FAKULTA INFORMATIKY, MASARYKOVA UNIVERSITA V BRNĚ

Text k přednášce

PV003 – Architektura relačních databází

Milan Drášil, 2004 – 2025

1 KOŘENY

1.1 DATOVÉ MODELOVÁNÍ

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

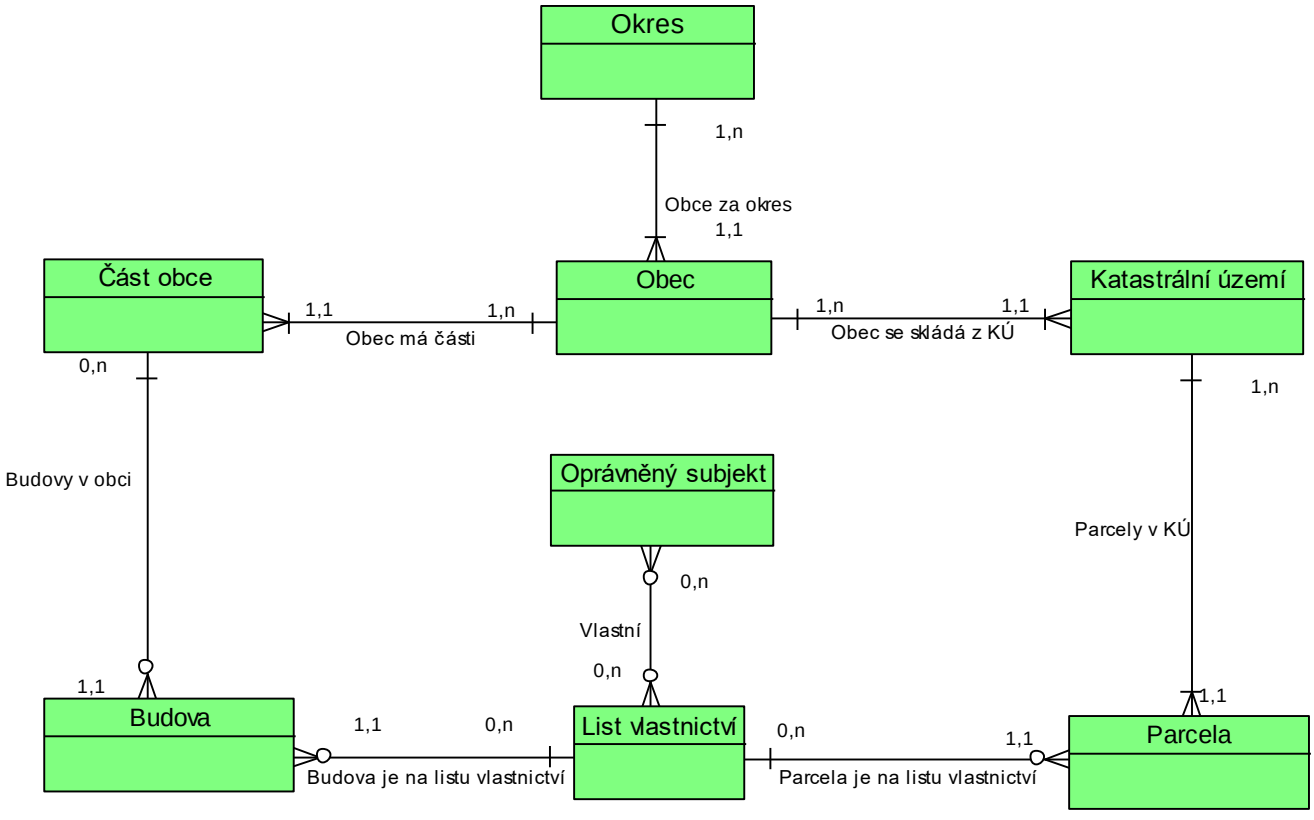
Entita (entitní typ) – je typ objektu zájmu informačního systému, který je jednoznačně identifikovatelný svými atributy. Instance entity může být fyzicky existující objekt (dům, osoba), nebo také událost (prodej auta).

Vztah (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 ($1:1, 1:n, m:n, \ldots$).

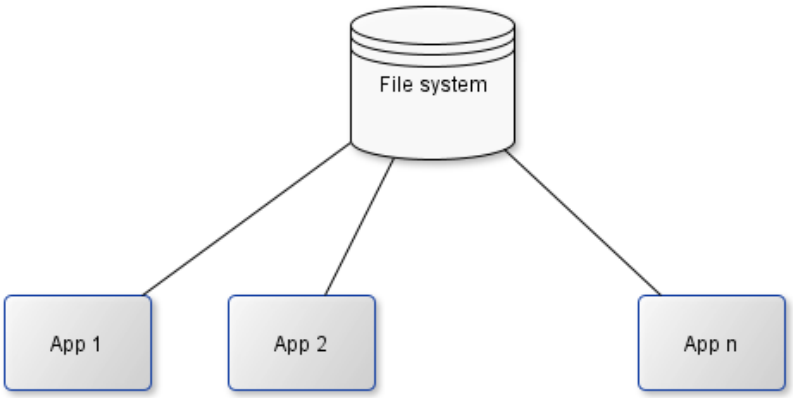
Příklad Konceptuální model vlastnických vztahů nemovitostí



□

1.2 PŘÍSTUP K INFORMACÍM, DATŮM

Souborově orientované systémy



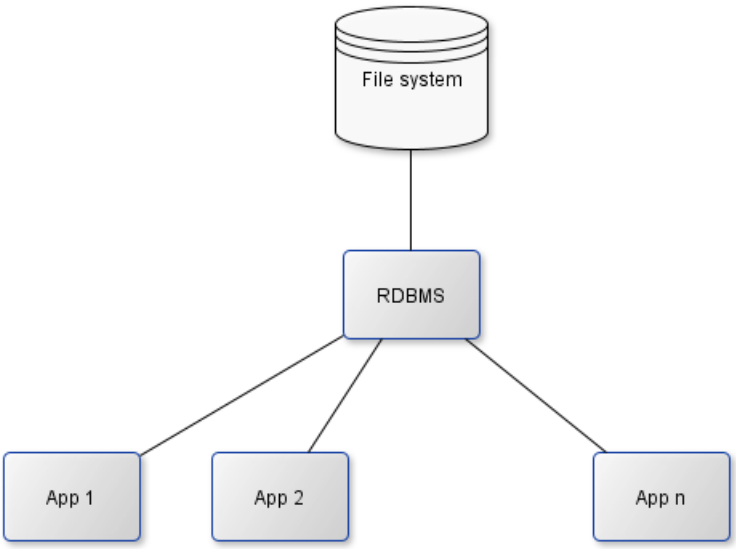
Výhody:

- Optimalizace datových struktur vzhledem k řešené problematice.
- Menší systémové nároky.

Nevýhody:

- Aplikační závislost, aplikace musí „znát“ interní strukturu souborů.
- Obtížné zabezpečení konzistence dat.
- Obtížná realizace konkurentních operací (zamykání souborů).
- Obtížná čitelnost, dokumentovatelnost datového modelu.
- Obtížně implementovatelný transakční přístup pro operaci s daty.
- Obtížné řízení přístupových práv.

Databázově orientované systémy



Tento přístup vyžaduje standardizované struktury dat

Výhody:

- Aplikační „nezávislost“.
- Snadné zabezpečení konzistence dat.
- Možnost 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 se podrobit normalizovaným datovým strukturám, a to může zkomplikovat situaci.
- Neprocedurální přístup k datům. Vyskytují se případy, kdy tvůrce je chytřejší než sebelepší strojová optimalizace.

2 PREDIKÁTOVÁ LOGIKA 1. ŘÁDU (FOL)

[viz např: https://cs.wikipedia.org/wiki/Predikátová_logika_prvního_řádu, https://cs.wikipedia.org/wiki/Predikátová_logika]

(Poznámka: tento odstavec nenahrazuje Úvod do logiky, obsahuje četné zkratky. Má sloužit pouze k uvedení následujících odstavců do souvislostí.)

V roce 1879 německý logik Gottlob Frege publikoval článek Begriffsschrift (jazyk formulí), popsal první formální jazyk predikátové logiky. Největší Fregeho pokrok od Aristotelovy logiky sylogismů byl dosažen zejména ve formalizaci a obecnosti jeho logiky. Tento článek položil základy Predikátové logiky 1. Řádu, FOL (=First Order Logic).

2.1 DEFINICE FOL

Jazyk FOL

Konstanty $c_0, c_1, \ldots$

Proměnné $x_0, x_1, \ldots$

Logické symboly $\wedge, \vee, \exists, \ldots$

Mimologické symboly (objekty universa, domény, relace)

N – ární relace

P – nad množinami (třídami) $D_1, .., D_n$ (domény relace) je jakákoli podmnožina:

$$P \subseteq D_1 \times \ldots \times D_n$$

Predikátový symbol

Pro n-ární relaci P zavedeme její predikátový symbol

$$P(x_1, x_2, \ldots, x_n)$$

Pro každou n-tici $c_1, c_2, \ldots, c_n$, kde $c_i \in D_i$ má smysl tvrdit, zda $P(c_1, \ldots, c_n)$ je pravdivé, tj. $[c_1, c_2, \ldots c_n] \in P$

Funkce

je relace $F \subseteq D_1 \times ..\times D_n \times Y$, kde pro každé $[c_1, .., c_n] \in D_1 \times ..\times D_n$ existuje právě jedno $c_y \in Y$ takové, že:

$$[c_1, .., c_n, c_y] \in F$$

Množinu Y nazýváme obor hodnot funkce F.

Funkční symbol

podobně, jako jsme zavedli pro relaci P její predikátový symbol, zavedeme pro funkci její n-ární funkční symbol $F(x_1, x_2, \dots, x_n)$ $(= y)$.

Universum

je jakákoli množina domén a relací popisující „oblast zájmu“.

Term

- Jakákoli konstanta a proměnná je term (např: x, y, c_0 …).
- Jakýkoli výraz $F(t_1, \dots, t_n)$, kde t_i je term a F je n-ární funkční symbol, je term
- Term je vytvořen konečnou aplikací předešlých pravidel.

Formule

- Pokud P je n-ární predikátový symbol a $t_1, \dots, t_n$ jsou termy, pak $P(t_1, \dots, t_n)$ je formule.
- Symbol rovnosti $=$. Pokud t_1 a t_2 jsou termy, potom $t_1 = t_2$ je formule
- Negace. Pokud φ je formule, pak její negace $\neg\, \varphi$ je také formule.
- Logické spojky. Pokud φ a ψ jsou formule a □ je binární logická spojka (→,∧, …), pak $\varphi \square \psi$ je také formule.
- Kvantifikátory. Pokud φ je formule a x je proměnná, potom je formule i $\forall x \varphi(x)$ (a $\exists x \varphi(x)$).
- Formule je to, co vznikne konečnou aplikací předchozích pravidel.
- Formule se nazývá otevřená, obsahuje-li nekvantifikovanou proměnnou.

Poznámka: FOL obsahuje dále logické axiomy a odvozovací pravidla, ostatně na ní „stojí” matematika. Tento aparát však pro potřeby této přednášky můžeme vynechat.

Poznámka: Pro přehlednost jsme vynechali definici domén funkcí a termů, budeme předpokládat, že studenti intuitivně vnímají, jak se vyhnout špatným konstrukcím, např.:

$$\sqrt[n]{\sin\,(Drášil^2)}$$

Příklad z historie

V hymně 35. Pěšího pluku (Karel Hašler, 1913) se v refrénu zpívá:

„..když jsme mašírovali, všechny panny plakaly..”

Pokusme se tuto situaci popsat prostředky FOL.

Univerzum:

H - množina všech lidí

Predikátové symboly:

$M(x), x \in H, x$ mašíruje

$V(x), x \in H, x$ je panna

$C(x), x \in H, x$ pláče

Funkce:

$B(x) = y,\ x \in H$, vrací y, číslo pěšího pluku pro x, nebo 0, když x není příslušníkem žádného pěšího pluku (jiný pěší pluk, artilerie, kavalerie, nevojáci,..)

Formule:

$$\forall x((B(x) = 35) \rightarrow M(x)) \rightarrow \forall y(V(y) \rightarrow C(y))$$

říká toto:

„Pokud z toho, že x je příslušníkem 35. pěšího pluku plyne, že x mašíruje, potom z toho, že y je panna plyne, že y pláče”

Neplačící panny za podmínky mašírujícího 35. PP potom lze definovat pomocí FOL jako množinu NCV:

$$NCV = \{y | y \in H, \forall x((B(x) = 35) \rightarrow M(x)) \wedge (V(y) \wedge \neg C(y))\}$$

□

Jazykem formální logiky 1. řádu lze tedy „popsat” množiny s určitými vlastnostmi.

Příklad Model popisující lidstvo

Universum:

- množina $Human$, reprezentující všechny lidské bytosti
- množina $Sex = \{male, female\}$, reprezentující pohlaví
- relace $SexOfHuman \subset Human \times Sex$, reprezentující pohlaví lidské bytosti
- relace $ParentOfHuman \subset Human \times Human$ s významem x je rodič y právě když $[x, y] \in ParentOfHuman$.

Zformulujme dále vlastnosti množiny $Human$. Uděláme to pomocí formulí FOL:

1) Každý člověk je buď muž, nebo žena:

$$\forall(x, s_1, s_2)\ x \in Human, (x, s_1) \in SexOfHuman \wedge (x, s_2) \in SexOfHuman \rightarrow$$

$$s_1 = s_2$$

Tato vlastnost nám umožní definovat funkci $y = sex(x)$

2) Každý člověk má maximálně jednoho otce a maximálně jednu matku:

$$\forall(x, y_1, y_2)\ x \in Human\ \wedge$$
$$[y_1, x] \in ParentOfHuman\ \wedge [y_2, x] \in ParentOfHuman \wedge sex(y_1) = sex(y_2)$$
$$\rightarrow y_1 = y_2$$

3) Každý člověk má buď oba, nebo žádného rodiče (může existovat prvek Eva resp. množina $Adam$):

$$\forall(x, y)[x, y] \in ParentOfHuman\ \exists(z) z \in Human \wedge [z, y] \in ParentOfHuman \wedge$$
$$\neg\ (sex(z) = sex(x))$$

4) Zkusme nyní zformulovat takovou vlastnost, že žádný člověk nemůže být sám sobě rodičem, je to jednoduché:

$$\forall(x)\ x \in Human \rightarrow [x, x] \notin ParentOfHuman$$

5) A konečně vlastnost, že žádný člověk nemůže být svým předkem:

$$\forall(\{x_i\}_{i=1}^{n})[x_{i+1}, x_i] \in ParentOfHuman \rightarrow x_1 \neq x_n$$

□

Pozor, 5) NENÍ formulí FOL

Kvantifikuje totiž „všechny konečné posloupnosti“ – proměnné FOL mohou zastupovat pouze pro elementy relací. FOL je tedy užitečný nástroj, jak popsat oblast zájmu, kterou chceme modelovat i když jsme si ukázali, že to není nástroj všemocný.

2.2 LIMITY FOL

Binární relaci $R \subseteq A$ x B nazveme transitivní, právě když

$$([a,b] \in R \wedge [b,c] \in R) \rightarrow [a,c] \in R$$

Transitivní uzávěr $T(R)$ relace R je transitivní relace $T(R) \supseteq R$ taková, že pro každou transitivní relaci $Q \supseteq R$ je $Q \supseteq T(R)$.

Transitivní uzávěr binární relace ***NELZE*** vyjádřit prostředky relační algebry.

K tomuto tvrzení lze dospět (uvádíme jen schéma rigorózního důkazu) takto. Uvažme binární relaci:

$$R = \{[c_0, c_1], [c_1, c_2], \ldots, [c_{n-1}, c_n]\}$$

a formulaci ve $T^\star$ FOL, která reprezentuje transitivní uzávěr R, tedy všechny dvojice $[c_i, c_j],\ i < j$:

$$T^\star(R) = \{[x,y] | \phi\}$$

kde například:

$$\phi =$$

$$[x,y] \in R \vee$$

$$\exists(x_1)([x,x_1] \in R \wedge [x_1,y] \in R) \vee$$

$$\exists(x_1,x_2)([x,x_1] \in R \wedge [x_1,x_2] \in R \wedge [x_2,y] \in R) \ldots$$

Nechť $l = |\phi|$ je délka (ve znacích) formule ϕ. Lze dokázat (Aho, Ullman 1979), že pro každou formuli FOL délky l existuje $n \in \mathbb{N}$ takové, že pro:

$$R = \{[c_0, c_1], [c_1, c_2], \ldots, [c_{n-1}, c_n]\}$$

$$[c_0, c_n] \notin T^\star(R)$$

tedy

$$T^\star(R) \neq T(R)$$

3 RELAČNÍ ALGEBRA

Běžně lze ukládat data („instance entit“) ve formě záznamů v tabelární formě, jejichž jednotlivé položky mají týž (sémantický) význam jako atributy entit. Zkusme si na tomto místě navíc uvědomit, že tabulky v podstatě reprezentují relace univerza, a to „výčtem pravdivých n-tic“.

Příklad Data o podniku

Zaměstanci - Zam ⊆ int x string x string x string

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

Oddělení - Odd ⊆ string x string

ID	Nazev
Adm	Administrativa
Pro	Provoz
..	

Projekty - Proj ⊆ string x string

ID	Nazev
ÚAP	Územní analytika
..	

Alokace zaměstnance na projektech - ZaP ⊆ int x string x int

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

Odpracovaná doba - Prac ⊆ date x int x number

IdZam	Datum	Hod
1	01-08-2024	4
1	01-08-2024	2.5
2	01-08-2024	8
..		

Nad takovými strukturami, tabulkami, 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 (Edgar F. Codd červen 1970, ACM Journal, Communications of ACM):

- datové struktury jsou n-ární relace
- relace univerza jsou reprezentovány tabulkami, výčtem pravdivých n-tic relace
- nad relacemi jsou proveditelné operace relační algebry

Relační algebra (primitivní operátory):

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

$$R \subseteq M_1 \text{ x .. x } M_n$$

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

Selekce $\sigma(R, \varphi)$ – je podmnožina relace R splňující danou formuli φ (viz FOL).

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.

Rozdíl relací (stejných typů) - je běžný množinový rozdíl.

Relační algebra (rozšiřující operátory):

Přejmenování – Má význam při násobném výskytu jedné relace v operaci součinu (např. přejmenujeme relaci $name$ na $name_1$ a $name_2$ podle toho o jaký výskyt jde).

Operátor agregace a grupování – $\Gamma(R, \gamma), \alpha(x)$ …,kde R je relace, γ je seskupovací atribut, x je agregovaný atribut a α je agregační funkce (například suma, průměr). Γ pro všechny výskyty elementů relace R se stejnou hodnotou seskupovacího atributu vypočítá hodnotu $\alpha(x)$ a vrací jako jeden prvek výsledné relace.

Coddova věta (zjednodušeně):

Relační algebra je, so se týče vyjadřovací síly, ekvivalentní s vyjadřovací silou FOL.

Limity relační algebry:

Relační algebra je silným dotazovacím nástrojem, i když obecně existují „dotazy“, které nelze vyjádřit jejími prostředky.

Na aparát relační algebry se budeme odkazovat zkratkou RA

Požadavek na data z Data o podniku Lze formulovat v relační algebře (jména sloupců relací jsou proměnné):

$$
\begin{aligned}
&\Pi(\\
&\sigma\ (\\
&\quad Zam \times Odd \times ZaP, \\
&\quad Zam.Odd = Odd.ID \wedge\ Zam.ID\ =\ Zap.Zam \\
&\ ), \\
&\ (Zam.Jmeno, Zam.Prijmeni, Odd.Nazev, ZaP.Projekt, ZaP.Procent) \\
&\ )
\end{aligned}
$$

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

Následující výraz v relační algebře s agregační funkcí $count(*)$

$\Gamma(\Pi(Odd \times ZaP, Odd.Nazev), Odd.Nazev, count(*))$

potom vrátí relaci, která obsahuje názvy oddělení a jejich počet zaměstnanců

Cvičení 1

Předpokládejme, že máme k dispozici funkce, která pro argument typu `date` vrací den, resp. měsíc, resp. rok. $Day(x)$**,** $Month(x)$, $Year(x)$ a agregační funkci $sum(number)$ Sestrojte výraz v relační algebře, který vrátí seznam zaměstnanců a jejich celkovou odpracovanou dobu za zadaný měsíc a rok.

4 JAZYK RELAČNÍ DATABÁZE

4.1 KRÁTKÁ HISTORIE SQL

- Firma 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 **Q**uery **L**anguage

- dnes všeobecně uznáván za standard pro komunikaci s relačními databázemi.
- Jsou kodifikovány standardy SQL (ANSI, ISO/IEC)

U komerčních producentů jde vývoj (pochopitelně!) rychleji, než práce standardizačních komisí $\Rightarrow$ univerzální standard **neexistuje** jednotlivé implementace se liší (ORACLE, MS-SQL, INFORMIX, DB2, MySQL.Postgresql, Sybase)

4.2 POŽADAVKY NA JAZYK RELAČNÍ DATABÁZE

- vytváření, modifikace a rušení relací, definice (schématu) univerza
- dotazy nad tabulkami, tj. implementace relační algebry
- vkládání, změna, odstranění řádku v tabulkách
- garance konzistence dat
- řízení přístupových práv
- řízení transakcí

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)

Příklad Souborový přístup k datům

```
FILE *inf;
inf=fopen(…);
while(  )
 {
  fseek(inf,…);
  fread(inf,…);
 }
```

□

Příklad Použití databázového stroje

```
string sql =
 "select jmeno, prijmeni, plat  from zamestnanci";
Cursor cursorRes = OpenCursor(sql);
while
(
 (object fetchRes = FetchCursor(cursorRes) ) !=null
)
{
 //Zpracuj..
}
CursorRes.Close();
```

□

4.3 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.

Některé databázové stroje 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"
```

4.4 ZÁKLADNÍ ELEMENTY JAZYKA SQL

- Datové typy – netabelární unární relace, primitivní domény patřící každému „univerzu", např.: `int, number(m,n), date, varchar(n),long`
- Konstanty – nulární funkce, např.: `101,'text'`
- Netabelární binární relace nad datovými typy, např.: „<" pro typ `number`, `date`, `varchar`..
- Netabelární funkce nad datovými typy, např.: `UPPER(varchar),sin(number)`
- Logické operátory `AND`, `OR` a `NOT`
- Agregační funkce, `COUNT(*),SUM(number)`…
- Komentáře (`/* */`)
- `NULL` speciální hodnota pro prázdnou hodnotu
- Klíčová slova `CREATE, DROP, SELECT, GROUP BY`…
- Objekty databázového schématu, popisující univerzum (tabulky relací, pohledy, indexy, sekvence, …)

Hodnota NULL stojí za pozornost, zavádí do světa relační databáze (v jistém smyslu) tříhodnotovou logiku. Například sloupec typu **bool** může nabývat hodnoty **true**, **false** nebo **NULL**.

Výsledky logických operací s null argumentem (operace jsou komutativní, na pořadí nezáleží):

p	q	p AND q	p OR q
`true`	`null`	`null`	`true`
`false`	`null`	`false`	`null`
`null`	`null`	`null`	`null`

Ostatní operace, do nichž vstupuje hodnota null dávají výsledek také null, např:

```
1+null=null
```

Z uvedeného také 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.

Názvy základních datových typů se mohou v závislosti na konkrétním výrobci mírně lišit jak názvem (`int`, `integer`), tak implementací (`numeric`, `float`, `double`).

5 DDL – DATA DEFINITION LANGUAGE

5.1 VYTVÁŘENÍ TABULEK PŘÍKAZ CREATE TABLE

Zam ⊆ int x **string** x **string** x **string**

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

```
CREATE TABLE Zam
 (
  id       NUMBER,
  Jmeno          VARCHAR2(64),
  Prijmeni       VARCHAR2(64),
  Odd      VARCHAR2(16) default 'Prov',
 );
```

5.2 MODIFIKACE TABULEK - PŘÍKAZ ALTER TABLE

Přidání sloupce:

```
ALTER TABLE Zam ADD rc varchar2(32);
```

Změna typu sloupce:

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

Odebrání sloupce:

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

5.3 SEKVENCE

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 SEQUENCE SEQ1 START WITH 32 INCREMENT BY 100;
```

Poznámka

Některé databázové stroje mají implementované sekvence přímo jako speciální datový typ, např. v Postgresql při vytvoření tabulky můžeme použít:

```
id   bigserial
```

Při vkládaní hodnot a absenci hodnoty sloupce ID se automaticky vkládá hodnota ze sekvence (poslední + 1).

5.4 INTEGRITNÍ OMEZENÍ (PRAVIDLA PRO KONZISTENCI DAT):

Povinnost:

```
ALTER TABLE ZAM ADD CONSTRAINT nn_zam NOT NULL (id);
```

Jednoznačnost:

```
ALTER TABLE ZAM ADD CONSTRAINT pk_zam UNIQUE (id);
```

Primární klíč – jednoznačná identifikace řádku:

```
ALTER TABLE ZAM ADD CONSTRAINT
 pk_emp PRIMARY KEY (id);
```

Cizí klíč – odkaz na hodnotu v jiném sloupci (jiné) tabulky:

```
ALTER TABLE ZAM ADD CONSTRAINT fk_deptno
 FOREIGN KEY (Odd) REFERENCES Odd(id);
```

Kontrola vkládaných dat:

```
ALTER TABLE ZAM ADD CONSTRAINT chk_id CHECK (id>0);
```

Přehled integritních omezení:

`NOT NULL`	Vyplnění sloupce je povinné
`UNIQUE`	Sloupec (sloupce) má unikátní hodnoty v celé tabulce
`PRIMARY KEY`	Primární klíč tabulky, řádek je jednoznačně identifikován hodnotami sloupce/ů
`REFERENCES`	Referenční integrita, hodnota sloupce/ů je hodnotou primárního klíče jiné (stejné) tabulky
`CHECK`	Kontrola vkládaného/modifikovaného řádku

Cvičení 2

Vytvořte v jazyku SQL schéma (včetně integritních omezení) pro příklad Data o podniku.

6 INDEXY

Index je interní struktura nad sloupcem/sloupci tabulky, která pomáhá k efektivnějšímu přístupu na její řádky, je-li použit „uplatnitelný“ predikát (podmínka).

V původních implementacích SQL databází byl implementován jediný typ indexu (struktura B^+-tree) na úplně uspořádaných datových typech, kdy libovolné dvě hodnoty daného typu byly srovnatelné – platila vždy jedna z možností „=“ „<“ „>“.

„Uplatnitelné“ predikáty na tomto typu indexu jsou:

```
INDEXOVANY_SLOUPEC = výraz
INDEXOVANY_SLOUPEC > výraz
INDEXOVANY_SLOUPEC < výraz
INDEXOVANY_SLOUPEC BETWEEN dolní_mez AND horní_mez
```

Při jednom průchodu tabulkou je vždy uplatněn maximálně jeden index, RDBMS prochází primárně indexovou strukturu a z ní získává odkazy na řádky tabulky.

Například, mějme ve schématu tabulku a indexy:

```
CREATE TABLE zam
(jmeno VARCHAR(128),pozice VARCHAR(128));

CREATE INDEX zam_idx1 ON zam (jmeno);
CREATE INDEX zam_idx2 ON zam (pozice);
```

Při podmínce:

```
jmeno=jmeno AND pozice=pozice
```

se uplatní jen jeden index, buď `zam_idx1`, nebo `zam_idx2`

K „efektivnějšímu“ zásahu prvního řádku tabulky na takový dotaz slouží složený index:

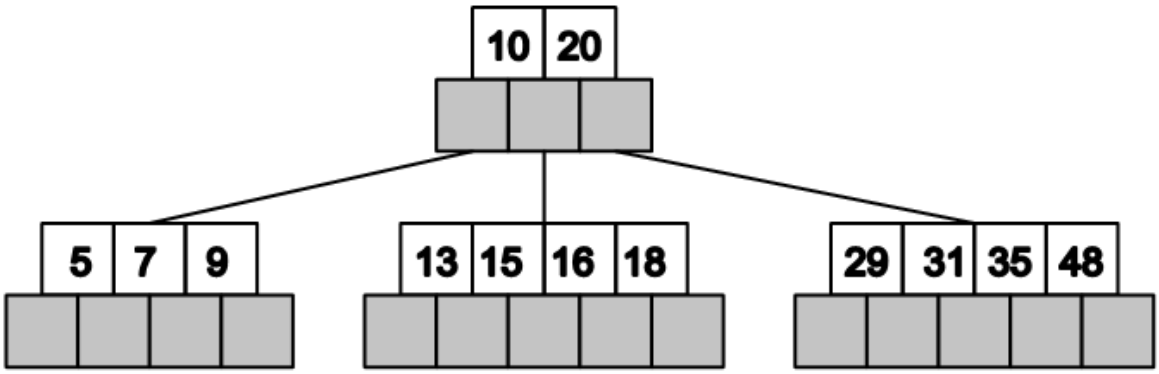
```
CREATE INDEX zam_idx1 ON zam (jmeno, pozice);
CREATE INDEX zam_idx2 ON zam (pozice, jmeno);
```

6.1 B-TREE (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 $key_1, .., key_k$ 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 $p_0 key_1 p_1 \ldots p_{k-1} key_k p_k$.



6.2 R-TREE

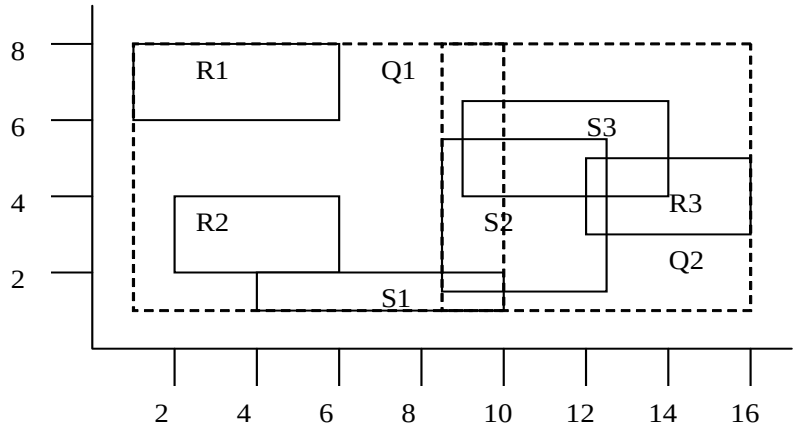
Analogie k B-stromům, klíče jsou $2D$ (obecně, nD) obdélníky.

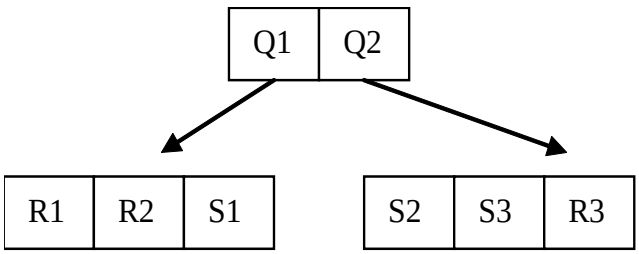
M – maximální počet klíčů v uzlu,

$m \leq M/2$ – minimální počet klíčů v uzlu

Definice R-tree:

- Každý uzel obsahuje minimálně m klíčů a maximálně M klíčů, pokud není kořen.
- Klíče v R-tree 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.





6.3 GiST stromy (Generalized Index 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.

- Uzly GiST jsou tvořeny n-ticemi tvaru $(key, next)$, kde key je instance libovolného typu (třídy) a $next$ je následný uzel (může být $null$ pro list).
- Uzel obsahuje maximálně N klíčů a pokud není kořen minimálně $M \leq N/2$ klíčů.

V „objektové terminologii“ se jedná o jakoukoli třídu obsahující definici klíčů a operátorů, která implementuje následující metody (zjednodušeno, podrobněji na http://www.postgresql.org):

- $\boldsymbol{Consistent}$: pro daný klíč key a dotaz $query$ vrací $false$, je-li zaručeno, že $query$ není pravdivý pro libovolný klíč $key_1 \subseteq key$. V opačném případě vrací $true$.
- $\boldsymbol{Union}$: pro sadu klíčů $\{key_i\}$ vrací jejich sjednocení $KEY = \cup \{key_i\}$, přičemž:

  $$\boldsymbol{Consistent}(key_i, query) \rightarrow \boldsymbol{Consistent}(KEY, query)$$

  pro každý klíč key_i a pro libovolný možný dotaz $query$
- $\boldsymbol{Penalty}(key_1, key_2)$, vrací míru rozšíření klíče key_1 o klíč key_2
- $\boldsymbol{PickSplit}(nod)$, rozdělí sadu klíčů uzlu nod na dvě sady K_1 a K_2 tak, aby pro všechny možné dotazy q v co nejvyšší míře:

  $\boldsymbol{Consistent}(\cup K_1, q)$ vylučovala pravdivost $\boldsymbol{Consistent}(\cup K_2, q)$

  a naopak
- $\boldsymbol{Same}$: operátor rovnosti.
- $\boldsymbol{Compres}, \boldsymbol{Decompres}$: binární podoba uzlu v souborovém systému.

Poznámka

V R-tree je mírou pro $\boldsymbol{PickSplit}$ je obsah průniku obdélníků $(\cup K_1) \cap (\cup K_2)$.

Algoritmus Search:

Vstup: Uzel stromu GiST R, dotaz $query$

Výstup: n-tice všech klíčů, pro které je $\boldsymbol{Consistent}(key_i, query)$

1. Je-li R list, potom dej na výstup všechny klíče key_i, pro které je $query$ pravdivý na key_i.
2. Není-li R list, potom proveď algoritmus pro všechny následníky určené těmi uzly key_i, pro které je $\boldsymbol{Consistent}(key_i, query)$.

Algoritmus Insert:

Vstup: Uzel stromu (GiST) R a klíč key

1. Není-li R list, potom:

    1.1. Vyber z R takový klíč key_i pro který je $\boldsymbol{Penalty}(key_i, key)$ minimální.

    1.2. Polož key_i = $\boldsymbol{Union}(\{key_i, key\})$

    1.3. Proveď algoritmus pro následníka key_i.

2. Je-li R list, potom vlož key do R.

    2.1. Není-li překročena maximální kapacita uzlu v GiST potom konec, jinak pokračuj 2.2.

    2.2. Je-li překročena maximální kapacita uzlu, nechť K_1 a K_2 jsou dvě sady klíčů z metody $\boldsymbol{PickSplit}(R)$. Vytvoř dva nové uzly nod_1 a nod_2 ze sad klíčů K_1 a K_2.

    2.2.1. Není-li R kořen, nechť P je rodičovský uzel R, jinak vytvoř nový prázdný kořen P.

    2.2.2. Odstraň z P ten klíč, jehož je R následníkem a vlož do něj klíče $(\boldsymbol{Union}(K_1), nod_1)$, $(\boldsymbol{Union}(K_2), nod_2)$ a opakuj krok 2.1. pro uzel *P*.

Indexy tohoto typu jsou například využitelné například v „krychlových n-dimensionálních dotazech“ tzv. OLAP (OnLine Analytical Processing) architektur. Jeden index je vybudován pro všechny sloupce tabulky a je použit v dotazech typu:

```
SLOUPEC_1 BETWEEN LOW_1 AND HIGH_1 AND
SLOUPEC_2 BETWEEN LOW_2 AND HIGH_2 AND…
SLOUPEC_N BETWEEN LOW_N AND HIGH_N
```

Cvičení 3

Doplňte potřebné indexy do vytvořeného schématu z příkladu Data podniku.

7 DML – DATA MANIPULATION LANGUAGE

7.1 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.

7.2 ZMĚNA HODNOT V ŘÁDCÍCH TABULKY, PŘÍKAZ UPDATE

UPDATE *tabulka* SET

sloupec$_1$=hodnota$_1$,

…

sloupec$_n$= hodnota$_n$

změní hodnoty na všech řádcích tabulky

UPDATE *tabulka* SET *sloupec$_1$=hodnota$_1$,…,sloupec$_n$= hodnota$_n$*

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!

7.3 ODSTRANĚNÍ ŘÁDKŮ Z TABULKY, PŘÍKAZ DELETE

`DELETE FROM` *`tabulka`*

odstraní vše (!)

`DELETE FROM` *`tabulka`* `WHERE` *`podminka`*

odstraní vše, co vyhovuje podmínce.

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);
```

7.4 VÝBĚRY Z TABULEK, VYTVÁŘENÍ RELACÍ

Jednoduché příkazy `SELECT`:

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

Sloupce lze v rámci příkazu SELECT přejmenovat (implementace operace přejmenovaní z R):

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

Libovolný výraz (sloupec/konstantu) lze v SQL příkazu nahradit jedno řádko-sloupcovým `SELECT` příkazem.

```
create table OSOBA
(
 ID INT PRIMARY KEY,
 JMENO VARCHAR2(64),
 PRIJMENI VARCHAR2(64),
 ID_MATKA INT REFERENCES OSOBA(ID),
```

```
 ID_OTEC INT REFERENCES OSOBA(ID)
)
SELECT
 A.JMENO "Jméno",
 A.PRIJMENI "Příjmení",
 (SELECT JMENO FROM OSOBA B WHERE B.ID=A.ID_MATKA) "Jméno matky"
FROM
 OSOBA A
```

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 známého 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
```

7.5 MNOŽINOVÉ OPERACE NAD RELACEMI

Sjednocení:

```
select … union [all] select…
```

Průnik:

```
select … intersect select…
```

Diference:

```
select … minus select…
```

7.6 WHERE KLAUSULE

Realizuje operaci *selekce* z relační algebry. I pro `WHERE` klausuli platí „univerzální“ pravidlo:

Libovolný výraz (sloupec/konstantu) lze v SQL příkazu nahradit jedno řádkovým - jedno sloupcovým `SELECT` příkazem.

Porovnání výrazu s výrazem

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

Porovnání výrazu se seznamem výrazů

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

Příslušnost k množině

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

Rozsahový dotaz

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

NULL test

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

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
 );
```

Porovnání řetězců

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

Logická kombinace pomocí logických operátorů `AND OR NOT`

7.7 SKUPINOVÉ (AGREGAČNÍ) FUNKCE

Jsou funkce, které vrací jeden výsledek na základě vstupu z více řádků. Z pohledu relační algebry se jedná o implementaci operátoru Γ.

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`

Cvičení 4 Násobení „řídkých“ matic

Ve schématu máme dvě „řídké“ matice ve tvaru (jsou vyplněny pouze nenulové prvky matice, $a_{ij} = 100$ odpovídá `insert into A values (i,j,100)`)

```
create table A (resp. B)
(
 i int,
 j int,
 val numeric check (val<>0)
);
```

Sestrojte SQL dotaz, který vrátí maticový násobek (viz jakákoli učebnice lineární algebry) matic `A` a `B`.

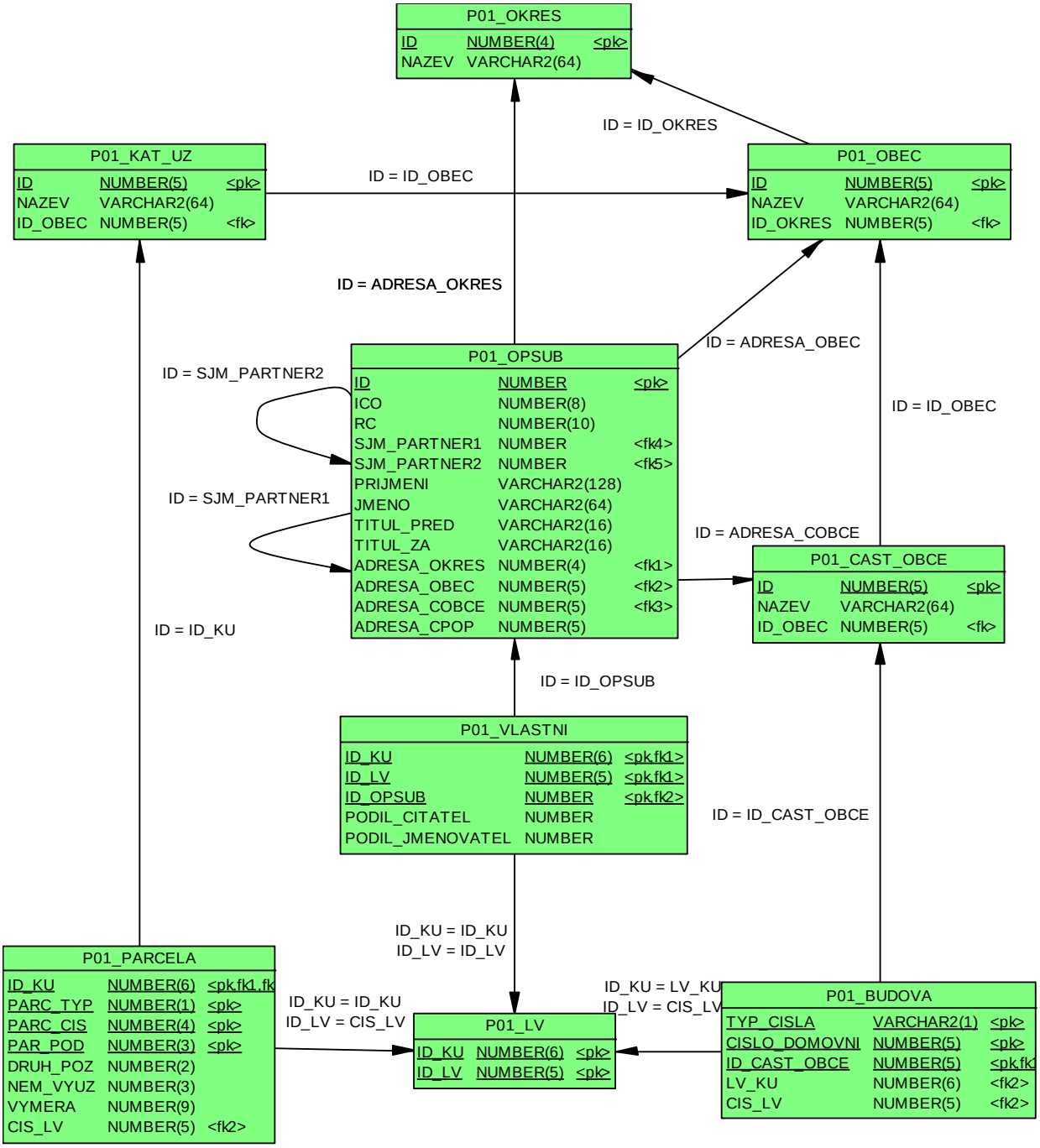
7.8 STRATEGIE VYHODNOCENÍ SQL DOTAZU

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.

7.9 SPOJOVÁNÍ TABULEK (JOIN) – NÁSOBENÍ A SELEKCE

Spojování tabulek (join) je pravděpodobně nejčastější konstrukce dotazů v prostředí SQL.

```
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
```


P01_OKRES
ID NUMBER(4) <pk>
NAZEV VARCHAR2(64)
ID = ID_OKRES
P01_KAT_UZ
ID NUMBER(5) <pk>
NAZEV VARCHAR2(64)
ID_OBEC NUMBER(5) <fk>
ID = ID_OBEC
P01_OBEC
ID NUMBER(5) <pk>
NAZEV VARCHAR2(64)
ID_OKRES NUMBER(5) <fk>
ID = ADRESA_OKRES
ID = ADRESA_OBEC
ID = SJM_PARTNER2
P01_OPSUB
ID NUMBER <pk>
ICO NUMBER(8)
RC NUMBER(10)
SJM_PARTNER1 NUMBER <fk4>
SJM_PARTNER2 NUMBER <fk5>
PRIJMENI VARCHAR2(128)
JMENO VARCHAR2(64)
TITUL_PRED VARCHAR2(16)
TITUL_ZA VARCHAR2(16)
ADRESA_OKRES NUMBER(4) <fk1>
ADRESA_OBEC NUMBER(5) <fk2>
ADRESA_COBCE NUMBER(5) <fk3>
ADRESA_CPOP NUMBER(5)
ID = SJM_PARTNER1
ID = ID_OBEC
ID = ADRESA_COBCE
P01_CAST_OBCE
ID NUMBER(5) <pk>
NAZEV VARCHAR2(64)
ID_OBEC NUMBER(5) <fk>
ID = ID_KU
ID = ID_OPSUB
P01_VLASTNI
ID_KU NUMBER(6) <pk,fk1>
ID_LV NUMBER(5) <pk,fk1>
ID_OPSUB NUMBER <pk,fk2>
PODIL_CITATEL NUMBER
PODIL_JMENOVATEL NUMBER
ID = ID_CAST_OBCE
ID_KU = ID_KU
ID_LV = ID_LV
P01_PARCELA
ID_KU NUMBER(6) <pk,fk1,fk
PARC_TYP NUMBER(1) <pk>
PARC_CIS NUMBER(4) <pk>
PAR_POD NUMBER(3) <pk>
DRUH_POZ NUMBER(2)
NEM_VYUZ NUMBER(3)
VYMERA NUMBER(9)
CIS_LV NUMBER(5) <fk2>
ID_KU = ID_KU
ID_LV = CIS_LV
P01_LV
ID_KU NUMBER(6) <pk>
ID_LV NUMBER(5) <pk>
ID_KU = LV_KU
ID_LV = CIS_LV
P01_BUDOVA
TYP_CISLA VARCHAR2(1) <pk>
CISLO_DOMOVNI NUMBER(5) <pk>
ID_CAST_OBCE NUMBER(5) <pk,fk
LV_KU NUMBER(6) <fk2>
CIS_LV NUMBER(5) <fk2>

7.10 TECHNIKY SPOJOVÁNÍ 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.

7.11 OUTER JOIN – VNĚJŠÍ SPOJENÍ TABULEK

Outer join (vnější spojení) dvou tabulek vrací povinně všechny řádky jedné z tabulek, i když nevyhovují podmínce. Sloupce, resp. výrazy v příkazu select z ostatních tabulek jsou v těchto případech vráceny jako NULL. Provádí se (+) operátorem ve `WHERE` klausuli, nebo frází `left/right outer join`

```
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
```

Vrátí:

I1	I2
2	2
3	3

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

nebo

```
SELECT t1.i i1,t2.i i2
FROM
 t1 left outer join t2 on t1.i=t2.i
```

Vrátí:

I1	I2
1	null
2	2
3	3

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

nebo

```
SELECT t1.i i1,t2.i i2
FROM
 t1 right outer join t2 on t1.i=t2.i
```

Vrátí:

I1	I2
2	2
3	3
null	4

7.12 ULOŽENÉ PŘÍKAZY SELECT, PŘÍKAZ CREATE VIEW

Objekt `VIEW` je uložený příkaz `select`.

`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 primární klíč je zároveň klíčem ve 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 odvolávat na sloupce náležící non-key-preserved tabulce, všechny vkládané sloupce náleží právě jedné key-preserved tabulce

Příklad CREATE 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;
```

□

Cvičení 5 DML příkazy nad VIEW

Projděte si **Příklad** CREATE VIEW a určete:

a) Které sloupce z tohoto VIEW jdou vkládat?

b) Které sloupce z tohoto VIEW jdou měnit?

c) Lze z tohoto VIEW mazat (DELETE), co se stane při pokusu o DELETE?

Příklad řízení přístupu k části tabulky pomocí VIEW

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
  )
```

□

7.13 MATERIALIZOVANÉ POHLEDY, MATERIALIZED VIEW

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

7.14 COMMON TABLE EXPRESSION (CTE)

Základní konstrukce CTE

Tabulkové výrazy v jazyku SQL slouží pro zpřehlednění dotazu. Jedná se o dočasný pojmenovaný výsledek dotazu. Základní syntax CTE je:

```
WITH temp_result(a,b,c) AS
(
  SELECT a,b,c
  FROM T1
)
SELECT a,c
FROM temp_result
```

Používá se v případech, když potřebujeme „zpřehlednit“ SQL dotaz, pokud se nám zdá, že formulace SQL dotazu pomocí CTE konstrukce je „čitelnější“.

Použitím CTE ovlivňujeme exekuční plán vykonání dotazu. V některých případech to může být užitečné, ale pozor, v některých situacích může být exekuční plán méně efektivní.

Výše uvedená základní konstrukce CTE **NEROZŠIŘUJE** dotazovací sílu relační algebry.

Rekurzivní CTE

Základní SQL jazyk (tj. implementace relační algebry) nepodporuje rekurzívní konstrukce dotazů. Jsou v něm tedy neřešitelné například běžné úlohy z teorie grafů typu „vyber podstrom grafu“, které jsou v datových modelech velmi užitečné, zejména v hierarchických strukturách. Přestavme si tabulku zaměstnanců, které má cizí klíč do sebe sama, s významem „nadřízená osoba“. Bohužel v relační algebře nelze sestavit vzorec, který by pro daného zaměstnance vybral všechny jeho podřízené. Částečnou nápravu absence rekurze v SQL jazyku představuje CTE s rekurzí. Základní schéma je následující:

```
WITH RECURSIVE temp_result (column1, column2, ...)
AS(
   --Inicializační část
   SELECT …
   UNION [ALL]
    --Rekurzivní část
    SELECT … FROM …, temp_result WHERE
)
SELECT * FROM temp_result;
```

Vyhodnocení rekurzivního CTE dotazu

1) Vykonání inicializační části dotazu, výsledek do R_0.
2) Vykonání rekurzivní části dotazu se vstupem R_i, výsledek do R_{i+1}.
3) Opakování 2) dokud je $R_{i+1} - R_i \neq \emptyset$.
4) Výsledek **`UNION[ALL]`** $(R_0 \dots R_n)$

Příklad – transitivní uzávěr relace:

Mějme relaci $tr \subseteq N \times N$ reprezentovanou tabulkou:

```
CREATE TABLE tr
(
 a int,
 b int
);
```

Následující CTE dotaz vytvoří transitivní uzávěr relace tr.

```
WITH recursive trcl (a,b) AS
(
 SELECT a,b FROM tr
  UNION
 SELECT trcl.a, tr.b FROM trcl, tr WHERE trcl.b=tr.a
)
SELECT * FROM trcl ORDER BY a,b
```

□

Doporučuji čtenáři, aby si tuto příkaz prostudoval a zdůvodnil, že tvrzení je pravdivé.
Výše uvedená rekurzivní konstrukce CTE je **ROZŠÍŘENÍM** relační algebry.

7.15 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 standardně (B^+-tree) 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`.

Databáze nepodporují vícenásobnou dědičnost, dědit lze pouze z jednoho nadřazeného typu.

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. Cenou je však zvýšená zátěž databázového serveru a řádově větší přenos dat v XML formátu.

7.16 DATOVÝ TYP JSON

Jiným způsobem, jak v databázové tabulce skladovat komplexní strukturované objekty je použití datového typu **J**ava**S**cript **O**bject **N**otation , v databázích typy `json` resp. `jsonb`. JSON je textový formát pro přenos a reprezentaci komplexních strukturovaných objektů. Je založený na syntaxi jazyka Java Script, který je hojně používaný ve webových aplikacích. Při použití těchto datových typů tedy odpadají konverze z/do formátu, ve kterém lze v prostředí webových aplikací komfortně pracovat. Oblíbenost tohoto formátu je způsobena jeho extrémní jednoduchostí a čitelností. V tomto textu nebudu uvádět jeho formální definici, studenti, kteří se s JSONem dosud nesetkali si mohou doplnit znalosti z např:

https://www.w3schools.com/whatis/whatis_json.asp

Příklad Liniová geometrie v JSON

```
{
 "type": "LineString",
 "coordinates":
  [[102.0, 0.0],
   [103.0, 1.0],
   [104.0, 0.0],
   [105.0, 1.0]]
}
```

□

Standard SQL:2023 poskytuje užitečné operátory pro manipulaci s JSON daty například:

@> - obsahuje

(úplný seznam funkcí a operátorů si čtenář může dohledat v dokumentaci použitého DB stroje)

Příklad Tabulka s typem `jsonb`

```
create table geometry_js
(
 id bigserial,
 geometry_js jsonb constraint geometry_js_chk01
 check
 (
  geometry_js@>'{"type":"Point"}' or
  geometry_js@>'{"type":"LineString"}' or
  geometry_js@>'{"type":"Polygon"}'
 )
);
```

□

Vlastnost (objekt) je odkazována v DML „**->**“, n-tý prvek pole je odkazován „**->>**“ například:

```
create index geometry_js_i1 on geometry_js((geometry_js->'type'));

insert into geometry_js (geometry_js)
values
(
 '{"type": "LineString",
 "coordinates":
 [[102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]]}'
);

select
 geometry_js->'type' as type,
 geometry_js->'coordinates'->>0 as coords
from geometry_js
where geometry_js->'type'='"LineString"'
```

Při použití datového typu JSON bychom měli mít jasno, zda objekty v JSON sloupci neskrývají položky, které by měly uplatnění v definici datového schématu, například primární/cizí klíče.

8 PROCEDURÁLNÍ JAZYKY RDBMS - PL/SQL

8.1 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;
```

8.2 DEKLARACE

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

Datový typ *tabulka*%ROWTYPE odpovídá struktuře tabulky.

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

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

8.3 PŘIŘAZENÍ, VÝRAZY:

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

8.4 DML A KURSORY

INTO fráze:

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

Kursor:

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

Ovládání kursorů:

Analogie k souborovému přístupu:

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

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;
```

8.5 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;
```

8.6 ASYNCHRONNÍ OŠETŘENÍ CHYB

```
begin
  select   …… into a,b,c;

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

8.7 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;
/
```

8.8 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;
```

8.9 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;
```

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]
```

9 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;
```

Příklad: trigger hlídající akce nad tabulkou:

```
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;
```

10 HOSTITELSKÉ NADSTAVBY SQL

10.1 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();
}
```

10.2 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);
…
```

10.3 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);
}
```

11 NORMALIZACE A SQL

11.1 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").

11.2 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 !!!

11.3 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:

Změna počtu obyvatel v tabulce OBEC vyvolá trigger, který přepočítá POCET_OBYV_OKRES v tabulce OKRES.

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!!!

11.4 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'))
 )))
```