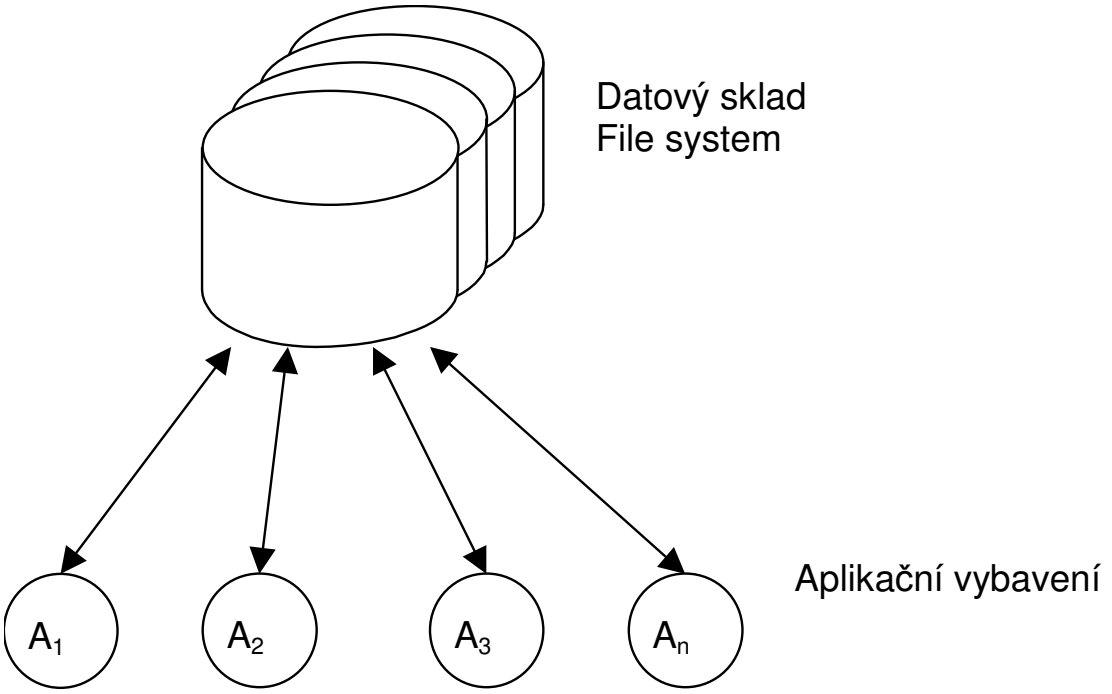
FAKULTA INFORMATIKY, MASARYKOVA UNIVERSITA V BRNĚ

Slide k přednášce

PV003 – Architektura relačních databází

Milan Drášil, únor 2004

Souborově orientované systémy:



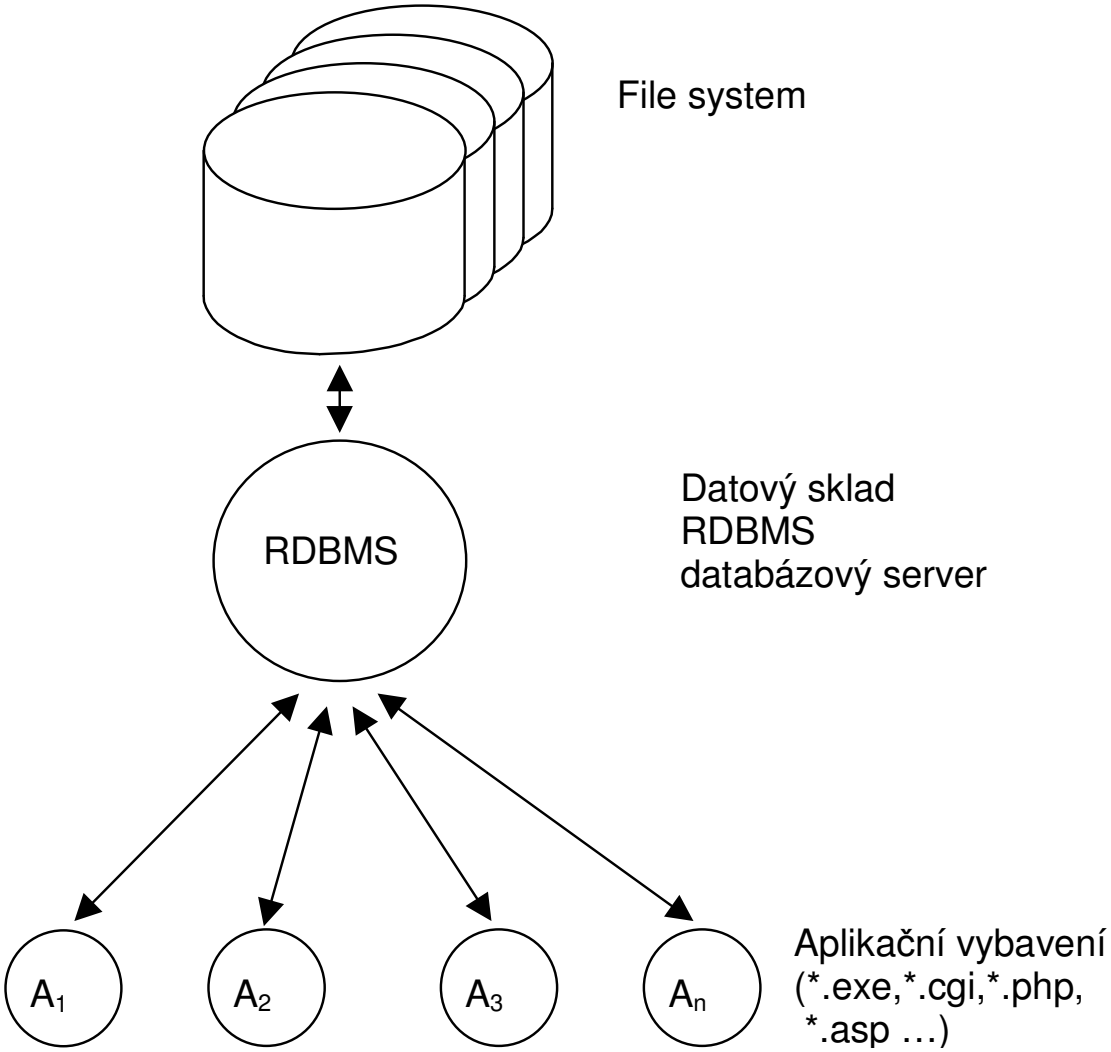
Výhody:

- **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 datum (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 datum**

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 algebra)**

    a) **projekce**
    b) **selekce**
    c) **násobení**
    d) **sjednocení**
    e) **průnik**

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**

Krátká historie SQL:

- **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 asi 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 $\Rightarrow$ univerzální standard neexistuje jednotlivé implementace se liší (ORACLE, MS-SQL, INFORMIX, DB2)

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,…);

 }
```

Embedded SQL:

```
Sprintf(sqlStmt,"select jmeno , prijmeni from …");

EXEC SQL PREPARE ST01 FROM sqlStmt;
EXEC SQL DECLARE CST01 CURSOR FOR ST01;
EXEC SQL DESCRIBE SELECT LIST FOR ST01 INTO selda;

EXEC SQL OPEN CURSOR CST01;
EXEC SQL WHENEVER NOT FOUND GOTO QUERY_FINISHED;

while(..)
 {
  EXEC SQL FETCH CST01 USING DESCRIPTOR selda;

 }
QUERY_FINISHED:
 EXEC SQL CLOSE CURSOR CST01;
```

Lexikální konvence SQL:

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.

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.

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;
```

DML – Data Manipulation Language

Vkládání řádků do tabulek

Příkaz `INSERT`:

`INSERT INTO`
tabulka
(sloupec$_1$,sloupec$_2$,…,sloupec$_n$)
`VALUES`
(hodnota$_1$,hodnota$_2$,…,hodnota$_n$)

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

`INSERT INTO` ***tabulka***
`VALUES (``hodnota`$_1$`,hodnota`$_2$`,…,hodnota`$_n$`)`

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);
```

Výběr z tabulek (vytváření relací) Jednoduché příkazy `SELECT`:

```
select all 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 nutné 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)

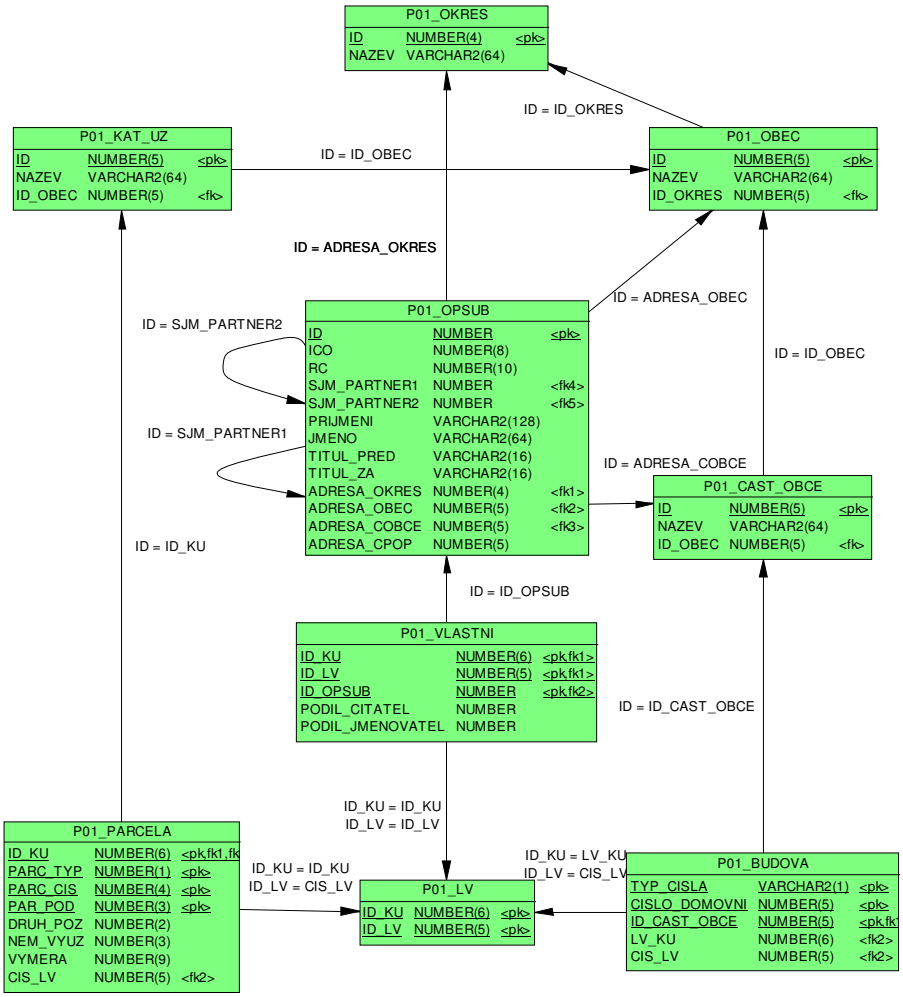
```
select distinct SL1 A,SL2 B from TABULKA;

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

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

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
```



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)

```
select id,nod2 from tp_hrany
where user_name=user and task_id=taskid and
nod1=curr_node and id <> curr_edge and switch=1
and exists
(select null from tp_uzly where user_name=user
and task_id=taskid and
tp_uzly.id=tp_hrany.nod2);
```

Množinové operace nad relacemi:

Sjednocení:

select … **union [all] select…**

Průnik:

select … **intersect select…**

Diference:

select … **minus select…**

Každý **select** je formálně tabulka

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

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

```
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?**

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
  AKTUALNI='A' AND
  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

Uživatelsky definované datové typy (ADT)

```
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)
                  )
           )
)
```

Obecně není možné `select geom from Streets`,
Ne všechny typy klientských rozhraní podporují ADT.
Jednou z možností je podpora pomocí XML:

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

resp:

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

vrátí:

```
<ROW>
<LINESTRING>
 <NUMPOINTS>3</NUMPOINTS>
  <VERTEXES>
   <POINT>
     <X>0</X>
     <Y>0</Y>
   </POINT>
   <POINT>
     <X>2000</X>
     <Y>123</Y>
   </POINT>
   <POINT>
    <X>2020</X>
    <Y>13460</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.

Další možnosti XML:

Metoda `xmlforest`

```
select xmlElement("ROW",
                    xmlforest(ID,GEOM)
                  ).GetStringVal()
from streets
```

```
XmlAgg()

Select
  xmlagg(
   xmlElement(
    "ROW",
    xmlforest(ID,GEOM)
   )
  ).GetStringVal() from streets
```

Poznámka:

ADT nelze indexovat.

XML – metoda `GetClobVal()` je řádově pomalejší, než `GetStringVal()` – a ta funguje jenom do 4KB.

Outer join

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
FROM  t1,t2
WHERE t1.i=t2.i (+)
```

```
I1  I2
------
 1
 2   2
 3   3
```

```
SELECT t1.i i1,t2.i i2
FROM  t1,t2
WHERE t1.i(+)=t2.i
```

```
I1  I2
------
 2   2
 3   3
     4
```

Hierarchické dotazy:

Vyberou 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);
```

CONNECT BY klausule

definuje relaci ve stromu

START WITH klausule

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

Skupinové funkce – funkce založené na více řádcích

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

`AVG([DISTINCT|ALL] 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`

Další skupinové funkce:

```
MAX([DISTINCT|ALL] expr)
MIN([DISTINCT|ALL] expr)
STDDEV([DISTINCT|ALL] expr)
SUM([DISTINCT|ALL] 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;
```

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**

Optimalizace příkazů

Exekuční plán:

```
explain plan for
select
 a.prijmeni,
 b.nazev
from
 p01_okres b,
 p01_opsub a
where
 a.ADRESA_OKRES=b.ID;

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


select
 a.prijmeni,
 b.nazev
from
 p01_okres b,
 p01_opsub a
where
 a.ADRESA_OKRES=b.ID AND
 b.nazev like 'KROM%' AND
 A.PRIJMENI LIKE 'NOV%';

SELECT STATEMENT Optimizer=CHOOSE
  NESTED LOOPS
    TABLE ACCESS (FULL) OF P01_OPSUB
    TABLE ACCESS (BY INDEX ROWID) OF P01_OKRES
      INDEX (UNIQUE SCAN) OF P01_OKRES_PK (UNIQUE)
```

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


select
 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_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)


CREATE INDEX P01_OKRES_I1 ON P01_OKRES(NAZEV);

select
 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_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)
```

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)
```

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 t-cí**
D	**– Durale**	**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 změní něco, co je změněno jinou nepotvrzenou transakcí, která začala dříve, potom transakce končí chybou.

`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;
```

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)

Jazyk - 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;
```

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;
```

INTO fráze:

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

Kursory:

```
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;
```

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 serial_num = serial_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;
```

Asynchronní ošetření chyb:

```
begin
  select   …… into a,b,c;

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

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;
/
```

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;
```

Balíky – Package:

```
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];
```

```
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;
```

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;
```

OPEN-FOR

```
DECLARE
 TYPE EmpCurTyp IS REF CURSOR;
 emp_cv EmpCurTyp;
 my_ename VARCHAR2(15);
 my_sal NUMBER := 1000;
BEGIN

sqlStmt=
'SELECT ename, sal FROM emp WHERE sal > :s'

OPEN emp_cv FOR sqlStmt
 USING my_sal;
...
END;

LOOP
 FETCH emp_cv INTO my_ename, my_sal;
 EXIT WHEN  emp_cv%NOTFOUND;
 .
 .
END LOOP;
```

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]
```

```
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;
```

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.

Zrušení práv

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

Postup 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.**

PRO*C

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 překladač:

```
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();
}
```

Dynamické příkazy v PRO*C

Metoda 1

Příkaz vrací pouze úspěch/neúspěch a nepoužívá hostitelské proměnné.

```
char dyn_stmt[132];
EXEC SQL WHENEVER SQLERROR do gsSqlError();

...
for (;;)
{
 printf("Enter SQL statement: ");
 gets(dyn_stmt);
 if (dyn_stmt[0] == '\0')
  break;
 EXEC SQL EXECUTE IMMEDIATE :dyn_stmt;
}
```

Metoda 2

Příkaz vrací pouze úspěch/neúspěch
a může používat hostitelské proměnné.

```
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;

for (;;)
{
 printf("Co chceš vymazat?");
 gets(temp);
 emp_number = atoi(temp);
 if (emp_number == 0)
  break;
 EXEC SQL EXECUTE sql_stmt USING :emp_number;
}
```

Metoda 3

SELECT dotaz, který vrací více řádků, pevný počet sloupců do hostitelských proměnných.

```
PREPARE statement_name FROM
{ :host_string | string_literal };

DECLARE cursor_name CURSOR FOR statement_name;

OPEN cursor_name [USING host_variable_list];

FETCH cursor_name INTO host_variable_list;

CLOSE cursor_name;
```

```
EXEC SQL PREPARE S1 FOR
  'SELECT RC,JMENO,PRIJMENI FROM P01_OPSUB
WHERE PRIJMENI LIKE ''':A1%'' ';

EXEC SQL DECLARE C1 CURSOR FOR S1;

gets(temp);

EXEC SQL OPEN C1 USING :temp;

for (;;)
{
 EXEC SQL FETCH C1 INTO :rc,:jmeno,:prijmeni;
 if ( …)
  break;
 .
 .
 .
}
```

Metoda 4

SELECT dotazy, který vrací více řádků, proměnný počet sloupců do vnitřních (C) proměnných. Je nutné použít popis těchto proměnných v deskriptoru – struktura SELDA.

```
EXEC SQL PREPARE statement_name
FROM { :host_string | string_literal };

EXEC SQL DECLARE cursor_name CURSOR FOR
statement_name;

EXEC SQL DESCRIBE BIND VARIABLES FOR
statement_name
INTO bind_descriptor_name;

… doplneni BIND deskriptoru …

EXEC SQL OPEN cursor_name
[USING DESCRIPTOR bind_descriptor_name];

EXEC SQL DESCRIBE [SELECT LIST FOR]
statement_name
INTO select_descriptor_name;

EXEC SQL FETCH cursor_name
USING DESCRIPTOR select_descriptor_name;

EXEC SQL CLOSE cursor_name;
```

```
#define ALLOCLEN 1024;

SQLDA *selda;
char   values[20][ALLOCLEN];

selda=sqlald(1,30,30);

gets(sql_stmt.arr)
sql_stmt.len=strlen(sql_stmt.arr);

EXEC SQL PREPARE stmt FROM :sql_stmt;
EXEC SQL DECLARE query_curs CURSOR FOR stmt;
EXEC SQL OPEN query_curs;
EXEC SQL DESCRIBE SELECT LIST FOR stmt INTO
selda;

if(selda->F)>20
 {
 return(
  'SELECT vraci vic nez 20 sloupcu');
 }

for(i=0;i<selda->F;++i)
 {
  .
  selda->V[i]=(char *) value[i];
  selda->L[i]=ALLOCLEN;
  selda->T[i]=DB_CHAR;
 }

while()
{
 EXEC SQL FETCH query_curs00
   USING DESCRIPTOR selda;
 for(i=0;i<selda->F;++i)
  printf('%s',value[i]);
}
EXEC SQL CLOSE query_curs00; ...
```

Návrh DB rozhraní:

```
nativeCode char *dbConnect
/* <= SUCCESS, or ptr to error string    */
(
  char *user,           /* user name       */
  char *pswd,           /* password        */
  char *service,        /* service         */
  long *sessionId       /* session ID      */
);


nativeCode char *dbDisconnect
/* <= SUCCESS, or ptr to error string*/
(
  void
);


nativeCode char *gsProcessSQL
/* <= SUCCESS, or ptr to error string*/
(
  char *stmtP,          /* =>non SELECT SQL    */
  long *rowsProcessed /* <= rows processed or
                          NULL if not
                          required            */
);
```

```
typedef struct
{
 int  nColumns; // numer of cols in descripotor
 char **name;   // ptr to array with col names
 char **value;  // ptr to array with values to
 int   *type;   // type of values in descriptor
 long  *allocLength;  // lengths of buffers
                      //(**value)
 long  *actLength; // actual lengths fetched
 int   *nullVal;   // TRUE if NULL
}
DB_sqldaT;



nativeCode char *dbInsert
/* <= SUCCESS, or ptr to error string       */
(
  char      *tableName,
  DB_sqldaT *values
);

nativeCode char *dbUpdate
/* <= SUCCESS, or ptr to error string       */
(
  long *rowsProcessed, // <= rows processed
  char      *tableName,// => table  name
  DB_sqldaT *values,   // => values
  char      *whereClause  // whereClause
                          // or NULL
);
```

```
nativeCode char *dbOpenCursor
/* <= SUCCESS, or ptr to error string*/
(
  int        *query_idP,  // <= query ID
  char       *selectStmtP,// => full select
  int        nValues      // => number of
);


nativeCode char *dbFetch
/* <= SUCCESS, or ptr to error string        */
(
  int        query_id,    // => query ID
  DBsqldaT *values        // <= values
);


nativeCode char *gsCloseCursor
/* <= SUCCESS, or ptr to error string   */
(
  int       query_id       // => query id.
);



nativeCode int gsGetNCols
(
 char       *selectStmtP  // => full select
);
```

Normalizace a SQL

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").

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 !!!

Druhá normální forma - 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

OKRES

ID_OKRES#	POCET_OBYV_OKRES	…
3702	1456024	

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!!!

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 takřka 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'))
   )
  )
)
```