

# III. Databázové a informační systémy (Úvod do databázových systémů, Databázové a informační systémy)

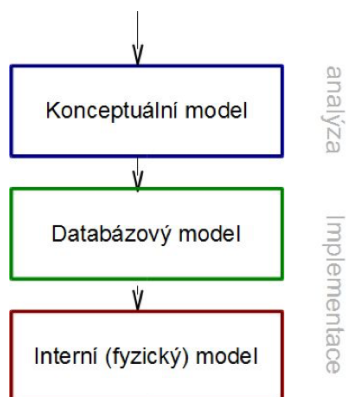
Update: 9. května 2018

## Obsah

1	Modelování databázových systémů, konceptuální modelování, datová analýza, funkční analýza; nástroje a modely.	2
2	Relační datový model, SQL; funkční závislosti, dekompozice a normální formy.	6
3	Transakce, zotavení, log, ACID, operace COMMIT a ROLLBACK; problémy souběhu, řízení souběhu: zamykání, úroveň izolace v SQL.	14
4	Procedurální rozšíření SQL, PL/SQL, T-SQL, trigger, funkce, procedury, kurzory, hromadné operace.	21
5	Základní fyzická implementace databázových systémů: tabulky a indexy; plán vykonávání dotazů.	26
6	Objektově-relační datový model a XML datový model: principy, dotazovací jazyky.	31
7	Datová vrstva informačního systému; existující API, rámce a implementace, bezpečnost; objektově-relační mapování.	35
8	Distribuované SŘBD, fragmentace a replikace.	38

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

## 1.1 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**.

### 1.1.1 Základní pojmy

- **Entita** – objekt reálného světa.
- **Atribut** – vlastnost entity (možné hodnoty jsou označeny jako doména atributu).
- **Entitní typ** – množina entit se stejnými atributy.
- **Vztah** – vztah mezi **dvěma entitními** typy.
- **Kardinalita vztahu** – dělení vztahů podle počtu entit vstupujících do vztahu – 1:1, 1:N, M:N.

## 1.2 Datová analýza a konceptuální model

Datová analýza **zkoumá objekty reálného světa, jejich vlastnosti a vztahy**. Zabývá se strukturou obsahové části systému (**strukturou databáze**). Výsledkem datové analýzy je **konceptuální model**. V rámci datové analýzy zpracováváme zadání (specifikaci požadavků na IS):

- podtrhneme **podstatná jména** = identifikujeme **objekty**,
- podtrhneme **slovesa** = identifikujeme **vazby** mezi objekty,

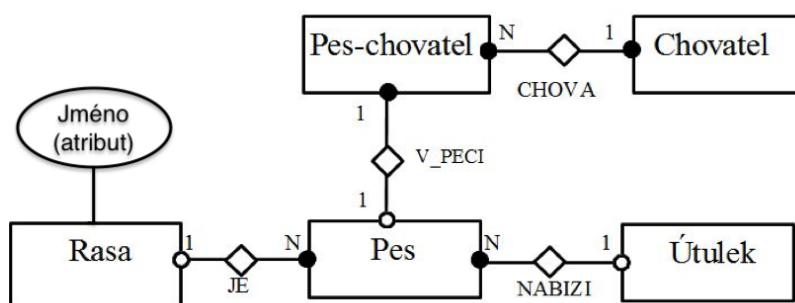
- najdeme **vlastnosti** a **stavy** nalezených objektu = identifikujeme **atributy**.

Z takto získaných informací sestavíme konceptuální model. **Konceptuální model** je jednoduchý **popis entit a jejich vzájemných vztahů**. Jedná se o jakýsi prvotní jednoduchý návrh námi vytvářené databáze. Je kladen důraz na zobrazení všech entit, jejich vztahů a je **nezávislý** na SŘBD. Skládá z:

- **ER Diagram**, lineární zápis **entit**, lineární zápis **vztahů**, **datový slovník**, popis dalších IO (**integritních omezení**).

### 1.2.1 ER (Entity-Relationship) Diagram

Grafické znázornění **konceptuálního modelu** (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 třídovému diagramu s atributy vepsanými do objektu.



### 1.2.2 Lineární zápis entit a vztahů

Lineárním 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, *ID*Utulek).
- Příklad lineárního zápisu vztahů: NABIZI (Útulek, Pes) 1:N.

### 1.2.3 Datový slovník

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

**Integritní omezení** obsahují další specifikace atributů, které nejsou dány typem a délkou. Nejčastěji se týkají formátu atributu (podmínka v jakém má být formátu) – např.: login se skládá z třech čísel a třech písmen, nebo rodné číslo je složeno z data narození, apod.

**Další integritní omezení** – 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.

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

Tabulka 1: Datový slovník pro tabulku Pes.

### 1.3 Funkční analýza

Zatímco datová analýza se zabývá strukturou obsahové části systému (strukturou databáze), **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.

Cílem je popsat vytvářený systém jako „černou skříňku“, definovat její **vnější chování** a strukturalizovat **okolí systému**, které se systémem komunikuje. **Popsat všechny funkce, které se budou s daty provádět.**

#### Otázky na požadavky

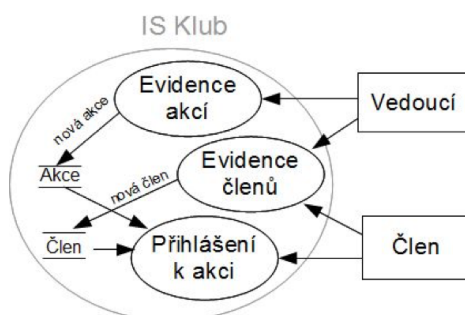
- **PROČ** nový systém.
- **ČEMU** má sloužit.
- **KDO** s ním pracuje - běžně, příležitostně, pravidelně zřídka.
- **VSTUPY** – objekty, atributy
- **VÝSTUPY** – výstupní sestavy, požadované informace
- **FUNKCE** – jaké výpočty, odvozování, výběry, třídění, ...
- **Vazby na OKOLÍ systému** – odkud data a kam.

#### Nefunkční požadavky

- Požadavky na **výsledný program**.
- **Vnější požadavky**: ostatní nefunkční implementační požadavky, použití **standardů**, **cenová omezení**, **časové požadavky**.

### 1.3.1 Diagram datových toků (DFD)

DFD je grafický nástroj pro **modelování funkcí a vztahů v systému**. Znáznorňuje 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**. Znáznorňuje 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. **Hlavní znaky DFD:**

- Má několik úrovní podrobnosti.
- Definuje **hranici systému**.
- Definuje **všechny akce**, které mezi systémem a jeho okolím probíhají.

### 1.3.2 Minispecifikace = algoritmy elementárních funkcí

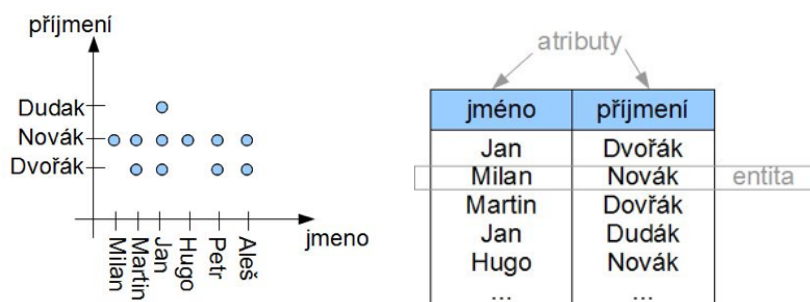
- Popisuje logiku každé z funkcí **poslední úrovně DFD**.
- Každému **elementárnímu (nerozložitelnému) procesu** z poslední úrovně DFD odpovídá **jedna minispecifikace**.
- Popisuje postup, jak jsou **vstupní data transformována na výstupní**.
- Popisuje, co **funkce znamená**, ne jak se to spočítá.
- Používá se **přirozený jazyk** s omezeným množstvím jasně definovaných pojmů, aby byla **srozumitelná** jak pro analytika, tak i uživatele a programátorovi.

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

### 2.1 Relační datový model

Relační datový model představuje **způsob uchování dat v tabulkách**. Relační se mu říká proto, jelikož tabulka je definována přes Relaci.



Obrázek 1: Tabulka s dvěma atributy jako relace (vlevo), relace zobrazena tabulkou (vpravo).

**Relace** je tabulka definována jako **podmnožina kartézského součinu domén**. Relace na obrázku je tedy podmnožina kartézského součinu množin  $\{\text{Dudak, Novák, Dvořák}\} \times \{\text{Milan, Martin, Jan, ..., Aleš}\}$ .

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.

- **Doména** je množina všech hodnot, kterých může daný atribut nabývat (obor hodnot atributu). V praxi je doména dána **integritním omezením** (IO). Doména atributu Příjmení z obrázků je množina  $\{\text{Dudak, Novák, Dvořák}\}$ .
- **Atribut** je vlastnost entity (z pohledu tabulky jde o sloupec).
- **Relační schéma** můžeme chápat jako strukturu tabulky (atributy a domény). Relační schéma  $R$  je výraz tvaru  $R(A, f)$ , kde  $R$  je jméno schématu,  $A = A_1, A_2, \dots, A_n$  je **konečná množina jmen atributů**,  $f$  je zobrazení přiřazující každému jménu atributu  $A_i$  neprázdnou množinu (obor hodnot atributu), kterou nazýváme **doménou atributu**  $D_i$ , tedy  $f(A_i) = D_i$ .

#### Příklad pro tabulku (relaci) Učitel

- **Atributy:** ID, jméno, příjmení, funkce, kancelář.
- **Domény:**
  - $D_1$  – tři písmena z příjmení, tři cifry pořadového čísla,
  - $D_2$  – 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), ...}

### 2.1.1 Základní úlohy relačního modelu:

1. Návrh „správné“ struktury databáze bez redundancí – funkční závislosti, normální formy.
2. Vyhledávání informací z databáze – (dotazovací) relační jazyky.

### 2.1.2 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).

### 2.1.3 Vazby relačního 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. Příklad vazby 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.

## 2.2 SQL (Structured Query Language)

SQL (Structured 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.

- Standardizovaný **strukturovaný dotazovací jazyk**, který je používán pro práci s daty v **relačních databázích**. (DQL - Data Query Language).
- Navržen IBM jako **dotazovací jazyk** (původní název Sequel).
- Základem je **n-ticový relační kalkul**.
- Standardy podporuje prakticky každá relační databáze, ale obvykle nejsou implementovány vždy **všechny požadavky normy**.
- Obsahuje i příkazy pro **vytvoření** a **modifikace** tabulek, pro **ukládání**, **modifikaci** a **rušení** dat v databázi a řadu dalších příkazů.
- **Příklad:** CREATE TABLE Drazitel (jmeno CHAR(20), adresa CHAR(30), aukce NUMBER(4), zisk NUMBER(4)); INSERT INTO clovek VALUES( 'nj001', 'Jan', 'Novotný', '777111222'); SELECT telefon FROM clovek WHERE prijmeni = "Novotný";

### 2.2.1 Příkazy SQL

- **Vytváření** a **modifikace** relačního schematu (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, shlu-kuj, 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,...).

## 2.3 Relační jazyky

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

- **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.



- **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.

## 2.4 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.

### 2.4.1 Základní operace relační algebry

Jsou dány relace **R** a **S**. **Množinové operace:**

- **Sjednocení** relací téhož stupně:  $R \cup S = \{x | x \in R \vee x \in S\}$
- **Průnik** relací:  $R \cap S = \{x | x \in R \wedge x \in S\}$
- **Rozdíl** relací:  $R - S = \{x | x \in R \wedge x \notin S\}$
- **Kartézský součin** relace **R** stupně **m** a relace **S** stupně **n**:  $R \times S = \{rs | r \in R \wedge s \in S\}$ , kde  $rs = \{r1, \dots, rm, s1, \dots, sn\}$

**Další relační operace:**

- **Projekce** (výběr sloupců) relace **R**, jedná se o unární operaci  $\Pi_{\mathbf{X}}(R)$ , kde **X** je množina názvů atributů.
- **Selekce** (výběr řádků) z relace **R** podle podmínky **P**. Selektce je unární relační operace  $\sigma_{\varphi(\mathbf{X})}(R)$ , kde **R** je relace,  $\varphi(\mathbf{X})$  predikátová formule hovořící o jednotlivých prvcích a jejich příslušnosti do relací.
- **Spojení** relací **R** s atributy **A** a **S** s atributy **B** (join). Značí se  $R \bowtie S$ , výsledkem je množina všech kombinací prvků relace **R** a **S**. Takto definovaný join se nazývá Přirozené spojení (natural join). Existují i další (outer, inner, left, right ...).

## 2.5 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**  $\{x|F(x)\}$  nahradíme zápisem **x WHERE F(x)**
  - Kde x je proměnná pro hledané n-tice (struktura relace).
  - F(x) je **podmínka**, kterou má x splňovat (výběr prvků relace).

### 2.5.1 Definice

Výraz n-ticového relačního kalkulu je výraz tvaru **x WHERE F(x)**, kde x je jediná volná proměnná ve formuli F. Základní operace relační algebry se dají vyjádřit pomocí výrazů n-ticového relačního kalkulu, tedy n-ticový relační kalkul je relačně úplný.

Platí:	$R \cup S$	$\Rightarrow x \text{ WHERE } R(x) \text{ OR } S(x)$
	$R \cap S$	$\Rightarrow x \text{ WHERE } R(x) \text{ AND } S(x)$
	$R - S$	$\Rightarrow x \text{ WHERE } R(x) \text{ AND NOT } S(x)$
	$R \times S$	$\Rightarrow x, y \text{ WHERE } R(x) \text{ AND } S(y)$
	$R[a_1, a_2, \dots, a_k]$	$\Rightarrow x.a_1, x.a_2, \dots, x.a_k \text{ WHERE } R(x)$
	$R(P)$	$\Rightarrow x \text{ WHERE } R(x) \text{ AND } P$
	$R[A*B]S$	$\Rightarrow x, y \text{ WHERE } R(x) \text{ AND } S(y) \text{ AND } x.A * y.B$

## 2.6 Funkční závislost

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**. Je-li  $Y, X$  říkáme, že závislost  $X \rightarrow Y$  je **triviální**.

- FZ je definována **mezi dvěma podmnožinami atributů** v rámci jednoho schématu relace. Jde o vztah mezi atributy, nikoliv mezi entitami.
- FZ je definována na **základě všech možných aktuálních relací**, není tedy možné soudit na funkční závislost z vlastností jediné relace. Tak můžeme poznat jen neplatnost funkční závislosti.
- FZ jsou **tvrzení o reálném světě**, o významu atributů nebo **vztahů mezi entitami**, je nutné realitu brát v úvahu při návrhu schématu databáze.

**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.

## 2.7 Armstrongovy axiomy

K určení **klíče schématu** a logických implikací množiny závislostí potřebujeme **nalézt uzávěr  $F^+$** , nebo určit, zda daná závislost  $X \rightarrow Y$  je prvkem  $F^+$ . K tomu existují pravidla zvaná Armstrongovy axiomy. Jsou **úplná** (dovolují odvodit z dané množiny závislostí  $F$  všechny závislosti patřící do  $F^+$ ) a **bezesporná** (dovolují z  $F$  odvodit pouze závislosti patřící do  $F^+$ ).

- **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  $X \rightarrow Z$

Závislost, která má na pravé straně pouze jeden atribut, nazýváme **elementární**.

### 2.7.1 Určení klíče pomoci funkčních závislostí

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

- $\text{učitel} \rightarrow \text{jméno, příjmení, email}$
- $\text{předmět} \rightarrow \text{název, kredity}$
- $\text{místnost, čas} \rightarrow \text{učitel, předmět}$

**Rozšíření:**

- $\text{učitel, místnost, čas} \rightarrow \text{jméno, příjmení, email, místnost, čas}$
- $\text{předmět} \rightarrow \text{název, kredity}$
- $\text{místnost, čas} \rightarrow \text{učitel, předmět}$

**Dekompozice 1:**

- $\text{učitel, místnost, čas} \rightarrow \text{jméno, příjmení, email, místnost, čas, učitel, předmět}$
- $\text{předmět} \rightarrow \text{název, kredity}$

**Dekompozice 2:**

- učitel, místnost, čas → 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.

## 2.8 Dekompozice

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**.

- **Algoritmus dekompozice (metoda shora dolů)** – na počátku máme celé relační schéma se všemi atributy, snažíme se od tohoto schématu odebírat funkční závislosti a tvořit schémata nová. **Exponenciální složitost, BCNF**.
- **Algoritmus syntézy (zdola nahoru)** – vytvoří pro každou funkční závislost novou relaci. Pak tyto malé relace spojuje do větších celků. **Menší složitost, 3NF**.

**Binární dekompozice**, kterou budeme dále řešit je rozklad jednoho relačního schématu na dvě. Obecná dekompozice vznikne postupnou aplikací binárních. Dekompozice relačního schématu  $R(A, f)$  je množina relačních  $RO = \{R_1(A_1, f_2), R_2(A_2, f_2), \dots\}$ , kde  $A = A_1 \cup A_2 \cup A_3 \cup \dots$

## 2.9 Normální formy

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

- **1 NF** – definuje tabulky, které obsahují **pouze atomické atributy**. Žádné složené atributy - např. v jednom atributu je Jméno i Příjmení.
- **2 NF** – je v 1NF + **každý sekundární atribut je úplně závislý na každém klíči schématu**. Neboli neexistuje závislost sekundárních na podklíči (pokud se klíč skládá z více atributů). Např.: když  $AB \rightarrow CD$ , pak nesmí být  $B \rightarrow C$ . Atribut adresa není závislý na všech klíčích FZ, ale pouze na F.

firma	adresa	zboží	cena
F1	A1	Z10	100
F1	A1	Z20	50
F2	A2	Z20	80

- **3 NF** – 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

- **BCNF** (Boyce-Coddova normální forma) – 3NF + je-li funkční závislost  $(X \rightarrow Y) \in F^+$  a  $Y \notin X$ , pak X obsahuje klíč schématu. **Musí být závislost sekundárních atributů na primárních nikoli naopak.** 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.

#### 3.1 Transakce

**Logická (nedělitelná, atomická) jednotka práce s databází**, která musí proběhnout buď celá, nebo (v případě že je přerušena) obnovit původní stav databáze a spustit se znovu. Začíná operací **BEGIN TRANSACTION** a končí provedením operací **COMMIT** nebo **ROLLBACK**.

- Obecně zahrnuje posloupnost operací.
- Jejím úkolem je převést **korektní stav databáze** na jiný korektní stav.
- O řízení se stará **manager transakcí** nebo **monitor transakčního zpracování**.
- Operace transakce jsou nejprve zaznamenávány do **logu**.
- Transakce nemohou být vnořovány.

##### 3.1.1 COMMIT

- Transakce doběhla úspěšně a změny mohou být **trvale uloženy**, zámky a adresace uvolněny (kromě WITH HOLD).
- Zavádí **potvrzovací bod**.
- Odpovídá úspěšnému ukončení logické jednotky práce a **označuje korektní stav DB**.

##### 3.1.2 ROLLBACK

- Označuje, že databáze může být v **nekorektním stavu** a všechny změny transakce musí být **zrušeny**.

#### 3.2 Zotavení

- Nastává po **chybě SŘBD** => Zotavení databáze z nějaké chyby.
- Výsledkem musí být **korektní stav DB**.
- Využívají se **skryté redundantní informace**.
- Jednotkou zotavení je **transakce**.
- Všechny změny jsou zapisovány do logu před zápisem změn do DB => **pravidlo dopředného zápisu do logu**.
- Do logu se zapisuje **sekvenčně**, proto poskytuje **vyšší výkon** než přímý zápis dat.

### 3.2.1 Chyby zotavení

- **Lokální** - pouze v rámci jedné transakce (chyba v dotazu, přetečení hodnoty atributu).
- **Globální** - ovlivňují více transakcí najednou:
  - **Systémové (soft crash)** (výpadek proudu, pád systému).
  - **Chyby média (hard crash)** - chyba disku (zotavení probíhá ze záložní kopie a z logu jsou obnoveny potvrzené transakce po vytvoření zálohy).

### 3.2.2 Průběh zotavení

Základním problémem vzniklým při systémové chybě je ztráta obsahu hlavní paměti, tedy ztráta obsahu vyrovnávací paměti SŘBD. Přesný stav transakce přerušené chybou není znám a transakce musí být **zrušena (UNDO)**. Někdy je transakce úspěšně dokončena, ovšem změny, nejsou přeneseny z vyrovnávací paměti na disk. V tomto případě musí být transakce po restartu systému přepracována (**REDO**). **Typy zotavení:**

#### Odloženou aktualizací (deferred update, NO-UNDO/REDO)

- Neprovádí aktualizace databáze na disk dokud transakce nedosáhne **potvrzovacího bodu**. Všechny změny jsou v paměťovém bufferu.
- Jakmile transakce dosáhne potvrzovacího bodu se **nejprve zapíše vše do logu** a pak do DB (**pravidlo dopředného zápisu do logu**).
- Při selhání transakce není nutné provádět **undo**, změny jsou ztraceny spolu s vyrovnávací pamětí.
- **Redo** se provádí při chybě během zápisu do DB.
- Do logu jsou v případě odložené aktualizace zapsány nové hodnoty (kvůli REDO).
- Minimální I/O operace, používá se pouze pro **krátké a nenáročné transakce** - **hrozí přetečení bufferu**.

#### Okamžitou aktualizací (immediate update, UNDO/NO-REDO)

- **Může provádět aktualizace DB** než transakce dosáhne potvrzovacího bodu.
- Operace jsou zapsány do logu a poté je aktualizována DB (pravidlo dopředného zápisu do logu).
- Při chybě je nutné provést **undo**, protože **došlo k aktualizaci DB**.
- Do logu se zapisují **původní hodnoty**, což umožní systému provést UNDO.
- **Velká zátěž disku**.

### Kombinovanou aktualizací (UNDO/REDO)

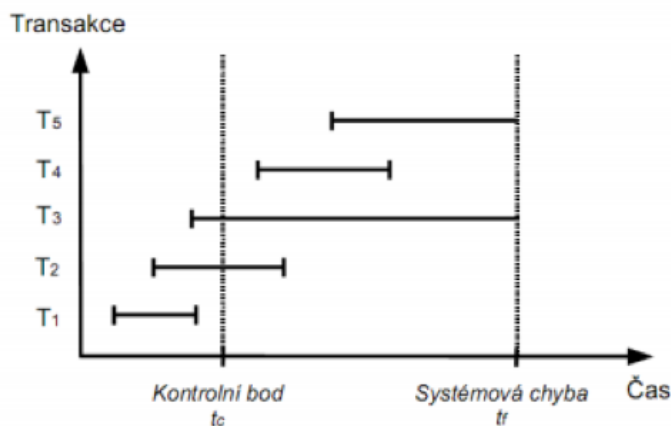
- Používaná v praxi. Využívá obou operací v kombinaci s technikou kontrolních bodů.
- Nezapisuje všechny potvrzené operace na disk, místo toho vytváří **kontrolní body**.
  - Zápis operací hromadně po určitém počtu záznamů.
  - Zapisuje se obsah vyrovnávací paměti na disk a záznam o kontrolním bodu do logu.
- Po restartu systému se provádí: **undo** na všechny transakce, které se **nestihly potvrdit** a **redo** na všechny transakce, které **se potvrdily** po vytvoření kontrolního bodu.

### 3.3 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.

V případě následující situace musí být:



- 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.

### 3.4 ACID

Každá transakce by měla splňovat následující vlastnosti:



- **Atomičnost (Atomicity)** – transakce musí být atomická: jsou provedeny všechny operace transakce nebo žádná.
- **Korektnost (Correctness)** – transakce převádí korektní stav databáze do jiného korektního stavu databáze, mezi začátkem a koncem transakce nemusí být databáze v korektním stavu.
- **Izolovanost (Isolation)** – transakce jsou navzájem izolovány: změny provedené jednou transakcí jsou pro ostatní transakce viditelné až po provedení COMMIT.
- **Trvalost (Durability)** – jakmile je transakce potvrzena, změny v databázi se stávají trvalými i po případném pádu systému.

### 3.5 Problémy souběhu

- Jednouživatelský/víceuživatelský DB systém (kolik současně).
- Souběh umožňuje SŘBD **zpřístupnit databázi mnoha transakcím ve stejném čase**.
- Souběh také přináší mnoho **problémů**, které je nutné řešit i na **aplikační úrovni**.

#### 3.5.1 Plán provádění transakce a anomálie

Plán provádění transakce = posloupnost operací transakce, při souběžném provedení - **plán souběžný/paralelní**. Vznikají **3 problémy**:

- **Problém ztráty aktualizace** – jedna transakce **přepíše právě prováděnou hodnotu**. Časová posloupnost: read\_A, read\_B, write\_A, write\_B.

Transakce A	Čas	Transakce B
READ $t$	$t_1$	-
-	$t_2$	READ $t$
WRITE $t$	$t_3$	-
-	$t_4$	WRITE $t$

Obrázek 9.1: Aktualizace transakce A je ztracena v čase  $t_4$ .

- **Problém nepotvrzené závislosti** – A přečte **nepotvrzená** (non-commited) data zapsaná B, B pak provede **callback**. A má **neplatná data!** Nebo B zapíše, A zapíše, B rollback - vrátí do stavu před 1. zápisem B.

Transakce A	Čas	Transakce B
-	$t_1$	WRITE $t$
READ $t$	$t_2$	-
-	$t_3$	ROLLBACK

Obrázek 9.2: Transakce A se stala v čase  $t_2$  závislou na nepotvrzené změně transakce B.

- **Problém nekonzistentní analýzy** – A provádí součet na účtech, před dokončením B provede přesun z účtu na účet, přičemž 1 už byl započítán a druhý ne. **Špatný**

**součet zůstatků!** A čte committed data (B provede commit, než si A vyžádá další účet), ale i tak to není správné.

$acc_1 = 30$	$acc_2 = 20$	$acc_1 = 50$
Transakce A	Čas	Transakce B
READ $acc_1$ $suma = 30$	$t_1$	-
READ $acc_2$ $suma = 50$	$t_2$	-
-	$t_3$	READ $acc_3$
-	$t_4$	WRITE $acc_3 = 60$
-	$t_5$	READ $acc_1$
-	$t_6$	WRITE $acc_1 = 20$
-	$t_7$	COMMIT
READ $acc_3$ $suma = 110$ <b>ne 100</b>	$t_8$	-

Obrázek 9.4: Transakce A provedla nekonzistentní analýzu.

### 3.6 Konflikty čtení/zápis

A a B chtějí číst/zapisovat stejnou entici (záznam). Nastávají 4 možnosti konfliktu:

- **RR (READ-READ)** – negativně se neovlivní, není problém.
- **RW (READ-WRITE)** –  $read\_A, write\_B = A$  dále počítá s daty  $\rightarrow$  RW zapříčiňuje **problém nekonzistentní analýzy**.  $read\_A, write\_B, read\_A \rightarrow A$  načte odlišené hodnoty = **neopakovatelné čtení (non repeatable read)**
- **WR (WRITE-READ)** –  $write\_A, read\_B, rollback\_A?$   $\rightarrow$  **Problém nepotvrzené závislosti**. Pokud B přečte data  $\rightarrow$  **Špinavé čtení (dirty read)** = čtení non-committed dat.
- **WW (WRITE-WRITE)** –  $write\_A, write\_B, rollback\_A?$   $\rightarrow$  **Ztráta aktualizace** (pro A) a **nepotvrzená závislost** pro B. **Špinavý zápis (dirty write)** - přepisování non-committed dat.

### 3.7 Techniky řízení souběhu

#### 3.7.1 Správa verzí

Předpoklad, že se paralelní **transakce ovlivňovat nebudou**. Systém **vytváří** při aktualizaci **kopie dat** a sleduje, která z verzí má být viditelná pro ostatní transakce (podle úrovně izolace).

#### 3.7.2 Zamykání

Předpokládáme, že se paralelní **transakce budou ovlivňovat**. Systém spravuje jednu kopii dat a jednotlivým transakcím přiděluje **zámky**.

- Chce-li transakce A provést čtení/zápis nějakého objektu v DB (nejčastěji n-tice), **požádá o zámek** na tento objekt. Žádná jiná paralelní transakce zámek získat nemůže, dokud jej A **neuvolní**.
- **2 typy zámků (existuje jich i více):**
  - **výlučný zámek** (exclusive lock / write lock) **X**.
  - **sdílený zámek** (shared lock / read lock) **S**.
- A má zámek X a B **nedostane žádný zámek** hned. A má zámek S, B **může hned dostat S**, X nikoliv.
- **Matice kompability** - vzájemné vztahy typů zámků, sloupce a řádky: X, S, -, A (okamžitě), N (ne)

	X	S	-
X	N	N	A
S	N	A	A
-	A	A	A

- **Operace aktualizace** - mění obsah DB - UPDATE, INSERT i DELETE.
- **Uzamykací protokol** - většinou žádání zámků implicitně → při **získání** n-tice z DB žádán **zámek S**. Při aktualizaci **zámek X**; žádá-li zámek X a má už S, je mu **S změněn na X**; když nemůže být zámek přidělen okamžitě, transakce přechází do stavu **čekání** (wait state).
- Systém musí zajistit aby v tomto stavu nesetřvala navždy - situace "**livelock**" nebo "**starvation**" → řadit požadavky do **fronty** (FIFO). Zámky uvolněny až po operaci COMMIT nebo ROLLBACK.
- **Explicitní uzamykání** - LOCK TABLE <names> IN [ROW SHARE|ROW EXCLUSIVE|SHARE UPDATE|SHARE|SHARE ROW EXCLUSIVE|EXCLUSIVE] MODE [NOWAIT]

### 3.8 Uvážnutí

- **Deadlock** - dvě nebo více transakcí jsou ve stavu **čekání** na uvolnění zámků držených jinou transakcí.
- **Detekce uvážnutí - časové limity** (nastavení max. času pro vykonání transakce), **detekce cyklu v grafu Wait-For** (zaznamenává, které transakce na sebe čekají → u jedné provede ROLLBACK).
- **Prevence uvážnutí pomocí časových razítek** - 2 verze uzamykacího protokolu. Každá transakce na začátku dostane časové razítko (unikátní). Pokud A požaduje zámek na entici, která je zamčená B pak:

- při **Wait-Die** - pokud je A starší než B, A přejde na čekání; je-li mladší, A je zrušena ROLLBACK a spuštěna znovu.
- při **Wound-Die** - pokud A je mladší než B, A přejde na čekání; starší → B zrušena ROLLBACK a spuštěna znovu.
- Při opětovném spuštění si transakce nechá své časové razítko. **Nevýhodou** je velký počet operací ROLLBACK. První část jména - situace kdy A je starší než B. **Nemůže nikdy dojít k uváznutí.**

### 3.9 Sériový a serializovatelný plán

- **Sériový plán** - n-tice uspořádaná dle **pořadí vykonávání** jednotlivých transakcí. (transakce jsou provedeny zasebou).
- **Serializovatelný plán** - **plán vykonávání dvou transakcí** je korektní jen tehdy, pokud je serializovatelný → plán ekvivalentní s výsledkem libovolného sériového plánu.

### 3.10 Úroveň izolace transakce

Serializovatelnost garantuje izolaci transakcí ve smyslu podmínky **ACID**. Je-li plán transakcí serializovatelný, neprojeví se negativní vlivy souběhu. Za izolovanost transakcí se platí **menším výkonem** souběhu → **nižší propustností**. SŘBD umožňuje nastavit úroveň izolace - ta **sníží míru izolace** transakce a **zvýší propustnost**.

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

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

### 4.1 Procedurální rozšíření SQL

Kromě základních příkazů pro vytváření a modifikaci dat obsahuje PSQL trigger, funkce, procedury, kurzory. To umožňuje přenést část aplikační logiky přímo do databází. Je **závislé na SŘBD** a její různé implementace se mnohdy velice liší:

- **PL/SQL** pro Oracle
- **Transact-SQL** (T-SQL) pro Sybase a MSSQL
- **PL/pgSQL** pro PostgreSQL
- **SQL PL** pro DB2

### 4.2 PL/SQL

- Založeno na jazyku ADA.
- Kód uložen a **prováděn v SŘBD**, může být sdílen více aplikacemi.
- **Nezávislý na aplikační platformě** (pouze na SŘBD).

#### 4.2.1 Proměnné, procedury a funkce

- **Proměnné**
  - Proměnné můžeme rozdělit do několika skupin, dle různých kritérií, nejčastějším dělením je podle **datového typu** na **číselné** (NUMBER), **stringové** (CHAR, VARCHAR2), **datumové** (DATE, TIMESTAMP).
  - Definujeme proměnné `part_no NUMBER(4); in_stock BOOLEAN;`
  - Přiřazení do proměnných je pomocí operátoru `:=`.
- **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.

- `CREATE [OR REPLACE] PROCEDURE jmeno_procedury [( jmeno_parametru [mod] datovy_typ , ... )] IS|AS definice lokálních proměnných BEGIN tělo procedury END [jmeno_procedury]`

- **Funkce**

- Na rozdíl od procedury **vrací hodnotu**. Kromě standardních funkcí (`TO.CHAR`, `TO.DATE`, `SUBSTR`, apod.) si můžeme definovat **vlastní funkce**.
- `CREATE [OR REPLACE] FUNCTION jmeno_funkce [( jmeno_parametru [mod] datovy_typ , ... )] RETURN navratovy_datovy_typ IS |AS definice lokálních proměnných BEGIN tělo procedury END [jmeno_procedury]`

#### 4.2.2 Dynamické a statické PL/SQL

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

#### 4.2.3 Výjimky, podmínky, cykly

- **Výjimky**

- Vznikají ručně i ze systému.
- Zpracování v bloku `EXCEPTION`
- Pro ruční vyvolání je nutné ji deklarovat (`DECLARE vyjimka EXCEPTION;`) a vyhodit (`RAISE vyjimka`)

- **Podmínky**

- `IF podminka1 THEN příkazy [ELSIF podminka2 THEN příkazy ] [ELSE příkazy ] END IF ;`

- **Cykly**

- `do while` - `LOOP` příkazy cyklu [`EXIT;` | `EXIT WHEN podminka ;`] `END LOOP;`
- `while do` - `WHILE` podminka `LOOP` příkazy cyklu `END LOOP;`
- `for` - `FOR` jmeno\_promenne `IN` [`REVERSE`] value1 .. value2 `LOOP` příkazy cyklu `END LOOP;`

#### 4.2.4 Kurzory, trigger, hromadné operace

- **Trigger**

- Kód spouštěný v reakci na **událost** (DML, DDL, systémové eventy).
- Pokud se pokusíme v triggeru číst či modifikovat stejnou tabulku dostaneme **mutating table error**.

- CREATE [OR REPLACE ] TRIGGER jmeno\_triggeru BEFORE | AFTER | INSTEAD OF INSERT [OR] | UPDATE [OR] | DELETE [OF jmeno\_sloupce] ON jmeno\_tabulky [REFERENCING OLD AS stara\_hodnota NEW AS nova\_hodnota] [FOR EACH ROW [WHEN (podminka )]] BEGIN příkazy END;

## • Kurzory

- 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:
  - \* **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:**
  - \* CURSOR kurzor IS select; - vytvoření kurzoru.
  - \* 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.

## • Vázané proměnné

- SŘBD kontroluje **jedinečnost dotazu**, pokud už byl dotaz v minulosti proveden, použije se **dříve použitý plán** dotazu místo nového vytváření plánu.
- Vázané proměnné umožňují **parametrizaci hodnot v dotazu**, odpadá tedy opětovné vytváření plánu pro stejný dotaz s jinou hodnotou.
- Lze používat i v dynamickém PL/SQL (pomocí **USING**).
- Použití i při volání z aplikace (podpora v C# i Java).

## • Hromadné operace

- Snížení režie na zotavení (zápis do logu) a aktualizace DB (datových struktur). Výsledkem je rychlejší vkládání záznamů do DB.
- Lze použít pro statické i dynamické SQL.
- BULK COLLECT
  - \* **Hromadné načtení** (navázání vstupní kolekce s PL/SQL enginem).
  - \* BULK COLLECT INTO collection\_name[, collection\_name]
- FORALL
  - \* Hromadná **operace** (navázání vstupní kolekce před posláním do SQL enginu)
  - \* FORALL index IN lower\_bound..upper\_bound [INSERT, UPDATE nebo DELETE];

- **Balíky**

- Obdoba kníhoven v programovacích jazycích.
- Specifikace balíku a následně tělo.

### 4.3 T-SQL (Transact-SQL)

Transact-SQL (T-SQL) je proprietární rozšíření jazyka SQL od společností **Microsoft** a **Sybase**, které Microsoft používá v produktu **Microsoft SQL Server**, Sybase Software pak v Adaptive Server Enterprise.

#### 4.3.1 Proměnné, procedury a funkce

- **Proměnné**

- Deklarace pomocí `DECLARE @TMP INT`.
- Inicializace pomocí `SET` nebo `SELECT`.

- **Podmínky**

- `IF <boolean condition> <statement> ELSE <statement>`
- Více příkazů v jedné větvi musíme obalit do `BEGIN/END`.

- **Cykly**

- `WHILE <Boolean expression> <code block>`

- **Transakce**

- **Začátek** - `BEGIN TRANSACTION <nazev_transakce>`
- **Konec** - `ROLLBACK` nebo `COMMIT`
- **Nastavení úrovně izolace** - `SET TRANSACTION ISOLATION LEVEL <level>`

- **Výjimky**

- Bloky `try/catch` (`BEGIN TRY/END TRY` a `BEGIN CATCH/END CATCH`).

- **Uložené procedury**

- Uloženy v SŘBD.
- Předkompilovány, tudíž jsou rychlejší.
- `CREATE PROC[EDURE] procedure_name [;number] [ @parameter data_type [VARYING] [= default] ] [OUTPUT] ] [ , . . . ] [ WITH RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION ] [FOR REPLICATION] AS sql_statement`

- **Uložené funkce**

- Není možné použít `try/catch`, `DML` atd.



- Musí vracet hodnotu.
- `CREATE FUNCTION [ schema_name. ] function_name ( [ @parameter_name [ AS ] [ type_schema_name . ] parameter_data_type [ = default ] [ READONLY ] [ ,...n ] ) RETURNS return_data_type [ WITH <function_option > [ ,...n ] ] [ AS ] BEGIN function_body RETURN scalar_expression END [ ; ]`

#### 4.3.2 Kurzory, trigger, dynamické SQL

- **Kurzory**

- `DECLARE cursor_name CURSOR FOR select_statement`
- **Musíme použít** sekvenci příkazů: `OPEN`, `FETCH`, `CLOSE`, `DEALLOCATE`
- Pro testování prázdného kurzoru používáme `@@FETCH_STATUS`

- **Trigger**

- `CREATE TRIGGER [ schema_name . ] trigger_name ON table | view [ WITH <dml_trigger_option > [ ,...n ] ] FOR | AFTER | INSTEAD OF [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] [ WITH APPEND ] [ NOT FOR REPLICATION ] AS sql_statement [ ; ] [ ,...n ] | EXTERNAL NAME <method specifier [ ; ] >`

- **Dynamické SQL**

- Podobné PL/SQL, pomocí příkazu `sp_executesql`
- `sp_executesql [@stmt =] stmt [ , [@params =] N'@parameter_name data_type [ ,... n ] ' , [@param1 =] 'value1 ' [ ,... n ] ]`

#### 4.4 Rozdíly PL/SQL vs T-SQL

- **T-SQL neposkytuje operátory** jako `%TYPE` a `%ROWTYPE` pro získání datových typů existujících záznamů.
- **T-SQL nepodporuje** `CREATE OR REPLACE PROCEDURE`, což nás nutí k tomuto zápisu: `/*CREATE*/ ALTER PROCEDURE.`
- **T-SQL podstatně omezuje konstrukce**, které můžeme využívat ve funkcích.
- V **T-SQL** musím při práci s kurzory používat `OPEN`, `FETCH`, `CLOSE`, `DEALLOCATE`.
- **T-SQL** nás nutí k dvojitému `FETCH` u kurzorů.
- V **T-SQL** musíme definovat u parametrů procedur a funkcí **délku datového typu** (pokud se u datového typu udává).
- V **T-SQL** musíme více příkazů v jedné větvi obalit do `BEGIN/END`.

## 5 Základní fyzická implementace databázových systémů: tabulky a indexy; plán vykonávání dotazů.

### 5.1 Fyzická implementace

Fyzická implementace definuje **datové struktury** pro základní logické objekty:

- **tabulky**,
- **indexy**,
- **materializované pohledy** (materialized views),
- **rozdělení dat** (data partitioning) – data s dlouhou historií. Fyzické rozdělení datových struktur a souboru na více částí.

Fyzická implementace tedy **řeší uložení dat na nejnižší úrovni databáze**. Na úrovni databáze můžeme ovlivnit výběr efektivnějšího plánu vykonávání dotazu případně **dobu vykonávání operací**.

- **ROWID** – jedinečné číslo **označující záznam tabulky**.
- Větší část teorie fyzického návrhu DB je platná pro libovolné SŘBD, dále se musíme řídit **doporučením výrobce**.
- Každé SŘBD obsahuje specifickou reprezentaci tabulek a indexů:
  - **CREATE TABLE** - vytvoření tabulky typu **halda** (Oracle), **shlukování záznamů** (SQL Server 2012).
  - **CREATE INDEX** - vytvoření **B+-stromu**, kde každá položka odkazuje na ROWID záznamu v tabulce.

### 5.2 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 doprostřed tabulky musím posunout milion záznamů). Základní datové struktury:

- **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.  $\text{Stranka.velikost} = X * \text{Blok.velikost}$  (je-li velikost bloku = 4KB je odpovídá velikost stránky násobkům 4KB).
- **Datový soubor** – fyzický prostor na disku s daty.

## 5.3 Typy tabulek

### 5.3.1 Heap Table

- Implicitní pro CREATE TABLE (v MSSQL), záznamy nejsou nijak uspořádány.
- **Stránkové perzistentní pole** s velikostí **bloku** nejčastěji **8kB**.
- Záznamy pouze **označovány jako smazané**, pro fyzické smazání slouží speciální operace **shrinking**.
- Při vkládání záznam uložen na **první volnou pozici v tabulce** nebo na **konec pole**.
- Složitost operací: neefektivní vyhledávání  $O(n)$ , velmi **efektivní vkládání**  $O(1)$  i **využití místa**.

### 5.3.2 Shlukování záznamů (Data Clustering)

- Záznamy v **datovém souboru** jsou **seřazeny podle zvoleného klíče**, pro implementaci nejčastěji využita nějaká **varianta B-stromu**.
- Listové uzly stromu (bloky) obsahují **kromě klíče i další záznamy** tabulky.
- Používá se všude, kde potřebujeme **získat i hodnoty ostatních atributů** (kromě klíče - např. SELECT neklíčových atributů).
- **Zhoršený výkon INSERT** (data se musí zatřizovat).
- **Oracle:** Index Organized Table (IOT), **SQL Server:** Clustered Index.

### 5.3.3 Hašovaná tabulka (Hash Table)

- Záznamy se stejnou hashovanou hodnotou jsou uloženy ve stejném nebo velmi blízkém bloku.
- Trochu **plýtvá místem**.
- Musíme znát předem velikost záznamů (alespoň přibližně).

### 5.3.4 Materializované pohled (Materialized views)

- Uložené **výsledky dotazů**, které bývají často v DB vyhodnocovány.
- Jde spíše o **fragment z tabulky** či několika tabulek.

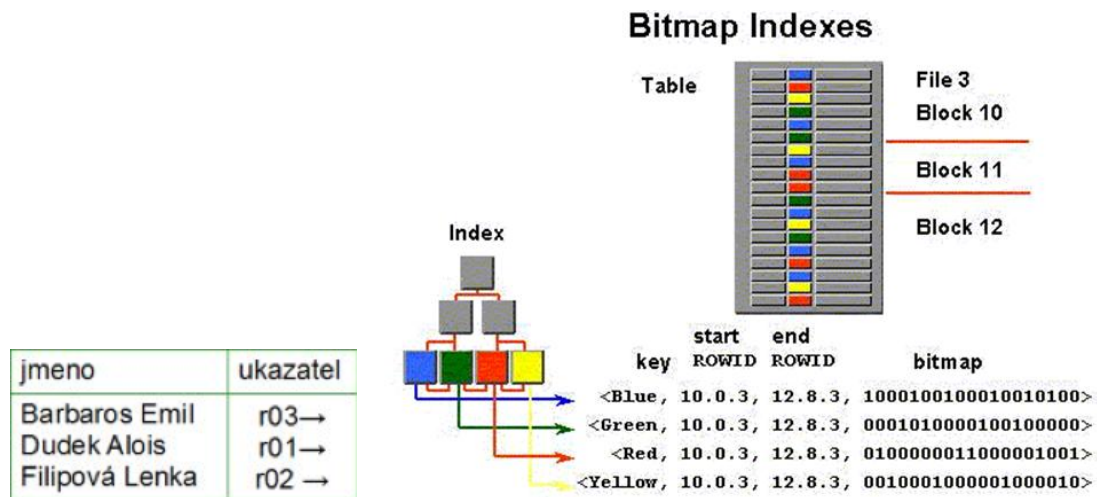
## 5.4 Indexy

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. Index umožňuje **rychlé vyhledávání** dle klíče, **ROWID** pak odkazuje na **kompletní záznam v heap tabulce**. `CREATE [BITMAP] INDEX login ON Student;`

- **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)** – klasické index popsány níže.
- **Fulltextový index** – používá se pro optimalizaci fulltextového vyhledávání v daném sloupci.

#### 5.4.1 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.



#### 5.4.2 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 řídicí tabulky a zároveň i závislé záznamy. (v normálním případě obsahuje diskový blok pouze záznamy jedné tabulky).

#### 5.4.3 Kandidáti na index

- **Primární klíče a cizí klíče.**
- Pokud je index používán pro nalezení malého počtu záznamů.
- Pokud index pokryje jeden nebo více častých dotazů.
- Atributy často se vyskytující v konstrukci WHERE.

## 5.5 Vykonávání dotazu

**Ovlivnění času vykonávání dotazu** - parametrizované dotazy, hromadné operace, nastavení transakcí. Na úrovni DB můžeme ovlivnit výběr efektivnějšího plánu vykonávání dotazu, případně dobu vykonávání operací -> **fyzický návrh DB**. 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** (materializované pohledy – fragmenty/výsledky dotazů).
- 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.
- **Optimalizátor** – transformační pravidla (převádí výraz na ekvivalentní). Fáze transformace/přepsání dotazu (query rewrite). Dotaz tedy není ve skutečnosti vykonán přesně tak, jak byl zadán!

### 3. Výběr nízkoúrovňových 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**.

## 5.6 Ladění dotazů

- Po vybrání nejlevnějšího plánu je **dotaz proveden** a uživateli vrácen výsledek.

- **Plán** lze v SŘBD většinou **zobrazit** - operace jako průchod tabulkou, přístup k indexu, třídění spojení atd. To lze využít pro **odladění dotazu** - např. uvidíme, že musíme použít index na neindexovaný atribut.
- 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**.

#### 5.6.1 Přístupy k ladění

- **Proaktivní** – Analyzujeme fyzický návrh a provádíme změny k lepšímu fungování.
- **Reaktivní** – Reagujeme na problém.

#### 5.6.2 Ladění výkonu SŘBD

- **HW** – RAID, RAM, CPU (Poslení možnost, je to drahé, efektivnější je **vyladit fyzický návrh**).
- **Parametry SŘBD** – Velikosti **cache**, maximum zámků, atd. Nutné optimalizovat na určité použití.
- **ORM** – Minimum SQL příkazů a objemu přenášených dat. Úroveň izolace transakce.

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

### 6.1 Objektový datový model

**Objektový datový model** představuje 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.

Objektové databázové systémy umožňují využití hostitelského objektového jazyka jako je třeba C++, Java, nebo Smalltalk přímo na objekty (skládají se z **atributů a metod**) "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 (Object Database Management System) 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.

### 6.2 Objektově relační datový model

Objektově orientované DB → **nákladná migrace relačních dat**, proto vznikly **objektově-relační rysy** v relačních DB. (standart **SQL99** - víceméně se nedodržuje, musíme se bavit o konkrétním SŘBD - Oracle).

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.

#### 6.2.1 Prvky objektově-orientovaných SŘBD

- Uživatelsky definované **datové typy** (s atributy i metodami).
- Uložené **procedury, trigger**y (přenesení funkcionality na server).
- Kolekce (**pole** proměnné délky, **vnořené tabulky**).
- Datový typ **ukazatel, reference** a **dereference**.
- **Dědičnost** – `CREATE TYPE student_type UNDER person_type ( ... OVERRIDING MEMBER FUNCTION show...`

### 6.2.2 Výhody OOSŘBD

- **Objektové typy** a jejich **metody** jsou **uloženy spolu s daty** v databázi -> není nutné vytvářet podobné objekty v každé aplikaci.
- Na data je možné nahlížet z **relačního i objektového pohledu**.
- **Objekty** mohou **reprezentovat vazby**, kdy entita se skládá z jiných entit.
- **Metody** jsou **spouštěny na serveru** – nedochází k neefektivnímu přenosu dat po síti, část výkonu přenesena na server.
- Programátor může přistupovat k množině objektů jako by se jednalo o jeden objekt.

### 6.3 Typy tabulek a datové typy

- **Objektové tabulky** – obsahují pouze objekty, **každý záznam = objekt** (řádkový objekt).
  - **Objekty sdíleny dalšími objekty** by měly být uloženy v objektových tabulkách (mohou být **referencovány**).
  - Buď tabulka s **1 sloupcem objektů** (nad kterými lze provádět objektové metody), nebo tabulka obsahující atributy obj. dat. typu, nad kterou lze provádět relační operace (kompromis mezi O/RDM).
- **Relační tabulky** – Obsahují **objekty spolu s ostatními daty**, mluvíme o tzv. **sloupcovém objektu**.

#### 6.3.1 Objektově relační datové typy a metody

- Data/atributy, operace/metody.
- Normální používání **objektových typů** při definici atributů – `INSERT INTO contacts VALUES (person_type(65, ,John', ,Smith'), 2014-05-29)`
- **Objektový identifikátor (OID)** - identifikuje objekty objektových tabulek, není přístupné přímo, ale jen pomocí reference (typ REF). U relačních tabulek s objekty OID nepotřebujeme.
- **Reference na objekt** - REF, ukazuje na objekty stejného typu nebo hodnota NULL, **nahrazuje FK (cizí klíč), asociace/vztahy**. Pokud tabulka obsahuje referenci na objekt jedné tabulky, lze využít IO: `SCOPE IS tabulka`.
- **Dereference** - `SELECT Deref(e.manager) FROM emp_person_obj_table e`; **Implicitní dereference**: `SELECT e.name, e.manager.name FROM ....`

#### 6.3.2 Kolekce a dědičnost

Používané pro více hodnotové atributy nebo vazby M:N.



- **Pole** – VARRAY – variable-size array - pole pevné délky s definovanou kapacitou aktuální velikostí. `DECLARE TYPE Calendar IS VARRAY(366) OF DATE.`
- **Zahnížděná (nested) tabulka** – pole proměnné délky, záznamy ukládány bez ohledu na pořadí, ale po načtení z disku jsou záznamy očíslovány (nezachová hodnotu indexu, jen udrží pořadí).
- **Asociativní pole** – jako hash table - obsahuje dvojice <key, value>, kde key = číslo/řetězec, klíče jsou indexovány (žádné sekvenční vyhledávání). Nelze uložit do tabulky, spíše pro malé množství záznamů.
- **Hierarchická dědičnost** – v SŘBD můžeme vytvářet hierarchie typů pomocí dědičnosti. Tato vlastnost nám pak umožní využít všechny rysy objektově-orientovaných technologií, např. mnohotvárnost, polymorfismus.

### 6.3.3 Typy metod

- **Členské, statické, konstruktor** (pro každý obj. typ definován implicitní).
- **Volání:** `SELECT c.contact.getID() FROM contacts c;`

```
-- Vytváření objektu s atributy a metodou
CREATE TYPE person_type AS OBJECT (idno NUMBER, first_name VARCHAR2(20),
last_name VARCHAR2(25) , MAP MEMBER FUNCTION get_idno RETURN NUMBER);

-- Definice těla metody
CREATE TYPE BODY person_type AS MAP MEMBER FUNCTION get_idno RETURN
NUMBER IS BEGIN RETURN idno; -- vraci id END; END;

-- Objektové datové typy lze 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') ;
```

## 6.4 XML (eXtensible Markup Language)

- Značkový jazyk - W3C standard.
- 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** vnořeném uvnitř (ukládáné informace).

## 6.5 Dotazovací jazyky XML

Dotazovací jazyk XML by měl umět: vybrat požadované části dokumentů, transformovat je do požadované podoby, manipulovat s kolekcemi dokumentů a další vlastnosti specifické pro konkrétní problémovou doménu.

### 6.5.1 XPath

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.

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

### 6.5.2 XQuery

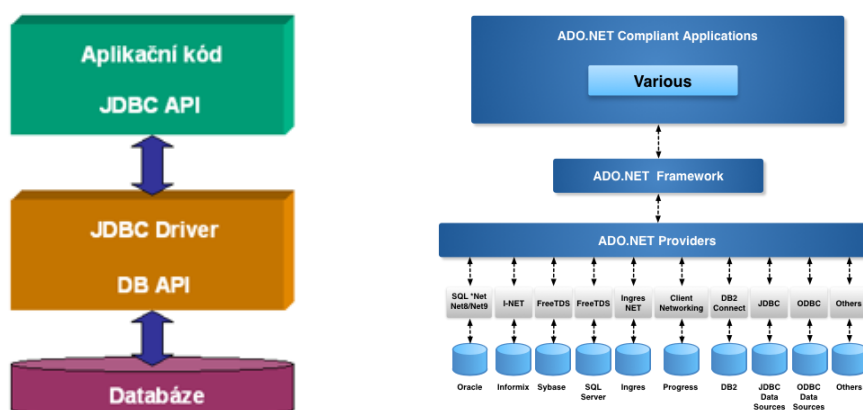
Připravovaný standard W3C (verze 1.0). XQuery **je silně typovaný jazyk**. Vychází z jazyka Quilt, inspirace XPath 1.0 a pro výběr částí dokumentů používá XPath 2.0.

- `$` – 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
```

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

### 7.1 Datová vrstva IS

Datová vrstva informačního systému **odděluje aplikaci od databáze**. Jde o třídy a funkce zajišťující komunikaci s databází. **Překonává propast mezi SŘBD a programovacím jazykem**.



- **Nástroje** – programovací jazyky + **SQL**, staví na **API pro dotazování DB** (JDBC, ADO.NET), **embedded SQL** - hostitelský jazyk obsahuje SQL.
  - **ODBC** (Open DataBase Connectivity) – 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, DB2, Ingress, ... – se s nimi přes JDBC pracuje stejně.
  - **ADO.NET** (Active Data Object) – specifikace (knihovna) pro přístupu k datový 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.
- **Speciální prostředí** – implementováno výrobcem SŘBD (Oracle Forms, APEX). Jednoduché, šité na míru, není připravené pro velké projekty a týmový vývoj.
- **Architektury** – jsou **komplikované, vícevrstvé**, ale umožňují jednodušší vývoj a zapojení více vývojářů. (3-vrstvá architektura => Business, Data, Presentation layer).

#### 7.1.1 JDBC

- Rozhraní pro **unifikovaný přístup k datům v DB** na platformě **Java**.

- Inspirováno rozhraním **ODBC**.
- Zprostředkování komunikace aplikace s konkrétním typem DB (ovladače k většině).
- **Dotazovací jazyk – SQL**. Předá se DB, ovladač vyhodnotí přímo.
- **Schéma provedení dotazu**: 1) Připojení k datovému zdroji. 2) Inicializace. 3) Vytvoření a provedení dotazu. 4) Získání výsledku. 5) Ukončení transakce. 6) Odpojení od datového zdroje.
- Hlavní třída **Connection** - odesílá **Statementy** a obdrží **ResultSety**.
- **Podpora transakcí** (implicitně, každý příkaz jedna transakce, jsou vázány k instanci connection).

### 7.1.2 ADO.NET

- **Connection** spouští **Commandy**, výsledkem je **DataSet** (použití **DataAdapteru**) nebo čistá data čtená **DataReaderem**.
- **DataSet** -> **DataTable[]** -> **DataRow[]** -> **DataColumn[]**
- **DataRelation** vazba mezi **DataSety** s vazebním **DataColumn**.
- Umožňuje **parametrizovat commandy**.
- Třída **TransactionScope** pro distribuované transakce.

## 7.2 ORM (Objektově-relační mapování)

Programovací technika **zpřístupňující relační či objektově-relační data pro objektové prostředí**. 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. Vlastnosti ORM rámců:

- Práce s objektovým modelem (přenositelnost mezi různými SŘBD).
- Překonává propast mezi SŘBD a programovacím jazykem.
- **Rychlejší vytváření aplikací** vs Menší výkon aplikace.
- Nevyužívá všechny vlastnosti SŘBD (efektivita, bezpečnost apod.).
- Typová kontrola (nevznikají chyby v SQL).
- Jednodušší testování.

### 7.2.1 Pravidla ORM

- **Minimalizace počtu dotazů** zasílaných na server.
- **Minimalizovat objem dat** získaných z DB, stahovat pouze ta data, které potřebujeme, a které jsou zobrazeny uživateli (Lazy Loading).

### 7.2.2 Vlastní implementace ORM

- Využití **API pro komunikaci s DB** (JDBC, ADO.NET).
- Používá se **SQL**.
- **Výhody:** plná kontrola nad výkonem (stahujeme jen to co chceme) a prováděnými příkazy. Můžeme využít konkrétní prvky daného SŘBD.
- **Nevýhody:** aplikace je závislá na určitém typu SŘBD.

### 7.2.3 Automatizované

- Frameworky **Hibernate** (Java), **LINQ2SQL**, **EntityFramework** (ASP.NET).
- Může výrazně **degradovat výkon** (musí provádět interpretaci do SQL, vytváří objektový model atd.).
- Většinou mají vlastní **cache**. Většinou jsou **nezávislé na daném SŘBD**.

## 7.3 Bezpečnost

**SQL Injection** - zranitelnost vznikající při nedostatečném **ošetření vstupů užívaných** v SQL dotazech. Např.: při přihlašování na webových stránkách zadá útoční příkaz SQL, které změní funkčnost dotazovaného příkazu. **Řešení:**

- **Parametrizované dotazy** (hodnoty jsou předány zvlášť, nemůže dojít k úpravě).
- Uložené procedury.
- Ošetření vstupů (`htmlspecialchars`, `mysql_real_escape_string`) - zneužití např. pomocí jiného kódování!
- Uživatel musí zadávat komplikovaná (neslovníková) hesla, **časové omezení počtu** pokusů. Logování neúspěšných pokusů o autentizaci.

## 8 Distribuované SŘBD, fragmentace a replikace.

### 8.1 Distribuované SŘBD

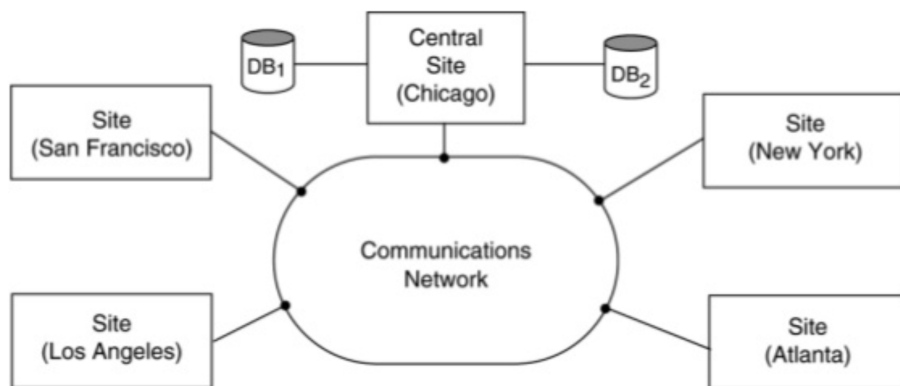
Distribuovaný databázový systém (DDBS) je tvořen **distribuovanou bází dat** a **programovým vybavením**, skládajícím se z **lokálních SŘBD** a dalších programů potřebných k jejich **koordinaci a řízení**, k zabezpečení celosystémových úloh spojených s přístupem uživatelů k DDBS, s udržováním integrity a provozuschopnosti v prostředí počítačové sítě. **Uživatelé se jeví celá distribuovaná báze jako by byla lokální** (na jednom místě) a přistupuje k ní stejným způsobem.

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

#### 8.1.1 Centralizované

Mají popis a řízení DDBS soustředěno na **1 centrální počítač**. Toto centrum nemusí být v centru počítačové sítě. Jsou zde soustředěny **popisky všech dat** tvořících DDBS a **centrálně se řídí: přístup k datům, provádění změn** ve struktuře dat, **provádění a synchronizace transakcí** + všechny další činnosti systému.

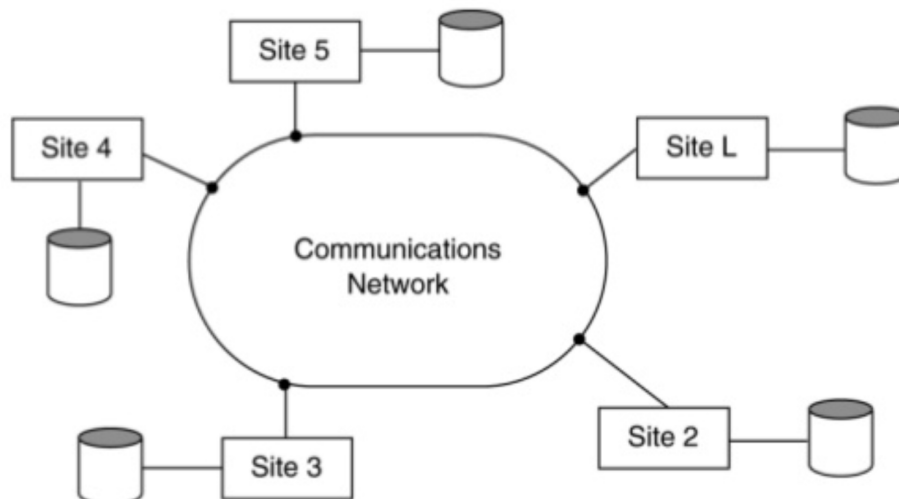
- **Výhody** – jednoduchost řízení.
- **Nevýhody** – vysoké náklady na komunikaci, pomalost (každý přístup k datům musí být povolen centrem) a nebezpečí výpadku tohoto jednoho prvku.



#### 8.1.2 Decentralizované

Jsou tvořeny počítačovou sítí, kde žádný uzel nemá privilegované postavení. **Všechny počítače mají stejné informace o DDBS a stejnou odpovědnost za integritu DDBS.**

- **Výhody** – větší stabilita (výpadek počítače způsobí max. ztrátu přístupu k lokálním datům).
- **Nevýhody** – složitější algoritmy pro řízení transakcí než v centralizovaných DDBS.



## 8.2 Rozmístění dat (fragmentace a replikace)

U DDBS jsou nejvíce časově náročné operace přesunu dat po komunikační lince. Vzhledem k vyšší možnosti poruch je třeba relace ukládat tak, aby jejich **dostupnost byla zajištěna** i v případě výpadku části sítě. Relací zde rozumíme logický celek, ale fyzicky i více souborů. Používají se **2 hlavní způsoby** fyzické implementace:

- **Replikace** – (přítomnost duplicitních dat) **uchovávání kopií relací v různých uzlech, porucha uzlu neznemožní přístup k relaci** (díky tomu se dist. databáze i samozálohuje), problém je v aktualizaci všech kopií relací. Proto se v DDBS ukládají v kopiích ta data, ke kterým je třeba **rychlý přístup** a nejsou často aktualizována a jsou velmi důležitá.
- **Fragmentace** – znamená **rozložení relací na menší části** (fragmenty), které jsou umístěny v různých uzlech sítě. Může jít o **horizontální** fragmentaci, kdy se v různých uzlech sítě ukládají části relace rozložené do skupin řádků nebo **vertikální** fragmentaci, kdy se v různých uzlech ukládají různé projekce relací. Fragmentace se provádí, aby se **původní relace získala zpět standardními operacemi nad relační databází** (sjednocení, spojení).

Obě metody se často kombinují tak, že se v distribuované bázi uchovávají kopie fragmentů v různých uzlech. V distribuované databázi musí být unikátní jména všech položek. Tedy ani 2 lokální položky nesmí používat stejná jména pro různé položky. Jednou z možností jak tento problém vyřešit je **centrální slovník**, ovšem ten je zase úzkým místem systému.