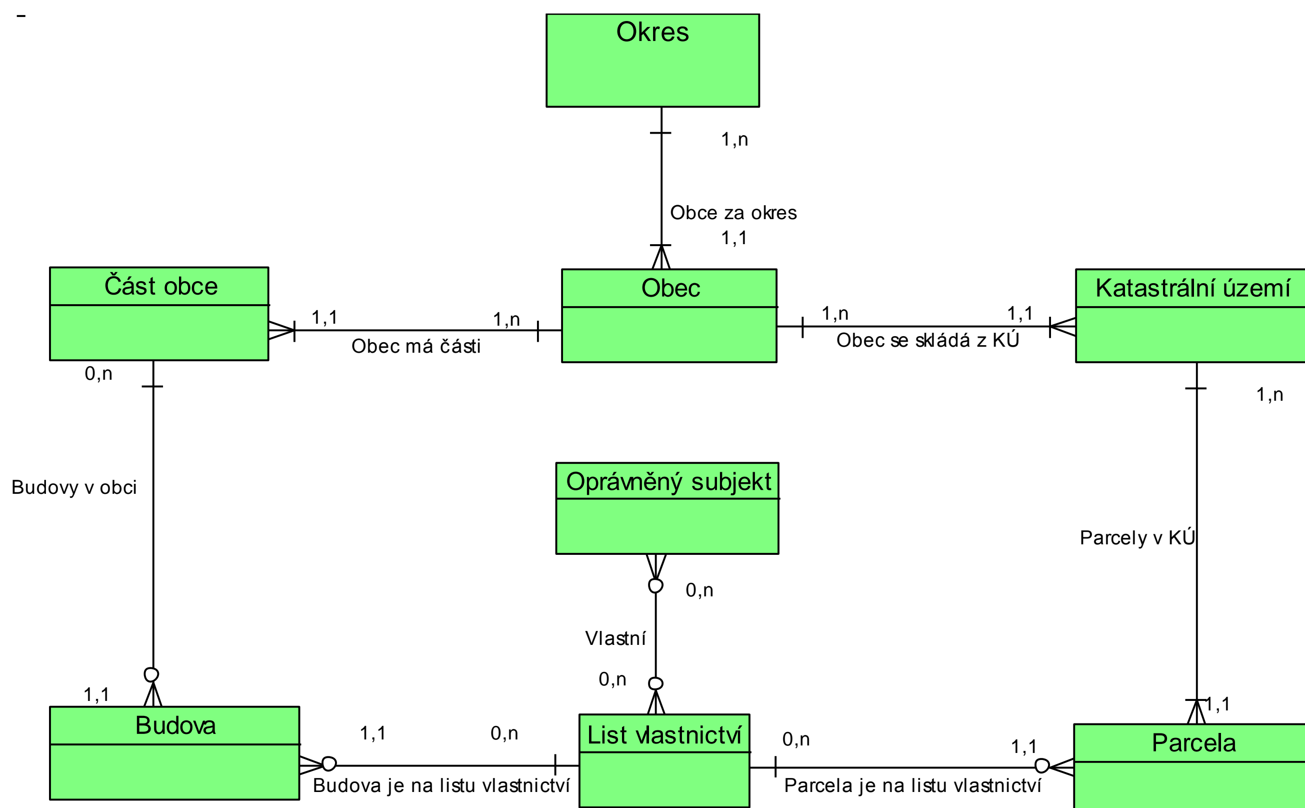
<u>Entita</u> (entitní typ) – je typ objektu zájmu informačního systému, který je jednoznačně identifikovatelný. Instance entity múže být fyzicky existující objekt (dům, osoba), nebo také událost (prodej auta).

<u>Vztah</u> (relationship) – entity mohou být spolu svázány vztahy (auto „bylo prodáno“ osobě).

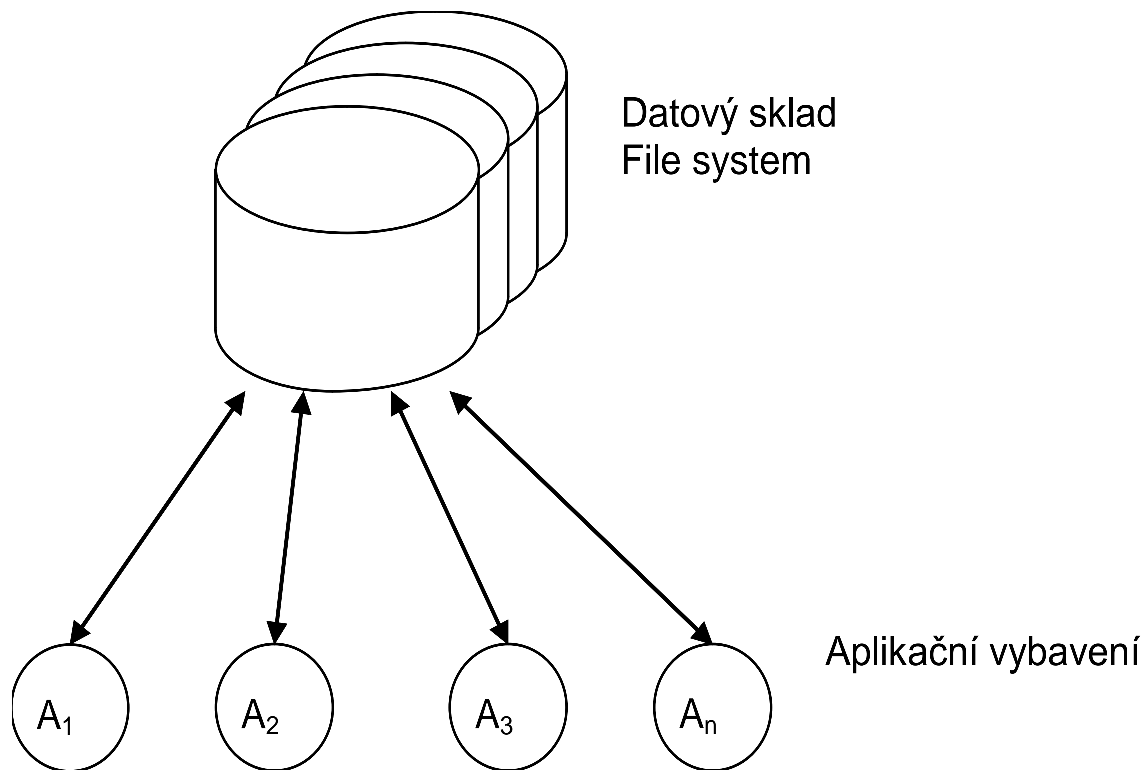
Entity (a vztahy) mohou obsahovat atributy.

Entity a vztahy znázorňujeme E-R diagramy, entity vyjadřujeme jako obdélníky, vztahy jako spojnice. Vztahy mohou mít určenu kardinalitu.

-



Souborově orientované systémy:

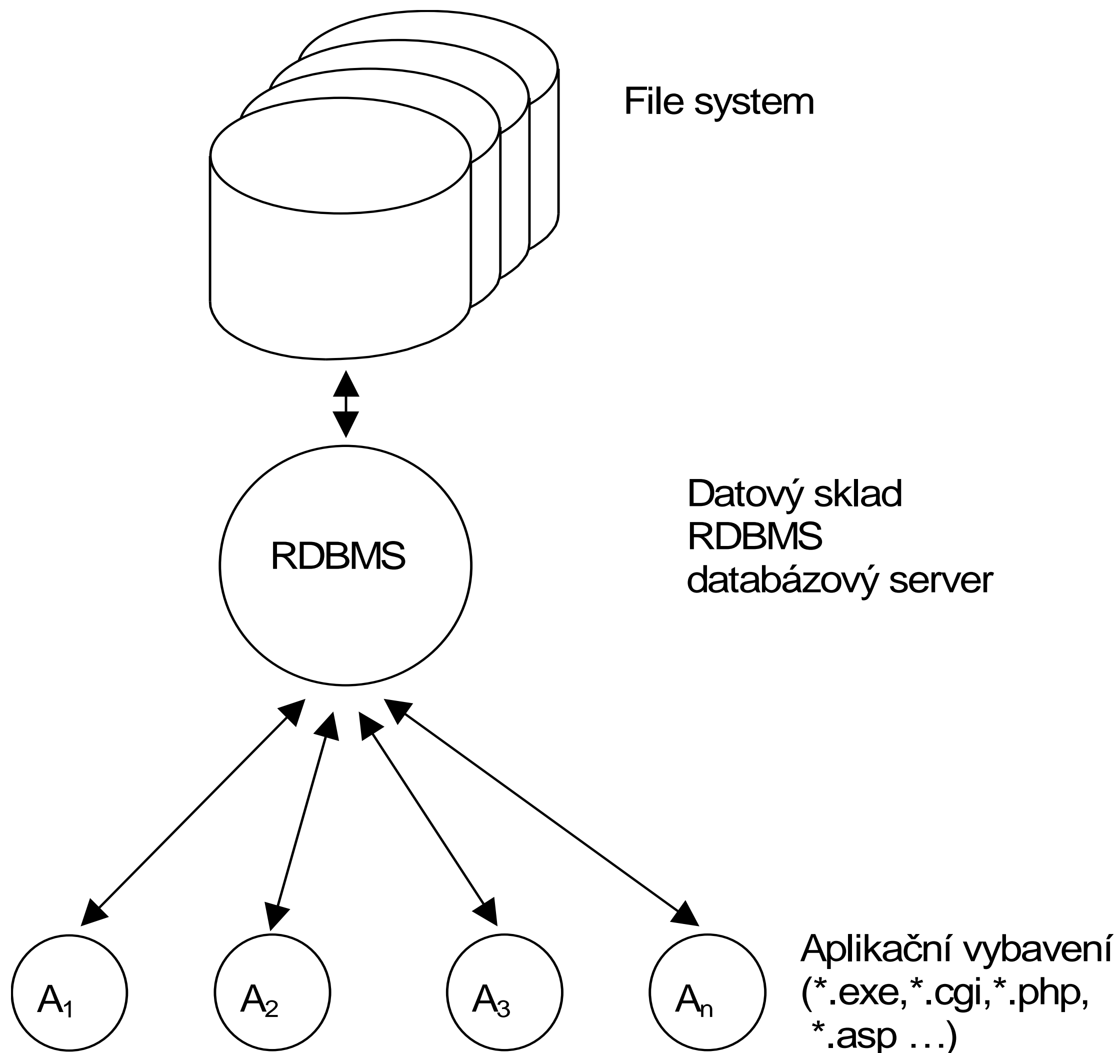


Výhody:
- **optimalizace datových struktur vzhledem k řešené problematice**
- **menší systémové nároky**

Nevýhody:

- **aplikační závislost**
- **obtížné zabezpečení konzistence dat**
- **obtížná realizace konkurentních operací (zamykání souborů)**
- **obtížná čitelnost – dokumentovatelnost datového modelu**
- **téměř nemožný transakční přístup pro operaci s daty**
- **obtížné řízení přístupových práv**

Databázově orientované systémy:



Výhody:

- **aplikační „nezávislost“**
- **snadné zabezpečení konzistence dat**
- **snadná realizace konkurentních operací**
- **snadná čitelnost - dokumentovatelnost datového modelu**
- **standardizovaná data umožňují i standardní vývoj IS, strukturovanou analýzu problematiky (vývoj pomocí prostředků CASE), od konceptuálního datového modelu je přechod do fyzického DM takřka automatizovatelný.**
- **neprocedurální přístup k datům (tj. neříkám „jak“ to chci, ale „co“ chci)**

Nevýhody:

- **obtížná implementace nestandardních přístupových technik**
- **obtížná implementace komplikovanějších datových struktur, je nutné je normalizovat do tabulek a to může zpomalit přístup k datům**
- **neprocedurální přístup k datům**

2 RELAČNÍ ALGEBRA

Aby bylo možné standardizovat přístup k datům pomocí databáze, bylo nutné standardizovat datové struktury.

Běžně lze ukládat data ve formě záznamů, jejichž jednotlivé položky mají týž (sémantický) význam (převod z E-R modelu 1:1):

<u>Příklad 1. (data o podniku):</u>

Zam:

ID	Jmeno	Prijmeni	Odd
1	Přemysl	Novák	Pro
2	Jan	Procházka	Pro
3	Jiřina	Tichá	Adm
..	..		

Odd:

ID	Nazev
Adm	Administrativa
Pro	Provoz
..	

Proj:

DP-ISKN	Dálkový přístup ISKN
ÚAP	Územní analytika
..	

ZaP:

Zam	Projekt	Procent
1	ÚAP	50
1	DP-ISKN	50
2	DP-ISKN	100
..		

Nad těmito strukturami budeme požadovat obecný aparát, který nám bude poskytovat data například v této formě:

Přemysl	Novák	Provoz	ÚAP	50
Přemysl	Novák	Provoz	DP-ISKN	50
Jan	Procházka	Provoz	DP-ISKN	100

Relační databáze (Codd červen 1970, ACM Journal, Communications of ACM):

- **datové struktury jsou n-ární relace**
- **relace je reprezentována tabulkou, výčtem pravdivých n-tic relace**
- **nad relacemi jsou proveditelné operace relační algebry**

Relační algebra:

n - ární relace R - nad množinami $M_1, .., M_n$ (domény relace) je jakákoli podmnožina:

$$R \subseteq M_1 \times .. \times M_n$$

Projekce relace $\Pi(R, (M_x..M_y))$ - vznikne z relace R tak, že do ní zahrneme pouze vyjmenované domény ($M_x..M_y$).

Selekce $\sigma(R, p)$ – je podmnožina relace R splňující danou podmínku (predikát) *p*.

Součin relací A x B jsou všechny m+n-tice $(a_1, ..a_m, b_1.., b_n)$, kde $(a_1, ..a_m) \in A$ a $(b_1.., b_n) \in B$.

Sjednocení relací (stejných typů) - je běžné množinové sjednocení

Průnik relací (stejných typů) - je běžný množinový průnik

Příklad 2.:

Požadavek na data z Příkladu 1. Lze formulovat v relační algebře:

Π(σ(Zam x Odd x ZaP,Zam.Odd=Odd.ID ∧ Zam.ID=Zap.Zam),(Zam.Jmeno, Zam.Prijmeni, Odd.Nazev, ZaP.Projekt,ZaP.Procent))

Zam.ID	Zam.Jmeno	Zam.Prijmeni	Zam.Odd	Odd.ID	Odd.Nazev	ZaP.Zam	ZaP.Projekt	Zap.Procent
1	Přemysl	Novák	Pro	Adm	Administrativa	1	ÚAP	50
1	Přemysl	Novák	Pro	Adm	Administrativa	1	DP-ISKN	50
1	Přemysl	Novák	Pro	Adm	Administrativa	2	DP-ISKN	100
1	Přemysl	Novák	Pro	Pro	Provoz	1	ÚAP	50
1	Přemysl	Novák	Pro	Pro	Provoz	1	DP-ISKN	50
1	Přemysl	Novák	Pro	Pro	Provoz	2	DP-ISKN	100
2	Jan	Procházka	Pro	Adm	Administrativa	1	ÚAP	50
2	Jan	Procházka	Pro	Adm	Administrativa	1	DP-ISKN	50
2	Jan	Procházka	Pro	Adm	Administrativa	2	DP-ISKN	100
2	Jan	Procházka	Pro	Pro	Provoz	1	ÚAP	50
2	Jan	Procházka	Pro	Pro	Provoz	1	DP-ISKN	50
2	Jan	Procházka	Pro	Pro	Provoz	2	DP-ISKN	100
3	Jiřina	Tichá	Adm	Adm	Administrativa	1	ÚAP	50
3	Jiřina	Tichá	Adm	Adm	Administrativa	1	DP-ISKN	50
3	Jiřina	Tichá	Adm	Adm	Administrativa	2	DP-ISKN	100
3	Jiřina	Tichá	Adm	Pro	Provoz	1	ÚAP	50
3	Jiřina	Tichá	Adm	Pro	Provoz	1	DP-ISKN	50
3	Jiřina	Tichá	Adm	Pro	Provoz	2	DP-ISKN	100

3 JAZYK RELAČNÍ DATABÁZE

Požadavky na jazyk relační databáze:

- **vytváření, modifikace a rušení relací**

- **dotazy nad tabulkami tj. implementace relační algebry**

- **vkládání, změna, odstranění řádku v tabulce**

- **garance konzistence dat**

- **řízení přístupových práv**

- **řízení transakcí**

<u>Krátká historie SQL:</u>

- **IBM se věnovala vývoji jazyka, který by "lidským" způsobem zabezpečil operace nad relacemi, vznikl jazyk SEQUEL (Structured English Query Language)**
-
- **Z SEQUEL (už se angličtině moc nepodobal) později vznikl jazyk SQL**

Structured Query Language

- **dnes všeobecně uznáván za standard pro komunikaci s relačními databázemi.**

- **Jsou kodifikovány standardy SQL (ANSI, ISO/IEC)**

Vývoj v komerčních firmách jde vývoj (pochopitelně!) rychleji, než práce standardizačních komisí ⇒ univerzální standard neexistuje jednotlivé implementace se liší (ORACLE, MS-SQL, INFORMIX, DB2, MySQL.Postgre, Sybase)

Části jazyka SQL

- **Definiční část – Data Definition Language**
- **Manipulační část – Data Manipulation Language**
- **Řízení transakcí – Transaction Control**

Procedurální nadstavby

- **Transact SQL (MS-SQL,Sybase)**
- **PL/SQL (Procedural Language/SQL, ORACLE)**

Souborový přístup k datům:

```
FILE *inf;
inf=fopen(…);
while(  )
 {

  fseek(inf,…);
  fread(inf,…);

 }
```

Použití databázového stroje:

```
string sql =
 "select jmeno, prijmeni, plat  from zamestnanci";

Cursor cursorRes = OpenCursor(sql);

while
(
 (object fetchRes = FetchCursor(cursorRes) ) !=null
)
{
 //Zpracuj..

}

CursorRes.Close();
```

4 LEXIKÁLNÍ KONVENCE SQL A ZÁKLADNÍ DATOVÉ TYPY

Příkaz jazyka SQL může být víceřádkový mohou být použity tabelátory. Tedy příkaz

```
SELECT ENAME, SAL*12, MONTHS_BETWEEN
(HIREDATE,SYSDATE) FROM EMP;
```

a příkaz

```
SELECT
    ENAME,
    SAL * 12,
    MONTHS_BETWEEN( HIREDATE, SYSDATE )
FROM EMP;
```

jsou ekvivalentní.

Velká a malá písmena nejsou podstatná v rezervovaných slovech jazyka SQL a identifikátorech. Tedy příkaz:

```
SELECT
    ename,
    sal * 12,
    month_between( HIREDATE, SYSDATE )
FROM emp;
```

je ekvivalentní s předchozími příkazy.

Některé databázové stroje (MySQL) lze instalovat jako „case/accent sensitive“, tedy rozlišuje i diakritická znaménka a malá/velká písmena.

V databázích je zavedena konvence pro vynucení case/accent sensitivity, například v DB ORACLE jsou to uvozovky:

```
CASE_ACCENT_INSENSITIVE_IDENTIFIER

“Identifikátor zohledňující velká/malá a diakritiku“
```

Základní elementy jazyka SQL:

- **Konstanty (101,`'text'`,`'''něco jiného'''`)**
- **Integer (5803042157)**
- **Number (580304.2157)**
- **Datové typy (`int, number(m,n), date, varchar(n),long, long raw`)**
- **`NULL` speciální hodnota pro prázdnou hodnotu**
- **Komentáře (`/*     */`)**
- **Objekty databázového schématu (tabulky, pohledy, indexy, sekvence, …)**

Z uvedeného vyplývá, že příkazy jazyka jsou závislé na zadaném databázovém schématu, tedy jeden příkaz SQL může být syntakticky správný v jednom schématu a v jiném nikoli. Například dotaz na tabulku je syntakticky špatně, když ve schématu tabulka daného jména neexistuje.

5 DDL – DATA DEFINITION LANGUAGE

Vytváření tabulek příkaz `create table`

```
CREATE TABLE scott.emp
 (
  empno NUMBER,
  ename VARCHAR2(10)
 );
```

```
CREATE TABLE emp
 (
  empno  NUMBER        CONSTRAINT pk_emp PRIMARY KEY,
  ename  VARCHAR2(10)  CONSTRAINT nn_ename NOT NULL
                       CONSTRAINT upper_ename
                       CHECK (ename = UPPER(ename)),
  job    VARCHAR2(9),
  mgr    NUMBER        CONSTRAINT fk_mgr
                        REFERENCES scott.emp(empno),
  hiredt DATE          DEFAULT SYSDATE,
  sal    NUMBER(10,2)  CONSTRAINT ck_sal
                        CHECK (sal > 500),
  comm   NUMBER(9,0)   DEFAULT NULL,
  deptno NUMBER(2)     CONSTRAINT  nn_deptno NOT NULL
                       CONSTRAINT fk_deptno
                         REFERENCES scott.dept(deptno)
);
```

Modifikace tabulek - příkaz `alter table`

Přidání sloupce:

```
ALTER TABLE emp ADD ssn varchar2(32);
```

Změna typu sloupce:

```
ALTER TABLE emp modify date_of_birth (26);
```

Odebrání sloupce:

```
ALTER TABLE emp DROP COLUMN date_of_birth;
```

Integritní omezení:

Primární klíč:

```
ALTER TABLE EMP ADD CONSTRAINT
 pk_emp PRIMARY KEY (empno,deptno);
```

Cizí klíč:

```
ALTER TABLE EMP ADD CONSTRAINT fk_deptno
 FOREIGN KEY (deptno) REFERENCES scott.dept(deptno);
```

Přehled integritních omezení:

NOT NULL	**Vyplnění sloupce je povinné**
UNIQUE	**Sloupec (sloupce) má unikátní hodnoty v celé tabulce**
PRIMARY KEY	**Primární klíč tabulky**
REFERENCES	**Referenční integrita, hodnota sloupce je hodnotou primárního klíče jiné (stejné) tabulky**
CHECK	**Kontrola vloženého řádku**

Indexování tabulek příkaz - `create index`

Index je uspořádaný seznam všech hodnot jednoho nebo více sloupců:

- **rychlý přístupu k řádkům tabulek**
- **přístupu do tabulek v po řadě podle uspořádání**

```
CREATE INDEX emp_idx1 ON emp (ename, job);
```

není totéž, co

```
CREATE INDEX emp_idx1 ON emp (job, ename);
```

Pro získání jednoznačné hodnoty typu INT (celé číslo) slouží tzv. sekvence. Obyčejně jsou využívány v těch situacích, kde neexistuje objektivní primární klíč v relační tabulce. Hodnota sekvence je generována nezávisle na transakčním zpracování. Ke každé sekvenci přistupujeme pomocí pseudosloupců:

`CURRVAL` **vrací současný stav sekvence**
`NEXTVAL` **vrací následný stav sekvence**

```
CREATE SEQUENCE SEQ1;
```

```
CREATE SEQUEMCE SEQ1
 START WITH 32 INCREMENT BY 100;
```

Příkaz `DROP`:

`DROP` *`typ_objektu jméno_objektu`* **odstraní objekt z datového schématu.**

Např.

`DROP PUBLIC SYNONYM S1;` **odstraní ze schématu synonymum s1**

Klauzule `CASCADE CONSTRAINTS` odstraní intergritní omezení související s touto tabulkou.

```
DROP TABLE OKRES CASCADE CONSTRAINTS;
```

odstraní i integritní omezení `P01_OBEC_FK01`

Synonyma:

```
CREATE PUBLIC SYNONYM T1 FOR TABULKA1;
CREATE PUBLIC SYNONYM TABULKA1
 FOR U1.TABULKA1;
```

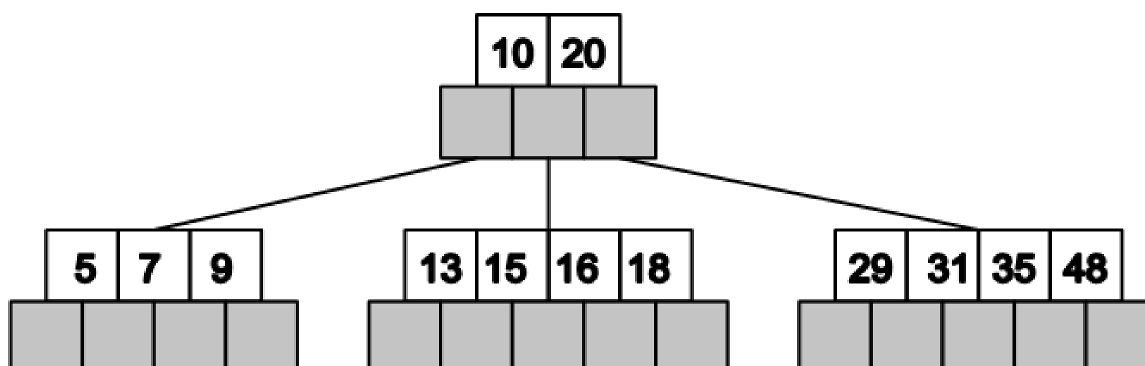
6 INDEXOVACÍ METODY

B-tree

Je základní vyhledávací metoda implementovaná ve všech DB strojích. Je stromová struktura, ve které uzly (indexové stránky, index pages), uchovávají více klíčů z úplně uspořádané množiny s těmito vlastnostmi:

- každý uzel má maximálně `m` synů
- každý uzel, s výjimkou kořene a listů, má minimálně `m/2` synů
- kořen má minimálně 2 syny, pokud není list
- všechny listy jsou na stejné úrovni
- nelistový uzel s `k` syny obsahuje `k-1` klíčů
- pro klíče v uzlu $key_1, \ldots, key_k$ jsou vzestupně uspořádány
- ukazatel p_i ukazuje na uzel, jehož všechny klíče jsou v intervalu $[key_i, key_{i+1}]$ (formálně předpokládáme, že $key_0 = -\infty$ a $key_{k+1} = \infty$).

Uzly B$^+$ stromu mají tedy tvar $p_0 key_1 p_1 \ldots p_{k-1} key_k p_k$.



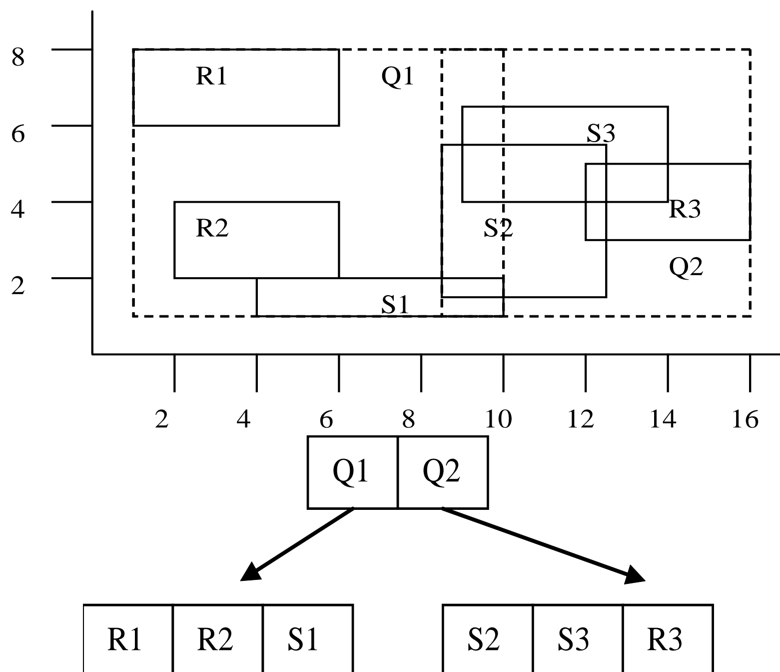
<u>R-tree</u>

Analogie k B-stromům, klíče jsou 2D (obecně, nD) obdélníky.

M – maximální počet klíčů v uzlu,
m ≤ M/2 – minimální počet klíčů v uzlu

<u>**Definice R-strom:**</u>

- **Každý uzel obsahuje minimálně m klíčů a maximálně M klíčů pokud není kořen.**
- **Klíče v R-stromech jsou obdélníky s ukazateli na synovské uzly, v listech obdélníky.**
- **Pro synovské uzly platí, že jejich klíče (tj. obdélníky) jsou uvnitř “otcovského” obdélníku.**
- **Listy stromu jsou na téže úrovni.**
- **Kořen obsahuje minimálně dva klíče, pokud není list.**



GiST stromy (Generalized Search Tree)

(Joseph M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer - Generalized Search Tree Proceedings of the 21st VLDB Conference Zurich, Switzerland, 1995)

Jsou zobecněním předchozích metod, stromová struktura má stejnou organizaci jako R-tree. Implementaci nalezneme např. v RDBMS PosgreSQL. V „objektovém jazyku“ se jedná o jakoukoli třídu obsahující definici klíčů a operátorů, která implementuje nasledující metody (zjednodušeno, podrobněji na http://www.postgresql.org):

- Consistent, pro daný klíč a operátor (*query*) vrací *false*, je-li zaručeno, že dotaz *query* nemůže být pravdivý.

- Union, pro sadu klíčů vrací jejich sjednocení, přičemž $\text{Consistent}(key, query) \rightarrow \text{Consistent}(KEY, query)\ \forall\ KEY \supseteq key$

- Penalty, vrací míru rozšíření uzlu při vložení nového klíče.

- PickSplit, rozdělí sadu klíčů na dvě sady K_1 a K_2 tak, aby

  $\neg\text{Consistent}(\cup K_1, q) \rightarrow \text{Consistent}(K_2, q)$ a
  $\neg\text{Consistent}(\cup K_2, q) \rightarrow \text{Consistent}(K_1, q)$

  platilo v „co nevyšší“ míře (dotaz *q* může být pravdivý na jednom klíči vylučuje v co nevyšší míře pravdivost *q* na klíči druhém).

- Same, operátor rovnosti.

- Compres, Decompres, binární podoba uzlu ve file systému.

7 DML – DATA MANIPULATION LANGUAGE

Vkládání řádků do tabulek

Příkaz `INSERT`:

```
INSERT INTO
 tabulka
  (sloupec1,sloupec2,…,sloupecn)
VALUES
  (hodnota1,hodnota2,…,hodnotan)
```

Pořadí sloupců nemusí odpovídat pořadí v definici tabulky a nemusí být všechny.

```
INSERT INTO  tabulka
VALUES (hodnota1,hodnota2,…,hodnotan)
```

Pořadí sloupců musí odpovídat pořadí v definici tabulky, nedoporučuje se – změna struktury tabulky, přidání sloupců vynucuje změnu všech aplikací, které takový insert používají.

Při příkazu `INSERT` se kontrolují všechna integritní omezení na tabulce.

V případě, že není dodána hodnota a v definici tabulky je použita `DEFAULT` klausule, potom je dosazena příslušná hodnota z `DEFAULT` klausule.

Sloupce které jsou primárním nebo unikátním klíčem jsou vždy indexovány, kontrola je rychlá.

Kontrola referenční integrity - sloupce, na které odkazuje referenční integrita jsou buď primární, nebo unikátní klíče, proto je kontrola referenční integrity rychlá.

Změna hodnot v řádcích tabulky

Příkaz `UPDATE`:

```
UPDATE tabulka SET
 sloupec1=hodnota1,
  .
  .
 sloupecn= hodnotan
```

Změní hodnoty na všech řádcích

```
UPDATE tabulka SET
 sloupec1=hodnota1,
  .
  .
 sloupecn= hodnotan

WHERE
 logická_podmínka
```

např. `WHERE (VEK>40) and (VZDELANI='MUNI')`

Při příkazu `UPDATE` se kontrolují všechna dotčená integritní omezení na tabulce.

Při změně hodnoty sloupce, který je primárním nebo unikátním klíčem je kontrola rychlá, sloupce jsou indexovány.

Při změně hodnoty sloupce, na který odkazuje jiná tabulka cizím klíčem je kontrolována korektnost této operace, tedy prochází se "detailová" tabulka a kontroluje se výskyt staré hodnoty, v případě jeho nalezení operace končí chybou. Z toho plyne nutnost vytvořit indexy na každý cizí klíč!

Odstranění řádků z tabulky

Příkaz `DELETE`:

```
DELETE FROM tabulka
```

Odstraní vše!

```
DELETE FROM tabulka WHERE podminka
```

Při mazání řádku z tabulky, na kterou odkazuje jiná tabulka cizím klíčem je kontrolována korektnost této operace, tedy prochází se "detailová" tabulka a kontroluje se výskyt mazané hodnoty, v případě jeho nalezení operace končí chybou. Další důvod, proč vytvářet index na každý cizí klíč!

`ON DELETE` klausule

`CASCADE` – při odstranění řádků z nadřízené tabulky (`a1`) se odstraní i řádky z tabulky podřízené (`b1`).

```
create table a1
 (i int primary key);

create table b1
(i int references a1(i) on delete cascade);
```

`SET NULL` – při odstranění řádků z nadřízené tabulky (`a1`) se odstraní je nastavena hodnota cizích klíčů podřízené tabulky (`b1`) na hodnotu `NULL`.

```
create table a1
 (i int primary key);

create table b1
(i int references a1(i) on delete set null);
```

8 VÝBĚRY Z TABULEK, VYTVÁŘENÍ RELACÍ

Jednoduché příkazy `SELECT`:

```
select SL1, SL2  from TABULKA;
```

Sloupce lze v rámci příkazu SELECT přejmenovat:

```
select SL1 A,SL2 B from TABULKA;
```

Výstup lze uspořádat (při velkých tabulkách je vhodné na sloupce vytvořit index):

```
select SL1 A,SL2 B from TABULKA order by SL1;

select SL1 A,SL2 B from TABULKA order by SL1 DESC;
```

Fráze `distinct` neopakuje stejné řádky

```
select distinct SL1 A,SL2 B from TABULKA;
```

V příkazu `SELECT` lze použít funkce (pozor jejich repertoár a jména se mohou lišit v závislosti na implementaci RDBMS stroje).

```
select SL1,SL2 from TABULKA
 where SL1 ='BRNO' and SL2>0;

select SL1,SL2 from TABULKA
 where upper(SL1)='BRNO';
```

Výsledek každého `SELECT` příkazu je formálně tabulka lze s ním tedy v příkazech takto nakládat:

```
select * from
 (
  select JMENO, PRIJMENI
  FROM …
  ORDER BY PRIJMENI||' '||JMENO
 )
WHERE PRIJMENI||' '||JMENO BETWEEN 'xxxx' AND 'yyyy';
```

Přepínač - výraz `CASE`:

Přepínač typu `CASE` může být použit v části `SELECT` tam kde vybíráme sloupce (nebo i ve `WHERE` klausuli) a potřebujeme „rozskok“ podle konečného počtu podmínek.

Například:

```
SELECT
 case
  when poc_obyv>=500     and poc_obyv<1000   then  9
  when poc_obyv>=1000    and poc_obyv<5000   then 10
  when poc_obyv>=5000    and poc_obyv<10000  then 11
  when poc_obyv>=10000   and poc_obyv<50000  then 12
  when poc_obyv>=50000   and poc_obyv<100000 then 13
  when poc_obyv>=100000 then                      14
  else 0
 end SET_TEXT_HEIGHT
FROM
 NAZVY_OBCI
```

Množinové operace nad relacemi:

Sjednocení:

`select … union [all] select…`

Průnik:

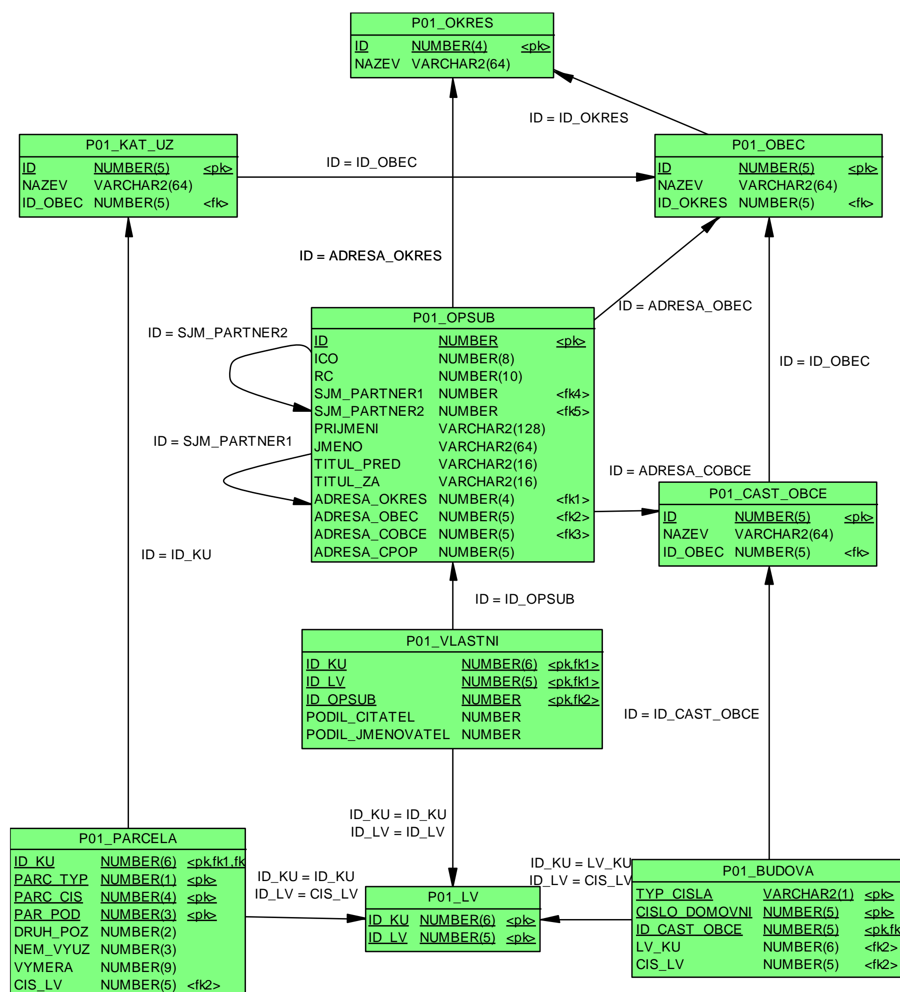
`select … intersect select…`

Diference:

`select … minus select…`

9 SPOJOVÁNÍ TABULEK (JOIN) – NÁSOBENÍ A SELEKCE

```
select
 OS.JMENO        "Jméno",
 OS.PRIJMENI     "Příjmení",
 OK.NAZEV        "Okres",
 OB.NAZEV        "Obec",
 CO.NAZEV        "Část obce",
 OS.ADRESA_CPOP "Číslo popisné"
from
 P01_OKRES OK,
 P01_OBEC OB,
 P01_CAST_OBCE CO,
 P01_OPSUB OS
where
 OS.ID =58342157 AND
 OS.ADRESA_OKRES=OK.ID AND
 OS.ADRESA_OBEC =OB.ID AND
 OS.ADRESA_COBCE=CO.ID
```



10 WHERE KLAUSULE

1) Porovnání výrazu s výrazem nebo poddotazem (subquery)

```
select * from
 P01_OPSUB
where ADRESA_OBEC=
 (select ID from P01_OBEC where nazev='Praha');
```

2) Porovnání výrazu se seznamem výrazů nebo poddotazem

```
select * from
 P01_OPSUB
where
 ADRESA_OBEC = SOME(3701,3801,3201);
```

```
select * from
 P01_OPSUB
where
 ADRESA_OBEC <> ALL(3701,3801,3201);
```

3) Příslušnost k množině

```
select * from
 P01_OPSUB
where ADRESA_OBEC IN
 (select ID from P01_OBEC where počet_obyv>2000);
```

4) Rozsahový dotaz

```
 select * from
  P01_OPSUB
 where
  RC BETWEEN 5800000000 AND 5899999999;
```

5) NULL test

```
select * from
 P01_OPSUB
where TITUL_PRED IS NOT NULL;
```

6) Existence v poddotazu

```
select * from
 P01_OPSUB A
where exists
 (
  select NULL from
   P01_OBEC  B
  where
   B.ID=A.ADRESA_OBEC AND
   B.ID_OKRES<>A.ADRESA_OBEC
 );
```

7) Srovnání řetězců

```
select * from
 P01_OPSUB
where
 PRIJMENI LIKE 'Nov%';
```

8) Logická kombinace 1) – 7) pomocí logických operátorů
   `AND OR NOT`

11 TECHNIKY SPOJOVÁNÍ (JOIN) TABULEK:

Nested loops (vnořené cykly):

Prochází se celá „vnější“ tabulka a hledá se odpovídající klíč v tabulce „vnitřní“. Podle existence indexu ve vnitřní tabulce se hledá odpovídající řádek podle indexu, nebo opět plným průchodem.

Sort merge join :

Uspořádá obě tabulky (vrácené řádky) podle klíče, kterým tabulky spojujeme, v případě existence indexu použije vhodný index. Poté prochází obě uspořádané tabulky a vrací kombinace řádků se stejnými klíči.

Hash join :

Pro menší tabulku se z klíčů vytvoří hash tabulka. Poté se prochází větší tabulka a hledá se odpovídající klíč v hash tabulce.

Databáze ORACLE se rozhoduje pro strategii na základě naplnění tabulek, frekvence výskytu klíčů.

12 OUTER JOIN – VNĚJŠÍ SPOJENÍ TABULEK

Outer join vrací všechny řádky, které vyhovují podmínce a takové řádky z jedné tabulky, které ji nesplňují. Sloupce, resp. výrazy v příkazu select z ostatních tabulek jsou v těchto případech vraceny jako `NULL`. Provádí se (+) operátorem ve `WHERE` klausuli.

```
create table t1 (i int);
create table t2 (i int);

insert into t1 values (1); ..(2); ..(3);
insert into t2 values (2); ..(3); ..(4);

SELECT t1.i i1,t2.i i2
FROM  t1,t2
WHERE t1.i=t2.i

SELECT t1.i i1,t2.i i2    I1  I2
FROM  t1,t2               ------
WHERE t1.i=t2.i (+)        1
                           2   2
                           3   3

SELECT t1.i i1,t2.i i2    I1  I2
FROM  t1,t2               ------
WHERE t1.i(+)=t2.i         2   2
                           3   3
                               4
```

13 ULOŽENÉ PŘÍKAZY `SELECT = VIEW`

`create view` *`jmeno`* `as select …`

S objekty typu `view` se v DML zachází:

`SELECT:` **stejně jako s tabulkami**

`UPDATE:` **všechny sloupce jsou jednoznačně přiřazeny key-preserved tabulkám – tj. takovým tabulkám jejichž každý klíč je zároveň klíčem view, příkaz mění řádky právě jedné tabulky**

`DELETE:` **řádky view odkazují na právě jednu key-preserved tabulku, z ní jsou řádky vymazány**

`INSERT:` **nesmí se explicitně nebo implicitně odvolávat na sloupce náležící non-key-preserved tabulce, všechny vkládané sloupce náleží právě jedné key-preserved tabulce**

Příklad VIEW:

```
CREATE TABLE ODDELENI
 (
  ID          INT PRIMARY KEY,
  NAZEV       VARCHAR2(256)
 );

CREATE TABLE PRACOVNIK
 (
   ID         INT              PRIMARY KEY,
   JMENO      VARCHAR2(32),
   PRIJMENI   VARCHAR2(32),
   ID_ODD     INT,
   CONSTRAINT PFK1 FOREIGN KEY (ID_ODD)
     REFERENCES ODDELENI(ID)
 );


CREATE VIEW PRAC_EXT AS
SELECT
 A.ID         ID_PRAC,
 A.PRIJMENI   PRIJMENI,
 A.JMENO      JMENO,
 B.ID         ID_ODD,
 B.NAZEV      NAZ_ODD
FROM
 PRACOVNIK A,
 ODDELENI B
WHERE
 A.ID_ODD=B.ID;
```

1) **Které sloupce z tohoto VIEW jdou vkládat?**

2) **Které sloupce z tohoto VIEW jdou měnit?**

3) **Lze z tohoto VIEW mazat (DELETE), co se stane při?**

Příklad: Jeden až několik pracovníků ze stejné oblasti má přidělen účet a může vidět jen svou oblast:

```
CREATE TABLE PVP_PRACOVNIK (
  ID_PRACOVNIK     NUMBER             NOT NULL,
  OBLAST           VARCHAR2 (8)   NOT NULL,
  ORG_JEDN_HR      VARCHAR2 (10)  NOT NULL,
  HARMONOGRAM      VARCHAR2 (9)   NOT NULL,
  USER_NAME        VARCHAR2 (16),
  PRIJMENI         VARCHAR2 (50)  NOT NULL,
  JMENO            VARCHAR2 (25)  NOT NULL,
  TARIFNI_TRIDA    VARCHAR2 (4),
  TARIFNI_STUPEN   VARCHAR2 (2),
  DATUM_NASTUPU    DATE               NOT NULL,
  DATUM_VYSTUPU    DATE,
  …
 ) ;


CREATE OR REPLACE VIEW U_PVP_PRACOVNIK  AS select *
from
 PVP_PRACOVNIK
 WHERE
  OBLAST IN
  (SELECT OBLAST
   FROM PVP_PRACOVNIK
   WHERE USER_NAME=USER
  )
```

Materializované pohledy

Jsou uložené výsledky dotazů (**`select`**), narozdíl od **`view`** výsledky jsou skutečně fyzicky uloženy. Je možnost výsledky dotazu obnovovat.

```
create materialized view v1
REFRESH FORCE
START WITH SYSDATE
      NEXT  SYSDATE + 1/1440
as select ...
```

`REFRESH` metoda obnovy

`FAST`	pohled musí mít primární klíč, musí exitsovat **`MATERIALIZED VIEW LOG`** na detailové tabulce
`COMPLETE`	provedeni celého dotazu znovu
`FORCE`	server vybere rychlejší metodu

`START WITH .. NEXT` interval obnovy

14 UŽIVATELSKY DEFINOVANÉ DATOVÉ TYPY (ADT), OBJEKTOVĚ RELAČNÍ DATABÁZE

Vytvoření typu:

```
create type Point as object
(
  x number,
  y number
)

create type Points as varray (10000) of Point;

create type LineString as object
(
  NumPoints int,
  Vertexes Points
)

create table Streets
(
 id    int,
 geom  LineString,
 constraint Streets_pk primary key (id)
)

insert into Streets (id,geom)
values
(1,
 Linestring(3,
            Points(
                     Point(0    ,    0),
                     Point(2000,  123),
                     Point(2020,13460)
                    )
             )
)
```

Vytvoření typu s metodami:

```
create or replace type AType as object (
  x number,
  y varchar2(10),
  member function ToString
          return varchar DETERMINISTIC
) NOT FINAL --[, NOT INSTANTIABLE]
/

create type body AType is
  member function ToString  return varchar
DETERMINISTIC is
  begin
   return y; -- PL/SQL tělo metody viz. funkce
  end;
end;
/
```

ADT nelze indexovat, lze však použít deterministickou funkci (viz. deterministické funkce PL/SQL), jejíž argument je ADT a která vrací „indexovatelný“ typ.

```
create table ATable(a Atype);

create index ATable_i1 on ATable(a.ToString());
```

Typ použitý jako typ sloupce by neměl být `NOT INSTANTIABLE`, jinak do něj nelze vkládat hodnoty.

Dědičnost typů:

```
create or replace type BType under AType
(
 z varchar2(200)
);
```

Typ, ze kterého dědíme musí být `NOT FINAL`.

ORACLE nepodporuje vícenásobnou dědičnost.

Obecně není možné `SELECT` na celý ADT, neboť ne všechny typy klientských rozhraní podporují ADT. Musíme vybírat jeho jednotlivé složky

```
select AT.A.x from ATable AT;
```

V případě neznáme položky ADT, možností je využití podpory pomocí XML:

```
select
 xmlelement("ROW",geom).getStringVal()from Streets
```

```
resp:
```

```
select xmlelement("ROW",geom).getClobVal() from
Streets
```

vrátí:

```
<ROW>
<LINESTRING>
 <NUMPOINTS>2</NUMPOINTS>
  <VERTEXES>
   <POINT>
     <X>0</X>
     <Y>0</Y>
   </POINT>
   <POINT>
     <X>2000</X>
     <Y>123</Y>
   </POINT>
  </VERTEXES>
 </LINESTRING>
</ROW>
```

Vzhledem, tomu, že v moderních vývojových prostředí klientský aplikací (C++, C# .NET) je implementována masivní podpora parsingu XML, jedná se o poměrně silný a univerzální prostředek. Pozor, cenou je zvýšená zátěž databázového serveru a řádově větší přenosu dat v XML formátu.

15 HIERARCHICKÉ DOTAZY

Vybere „podstrom“ ze stromové struktury v tabulce.

```
create table HI
(
  ID     INT,
  PARENT INT,
  .
  .
  .
  CONSTRAINT HI_FK01 FOREIGN KEY
   (PARENT) REFERENCES HI(ID);
 );

INSERT INTO HI VALUES (1,null);
INSERT INTO HI VALUES (2,1);
INSERT INTO HI VALUES (3,1);
INSERT INTO HI VALUES (4,3);
INSERT INTO HI VALUES (5,3);
```

<u>CONNECT BY klausule:</u>

definuje relaci „rodič“ ve stromu

<u>START WITH klausule:</u>

Příklad:

```
select level,id,parent from hi
 connect by prior ID=PARENT
start with id=3;

    LEVEL         ID    PARENT
--------- --------- ---------
        1          3         1
        2          4         3
        2          5         3
```

16 SKUPINOVÉ (AGREGAČNÍ) FUNKCE

Jsou funkce které vrací jeden výsledek na základě vstupu z více řádků.

Pokud není uvedena `group by` klausule potom je výsledek funkce aplikován na celý výsledek `SELECT` dotazu.

`AVG(expr)` – průměr z `expr`

```
select
 AVG(PLAT)
from
 ZAMESTNANCI
where
 VEK between 25 and 30;
```

`COUNT({* | [DISTINCT|ALL] expr})` - počet řádků, ve kterých je `expr NOT NULL`

```
select count(*) from P01_VL
```

vrátí počet řádků z tabulky `P01_VL`

<u>Další skupinové funkce:</u>

```
MAX(expr)
MIN(expr)
STDDEV(expr)
SUM(n)
```

`group by` klausule

Použitím `group by` klausule jsou podle výrazu v této klausuli agregovány řádky výsledku

```
select
 PRIJMENI
from
 P01_OPSUB
group by PRIJMENI;
```

Seznam sloupců v select příkazu, který obsahuje `group by` může obsahovat pouze:

- **Konstanty**
- **skupinové funkce**
- **výrazy, které jsou identické s výrazy v `group by`**
- **výrazy, které jsou založeny na předešlých výrazech**

```
select
 rtrim(upper(PRIJMENI),40)||'-'||count(*)
from
 P01_OPSUB
group by PRIJMENI;
```

`having` klausule

Používá se k omezení výstupu na základě skupinových funkcí

```
select
 rtrim(upper(PRIJMENI),40)||'-', count(*)
from
 P01_OPSUB
group by PRIJMENI
having count(*)>2;
```

Strategie vyhodnocení SQL dotazu

1. Obsahuje-li `SELECT WHERE` klausuli zpracují se pouze řádky které vyhovují `WHERE`

2. Obsahuje-li `SELECT GROUP BY` klausuli, vytvářejí se skupiny podle výrazů `group by`

3. Obsahuje-li `SELECT HAVING` klausuli, potom jsou vyřazeny ty skupiny, které podmínku `having` nesplňují

17 OPTIMALIZACE PŘÍKAZŮ

Exekuční plán:

```
delete from plan_table;

commit;

explain plan
 SET STATEMENT_ID = 'Adresa 01'
 INTO plan_table
for
select
 a.NAZEV  Okres,
 b.NAZEV  Obec,
 c.NAZEV  Ulice,
 d.CIS_OR COR
from
 u_okresy A,
 u_obce   B,
 u_ulice  C,
 u_adresy D
where
 b.kodok=a.kodok and
 c.iczuj=b.iczuj and
 d.kodul=c.kodul and
 d.PRIZNAK_ulice=c.priznak and
 b.nazev like 'Brno%' and
 c.NAZEV like 'Podlesná%'
```

```
SELECT LPAD(' ',2*(LEVEL-1))||operation operation,
options,
object_name,object_type
    FROM plan_table
    START WITH id = 0 AND statement_id = 'Adresa 01'
    CONNECT BY PRIOR id = parent_id AND statement_id
= 'Adresa 01';
```

SELECT STATEMENT			
NESTED LOOPS			
NESTED LOOPS			
HASH JOIN			
TABLE ACCESS	BY INDEX ROWID	U_OBCE	TABLE
INDEX	RANGE SCAN	IX_U_OBCE	INDEX
TABLE ACCESS	FULL	U_ULICE	TABLE
TABLE ACCESS	BY INDEX ROWID	U_OKRESY	TABLE
INDEX	UNIQUE SCAN	PK_U_OKRESY	INDEX (UNIQUE)
INDEX	RANGE SCAN	IX_U_ADRESY_2	INDEX

Přístup k tabulce:

FULL
INDEX RANGE SCAN
UNIQUE SCAN

Řízení přístupu - HINT

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

/*+ INDEX(*jméno_indexu*) */ - vynutí použití indexu

```
CREATE INDEX P01_OPSUB_I2 ON P01_OPSUB(ADRESA_OKRES);
CREATE INDEX P01_OPSUB_I1 ON P01_OPSUB(PRIJMENI);
CREATE INDEX P01_OKRES_I1 ON P01_OKRES(NAZEV);

select /*+ INDEX(P01_OKRES_I1) */
 a.prijmeni,
 b.nazev
from
 p01_okres b,
 p01_opsub a
where
 a.ADRESA_OKRES=b.ID AND
 B.NAZEV='KROMĚŘÍŽ'  AND
 A.PRIJMENI = 'NOVÁK'


SELECT STATEMENT Optimizer=CHOOSE
  NESTED LOOPS
    TABLE ACCESS (BY INDEX ROWID) OF P01_OKRES
      INDEX (RANGE SCAN) OF P01_OKRES_I1 (NON-UNIQUE)
    TABLE ACCESS (BY INDEX ROWID) OF P01_OPSUB
      INDEX (RANGE SCAN) OF P01_OPSUB_I1 (NON-UNIQUE)
```

/*+ ORDERED */

Spojení (JOIN) tabulek probíhá v pořadí podle FROM klausule.

```
select /*+ ORDERED */
 a.prijmeni,
 b.nazev
from
 p01_opsub a,
 p01_okres b
where
 b.ID=a.ADRESA_OKRES AND
 B.NAZEV='KROMĚŘÍŽ'  AND
 A.PRIJMENI = 'NOVÁK'

SELECT STATEMENT Optimizer=CHOOSE
  NESTED LOOPS
    TABLE ACCESS (BY INDEX ROWID) OF P01_OPSUB
      INDEX (RANGE SCAN) OF P01_OPSUB_I1 (NON-UNIQUE)
    TABLE ACCESS (BY INDEX ROWID) OF P01_OKRES
      INDEX (UNIQUE SCAN) OF P01_OKRES_PK (UNIQUE)

select /*+ ORDERED */
 a.prijmeni,
 b.nazev
from
 p01_okres b,
 p01_opsub a
where
 b.ID=a.ADRESA_OKRES AND
 B.NAZEV='KROMÌŘÍŽ'  AND
 A.PRIJMENI = 'NOVÁK'

SELECT STATEMENT Optimizer=CHOOSE
  NESTED LOOPS
    TABLE ACCESS (BY INDEX ROWID) OF P01_OKRES
      INDEX (RANGE SCAN) OF P01_OKRES_I1 (NON-UNIQUE)
    TABLE ACCESS (BY INDEX ROWID) OF P01_OPSUB
      INDEX (RANGE SCAN) OF P01_OPSUB_I1 (NON-UNIQUE)
```

```
EXPLAIN PLAN FOR
select
 KU.NAZEV,
 PA.PARC_TYP,
 PA.PARC_CIS,
 PA.PAR_POD,
 VL.PODIL_CITATEL||'/'||VL.PODIL_JMENOVATEL
from
 P01_VLASTNI VL,
 P01_PARCELA PA,
 P01_KAT_UZ  KU
where
 VL.ID_OPSUB=1        AND
 VL.ID_KU   =PA.ID_KU  AND
 VL.ID_LV   =PA.CIS_LV AND
 PA.ID_KU   =KU.ID;

SELECT STATEMENT Optimizer=CHOOSE
  NESTED LOOPS
    NESTED LOOPS
      TABLE ACCESS (FULL) OF P01_KAT_UZ
      TABLE ACCESS (BY INDEX ROWID) OF P01_PARCELA
        INDEX (RANGE SCAN) OF P01_PARCELA_PK (UNIQUE)
    TABLE ACCESS (BY INDEX ROWID) OF P01_VLASTNI
      INDEX (UNIQUE SCAN) OF P01_VLASTNI_PK (UNIQUE)
```

```
create index P01_VLASTNI_i1 on P01_VLASTNI(ID_OPSUB);

EXPLAIN PLAN FOR
select
 KU.NAZEV,
 PA.PARC_TYP,
 PA.PARC_CIS,
 PA.PAR_POD,
 VL.PODIL_CITATEL||'/'||VL.PODIL_JMENOVATEL
from
 P01_VLASTNI VL,
 P01_PARCELA PA,
 P01_KAT_UZ  KU
where
 VL.ID_OPSUB=1        AND
 VL.ID_KU   =PA.ID_KU  AND
 VL.ID_LV   =PA.CIS_LV AND
 PA.ID_KU   =KU.ID;


SELECT STATEMENT Optimizer=CHOOSE
 NESTED LOOPS
  NESTED LOOPS
   TABLE ACCESS (BY INDEX ROWID) OF P01_VLASTNI
    INDEX (RANGE SCAN) OF P01_VLASTNI_I1 (NON-UNIQUE)
      TABLE ACCESS (BY INDEX ROWID) OF P01_PARCELA
       INDEX (RANGE SCAN) OF P01_PARCELA_PK (UNIQUE)
   TABLE ACCESS (BY INDEX ROWID) OF P01_KAT_UZ
    INDEX (UNIQUE SCAN) OF P01_KAT_UZ_PK (UNIQUE)
```

18 TCC – TRANSACTION CONTROL COMMANDS

Transakce – je posloupnost DML příkazů, které převedou datové schéma z jednoho konzistentního stavu do druhého.

ACID

A	**– Atomic**	**celá se provede, nebo odvolá**
C	**– Consistent**	**na konci není porušeno žádné omezení**
I	**– Isolated**	**operace jsou izolovány od ostatních transakcí**
D	**– Durable**	**po ukončení transakce jsou data trvale uložena**

`COMMIT` – Potvrzení změn DML od počátku transakce.

`ROLLBACK [TO` *`savepoint`*`]` –Odvolá změny od počátku transakce/ `savepoitnt`

`SAVEPOINT` – Stanoví místo po které lze provést rollback

`SET TRANSACTION READ WRITE` – default nastavení transakcí

`SET TRANSACTION READ ONLY` – nastaví transakci tak, že nejsou povoleny příkazy `INSERT`, `UPDATE`, `DELETE` a `SELECT` s klausulí `FOR UPDATE`. Musí být prvním příkazem transakce.

Úrovně izolace:

`SET TRANSACTION ISOLATION LEVEL SERIALIZABLE` – Úroveň izolace podle normy SQL92. V případě že se transakce mění objekt, který je měněn jinou konkurentní transakcí, potom transakce končí chybou v případě, že konkurentní transakce je potvrzena.

`SET TRANSACTION ISOLATION LEVEL READ COMMITTED` – (default chování ORACLE). V případě, že transakce požaduje zámek na řádky, které jsou drženy jinou transakcí, potom transakce čeká na uvolnění, potom DML příkaz provede.

`SELECT` ...... `FOR UPDATE [NOWAIT];`
Uzamkne vybrané řádky/sloupce pro aktuální transakci až do `COMMIT` nebo `ROLLBACK`.

`LOCK TABLE` *`lock mode`* `MODE [NOWAIT];`

`ROW SHARE` – Zakazuje **`EXCLUSIVE LOCK`**, jinak nechává povolené konkurentní aktivity na tabulce

`EXCLUSIVE` – Výhradní právo na tabulku pro transakci, mimo **`SELECT`** zakazuje cokoli.

`SHARE` – Zakazuje **`UPDATE`** tabulky

```
CREATE TABLE I1
(
 I INT,
 C VARCHAR2(64),
 CONSTRAINT I1_PK PRIMARY KEY (I)
);

INSERT INTO I1 VALUES (1,‘A‘);
COMMIT;
```

READ COMMITED

```
SE #1 - UPDATE I1 SET C='B' WHERE I=1;[OK]
SE #2 - UPDATE I1 SET C='C' WHERE I=1;[OK – čeká]
SE #1 – COMMIT;                       [OK]
SE #2 – COMMIT;                       [OK]

SELECT * FROM I1
```

I	C
1	C

SERIALIZABLE

```
SE #1 - UPDATE I1 SET C='B' WHERE I=1;[OK]
SE #2 - UPDATE I1 SET C='C' WHERE I=1;[OK – čeká]
SE #1 – COMMIT;                       [OK]
SE #2 – havaruje

SELECT * FROM I1
```

I	C
1	B

Integritní omezení **INITIALLY DEFERRED** kontrolují se až v okamžiku **COMMIT** transakce.

Příklad: povinná vazba 1:1

```
CREATE TABLE T1
(
 I INT PRIMARY KEY
);

CREATE TABLE T2
(
 I INT PRIMARY KEY
);

ALTER TABLE T1 ADD CONSTRAINT T1_FK1
 FOREIGN KEY (I) REFERENCES T2(I);

ALTER TABLE T2 ADD CONSTRAINT T2_FK1
 FOREIGN KEY (I) REFERENCES T1(I) INITIALLY DEFERRED;
```

Proběhne:

```
INSERT INTO T2 VALUES (2);
INSERT INTO T1 VALUES (2);
COMMIT;
```

Havaruje:

```
INSERT INTO T1 VALUES (3);
INSERT INTO T2 VALUES (3);
COMMIT;
```

I SELECT příkaz je v jistém smyslu transakční, databáze musí vrátit stav dat, který byl platný v okamžiku jeho zadání!!

Deadlock:

```
CREATE TABLE I1
(
 I INT,
 C VARCHAR2(64),
 CONSTRAINT I1_PK PRIMARY KEY (I)
);

INSERT INTO I1 VALUES (1,'A');
INSERT INTO I1 VALUES (2,'B');
COMMIT;
```

```
SE #1 - UPDATE I1 SET C='C' WHERE I=1;[OK]
SE #2 - UPDATE I1 SET C='D' WHERE I=2;[OK]
SE #1 - UPDATE I1 SET C='E' WHERE I=2;[OK - čeká]
SE #2 - UPDATE I1 SET C='F' WHERE I=1;[OK - čeká]
```

SE #1 – během čekání na prostředek došlo k deadlocku

```
SE #1 – COMMIT;
SE #2 – COMMIT;

SELECT * FROM I1
```

I	C
1	F
2	D

Strategie transakcí z klientských aplikací.

Optimistický a pesimistický přístup transakcí.

Organizace rollback segmentů (snímek je příliš starý)

19 JAZYK - PL/SQL

19.1 STRUKTURA BLOKU

PL/SQL je součástí databázového stroje.

Je procedurální jazyk, tak jak je pojem procedurálního jazyka běžně chápán.

Je strukturován do bloků, tj. funkce a procedury jsou logické bloky, které mohou obsahovat bloky atd.

Příkazy: řídící příkazy jazyka PL/SQL, přiřazení-výrazy, SQL příkazy DML.

```
[DECLARE
-- declarations]
BEGIN
-- statements
[EXCEPTION
 -- handlers]
END;
```

19.2 DEKLARACE

```
Kolik_mi_zbyva_penez NUMBER(6);
skutecne             BOOLEAN;
```

Datový typ `tabulka%ROWTYPE` **odpovídá struktuře tabulky.**

Datový typ `tabulka.sloupec%ROWTYPE` **odpovídá typu sloupce v tabulce**

```
JM P01_OPSPUB.JMENO%TYPE;
OBSUB%ROWTYPE;
```

Přiřazení, výrazy:

```
tax := price * tax_rate;
bonus := current_salary * 0.10;
amount := TO_NUMBER(SUBSTR('750 dollars', 1, 3));
valid := FALSE;
```

19.3 DML A KURSORY

INTO fráze:

```
SELECT sal*0.10 INTO bonus
FROM
 emp
WHERE
 empno = emp_id;
```

Kursor:

```
DECLARE CURSOR c1 IS
SELECT empno, ename, job FROM emp WHERE deptno = 20;
```

Ovládání kursorů:

1) Analogie k souborovému přístupu:

```
  OPEN, FETCH, CLOSE

  OPEN C1;
  ..
  FETCH C1 into a,b,c;
  ..
  CLOSE C1;
```

2) For cykly pro kursory:

```
DECLARE CURSOR c1 IS
SELECT ename, sal, hiredate, deptno FROM emp;
...
BEGIN
FOR emp_rec IN c1 LOOP
 salary_total := salary_total + emp_rec.sal; …
END LOOP;
```

Použití ROWTYPE pro kursory:

```
DECLARE CURSOR c1 IS
SELECT ename, sal, hiredate, job FROM emp;

emp_rec c1%ROWTYPE;
```

Dynamické SQL příkazy:

Jsou dotazy jejichž konečný tvar vzniká až při běhu programu.

```
EXECUTE IMMEDIATE

sql_stmt := 'INSERT INTO dept VALUES (:1, :2, :3)';

EXECUTE IMMEDIATE sql_stmt
 USING dept_id, dept_name, location;
```

19.4 ŘÍDÍCÍ PŘÍKAZY

IF-THEN-ELSE:

```
IF acct_balance >= debit_amt THEN
  UPDATE accounts SET bal = bal - debit_amt
  WHERE account_id = acct;
  .
  .
 ELSE
  INSERT INTO temp VALUES
   (acct, acct_balance, 'Insufficient funds');
  .
  .
END IF;
```

FOR-LOOP:

```
FOR i IN 1..order_qty LOOP
 UPDATE sales SET custno = customer_id
 WHERE seriál_num = seriál_num_seq.NEXTVAL;
END LOOP;
```

WHILE-LOOP:

```
WHILE salary < 4000 LOOP
 SELECT sal, mgr, ename INTO salary, mgr_num,
 last_name FROM emp WHERE empno = mgr_num;

END LOOP;
```

19.5 ASYNCHRONNÍ OŠETŘENÍ CHYB

```
begin
  select   …… into a,b,c;

EXCEPTION
WHEN NO_DATA_FOUND THEN
-- process error
end;
```

19.6 FUNKCE A PROCEDURY

```
CREATE OR REPLACE PROCEDURE [FUNCTION] jmeno
(
 par1 IN  VARCHAR2,
 par2 OUT INT
)
[RETURN VARCHAR2]
IS
var1 VARCHAR2(1);
BEGIN
 .
 .
 RETURN [var1];
END jmeno;
/
```

Deterministické funkce:

Jsou funkce, které vrací pro stejné argumenty vždy stejný výsledek. Výsledek tedy není ovlivněn momentálním stavem databáze (schéma, data, čas..). Jen tyto funkce lze použít v indexech založených na funkcích.

```
create or replace function fun1(…) return varchar2
deterministic
```

Funkce lze použít v DML příkazech například:

```
SELECT moje_funkce(43) FROM DUAL;

SELECT moje_funkce(SL3+SL2);

DELETE FROM TAB1 WHERE SL1=moje_funkce(SL3+SL2);
```

Procedury spouštíme v rámci PL/SQL bloku:

```
Begin
  moje_procedura(argument,… …);
end;
```

19.7 BALÍKY – PACKAGE

Jsou pojmenované programové jednotky, které mohou obsahovat typy, proměnné, funkce a procedury.

```
CREATE PACKAGE name AS
-- public type and item declarations
-- subprogram specifications
END [name];

CREATE PACKAGE BODY name IS
-- private type and item declarations
-- subprogram bodies
END [name];
```

Instance objektů vznikají v rámci sezení, tj. nemohou vzniknout kolize zapříčiněné konkurentním používáním objektů balíku.

```
CREATE PACKAGE STEMIG AS
 C_MASTER_NAME    VARCHAR2(16):='S3';
 FUNCTION TO_NUMEXT (x in char) RETURN number;
 FUNCTION  ANG (X1 IN NUMBER,Y1 IN NUMBER,
               X2 IN NUMBER,Y2 NUMBER)
  RETURN NUMBER;
END STEMIG;

CREATE PACKAGE BODY STEMIG IS
FUNCTION TO_NUMEXT (x in char)
RETURN number
IS
R number;
BEGIN
 R:=TO_NUMBER(x);
 return(R);
  exception when VALUE_ERROR THEN
 return(NULL);
END;
END STEMIG;
```

20 TRIGGERY

PL/SQL bloky, které jsou přidruženy k tabulkám.

Události které spouští triggery:

`INSERT,UPDATE,DELETE`
Typy triggerů:

	STATEMENT	ROW
BEFORE	Trigger je spuštěn jednou před provedením příkazu	Trigger je spuštěn jednou před modifikací každého řádku
AFTER	Trigger je spuštěn jednou po provedení příkazu	Trigger je spuštěn jednou po modifikaci každého řádku

`:NEW` a `:OLD` proměnné v řádkovém triggeru odkazují na nové resp. staré hodnoty modifikovaného řádku.

Logické proměnné v každém řádkovém triggeru:

`INSERTING` - `true` jestliže trigger je spuštěn INSERT

`DELETING` - `true` jestliže trigger je spuštěn DELETE

`UPDATING` - `true` jestliže trigger je spuštěn UPDATE

`UPDATING(`*`column_name`*`)` modifikuje sloupec

PL/SQL bloky nesmí obsahovat příkazy řízení transakcí (`commit, rollback,` …)

Triggery by neměly "šifrovat" data tedy by neměly obsahovat bloky typu:

```
if UPDATING('STAV_KONTA')
    and
   JMENO_MAJITELE_UCTU='Drášil'
    and
   :NEW.STAV_KONTA < :OLD.STAV_KONTA
 THEN
  :NEW.STAV_KONTA := :OLD.STAV_KONTA;
end if;
```

Kódování zdrojových kódů balíků, těl balíků, procedur, funkcí – vznikne zašifrovaný zdrojový text (doporučuji – nikdy nepoužívat, programátoři svoje zdroje většinou šifrují dostatečně):

```
WRAP INAME=input_file [ONAME=output_file]
```

<u>Příklad: trigger hlídající akce nad tabulkou:</u>

```
CREATE TRIGGER audit_trigger
 BEFORE
  INSERT OR
  DELETE OR
  UPDATE
 ON nejaka_tabulka
FOR EACH ROW
BEGIN
 IF INSERTING THEN
  INSERT INTO audit_table
  VALUES (USER||' is inserting'||' new key: '||
          :new.key);
  :NEW.USER_NAME=USER;

 ELSIF DELETING THEN
  INSERT INTO audit_table
  VALUES (USER||' is deleting'||' old key: '||
          :old.key);

 ELSIF UPDATING('FORMULA') THEN
  INSERT INTO audit_table
  VALUES (USER||' is updating'||' old formula: '||
```

```
          :old.formula||' new formula: ' ||
          :new.formula);

 ELSIF UPDATING THEN
   IF :OLD.USER_NAME<>USER THEN
    RAISE_APPLICATION_ERROR('-20000',
                        'Přístup k řádku odmítnut')
   END_IF;
   INSERT INTO audit_table
   VALUES (USER||' is updating'||' old key: ' ||
           :old.key||' new key: ' || :new.key);
END IF;
END;
```

21 ADMINISTRACE PŘÍSTUPOVÝCH PRÁV

Role jsou seznamy práv:

```
CREATE ROLE jméno;

GRANT [system_priv|role,…] TO [user|role|PUBLIC];
```

Příklady systémových práv:

```
ALTER ANY TABLE, CREATE ANY SEQUENCE, CREATE
PROCEDURE, SELECT ANY TABLE …

GRANT [object_priv|ALL (column,…),… ] ON
schema.object TO [user|role|PUBLIC]
```

Příklady práv k objektům:

```
ALTER,EXECUTE,INSERT,READ,SELECT,UPDATE
```

Práva na tabulky končí na úrovni sloupců, pro práva na řádky tabulek musíme použít techniku triggerů nebo „updatable view".

Zrušení práv

```
REVOKE [priv] from [user|role|PUBLIC]
```

Příklad postupu administrace:

1) **Vytvoříme DB schéma master uživatele.**
2) **Vytvoříme PUBLIC synonyma pro každý objekt.**
3) **Stanovíme role pro přístup k objektům, podle typů uživatelů.**
4) **Rolím přidělíme práva pro jednotlivé objekty.**
5) **Každý nový uživatel systému nevlastní žádné objekty, “vidí” je prostřednictvím veřejných synonym.**
6) **Správce systému přidělí potřebné role každému uživateli.**

Nestandardní indexy

FullText:

Umožňuje efektivní vyhledání řádků podle „volného“ textu, který je obsažen ve sloupci typu `CLOB`.

Tabulka s dokumenty (text, *.doc, *.pdf ..)

```
create table FULLWORD
(
 ID INT primary key,
 TEXT CLOB
);
```

Full text index:

```
CREATE INDEX FULLWORD_FTI
 ON FULLWORD(TEXT)
 INDEXTYPE IS CTXSYS.CONTEXT;
```

Naplnění tabulky dokumenty:

```
insert into fullword (ID,TEXT) values (1,
'Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Vivamus ornare congue turpis.');

insert into fullword (ID,TEXT) values (2,
'Cras volutpat, neque et sodales congue, urna mauris
dignissim sem, blandit laoreet magna mauris in
nibh.');

insert into fullword (ID,TEXT) values (3,
'In ac felis eget velit elementum consectetur. Morbi
vitae sem. Proin varius luctus risus.');

insert into fullword (ID,TEXT) values (4,
'Fusce ut dui a massa congue vehicula.');

insert into fullword (ID,TEXT) values (5,
'Class aptent taciti sociosqu ad litora torquent per
conubia nostra, per inceptos himenaeos.');
```

```
insert into fullword (ID,TEXT) values (6,
'Phasellus ut lacus. Praesent nec nisl.');

insert into fullword (ID,TEXT) values (7,
'Pellentesque sapien lectus, ultrices rhoncus,
consequat at, elementum non, orci.');

insert into fullword (ID,TEXT) values (8,
'Ut consequat ipsum ut ante. Quisque diam sem,
posuere id, vehicula eu, tincidunt ac, neque.');

insert into fullword (ID,TEXT) values (9,
'Vestibulum leo. Fusce vel ante ac dui tincidunt
ornare.');

insert into fullword (ID,TEXT) values (10,
'Nam est augue, vehicula ut, molestie non, iaculis
quis, arcu.');

commit;
```

Obnovení indexu (není automatické):

```
EXEC CTX_DDL.SYNC_INDEX('FULLWORD_FTI');
```

Select příkaz z frází `CONTAINS`**:**

```
SELECT ID,text FROM FULLWORD
WHERE
 CONTAINS(TEXT, 'vehicula') > 0
```

Vrátí:

4	Fusce ut dui a massa congue vehicula.
8	Ut consequat ipsum ut ante. Quisque diam sem, posuere id, vehicula eu, tincidunt ac, neque.
10	Nam est augue, vehicula ut, molestie non, iaculis quis, arcu.

Prostorové indexy (Oracle Spatial Data Option)

Slouží k efektivnímu přístupu ke geometrickým datům (čáry, polygony..).

Zdrojem pro index je tzv. minimální omezující obdélník (maxima a minima souřadnic).

Použitá metoda pro tento typ indexu je R-Tree (modifikace B+ stromu).

Indexovatelné sloupce jsou typu `MDSYS.SDO_GEOMETRY`

Tabulka s geometrickým sloupcem:

```
ALTER TABLE AK_HRANICE_PARCEL
ADD GEOMETRY MDSYS.SDO_GEOMETRY;
```

Metadata pro geometrický sloupec:

```
insert into mdsys.user_sdo_geom_metadata
(table_name,column_name,diminfo)
values
('AK_HRANICE_PARCEL','GEOMETRY',
MDSYS.SDO_DIM_ARRAY
(MDSYS.SDO_DIM_ELEMENT('X',1230000000, 933000000,5),
 MDSYS.SDO_DIM_ELEMENT('Y', 906000000, 410000000,5)
));
```

Prostorový index:

```
create index AK_HRANICE_PARCEL_SPAT ON
 AK_HRANICE_PARCEL(GEOMETRY)
 INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

Prostorový dotaz:

```
select
 ID,GEOMETRIE
FROM ISKN.AK_HRANICE_PARCEL A
WHERE
 MDSYS.SDO_FILTER
 (GEOMETRIE,
  MDSYS.SDO_GEOMETRY
   (2003,
    (SELECT SRID FROM user_sdo_geom_metadata
      WHERE
       TABLE_NAME='AK_HRANICE_PARCEL' AND
       COLUMN_NAME='GEOMETRY'),
    NULL,
    MDSYS.SDO_ELEM_INFO_ARRAY(1,1003,3),
    MDSYS.SDO_ORDINATE_ARRAY
     (
      1144945653, -- dotazovací obdélník
       520792013,
      1145270564,
       521132433)
     ),
  'queryType=window')='TRUE';
```

22 HOSTITELSKÉ NADSTAVBY SQL

22.1 PRO*C

Výhoda: relativní platformová nezávislost, vznikne „čistý“ C kód.

Deklarační část:

```
EXEC SQL BEGIN DECLARE SECTION;
 VARCHAR DBuser[80];
 VARCHAR DBpswd[20];
 VARCHAR sql_stmt[8192];
EXEC SQL END DECLARE SECTION;

SQLDA *selda;
int i;
```

Výkonná část:

```
EXEC SQL WHENEVER SQLERROR do gsSqlError();

strcpy(DBuser.arr,"TEST@GB001");
strcpy(DBpswd.arr,"TEST");

DBuser.len=strlen(DBuser.arr);
DBpswd.len=strlen(DBpswd.arr);

printf("connect\n");

EXEC SQL CONNECT :DBuser IDENTIFIED BY :DBpswd;

srintf(stmtP,"DROP TABLE AUDIT");

strcpy(sql_stmt.arr,stmtP);
sql_stmt.len=strlen(sql_stmt.arr);

EXEC SQL PREPARE STMT FROM :sql_stmt;
EXEC SQL EXECUTE STMT;
```

PRO*C prekompilátor:

```
strcpy(sql_stmt.arr,stmtP);
sql_stmt.len=strlen(sql_stmt.arr);
/* EXEC SQL PREPARE STMT FROM :sql_stmt; */
{
 struct sqlexd sqlstm;
 sqlstm.sqlvsn = 10;
 sqlstm.sqhstv[0] = (void  *)&sql_stmt;
 sqlstm.sqlest = (unsigned char  *)&sqlca;
 sqlstm.sqlety = (unsigned short)256;
 sqlstm.occurs = (unsigned int  )0;
 sqlstm.sqhstl[0] = (unsigned int  )8194;
 sqlstm.sqhsts[0] = (         int  )0;
 sqlstm.sqindv[0] = (         void  *)0;
 sqlstm.sqinds[0] = (         int  )0;
 sqlstm.sqharm[0] = (unsigned int  )0;
 sqlstm.sqadto[0] = (unsigned short )0;
 sqlstm.sqtdso[0] = (unsigned short )0;
 sqlstm.sqphsv = sqlstm.sqhstv;
 sqlstm.sqphsl = sqlstm.sqhstl;
 sqlstm.sqphss = sqlstm.sqhsts;
 sqlstm.sqpind = sqlstm.sqindv;
 sqlstm.sqpins = sqlstm.sqinds;
 .
 sqlcxt((void **)0, &sqlctx, &sqlstm, &sqlfpn);
 if (sqlca.sqlcode < 0) gsSqlError();
}
 /* EXEC SQL EXECUTE STMT; */
{struct sqlexd sqlstm;
 sqlstm.sqlvsn = 10;
 sqlstm.arrsiz = 4;
 sqlstm.sqladtp = &sqladt;
  .
 sqlcxt((void **)0, &sqlctx, &sqlstm, &sqlfpn);
 if (sqlca.sqlcode < 0) gsSqlError();
}
```

22.2 OBJEKTOVÁ ROZHRANÍ SQL, ADO.NET

Výhoda: Moderní prostředí, relativní nezávislost na typu databáze (pokud používáme základ SQL).

```
this.oracleConnection =
 new Oracle.DataAccess.Client.OracleConnection();

this.oracleConnection.ConnectionString =
 MyConnectString;
this.oracleConnection.Open();

this.oracleConnection.SetCommand(sqlStmt);
```

...

22.3 VÁZANÉ PROMĚNNÉ (BIND), OBRANA PROTI SQL INJEKCÍM

Při vývoji aplikace je nutné dát si pozor na možné útoky, tzv. SQL injekce. V databázové schématu máme například tabulku, která obsluhuje přístupová práva:

```
CREATE TABLE REMOTE_USERS
(
 USER_NAME VARCHAR2(64),
 USER_PSWD VARCHAR2(64)
);
```

Aplikace potom “ověřuje“ uživatele například takto:

```
public bool Authenticate
(string userName, string userPassword)
{
 string sqlStmt =
 "SELECT COUNT(*) FROM REMOTE_USERS WHERE USER_NAME='" +
 userName +
 "' AND USER_PSWD='" +
 userPassword+ "'";

 System.Data.SqlClient.SqlCommand command =
  new SqlCommand(sqlStmt);

 object o = command.ExecuteScalar(sqlStmt);
 return (Convert.ToInt32(o) == 1);
}
```

Útok:

potom probíhá například takto:

```
userName="SQL injekce"

userPassword="je snadná' OR ROWNUM<'2"
```

Výsledný dotaz do tabulky uživatelů vždy vrací hodnotu 1:

```
SELECT COUNT(*)
FROM REMOTE_USERS
WHERE
 USER_NAME='SQL injekce' AND
 USER_PSWD='je snadná' OR ROWNUM<'2'
```

Obrana:

Použijeme tzv. vázané proměnné (bind), tj. využijeme možnosti nezávislého odeslání příkazu a jeho parametrů.

```
public bool Authenticate
(string userName, string userPassword)
{
 string sqlStmt =
 "SELECT COUNT(*) FROM REMOTE_USERS "+
 " WHERE USER_NAME=@a1 AND USER_PSWD=@a2";

 System.Data.SqlClient.SqlCommand command =
  new SqlCommand(sqlStmt);

 command.Parameters.AddWithValue("@a1", userName);
 command.Parameters.AddWithValue("@a2", userPassword);

 object o = command.ExecuteScalar(sqlStmt);

 return (Convert.ToInt32(o) == 1);
}
```

23 NORMALIZACE A SQL

23.1 NULTÁ NORMÁLNÍ FORMA

Žádné omezení (někdy se uvádí nutnost existence alespoň jednoho atributu, který může obsahovat více než jednu hodnotu, někdy se uvádí "entity jsou reprezentovány tabulkami, jejich atributy sloupci").

23.2 PRVNÍ NORMÁLNÍ FORMA

Všechny atributy tabulky jsou již dále nedělitelné, atomické.

PARCELA

KU#	TYP#	CISLO#	PODLOMENI#	VLASTNICI
523641	1	231	2	*ID1,ID2,ID3 …*

VLASTNIK

ID#	JMENO	…
5803042751		

- **nelze zaručit konzistenci databáze pomocí referenční integrity (lze ji však zajistit pomocí triggerů)**
- **nelze efektivně indexovat**
- **komplikované neefektivní SQL dotazy (i když jsou v principu možné)**

```
function vlast
(VLASTNICI IN VARCHAR2,PORADI IN INT)
 RETURN INT; /* vrací jedno ID z řetězce PORADI)  */

select … from  PARCELA A,VLASTNIK B
where
 vlast(A.VLASTNICI,1)=B.ID
union all
select … from  PARCELA A,VLASTNIK B
where
 vlast(A.VLASTNICI,2)=B.ID …
```

- **Problém vymezení domén – je “rodné číslo” doména nebo se skládá ze `DEN,MESIC,ROK,POHLAVI,PODLOMENI`…?**

Zásadně vždy dodržet !!!

23.3 DRUHÁ NORMÁLNÍ FORMA

Každá tabulka obsahuje primární klíč a každý neklíčový atribut je plně závislý na všech atributech tvořící primární klíč.

OBEC

ID_OKRES#	ID_OBEC#	POCET_OBYV_OBEC	POCET_OBYV_OKRES	…
3702	1	398456	1456024	

(není v 2. normální formě - `POCET_OBYV_OKRES` je závislý na části klíče signalizuje existenci entity “`OKRES`”)

V zásadě není bezpodmínečně nutné dodržet (někdy kvůli výkonnosti opravdu nebývá dodržena – v některých případech se vyhneme `join` operaci), musíme dát pozor na:

- **existenci entit, jejichž existenci signalizuje podklíč denormalizovaných tabulek, který způsobuje porušení 2. normální formy.**

- **zaručení konzistence atributů v denormalizované tabulce pomocí triggerů**

Někdy se jedná o netriviální systém triggerů viz. uvedený příklad:

a) **Změna počtu obyvatel v tabulce OBEC vyvolá trigger, který přepočítá `POCET_OBYV_OKRES v tabulce OKRES`.**

b) **Změna počtu obyvatel v tabulce OKRES se musí zpětně promítnout do tabulky OBEC .**

Uvedené nelze provádět řádkovými triggery – tabulka je měněna a

nelze v ní provádět `UPDATE` a `SELECT`!!!

23.4 TŘETÍ NORMÁLNÍ FORMA

Hodnoty atributů nejsou (funkčně) závislé na hodnotách jiných atributů.

VLASTNIK

ID#	JMENO	PRIJMENI	RODNE_CISLO	POHLAVI	…
1	Drášil	Milan	5803042751	M	
2	Drášilová	Dominika	6552104531	Ž	

(není v 3. Normální formě 3. cifra sloupce RODNE_CISLO je závislá na sloupci pohlaví)

VLASTNIK

ID#	…	POHLAVI	ROK_N	MESIC_A	DEN_N	RC
1		M	58	03	04	2751
2		Ž	65	02	10	4531

U rozsáhlejších systémů téměř nelze dodržet – 3. Normální forma zakazuje redundanci dat. Ta bývá někdy i užitečná – rodné číslo může sloužit i ke kontrole správnosti pořízení data narození a pohlaví.

Redundanci můžeme s klidným svědomím povolit, musíme však prostředky databáze zajistit její konsistenci (triggery,integritní omezení)

```
alter table VLASTNIK ADD constraint VLASTNIK_CH1
check
((POHLAVI in ('M','Z')) AND
  (
   (POHLAVI='M' AND
    SUBSTR(RC,3,1) IN ('0','1'))
   )
   OR
   (POHLAVI='Z' AND
    SUBSTR(RC,3,1) IN ('5','6'))
   )))
```