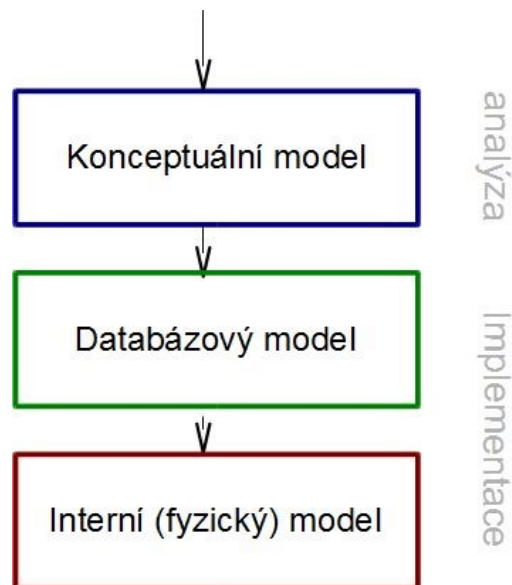


### 1) Modelování databázových systémů, konceptuální modelování, datová analýza, funkční analýza, nástroje a modely

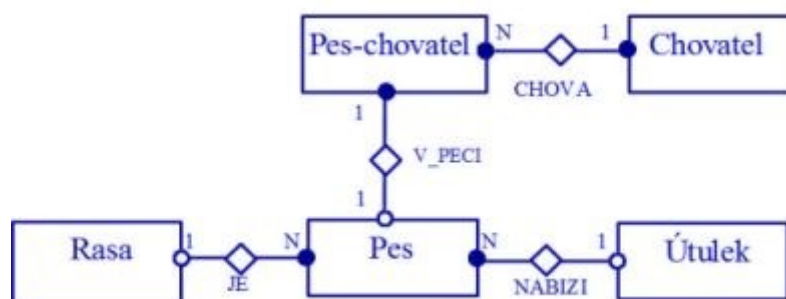
#### Modelování databázových systémů



Databázový systém můžeme modelovat třemi datovými modely.

Ve fázi analýzy se používá [konceptuální model](#), který modeluje realitu na logickou úroveň databáze. Konceptuální model je výsledkem [datové analýzy](#) a je nezávislý na konkrétní implementaci.

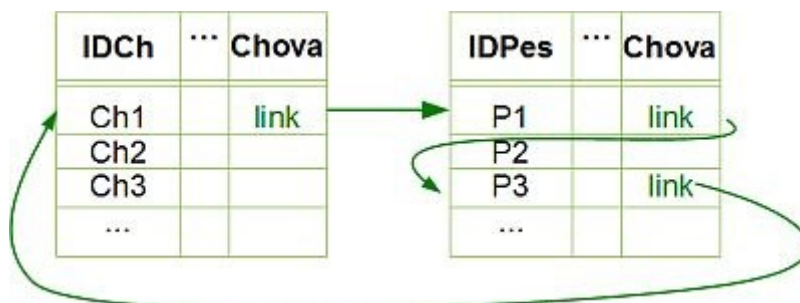
V implementační fázi si pak pomáháme **databázovými modely**, kde modelujeme vazby a vztahy (realitu) na konkrétní tabulky (obecně SŘBD). Databázový model můžeme dále dělit na **relační a síťový model**. Fyzickým uložením dat na paměťové médium se zabývá **interní model**.



Ukázka ER diagramu část konceptuálního datového modelu

IDPes	...	...	IDPes	IDCh	IDCh	...	...
P1			P1	Ch1	Ch1		
P2			P2	Ch1	Ch2		
P3			P3	Ch3	Ch3		
...			...	...	...		

Ukázka relačního datového modelu



## Datová analýza a konceptuální model

**Datová analýza** zkoumá objekty reálného světa, jejich vlastnosti a vztahy. Výsledkem datové analýzy je **konceptuální schéma**. V rámci datové analýzy zpracujeme zadání (specifikaci požadavků na IS):

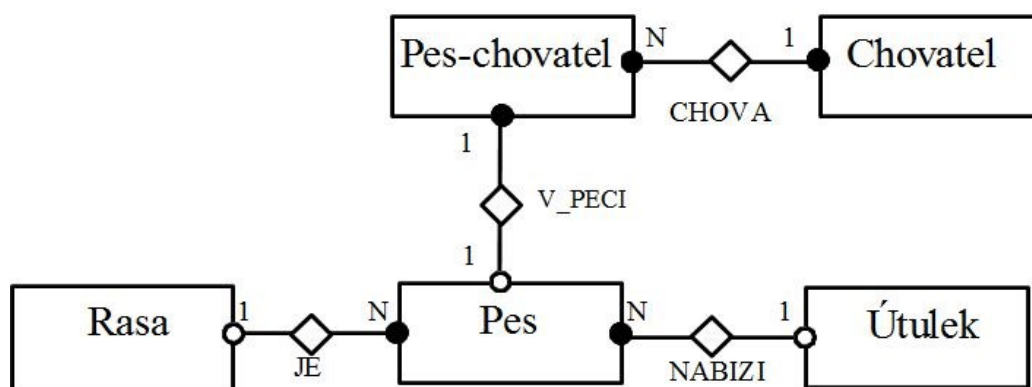
- podtrhneme podstatná jména = identifikujeme objekty
- podtrhneme slovesa = identifikujeme vazby mezi objekty
- najdeme vlastnosti a stavy nalezených objektu = identifikujeme atributy

Z takto získaných informací sestavíme konceptuální schéma. Konceptuální schéma se skládá z:

- ER Diagram
- Lineární zápis entit
- Lineární zápis vztahů
- Datového slovníku
- Popisu dalších IO (integritních omezení)

## ER Diagram

Grafické znázornění objektů a vztahů mezi nimi. Může mít několik podob v závislosti na používaném prostředí a detailnosti s jakou jej potřebujeme vypracovat. Atributy mohou být v grafu znázorněny ovály spojenými s objekty (obdélníky), vazba 1:N může být znázorněna "hráběmi" místo N, či celý diagram se může podobat třídnímu diagramu s atributy vepsanými do objektu.



## Lineární zápis entit a vztahů

Lineární zápisem popisujeme objekty, jejich vlastnosti a vztahy z pohledu implementačního. Lineárním zápisem entit jsou v podstatě definovány tabulky a jejich atributy včetně primárních a *cizích klíčů*.

Příklad lineárního zápisu entity: Pes (IDPes, jmeno, pohlavi, vek, CRasa, IDUtulek)

Příklad lineárního zápisu vztahů: NABIZI (Útulek, Pes) 1:N

## Datový slovník

Ukázka datového slovníku pro tabulku Pes

Pes	Typ	Délka	Klíč	NOT NULL	IO
IDPes	int	8	primarni	ano	pravidla pro tvar čipového čísla
jmeno	varchar	50			
vek	int	2			
CRasa	int	2	sekundarni		
...					

Podrobný rozpis jednotlivých atributů. Tabulka obsahuje typ atributů, velikost, integritní omezení, apod.

**Integritní omezení** obsahují další specifikaci atributů, které není dáno typem a délkou. Nejčastěji se týká formátu atributu. Např. login se skládá z třech čísel a třech písmen, nebo rodné číslo je složeno z data narození, apod.

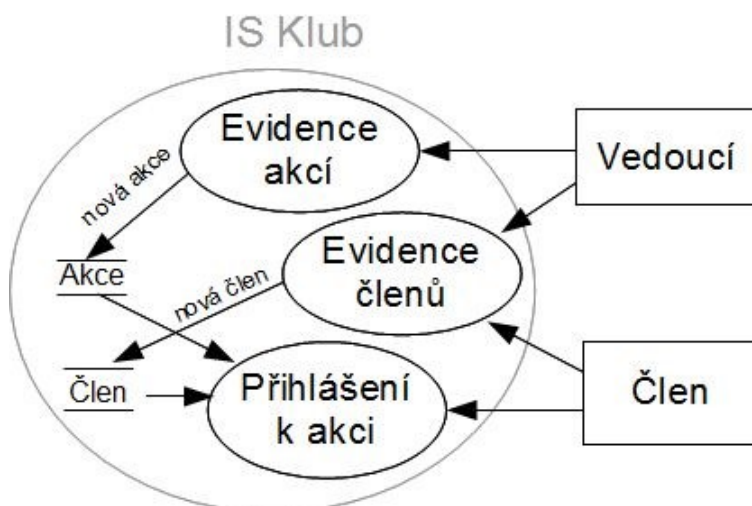
## Další integritní omezení (IO)

Konceptuální schéma obsahuje také soupis dalších IO, které se týkají entit (tabulek) a vazeb mezi nimi. Může jít například o omezení vícenásobné vazby, vyjádření hierarchie mezi entitama, apod.

## Funkční analýza

Zatímco datová analýza se zabývá strukturou obsahové části systému (strukturou databaze), funkční analýza řeší funkce systému. Funkční analýza tedy vyhodnocuje manipulaci s daty v systému. Skrze **DFD (Data Flow Diagramy)** analyzuje toky dat, základní funkce systému a aktéry, kteří se systémem pracují. Výstupem jsou pak **minispecifikace** - podrobné analýzy elementárních funkcí systému.

Diagram datových toků (DFD - Data Flow Diagram)



DFD je grafický nástroj pro modelování funkcí a vztahů v systému. Znázorňuje nejen procesy (funkce) a datové toky, ke kterým v systému dochází, ale definuje také hlavní aktéry a jejich omezení nad systémem. DFD diagram obsahuje tyto prvky: **aktér** (obdélník mimo systém), **proces** (kruh uvnitř systému), **datové toky** (šipky) a **paměť** (viz. obr. Akce a Člen).

DFD diagramy lze zakreslit v různých úrovních. Např. proces Evidence akcí na obrázku lze dále rozkreslit dalším DFD, obsahující procesy vytvoření a editace akce. DFD nejvyšší úrovně se nazývá **kontextový diagram**. Znázorňuje pouze práci aktérů se systémem jako celkem. Systém v kontextovém diagramu vystupuje jako černá skříňka a v diagramu tedy nejsou použity prvky procesu a paměti.

## Minispecifikace

Podrobná analýza algoritmů elementárních funkcí, tedy procesů nejnižší úrovně DFD. Při tvorbě minispecifikace se používá přirozený jazyk, tak aby byla minispecifikace nezávislá na implementaci v konkrétním prostředí. Měla by však být strukturovaná a používat standardní programové struktury (větvení, cykly,...). Pravidla minispecifikací:

- Existuje jedna pro každou elementární funkci
- popisuje algoritmus, jak jsou datové toky do funkce vstupující transformovány na výstupní datové toky
- Nezavádí redundandní popisy, nevyjadřuje znovu, co je popsáno v DFD nebo v datovém slovníku (domény atributů, klíče, ...)
- Množina výrazů pro popis by měla být jednoduchá a nepříliš rozsáhlá, má používat standardní vyjadřování.

IF všechny výrobky v objednávce jsou rezervovány,  
THEN pošli objednávku k dalšímu zpracování oddělení prodeje.  
OTHERWISE,

FOR EVERY nezarezervovaný výrobek v objednávce DO:

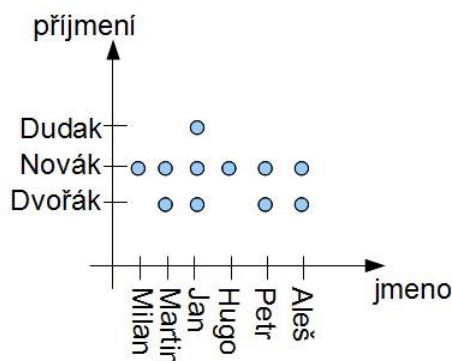
    Zkus najít volný výrobek a rezervuj ho.

    IF výrobek není na skladě,

    THEN informuj správce.

## 2) Relační datový model, SQL; funkční závislosti, dekompozice a normální formy.

### Relační datový model



Tabulka s dvěma atributy znázorněná jako relace

Relační datový model je způsob uchování dat v tabulkách. Relační se mu říká proto, že se tabulka se definuje přes relaci.

**Relace** je tedy v podstatě tabulka. Definována je jako podmnožina kartézského součinu domén. *Relace na obrázku je tedy podmnožina součinu množin {Dudak, Novák, Dvořák} x {Milan, Martin, Jan, ..., Aleš}*

Na rozdíl od [matematické relace](#) se ta databázová mění v čase (přidáváním a odebíráním prvků relace). Kromě základních množinových operací se u databázové relace setkáme s operací **selekce** - výběr řádků a **projekce** - výběr sloupců.

jméno	příjmení
Jan	Dvořák
Milan	Novák
Martin	Dvořák
Jan	Dudák
Hugo	Novák
...	...

Relace z předchozího obrázku zobrazena tabulkou

**Doména** je množina všech hodnot, kterých může nabývat atribut. Jinak řečeno obor hodnot atributu. V praxi je doména dána integritním omezením (IO).

*Doména atributu Příjmení z obrázků je množina {Dudak, Novák, Dvořák}. \*pozn.*

**Atribut** je vlastnost entity. Z pohledu tabulky jde o sloupec.

**Relační schéma** můžeme chápat jako strukturu tabulky (atributy a domény).

**Příklad pro tabulku (relaci) Učitel:**

*Atributy:* ID, jméno, příjmení, funkce, kancelář

*Domény:*

D1 - tři písmena z příjmení, tři cifry pořad. čísla

D2 - kalendář jmen

D3 - množina příjmení

D4 - množina funkcí (asistent, vědec, učitel,...)

D5 - A101, A102, ... A160

*Relační schéma:* Učitel (ID, jméno, příjmení, funkce, kancelář)

*Relace:* Učitel = {(nov001,lukas,novak,vědec,A135),(kom123,jan,komensky,učitel,A111),...}

Definováno je jako  $R(A,f)$ , kde  $A$  je množina atributů ( $A_1, A_2, \dots, A_n$ ) a funkce  $f(A_i) = D_i$  přiřazuje atributu doménu.

Vlastnosti relačního datového modelu

Z definice relace vyplývají tyto jejich tabulkové vlastnosti:

- Homogenita (stejnorodost) sloupců (prvky domény)
- každý údaj (hodnota atributu ve sloupci) je atomickou položkou
- na pořadí řádků a sloupců nezáleží (jsou to množiny prvků/atributů)
- každý řádek tabulky je jednoznačně identifikovatelný hodnotami jednoho nebo několika atributů (primárního klíče)

Vazby v relačním modelu

Obecně se vazby v relačním modelu realizují pomocí další relace (tabulky). Jedná se o tzv. **vazební tabulku**. Ta **obsahuje** ty atributy relací (tabulek, které se vazby účastní), které jednoznačně identifikují jejich entity - **primární klíče**. Obsahuje-li tabulka atribut, který slouží jako primární klíč v jiné tabulce, pak obsahuje **cizí klíč**. Vazební tabulka tedy obsahuje cizí klíče.

Na obrázku dole máme zobrazenou relaci Učitel s primárním klíčem *idu* a relaci Předmět s primárním klíčem *cp*. K vyjádření vztahu *Učitel UČÍ Předmět* byla vytvořena nová relace Učí, která obsahuje dva cizí klíče (*idu* odkazující na učitele a *cp* odkazující na entitu předmětu). Nyní tedy můžeme přes vazební tabulku pospojovat učitele s předměty, podle toho kdo co učí.

A proč jsme do tabulky Učitel nepřidali jen atribut předmět? Jednoduše proto, že učitel může učit více předmětů. Mezi učitelem a předmětem je vazba M:N. Takže ani kdybychom přidali do tabulky Předmět atribut učitel, bychom tento vztah nevyřešili (předmět může být učen více učiteli). Pro vztah 1:N by to však šlo a v praxi se to tak i dělá. Takže **vazební tabulka je nutná jen k realizaci vztahů M:N**.

Učitel			Učí		Předmět	
idu	jméno	příjmení	idu	cp	cp	nazev
dvo01	Jan	Dvořák	dvo01	3	1	matematika
kov01	Marie	Kovářová	dvo01	1	2	anglický j.
kov02	Martin	Kovadlina	kov01	2	3	fyzika
chy01	Jana	Chtrá	chy01	1	4	biologie
mal01	Libuše	Malinová	mal01	5	5	český j.

Ukázka vazební tabulky pro vztah Učí mezi tabulkami Učitel a Předmět. Vztah je M:N, tedy že jeden učitel může učit N předmětů a jeden předmět může být učen M učiteli.

## SQL

SQL (Strucrured Query Language) je **relační jazyk** založen na predikátovém kalkulu. Na rozdíl od jazyků založených na relační algebře, kde se dotaz zadává algoritmem, tyto jazyky se soustředí na to co se má hledat, ne jak.

SQL obsahuje příkazy pro:

- vytváření a modifikaci relačního schématu (tabulek, databází) - CREATE, ALTER (MODIFY, ADD, DROP), DROP = vytvoř, uprav, smaž
- modifikací dat - INSERT, UPDATE, DELETE = vlož, uprav, smaž
- vyhledávání v relacích - SELECT, ORDER BY, GROUP BY, JOIN = vyhledej, seřaď, shlukuj, spoj
- transakce - COMMIT, ROLBACK = úspěšně provedená transakce, save-point uvnitř transakce ke kterému se dá vrátit byla-li transakce přerušena
- další, pro podmínky, logické operatory,... (WHERE, LIKE, BETWEEN, IN, IS NULL, DISTINCT/UNIQUE, JOIN, INNER JOIN, OUTER JOIN, EXISTS, HAVING, COUNT, VIEW, INDEX,...)

```
CREATE TABLE clovek(id char(5) NOT NULL, jmeno varchar(50), prijmeni varchar(50), telefon varchar(15), PRIMARY KEY (id));
```

```
INSERT INTO clovek VALUES( 'nj001', 'Jan', 'Novotný', '777111222');
```

```
SELECT telefon FROM clovek WHERE prijmeni = "Novotný";
```

## Relační jazyk

Jazyky pro formulaci požadavků na výběr dat z relační databáze (dotazovací jazyky) se dělí do dvou skupin:

1. **jazyky založené na relační algebře**, kde jsou výběrové požadavky vyjádřeny jako posloupnost speciálních operací prováděných nad daty; dotaz je tedy zadán algoritmem, jak vyhledat požadované informace;

2. **jazyky založené na predikátovém kalkulu**, které požadavky na výběr zadávají jako predikát charakterizující vybranou relaci; je úlohou překladače jazyka nalézt odpovídající algoritmus; tyto jazyky se dále dělí na:
- n-ticové relační kalkuly
  - doménové relační kalkuly

## Relační algebra

Relační algebra je velmi silný dotazovací jazyk vysoké úrovně. Nepracuje s jednotlivými entitami relací, ale s celými relacemi. Operátory relační algebry se aplikují na relace, výsledkem jsou opět relace. Protože relace jsou množiny, přirozenými prostředky pro manipulaci s relacemi budou množinové operace.

I když relační algebra v této podobě není vždy implementována v jazycích SŘBD, je její zvládnutí nutnou podmínkou pro správnost manipulací s relacemi. I složitější dotazy jazyka SQL, který je deskriptivním dotazovacím jazykem, mohou být bez zkušeností s relační algebrou problematické.

### Základní operace relační algebry:

Jsou dány relace R a S.

- Množinové operace
  - sjednocení relací téhož stupně  $R \cup S = \{x \mid x \in R \vee x \in S\}$
  - průnik relací  $R \cap S = \{x \mid x \in R \wedge x \in S\}$
  - rozdíl relací  $R - S = \{x \mid x \in R \wedge x \notin S\}$
  - kartézský součin relace R stupně m a relace S stupně n  $R \times S = \{rs \mid r \in R \wedge s \in S\}$ , kde  $rs = \{r_1, \dots, r_m, s_1, \dots, s_n\}$
- Další operace relační:
  - **projekce** (výběr sloupců) relace R
  - **selekce** (výběr řádků) z relace R podle podmínky P
  - spojení relací R s atributy A a S s atributy B (**join**)
  - přirozené spojení relací R(A) a S(B)

## N-ticový relační kalkul

- Dr. Codd definoval n-ticový relační kalkul pro RDM jazyk matematické logiky - predikátový počet je využit pro výběr informací z relační databáze
- název odvozen z oboru hodnot jeho proměnných - relace je množina prvků = n-tic
- je základem pro jazyk typu SQL
- Syntaxe je přizpůsobena programovacímu jazyku

...matematické vyjádření  $\{x \mid F(x)\}$  ...nahradíme zápisem  $x \text{ WHERE } F(x)$

- kde  $x$  je proměnná pro hledané n-tice (struktura relace),  $F(x)$  je podmínka kterou má  $x$  splňovat (výběr prvků relace)

## Funkční závislosti

Funkční závislost je v databázi vztah mezi atributy takový, že máme-li **atribut Y je funkčně závislý na atributu X** píšeme  **$X \rightarrow Y$** , pak se nemůže stát, aby dva řádky mající stejnou hodnotu atributu X měly různou hodnotu Y.

Příklad:

Atribut 'datum narození' je **funkčně závislý** na atributu 'rodné číslo'.

Nemůže se stát, že u záznamů se stejnými rodnými čísly bude různé datum narození.

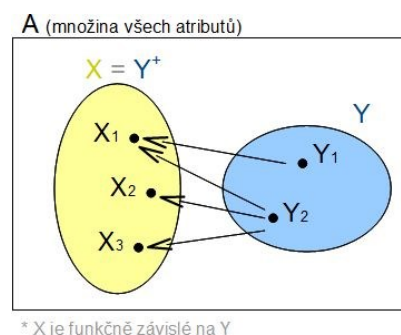
Pomocí funkčních závislostí můžeme automaticky navrhnout schéma databáze a předejít problémům jako je **redundance**, **nekonzistence** databáze, zablokování při vkládání záznamů, apod.

Postup datové analýzy s automatickým navržením struktury databáze je následující:

- ze zadání zjistím, co je třeba v databázi evidovat (objekty, atributy, vztahy, integritní omezení,...)
- funkční analýzou určím závislosti (vztahy) mezi atributy
- vytvořím jednu obrovskou tabulku (relaci), obsahující všechny atributy
- pomocí funkčních závislostí provedu dekompozici (rozbití) velké relace na menší, které vytvoří výsledné schéma databáze.

## Armstrongovy axiomy

Armstrongovy axiomy jsou odvozovací pravidla funkčních závislostí. Pomocí axiomu získáme uzávěr funkčních závislostí spolu s klíči schémat.





**Uzávěr  $X^+$**  je množina všech atributů funkčně závislá na attributech množiny  $X$ .

Armstrongovy axiomy -  $A$  = množina všech atributů,  $F$  = množina všech funkčních závislostí,  $XY = XUY$  ( $X$  nebo  $Y$ ):

- **Reflexivita** - je-li  $Y \subset X \subset A$ , pak  $X \rightarrow Y$
- **Tranzitivita** - pokud je  $X \rightarrow Y$  a  $Y \rightarrow Z$ , pak  $X \rightarrow Z$
- **Pseudotranzitivita** - pokud je  $X \rightarrow Y$  a  $WY \rightarrow Z$ , pak  $XW \rightarrow Z$
- **Sjednocení** - pokud je  $X \rightarrow Y$  a  $X \rightarrow Z$ , pak  $X \rightarrow YZ$
- **Dekompozice** - pokud je  $X \rightarrow YZ$ , pak  $X \rightarrow Y$  a  $X \rightarrow Z$
- **Rozšíření** - pokud je  $X \rightarrow Y$  a  $Z \subset A$ , pak  $XZ \rightarrow YZ$
- **Zúžení** - pokud je  $X \rightarrow Y$  a  $Z \subset Y$ , pak  $X \rightarrow Z$

**Určení klíče pomocí funkčních závislostí:**

Ze zadání jsme určili atributy  $A = \{\text{učitel}, \text{jméno}, \text{příjmení}, \text{email}, \text{předmět}, \text{název}, \text{kredity}, \text{místnost}, \text{čas}\}$  a funkční závislosti  $F$ :

*učitel  $\rightarrow$  jméno, příjmení, email*

*předmět  $\rightarrow$  název, kredity*

*místnost, čas  $\rightarrow$  učitel, předmět*

---

Rozšíření

*učitel, **místnost, čas**  $\rightarrow$  jméno, příjmení, email, **místnost, čas***

*předmět  $\rightarrow$  název, kredity*

*místnost, čas  $\rightarrow$  učitel, předmět*

---

Dekompozice

*učitel, **místnost, čas**  $\rightarrow$  jméno, příjmení, email, místnost, čas, **učitel, předmět***

*předmět  $\rightarrow$  název, kredity*

---

Dekompozice

*učitel, místnost, čas  $\rightarrow$  jméno, příjmení, email, místnost, čas, učitel, předmět, **název, kredity***

Atributy *učitel, místnost, čas* je klíč schématu velké relace. V dalším kroku je třeba provést dekompozici a tuto velkou relaci rozbít na menší relace.

## Dekompozice

Na počátku máme celé relační schéma se všemi atributy, snažíme se od tohoto schématu odebírat funkční závislosti a tvořit schémata nová. Dekompozice relačního schématu je rozklad relačního schématu na menší relač. sch. (rozloží velkou tabulku na menší) aniž by došlo k narušení redundance databáze. Mezi základní vlastnosti dekompozice patří - zachování informace a zachování funkčních závislostí.

---

Definice dekompozice relačního schématu :

Dekompozice relačního schématu  $R(A, f)$  je množina relačních  $RO = \{R_1(A_1, f_1), R_2(A_2, f_2), \dots\}$ , kde  $A = A_1 \cup A_2 \cup \dots$

---

Binární dekompozice, kterou budeme dále řešit je rozklad jednoho relačního schématu na dvě. Obecná dekompozice vznikne postupnou aplikací binárních.

Zachování informace

Je jedna ze základních vlastností dekompozice. K ověření že při rozkladu tabulky, nedošlo ke ztrátě informace slouží následující věta:

Je-li  $RO = \{R_1(B), R_2(C)\}$  dekompozice relačního schématu  $R(B \cup C)$ , pak musí platit

$$B \cap C \rightarrow B - C \text{ nebo } C \cap B \rightarrow C - B$$

---

Je rozklad dekompozice?

Je dán rozklad  $RO = \{R_1(U, J, \check{R}, E), R_2(P, U, N, K, M, \check{C})\}$  dekompozice pro  $R(\text{Učitel}, \text{Jméno}, \text{Příjmení}, \text{Email}, \text{Předmět}, \text{Název}, \text{Kredity}, \text{Mítnost}, \text{Čas})$  s klíčem UMČ a funkčními závislostmi  $U \rightarrow J\check{R}E, P \rightarrow NK, M\check{C} \rightarrow PU$  ?

- $UJ\check{R}E \cap PUNKM\check{C} = U$
- $UJ\check{R}E - PUNKM\check{C} = J\check{R}E$
- $PUNKM\check{C} - UJ\check{R}E = PNKM\check{C}$

Mezi funkční závislostmi musí existovat  $U \rightarrow J\check{R}E$  nebo  $U \rightarrow PNKM\check{C}$ , což existuje, takže uvedený rozklad je dekompozice.

Zachování funkčních závislostí

Pro náš příklad tedy zkontroluju, zda jsou všechny 3 funkční závislosti obsažené v jedné z rozložených tabulek  $R_1$  a  $R_2$ :

- $U \rightarrow J\check{R}E$  je obsažena v  $R_1$
- $P \rightarrow NK$  je obsažena v  $R_2$
- $M\check{C} \rightarrow PU$  je obsažena v  $R_2$

Dekompozice  $RO = \{R_1(B), R_2(C)\}$  zachovává množinu funkčních závislostí  $F$ , jestliže množina závislostí  $(F[B] \cup F[C])$  logicky implikuje závislosti v  $F$ , tedy

$$F^+ = (F[B] \cup F[C])^+$$

V praxi to znamená, že zkontroluju, zda každá funkční závislost (všechny atributy před šipkou i za šipkou) je obsažena alespoň v jedné z rozložených tabulek.

## Normální formy

Normální formy relací (NF) prozrazují jak pěkně je databáze navržena (čím vyšší NF tím lepší :).

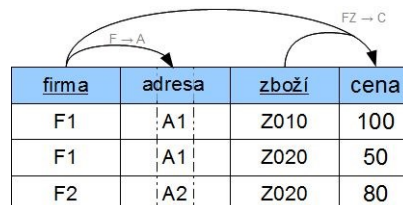
Seřazeny sestupně jsou definovány takto:

1. **První normální forma** definuje tabulky, které obsahují pouze atomické atributy.

Žádné složené atributy. Např. v jednom atributu je Jméno i Příjmení.

2. **Druhá normální forma** je 1NF + každý sekundární atribut je plně závislý na každém klíči schématu.

Tudíž nesmí existovat závislost na podklíči. Když  $AB \rightarrow CD$  ...pak nesmí být  $B \rightarrow C$ . Atribut adresa není závislý na všech klíčích FZ, ale pouze na F.



<u>firma</u>	adresa	zboží	cena
F1	A1	Z010	100
F1	A1	Z020	50
F2	A2	Z020	80

3. **Třetí normální forma** je 2NF + žádný sekundární atribut není tranzitivně závislý na žádném klíči schématu. Nesmí existovat závislosti mezi sekundárními atributy (Model auta -> značka auta).

Když  $AB \rightarrow CD$  ...pak nesmí  $C \rightarrow D$

**Příklad porušení 3NF:**

Atribut počet obyvatel je tranzitivně závislý (přes atr. město) na klíči.



<u>firma</u>	město	obyvatel
F1	M1	100 000
F2	M1	100 000
F3	M2	8 000

4. **Boyce-Coddová normální forma** - je-li funkční závislost  $(X \rightarrow Y) \in F^+$  a  $Y \not\subseteq X$ , pak X obsahuje klíč schématu. Musí být závislost sekundárních atributů na primárních nikoli naopak.

BCNF splňuje požadavky 3NF a zároveň nesmí mít závislosti mezi atributy klíče.

Když  $AB \rightarrow CD$  ...pak nesmí  $C \rightarrow A$

### 3) Transakce, zotavení, log, ACID, operace COMMIT a ROLLBACK; problémy souběhu, řízení souběhu: zamykání, úroveň izolace v SQL

#### Transakce

**Transakce** je základní dále nedělitelná jednotka zpracování dat, která musí proběhnout buď celá, nebo (v případě že je přerušena) obnovit původní stav databáze a spustit se znovu. Pokud k tomu nedojde, je ohrožena konzistence databáze.

## Vlastnosti ACID

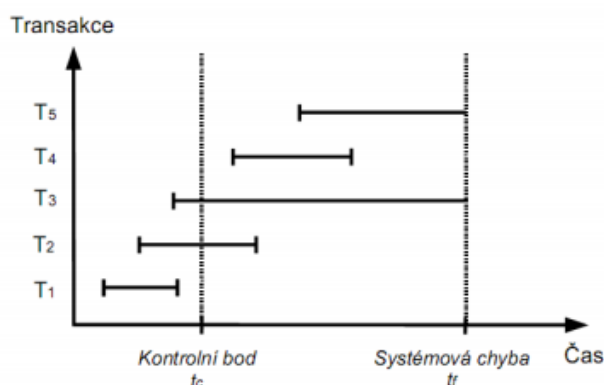
Transakce musí splňovat vlastnosti **ACID**:

- **A = Atomičnost** - transakce musí být atomická => proběhnou buď všechny operace transakce, nebo žádná.
- **C = Korektnost** - transakce převádí data z jednoho konkrétního stavu do druhého. Během transakce, nemusí být korektnost dodržena.
- **I = Izolovanost** - transakce se nesmí ovlivňovat - změny provedené jednou transakcí nesmí ovlivnit průběh druhých. Zajišťují zámky.
- **D = Trvalost** - po potvrzení transakce (COMMIT) jsou změny které transakce trvalé a zůstávají v databázi i po restartu.

## Kontrolní body

Kontrolní body jsou vytvářeny např. po určitém počtu záznamů, které byly zapsány do logu a zahrnují:

- zápis obsahu vyrovnávací paměti na disk,
- zápis záznamu o kontrolním bodu do logu



- Po restartu systému musí být transakce typu  $T_3$  a  $T_5$  zrušeny (undo).
- Transakce typu  $T_2$  a  $T_4$  musí být přepracovány (redo).
- Jelikož změny provedené transakcí  $T_1$  byly provedeny p kontrolním bodem  $t_c$ , tuto transakci při zotavení vůbec neuvažujeme

## Zabezpečení transakcí

Aby nedocházelo k narušení konzistence databáze vlivem přerušených transakcí, jak ukazuje příklad, je třeba ošetřit tak, aby buď proběhly celé, nebo obnovily původní stav databáze a spustily se znovu.

## Metoda zpožděné aktualizace

Zkopíruje si data z databáze do systémového logu souboru a tyto data zpracovává. Pokud dojde k přerušení transakce v tomto momentu, nic se nestane, neboť obsah databáze nebyl změněn. Znovu se tedy načtou data z databáze do logu a znovu se začnou zpracovávat.

Jsou-li data zpracována, je výsledek uložen v logu (REDO logu) a začne se přenášet do databáze. Dojde-li k přerušení v této chvíli. Po nahození systému se spustí **operace REDO**, která znovu začne přenášet konzistentní výsledky z logů do databáze.

## Metoda přímé aktualizace

Mění obsah databáze během výpočtů. Před zahájením si však do systémových log souboru (UNDO logu) zálohuje původní hodnoty databáze. V případě přerušení se pak z logu obnoví původní stav databáze. Operace, která po restartu systému, obnoví původní stav databáze z logů se nazývá **operace UNDO**.

## Metoda stínového stránkování

Ucet	
ID	stav
U1	200
U2	200
U3	400
...	...
U1	300
U2	200

Před zahájením transakce si původní data zálohuje do stínové tabulky (stínové stránky tabulky / stínového bloku). Pokud je transakce přerušena, obnoví se původní data ze stínové tabulky. Pokud transakce proběhla úspěšně smaže se (uvolní se pro další použití) záloha ve stínové tabulce.

Na obrázku můžeme vidět tabulku Ucet z předchozího příkladu se stínovou stránkou označenou šedě. Pokud by takto vypadala tabulka po restartu systému, obnoví se původní hodnoty ze stínu (U1=300, U2=200) a transakce převodu stovky z U1 na U2, se spustí znovu. Po úspěšně dokončené transakci se hodnoty z šedé části odstraní.

## Zotavení a zálohování

Pod pojmem zotavení rozumíme zotavení databáze z nějaké chyby. Způsoby zotavení po přerušené transakci, jsme si popsali v předchozí kapitole. Jde o obnovu databáze s použitím systémových logů (UNDO log, REDO log) či stínových tabulek.

Může však nastat větší katastrofa (hackrovský útok typu DROP DATABASE, či přírodní katastrofa která odrovná fyzické úložiště dat) a my přijdeme o celou databázi, ne jen její konzistenci. Z toho důvodu je nutné databázi důkladně a pravidelně zálohovat. Zálohuje se vždy **databáze ke konkrétnímu času + systémové logy**, kde jsou zaznamenány změny, které se nad databázi od poslední zálohy prováděly. Z těchto dvou informací, je pak možné databázi zrekonstruovat.

Nekonzistence vzniklá při zálohování databáze:

Představme si že během transakce z prvního příkladu dojde k zálohování databáze:

čas 1: Transakce odečetla z účtu U1 100Kč

čas 2: Zálohování si uložilo U1 | 200

čas 3: Zálohování si uložilo U2 | 200

čas 4: Transakce přičetla na účet U2 100Kč

Než došlo k připsání peněz na účet U2 tabulka se zálohovala a v záloze teď chybí 100Kč.

Během zálohy databáze, se většinou zastaví transakce např. uzamčením databáze či tabulek, aby záloha proběhla v klidu celá a zamezilo se nekonzistenci zálohy viz. ukázka vpravo. Pokud není možné zastavit transakce, řeší se to rekonstrukcí stavu 'databáze v konkrétním čase' z log souboru. V log souborech pak musí být uvedeno i datum změny.

## Transakční analýza

Podporuje-li SŘBD transakce, je dobré do minispecifikací doplnit transakce. Respektive označit jejich začátek, konec, případně checkpointy. Tímto se zabývá transakční analýza.

V rámci transakční analýzy je třeba vyhledat v minispecifikacích transakce, většinou co jedna minispecifikace to jedna transakce. A do minispecifikace doplnit do nich příkazy pro zahájení, ukončení a restart transakce a příkazy pro zamykání a odemykání záznamu.

Příkazy transakční analýzy:

- **begin transaction** - pro označení začátku transakce
- **end transaction** - pro označení konce
- **COMMIT** - oznámení o úspěšném dokončení transakce a signál k finálnímu uložení výsledků transakce do databáze
- **ROLLBACK** - oznámení o neúspěšném provedení transakce a signál k navrácení změn provedených buď celou transakcí (návrat na začátek transakce) nebo od kontrolního bodu (návrat transakce k poslednímu **savepointu**).
- **LS(A)** - sdílený zámek objektu A (záznamu, tabulky, databáze), umožňuje ostatním číst, ale pro psaní zablokuje objekt A.
- **LX(A)** - exkluzivní zámek objektu A. Nepovoluje ostatním transakcím ani čtení z uzamčeného objektu A.
- **UN(A)** - odemčení objektu A.

## Řízení souběhu transakcí, zamykání

Jelikož nad jednou databází většinou pracuje mnoho uživatelů, musí být databáze schopna řešit velké množství požadavků a transakcí. Zde může docházet k různým problémům vlivem špatné koordinace jednotlivých transakcí. Je třeba se zabývat **řízením souběhu** požadavků.

Hlavním problémem, ke kterému dochází, je že si dvě transakce probíhající zároveň nad stejnými daty budou přepisovat data. Nejjednodušším řešením problému souběhu transakcí je spouštět transakce sériově, tedy jednu po druhé. Jenže to by bylo zdlouhavé a zbytečně by se během dlouhých přenosů dat nevyužíval procesor, který by mohl zpracovávat další transakce. Proto se transakce provádějí paralelně, ale tak aby byl splněn **požadavek na sériovost transakcí**, který říká, aby výsledek paralelního zpracování byl stejný jako při zpracování sériovém.

Požadavek sériovosti transakcí můžeme zajistit:

- **Precedenčním grafem** - graf transakcí kde uzly reprezentují transakce a orientované hrany mezi nimi se pak objeví, pokud pracují nad stejnými daty. Pokud graf obsahuje cykly, pak testované schéma paralelního zpracování nesplňuje požadavek sériovosti.
- **Zamykáním** části databází proti přepisu, když s ní zrovna transakce pracuje.

### TECHNIKY ŘÍZENÍ SOUBĚHU

V minulosti byla vyvinuta celá řada technik pro řízení souběhu, které se snaží řešit výše zmíněné problémy.

Nejznámější techniky:

- **Zamykání** (angl. **locking**) – používá většina DBMS. Někdy se říká pesimistický přístup k souběžnému zpracování: předpokládáme, že paralelní transakce se budou pravděpodobně navzájem ovlivňovat. Při uzamykání spravuje systém jednu kopii dat a jednotlivým transakcím přiděluje zámky.

- **Správa verzí** (angl. *multiversion*) - je další přístup pro řízení souběžného zpracování – optimistický přístup: předpokládáme, že paralelní transakce se ovlivňovat nebudou. Při správě verzí systém vytváří při aktualizaci kopie dat a systém sleduje, která z verzí má být viditelná pro ostatní transakce (v závislosti na úrovni izolace).
- **Časová razítka** (angl. timestamps)
- **Validace**

## Zamykání

Transakce si při práci s daty může pozamykat části databáze proti přepisu. Většinou si zamkne záznamy, které zpracovává, ale lze zamknout celou tabulku či dokonce databázi například při [zálohování databáze](#). Rozlišujeme dva typy zámeků:

- **Zámek pro sdílený přístup** - zamkne prvek pouze proti přepisu a ostatní transakce z něj mohou číst. V transakční analýze označujeme tento typ zámku LS(A)
- **Exkluzivní zámek** - zamkne prvek jak pro přepis, tak pro čtení. V trans. analýze označujeme LX(A).

## Problém uváznutí

Ukázka uváznutí

T1: zamkne záznam A - LX(A)

T2: zamkne záznam B - LX(B)

T1: chce zamknout B a čeká na uvolnění...

T2: chce zamknout A a čeká na uvolnění...

...a jestli neumřely tak tam čekají do dnes.

Zámky sice řeší problém vzájemného přepisu, ale způsobují další problém a tím je **problém uváznutí**. K tomu dochází když si transakce vzájemně zamknou záznamy s kterými potřebují a pak čekají až jim ta druhá ten jejich záznam odemkne.

Tento problém uváznutí se dá řešit několika způsoby:

- v transakční analýze tak že **na začátku transakce vše potřebné pozamykáme a na konci to odemkneme**. Zbytečně však během celé transakce blokuje záznamy, které nepotřebujeme a pak před zahájením dlouho čekáme na odemknutí všech potřebných záznamů.
- v transakční analýze **při zamykání respektujeme nějakou lineární posloupnost**. Například zamykáme postupně podle ID.
- SŘBD umí **detekci uváznutí** a ukončí jednu z transakcí
- SŘBD obsahuje plánovač, který vykonává transakce tak aby k uváznutí nedocházelo. Například **plánovač pomocí časových razítek**. Jednotlivým transakcím přidělí časové razítko a podle toho pak řadí přístupy

## PROBLÉMY SOUBĚHU

Při souběhu mohou nastat tyto 3 problémy:

### 1. Ztráta aktualizace

Transakce A načte entitu t, transakce B načte taky entitu t. Transakce A entitu změní, pak ji změní i Transakce B a tím pádem je ztracena aktualizace, kterou provedla Transakce A.

*transakce A      transakce B*

read t	-
-	read t
write t	-
-	write t

Tento problém se také označuje jako RW konflikt neboli **špinavý zápis**.

## 2. Nepotvrzená závislost

Transakce A načte entitu t a provede její aktualizaci, Transakce B načte už aktualizovanou entitu t, ale Transakce A provede Rollback a tak Transakce B pracuje s neplatnými daty. Problém nepotvrzené závislosti nastane v případě kdy jedna transakce načte nebo v horším případě aktualizuje entitu, která byla aktualizována dosud nepotvrzenou transakcí. Jelikož tato transakce nebyla potvrzena, existuje možnost, že potvrzena nebude a naopak transakce bude zrušena operací ROLLBACK. V tomto případě ale první transakce pracuje s hodnotami které nejsou platné.

*transakce B      transakce A*

-	write t
read t	-
-	rollback

Tento problém je taky označován jako WR konflikt neboli **špinavé čtení**.

## 3. Nekonzistentní analýza - Jedna Transakce mění data pod rukama té druhé Transakce.

Rozdíl, mezi nepotvrzenou závislostí a nekonzistentní analýzou je, že u nepotvrzené závislosti pracuje transakce s daty, které už neexistují (byl proveden Rollback) a u nekonzistentní analýzy s daty, které existují (byl proveden Commit) ale jsou jiné než na začátku transakce.

*transakce A      transakce B*

read t	-
-	write t
read t	-

## Úroveň izolace

- READ UNCOMMITTED - špinavé čtení - čte nepotvrzená data
- READ COMMITTED - čisté čtení - čte potvrzená data (opakovatelné čtení)
- REPEATABLE READ - čisté neopakovatelné čtení - mohou se zde vyskytovat fantomy
- SERIALIZABLE - čisté neopakovatelné čtení bez fantomů

Vyšší úroveň značí vyšší míru izolace, ale nižší propustnost. Naopak nižší úroveň značí nižší míru izolace, ale vyšší propustnost. Pokud zvolíme maximální úroveň, SERIALIZABLE, pak jsou transakce maximálně izolovány od vlivů ostatních souběžných transakcí. Naopak při nižší úrovni izolace mohou nastat různé problémy souběhu.



Úroveň READ COMMITTED například povoluje následující chování zamykacího protokolu v transakci (jedná se o dříve popsanou výjimku neopakovatelného čtení):

1. Získáme z databáze entiti **t**.
2. Získáme **S** zámek na **t**.
3. Pokud nedojde k modifikaci a tedy k získání **X** zámku, pak zámek může být uvolněn ještě před ukončením transakce.

Pokud jiná transakce provede modifikace **t** a COMMIT, pak původní transakce po možném novém čtení **t** zjistí rozdílnou hodnotu oproti prvnímu čtení – databáze je tedy v nekonzistentním stavu.

Tento plán zjevně nedodržel dvoufázové zamykání: zámek byl uvolněn před ukončením transakce. Pokud by úroveň izolace byla SR nebo RR, pak by všechny získané zámky byly uvolněny až na konci transakce a k tomuto problému by nedošlo. SR je tedy jistější volba úrovně izolace, kdy díky dvoufázovému zamykacímu protokolu, máme zajištěnu serializovatelnost transakcí. Na druhou stranu nižší úroveň izolace mohou, za určitých okolností, poskytnout vyšší míru souběhu.

### VÝJIMKY SOUBĚHU PRO RŮZNÉ ÚROVNĚ IZOLACE

V následující tabulce vidíme různé výjimky souběhu, které mohou či nemohou nastat pro různé úrovně izolace transakcí:

Úroveň izolace	Špinavé čtení	Neopakovatelné čtení	Výskyt fantomů
READ UNCOMMITTED	Ano	Ano	Ano
READ COMMITTED	Ne	Ano	Ano
REPEATABLE READ	Ne	Ne	Ano
SERIALIZABLE	Ne	Ne	Ne

### NEOPAKOVATELNÉ ČTENÍ

V případě úrovně RC (a nižší) je umožněno tzv. neopakovatelné čtení. V tomto případě příkaz SELECT požaduje sdílený zámek na entiti, SŘBD ale nedodrží dvoufázový zamykací protokol a zámky mohou být uvolněny před ukončením transakce. Výlučné zámky jsou ovšem uvolněny až na konci transakce. V případě úrovně SR a RR k této výjimce nedochází, v případě úrovně RC a RU se může tento problém objevit.

### ŠPINAVÉ ČTENÍ

V případě úrovně RU může nastat tzv. špinavé čtení, tedy transakce může načíst data změněná a dosud nepotvrzená jinou transakcí.

## 4) Procedurální rozšíření SQL, PL/SQL, trigger, funkce, procedury, kurzory, hromadné operace.

### Procedurální rozšíření SQL

PL/SQL je procedurálně rozšířené SQL. Kromě základních příkazů pro vytváření a modifikaci dat obsahuje PL/SQL trigger, funkce, procedury, kurzory. To umožňuje přenést část aplikační logiky přímo do databáze.

#### Procedura

```
CREATE PROCEDURE jmeno_procedury (parametry)
IS|AS
    definice lokálních proměnných
BEGIN
    tělo procedury
END;
```

Program v databázi, jehož příkazy jsou uloženy v databázi.

**Anonymní procedury** - nepojmenované procedury, které nejde volat. Mohou být uloženy v souboru nebo spuštěny přímo z konzole. Jsou pomalejší než pojmenované procedury, protože nemohou být předkompilovány.

**Pojmenované procedury** - obsahují hlavičku se jménem a parametry. Díky tomu se dají volat z jiných procedur či triggerů nebo spuštěny příkazem EXECUTE. Jelikož jsou kompilovány jen jednou, jsou rychlejší než anonymní.

## Funkce

Jsou velmi podobné procedurám, ale na rozdíl od nich navíc *specifikují návratový typ* a musí vrátet hodnotu. Kromě standardních funkcí (TO.CHAR, TO.DATE, SUBSTR, apod.) si můžeme definovat vlastní funkce.

Funkce pro získání emailu uživatele

```
CREATE FUNCTION EmailUzivatele( ulogin IN Uzivatel.login%TYPE)
RETURN Uzivatel.email%TYPE AS v_email Uzivatel.email%TYPE;
BEGIN
    SELECT email INTO v_email FROM Uzivatel
    WHERE login = ulogin;
    RETURN v_email;
END EmailUzivatele;
```

## Trigger

Trigger je v podstatě procedura spojená s tabulkou. Přesněji s operací nad tabulkou, protože se spouští ve chvíli kdy je na tabulku zavolaný příkaz **INSERT, UPDATE, nebo DELETE** (může se ještě specifikovat podmínkou WHERE). Pokud se pokusíme v triggeru číst či modifikovat stejnou tabulku dostaneme **mutating table error**. Takovému triggeru bychom se měli vyhnout.

Trigger evidující mazání v tabulce Uzivatel

V případě že dojde ke smazání záznamu v tabulce Uživatel, do tabulky SmazUzivatel SE uloží login a email smazaného uživatele a údaje o mazateli.

```
CREATE TRIGGER smaz
BEFORE DELETE ON Uzivatel
FOR EACH ROW
BEGIN
    INSERT INTO SmazUzivate(login,email,uzitel)
    VALUES(:OLD.login,:OLD.email,USER);
END;
```

**BEFORE, AFTER** - jsou příkazy triggeru definující zda se má trigger spustit před nebo po provedení akčního příkazu

**FOR EACH ROW** - tělo triggeru se provede pro každý řádek tabulky které se týká

**OLD, NEW** - označuje staré či nové hodnoty.

## Kurzor

Kurzor je ukazatel na řádek víceřádkového výběru. Je třeba jej v programu deklarovat, pokud budeme zpracovávat víceřádkové výběry. Kurzorem mohu pohybovat a tak se dostanu na další řádky výběru. Jsou dva typy kurzoru:

Příklad explicitního kurzoru

```
CURSOR muj_kurzor IS Select * FROM Uzivatel;
```

Použití kurzoru - kurzor který prochází všechny e-maily

DECLARE

```
CURSOR muj_kurzor IS Select * FROM Uzivatel;  
  zaznam Uzivatel%ROWTYPE;  
  radek INTEGER := 0;  
BEGIN  
  OPEN muj_kurzor;  
  LOOP  
    FETCH muj_kurzor INTO zaznam;  
    EXIT WHEN muj_kurzor%NOTFOUND;  
    radek := muj_kurzor%ROWCOUNT;  
    DBMS_OUTPUT.PUT_LINE(radek || zaznam.email);  
  END LOOP;  
  CLOSE muj_kurzor;  
END;
```

Popř:

```
DECLARE  
CURSOR c_surname IS SELECT surname FROM Student ;  
BEGIN  
  FOR one_surname IN c_surname LOOP  
    DBMS_OUTPUT.PUT_LINE (one_surname.surname) ;  
  END LOOP;  
END ;
```

- **implicitní** - vytváří se automaticky po provedení příkazu INSERT, UPDATE, DELETE
- **explicitní** - ručně vytvořený kurzor. Vytváří se nejčastěji ve spojení s příkazem select

Příkazy pro práci s kurzorem:

- **OPEN** kurzor - otevře kurzor, tedy nastaví ho na první řádek
- **FETCH** kurzor **INTO** promena - příkaz pro pohyb kurzoru. Načte aktuální záznam do proměnné a posune se na další záznam.
- **CLOSE** kurzor - uzavře kurzor.

## Statické a dynamické PL/SQL

- **Statické** - klasické procedury, které mají vázané proměnné
- **Dynamické** - kód SQL příkazu je vytvářen dynamicky za běhu - vytvoření textového řetězce a jeho spuštění příkazem EXECUTE IMMEDIATE

Hromadné SQL operace v PL/SQL

Typické (neefektivní) vkládání většího počtu záznamů je vkládání po jednom záznamu. U většiny SŘBD můžeme ale využít tzv. **hromadných operací**. Vkládáním většího počtu záznamů najednou snížíme režii zotavení (zápis do logu) a datových struktur (menší počet aktualizací diskových stránek). Výsledkem je rychlejší vkládání záznamů.

Lze použít pro statické i dynamické SQL. U SŘBD Oracle používáme operace **BULK COLLECT** a **FORALL**.

## 5. Fyzická implementace databázových systémů: tabulky a indexy; plán vykonávání dotazů, ladění vykonávání dotazů.

### Fyzický návrh databáze

Fyzická implementace definuje datové struktury pro základní logické objekty:

- tabulky
- indexy
- materializované pohledy
- rozdělení dat (data partitioning)

Fyzická implementace tedy řeší uložení dat na nejnižší úrovni databáze.

Dostupných datových struktur je celá řada, implicitně nabízí SŘBD administrátorovi nějakou volbu.

Např. **CREATE TABLE** značí vytvoření tabulky typu halda, **CREATE INDEX** značí vytvoření B+-stromu.

Na úrovni databáze můžeme ovlivnit výběr efektivnějšího plánu vykonávání dotazu případně dobu vykonávání operací -> fyzický návrh databáze.

## Datové struktury

Datové struktury se skládají buďto ze stránek, nebo z uzlů v případě stromové struktury. Jsou realizovány tak aby **operace vyhledávání, vkládání, editace a mazání** byly **co nejefektivnější**.

Pro rychlé vyhledávání, které předchází i editaci a mazání je nutné udržovat datovou strukturu setříděnou. To však může zpomalit operaci vkládání (např. zařazením nového záznamu do prostřed tabulky musím posunout milion záznamu).

```
C:\Users\little>chkdsk F:
...
488384000 kB místa na disku celkem.
158126356 kB v 178066 souborech.
76232 kB v 18277 rejstřících.
0 kB v chybných sektorech
286784 kB používáno systémem
65536 kB zabírá soubor s protokolem.
329894628 kB na disku je volných.

4096 bajtů v každé alokační jednotce
122096000 alokačních jednotek na disku celkem
82473657 volných alokačních jednotek
```

Výpis příkazu chkdsk, ukazuje že disk F: je členěn do bloků o velikosti 4kB. Těchto bloků má na disku 122 milionu => disk má velikost akoro půl Terabajtu (4kB \* 122 = 488kB)

**Blok** (alokační jednotka) - je nejmenší jednotka, se kterou SŘBD manipuluje při zápisu a čtení dat z disku (obvykle 4KB nebo 8KB).

**Stránka** - nejmenší jednotka s kterou pracuje správce paměti.  $Stranka.velikost = X * Blok.velikost$  (je-li velikost bloku = 4KB je odpovídá velikost stránky násobkům 4KB)

**Datový soubor** - fyzický prostor na disku s daty.

## Typy Tabulek

Každý záznam tabulky má přiřazeno jedinečné číslo – ROWID.

### Tabulka typu halda (Heap table)

Záznamy v tabulce nejsou uspořádány. Je to implicitní volba CREATE TABLE. Jedná se o stránkované perzistentní pole s velikostí bloku nejčastěji 8kB. Záznamy nejsou fyzicky mazány, jsou pouze označeny jako smazané (pro fyzické smazání potřebujeme speciální operaci, tzv. shrinking). Při vkládání je záznam vždy umístěn na první nalezenou volnou pozici v tabulce nebo na konec pole.

- Není možné se spoléhat na uspořádání záznamů v tabulce -> neefektivní vyhledávání,  $O(n)$ .
- Tento typ tabulky je velmi efektivní z pohledu operace INSERT ( $O(1)$ ) a využití místa.

### Hašovaná tabulka (Hash table)

Záznamy se stejnou hašovanou hodnotou jsou uloženy ve stejném bloku, nebo ve velmi blízkém bloku.

**Indexy** - tabulka seříděna podle daného atributu. Jde o soubor obsahující seříděný indexovaný atribut a ukazatel do datového souboru, kde je daný záznam uložen.

## Indexy

Indexové soubory slouží k seřazení tabulky podle jiných atributů než je samotná tabulka defaultně seřazená. Index je tedy vázaný ke konkrétní tabulce a konkrétnímu atributu podle kterého data řadí.

Příklad použití indexů

V tabulce Student potřebujeme často vyhledávat podle loginu, jména a ročníku. Tabulka je automaticky indexována podle loginu, takže to nemusíme řešit.

jmeno	ukazatel
Barbaros Emil	r03→
Dudek Alois	r01→
Filipová Lenka	r02 →

Budeme řešit tedy indexy z hlediska jména a ročníku. Jména jsou vesměs originální, proto použijeme klasický index, který seřadí jména podle abecedy.

CREATE INDEX login ON Student;

ročník:	1	2	3	4	5
1.záznam:	0	0	0	1	0
2.záznam:	0	1	0	0	0
3.záznam:	1	0	0	0	0

Na indexaci ročníku je zbytečné použít klasický index, ročníku je pár a nepotřebuje je řadit od prvního do posledního. Zato studentů v jednom ročníku je hafo. Proto použijeme bitmapový index.

CREATE BITMAP INDEX BIRocnik ON Student(rocnik);

Rozlišujeme dva typy indexů:

- **Klasický index** - si můžeme představit jako tabulku s dvěma sloupci, v prvním je seřazený indexovaný atribut a v druhém je ukazatel na záznam. Ve skutečnosti je realizován nějakou efektivní datovou strukturou (B-stromy) aby se minimalizoval čas při aktualizaci indexu. Tu je třeba provést při vkládání a mazání záznamu v tabulce. V případě že visí na tabulce více indexů je třeba zaktualizovat všechny.
- **Bitmapový index** - odpovídá na všechny možné hodnoty daného atributu. Jde o tabulku jedniček a nul, kde řádky reprezentují záznamy v indexované tabulce a sloupce hodnoty indexovaného atributu. Jednička pak znamená true, že záznam má hodnotu danou sloupcem. Vyplatí se, pokud se používají logické operátory (AND,OR,XOR) nad několika bitmapovými indexy atributů.

**Primární index** (PRIMARY) automatický index který se váže k primárnímu klíči. Zajišťuje jedinečnost údajů.

**Unikátní index** (UNIQUE) stejně jako primární index zajišťuje jedinečnost údajů v atributu. Ale neváže se pouze ke klíči.

**Vedlejší index** (INDEX) klasický index popsány výše.

**Fultextový index** se používá pro optimalizaci fultextového vyhledávání v daném sloupci.

## SHLUKOVANÝ INDEX (Cluster Index)

Pokud se pro dvě tabulky často používá operace spojení (JOIN) pro jeden atribut. V tomto případě diskový blok obsahuje záznam z řídící tabulky a zároveň i závislé záznamy. (v normálním případě obsahuje diskový blok pouze záznamy jedné tabulky)

## Vykonávání dotazu

Identifikujeme 4 fáze vykonávání dotazu:

### 1. Převod dotazu do interní formy

- Převod původního dotazu do zvolené interní formy
- Eliminujeme syntaxi jazyka dotazu (např. SQL)
- Zpracování pohledů, které probíhá v této fázi, znamená, že nahradíme pohled jeho definicí

Interní forma je nejčastěji nějaký druh dotazovacího stromu (angl. query tree)

### 2. Převod do kanonické formy

- V této fázi optimalizátor provádí celou řadu optimalizací.
- Převodem do kanonické formy dochází k odstranění různých povrchních rozdílů a především nalezení efektivnějšího tvaru než nabízel původní dotaz

### 3. Výběr nízkourovňových procedur

- V této fázi se optimalizátor rozhoduje jak bude transformovaný dotaz vykonán.
- Nyní optimalizátor uvažuje: existenci indexů, distribuci hodnot, shlukování uložených dat.

### 4. Vygenerování plánů dotazu a výběr nejlevnějšího plánu

- Cena procedury je závislá na aktuální mohutnosti vstupních relací a na mohutnosti mezivýsledků jednotlivých procedur.
- Odhad velikosti mezivýsledků je však často problematický jelikož velmi ovlivňuje cenu operace, jedná se o jeden z nejřešenějších problémů.
- Z množiny dotazovacích plánů pak optimalizátor vybírá ten nejlepší, tedy nejlevnější

## Ladění dotazů

- Dvěma či více různými dotazy je možno obdržet stejná data.
- Rychlost různých dotazů ovšem nemusí být stejná i přesto, že vracejí stejná data.
- snažíme dosáhnout maximálního výkonu se stávajícími prostředky
- Plán vykonávání dotazu vybírá optimalizátor
- Snažíme se vytvořit dotaz, který bude načítat z úložiště pouze to, co potřebuje

## 6. Objektově-relační datový model a XML datový model: principy, dotazovací jazyky.

### Objektový datový model

**Objektový datový model** jsou data uložené v objektové struktuře. Jde většinou o objektovou mezivrstvu mezi kódem a databází, do které se nasypou data, s kterými pak aplikace pracuje a nezatěžuje dotazy DB server.

ADT abstraktní datový typ adresa

```
CREATE TYPE Adresa AS OBJECT(  
    ulice VARCHAR(50),  
    mesto VARCHAR(50),  
    psc VARCHAR(5),  
)
```

#### Objektově relační datový model

Objektově-relační datový model, je klasická tabulková databáze rozšířená o **abstraktní datové typy** (ADT). ADT jsou uživatelem definované typy skládající se ze základních datových typů databáze. Objektové databázové systémy umožňují využití hostitelského objektového jazyka jako je třeba C++, Java, nebo Smalltalk přímo na objekty "v databázi"; tj. místo věčného přeskakování mezi jazykem aplikace (např. C) a dotazovacím jazykem (např. SQL) může programátor jednoduše používat objektový jazyk k vytváření a přístupu k metodám. Krátce řečeno, ODBMS jsou výborné pro manipulaci s daty. Pokud navíc opomeneme programátorskou stránku, dá se říct, že některé typy dotazů jsou efektivnější než v RDBMS díky dědičnosti a referencím.

ORDBMS využívají datový model tak, že "přidávají objektovost do tabulek". Všechny trvalé informace jsou stále v tabulkách, ale některé položky mohou mít bohatší datovou strukturu, nazývanou abstraktní datové typy (ADT). ADT je datový typ, který vznikne zkombinováním základních datových typů. Podpora ADT je atraktivní, protože operace a funkce asociované s novými datovými typy mohou být použity k indexování, ukládání a získávání záznamů na základě obsahu nového datového typu. OOSŘBD umožňují používat uživatelské typy, dědičnost, metody tříd, rozlišují pojmy instance a ukazatel na instanci.

- Objektové rysy jsou dnes implementovány ve všech velkých SŘBD.
- Objektové typy a jejich metody jsou uloženy spolu s daty v databázi, programátor tedy nemusí vytvářet podobné struktury v každé aplikaci.
- Programátor může přistupovat k množině objektů jako by se jednalo o jeden objekt.
- Objekty mohou jednoduše reprezentovat vazby kdy jedna entita se skládá z jiných entit (bez nutnosti použít vazeb).
- Metody jsou spouštěny na serveru – nedochází k neefektivnímu přenosu dat po síti.

#### Objektový datové typy - Uložené v tabulkách

- Objektové tabulky - obsahují pouze objekty: každý záznam reprezentuje objekt. Mluvíme o tzv. řádkovém objektu. Objekty, které jsou sdíleny dalšími objekty by měly být uloženy v objektových tabulkách.
- Relací tabulky - obsahují objekty spolu s ostatními daty. Mluvíme o tzv. sloupcovém objektu

#### Objektové datové typy:

##### 1. Objektový identifikátor

Identifikuje objekty objektových tabulek. OID není přístupný přímo, pouze pomocí reference. SŘBD automaticky generuje OID pro záznamy objektových tabulek. Sloupcové objekty jsou identifikovány hodnotou primárního klíče, v případě relačních tabulek s objekty tedy OID nepotřebujeme

Ukazatel nebo reference na objekt objektové tabulky je reprezentována datovým typem REF. REF může ukazovat na různé objekty stejného typu, nebo má hodnotu null.

## 2. Kolekce

Používáme pro více hodnotové atributy nebo vazby M:N. Jako datový typ atributu můžeme použít:

- **pole proměnné délky**
- **vehnížděné tabulky**

## 3. Hierarchická dědičnost

V SŘBD můžeme vytvářet hierarchie typů pomocí dědičnosti. Tato vlastnost nám pak umožní využít všechny rysy objektově - orientovaných technologií, např. Mnohotvárnost, polymorfismus.

### Objektové datové typy

- mohou obsahovat:
  - data – atributy
  - operace – metody
- Specifické pro ORŠRBD je to, že se na data můžeme dívat jak z relačního (záznamy, operace) tak objektového pohledu (objekty, metody).

```
CREATE TYPE person_type AS OBJECT ( idno NUMBER, first_name VARCHAR2(20), last_name VARCHAR2(25) , MAP MEMBER FUNCTION get_idno RETURN NUMBER);
```

- pak definujeme těla metod:

```
CREATE TYPE BODY person_type AS MAP MEMBER FUNCTION get_idno RETURN NUMBER IS BEGIN RETURN idno ; // vrací id  
END; END;
```

- Objektové datové typy může používat při definici atributů podobně jako SQL datové typy.

```
CREATE TABLE contacts ( contact person_type , contact_date DATE);
```

- Záznam vložíme pomocí SQL INSERT:

```
INSERT INTO contacts VALUES ( person_type (65, 'Verna' , 'Mills') , ' 24 Jun 2003 ' ) ;
```

### Typy metod:

- **Členské metody** – jsou volány nad konkrétním objektem.
- **Statické metody** – jsou volány nad datovým typem.
- **Konstruktor** – pro každý objektový typ je definován implicitní konstruktor.

## XML

- značkovací jazyk - W3C standart
- Data jsou uložena v xml souboru formou stromu.
- Logika a význam dat je součástí xml souboru.
- Skládá se z hlavičky, kořenového elementu (právě jeden), vnořených elementů, atributů a textu vnořené uvnitř (ukládání informace)

Dotazovací jazyky jsou:

- **XQuery** - je silně typovaným jazykem.
  - \$ identifikuje proměnné
  - for - iterace přes něco
  - where - omezení výběru



- order by – řadí
- return - vrací výsledek
- let - nastavuje hodnotu pro proměnnou

```
for $x in doc("books.xml")/bookstore/book
```

```
where $x/price>30
```

```
order by $x/title
```

```
return $x/title
```

- **XPath** - je jazyk používaný pro identifikaci uzlů v XML dokumentu. Pravděpodobně nejdůležitějším rysem jazyka XPath je možnost vyjádření relativní cesty od uzlu k jinému uzlu či atributu. Připomíná dotazovací jazyk SQL, zejména díky tomu, že na základě podmínky vrátí jeden nebo množinu výrazů (nebo žádný) odpovídající vstupní podmínce.

```
//zboží[@sleva >= @cena div 2], který najde všechny elementy zboží, jejichž atribut sleva má hodnotu nejméně poloviny hodnoty atributu cena
```

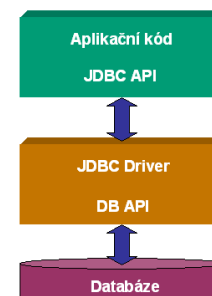
## 7. Datová vrstva informačního systému; existující API, rámce a implementace, bezpečnost; objektově-relační mapování.

### Datová vrstva IS

Datová vrstva informačního systému odděluje aplikaci od databáze. Jde o třídy a funkce zajišťující komunikaci s databází.

**ODBC** (Open DataBase Connectivity) je standartizované API pro přístup k databázovým systémům. ODBC se snaží poskytovat přístup nezávislý na programovacím jazyku, databázovém systému a operačním systému.

**JDBC** (Java DataBase Connectivity) javovské API pro unifikovaný přístup k databázi. Bez ohledu kde jsou data uložena - SQL databáze, Oracle, XML, CSV,... - se s nimi přes JDBC pracuje stejně.



**ADO.NET** (Active Data Object) specifikace (knihovna) pro přístup k datovým zdrojům na platformě .NET.

**ASP.NET** - knihovny pro informační systémy na platformě .NET. Pro komunikaci s databází využívají ADO.NET.

**Java2EE** - knihovny pro informační systémy na platformě Java.

**ORM** - Objektově relační mapování (ORM) je programovací technika zpřístupňující relační (či objektově-relační data) pro objektové prostředí (např. C# nebo Java).

**Hibernate** - ORM nástroj založený na Javě

## Vlastnosti ORM rámců

- práce s objektovým modelem,
- rychlejší vytváření aplikací vs menší výkon aplikace
- přenositelnost mezi různými SŘBD vs. nevyužívá všechny vlastnosti SŘBD (efektivita, bezpečnost atd.)
- typová kontrola
- nevznikají chyby v SQL
- jednodušší testování

Entita v ORM je objekt, který je uložen v SŘBD (nejčastěji jako jeden záznam). Nejčastěji je implementován jako třída. Díky rozhraní jsou jednotlivé implementace zaměnitelné.

## Bezpečnost

SQL injection: Testujeme hodnoty datových typů, zda neobsahují nekorektní znaky v daném kontextu (např. apostrof). Používáme parametrizované dotazy. V některých případech používáme uložené procedury.

uživatel musí zadávat komplikovaná (neslovníková) hesla, uživatel může zadat heslo nejvýše

2x, logování neúspěšných pokusů o autentizaci.

## 8. Distribuované SŘBD, fragmentace a replikace.

---

### Distribuované SŘBD

**SŘBD**: je softwarové vybavení, které zajišťuje práci s databází, tzn. tvoří rozhraní mezi aplikačními programy a uloženými daty. Občas se pojem zaměňuje s pojmem databázový systém. Databázový systém však je SŘBD dohromady s bází dat. Aby mohl být nějaký programový systém označený za SŘBD, musí být jednak schopen efektivně pracovat s velkým množstvím dat, ale také musí být schopný řídit (vkládat, modifikovat, mazat) a definovat strukturu těchto perzistentních dat (čímž se liší od prostého souborového systému).

Distribuovaná databáze má data rozdělené na více počítačích. Přičemž je zachována **transparentnost** uživateli se jeví jako by byla celá databáze na jednom místě. A **autonomnost** s lokálními bázemi dat je možné pracovat nezávisle na ostatních databázích.

**Replikace** dat (přítomnost duplicitních dat) zajišťuje plnou funkčnost databáze, i když nějaký uzel vypadne. Díky tomu se dist. databáze i samozalohuje.

**Fragmentace** - distribuovaná databáze má je rozdělená do více uzlů. Podle toho zda ji rozsekáme po sloupcích či po řádcích jde o fragmentaci **vertikální** či **horizontální**.

Podle toho jak je dist. databáze řešena a kam směřují dotazy ji dělíme na:

- **Centralizovanou** - všechny dotazy směřují do centrálního uzlu s řídicí jednotkou. Nebezpečí výpadku tohoto uzlu.
- **Decentralizovanou** - každý uzel má svou řídicí jednotku. Vysoké nároky na obsluhu.

Ideální je kombinovat oba přístupy.