Databázové systémy

Tomáš Skopal

- úvod
- konceptuální datové modelování

Osnova

- organizační záležitosti
- přehled kurzu
- konceptuální datové modelování

Organizační záležitosti

- Povinnosti
  - zápočet = složit zápočtový test (≥ 60% bodů)
    - uskuteční se 23. listopadu místo přednášky
    - bude pouze **jeden opravný** termín !! (tj. celkem 2)
  - zkouška = složit zkouškový test
    - termíny pouze do konce ZS
  - účast na přednáškách a cvičeních nepovinná, nicméně silně doporučená
    - materiály na webu vyučujícího nejsou vyčerpávající zdroj
- web: siret.ms.mff.cuni.cz/skopal/courses.htm

Studijní zdroje

- literatura
  - Pokorný, Halaška: Databázové systémy, skripta FEL ČVUT, 2003
  - Halaška, Pokorný: Databázové systémy – cvičení, skripta ČVUT, 2002
  - **Ramakrishnan, Gehrke: Database Systems Management, McGraw-Hill, 2003**
    (k dispozici v knihovně na Malé straně)
- slidy a příklady na stránkách cvičících/přednášejícího
- Google – univerzální zdroj **www.google.com**
- Wikipedia – univerzální zdroj **www.wikipedia.cz**
- stránky o různých RDBMS (vhodné pro syntaxi SQL, transakce, apod.) – pozor, většinou odchylky od ANSI SQL!
  - Oracle (**www-db.stanford.edu/~ullman/fcdb/oracle/or-nonstandard.html**)
  - MS SQL Server (**msdn2.microsoft.com/en-us/library/ms189826.aspx**)
  - PostgreSQL (**www.postgresql.org/docs/8.1/interactive/sql.html**)
  - MySQL (**dev.mysql.com/doc/refman/5.0/en**)
  - a mnoho dalších...

Přehled kurzu – o čem to (ne)bude

- kurz zahrnuje:
  všeobecný přehled základů klasických databázových technologií – „od všeho něco“
  - konceptuální datové modelování
  - relační model a relační datové modelování
  - fyzická implementace databází
  - transakce
  - úvod do databázových aplikací
- kurz nezahrnuje:
  - konkrétní DBMS sw. balík(y)
  - multimediální, textové a XML databáze
  - dobývání znalostí z databází
  - data warehousing, OLAP
  - zmíněné oblasti (a další) pokrývají specializované kurzy DBI*

Software ke cvičením

- **ERtoS** - **editor ER diagramů**
  - bakalářský projekt Hany Kozelkové.
  - Text práce může posloužit k teorii o ER modelování
- **DatAlg -** aplikace **databázové algoritmy** (relační)
  - umožní procvičit si funkční závislosti, normální formy, dekompozici a syntézu.
  - bakalářský projekt Dany Soukupové (vedl dr. Říha).
- **ReAl** - aplikace vyhodnocování výrazů v **relační algebře**.
  - bakalářský projekt Martina Lysíka.
  - Text práce může posloužit teorii o RA.
- **TCR** - aplikace vyhodnocování výrazů v **nticovém relačním kalkulu**.
  - bakalářský projekt Viliama Sabola.
  - **Text práce** může posloužit teorii o RK.
- **TSimul** - aplikace **rozvrhování a běhu transakcí**
  - bakalářský projekt Dung Nguyen Tiena.
  - Text práce může posloužit teorii o transakcích.
- **MS SQL Server 2005 Express Edition**
  - pro pokusy s SQL
  - volně stáhnutelný

(vše uvedené je ke stažení z webu přednášejícího)

DBI025 = základ pro výběrové DB kurzy

- Dotazovací jazyky I, II, Datalog
- Organizace a zpracování dat I, II
- Databázové aplikace, administrace Oracle, Caché I, II
- Dobývání znalostí (z databází)
- Vyhledávání v multimediálních databázích
- Transakce
- Stochastické metody v databázích, Statistické aspekty dobývání znalostí z dat
- (Pokročilé) technologie XML

Databázový systém

- databáze (data)
  - je logicky uspořádaná (integrovaná) kolekce navzájem souvisejících dat.
  - je sebevysvětlující, protože data jsou uchovávána společně s popisy, známými jako metadata (také schéma databáze).
- systém řízení báze dat (SŘBD, angl. database management system – DBMS)
  - je obecný softwarový systém, který řídí sdílený přístup k databázi, a poskytuje mechanizmy pomáhající zajistit bezpečnost a integritu uložených dat
- administrátor (správce DB systému)

Proč databázové systémy?

- sdílení dat
- unifikované rozhraní a jazyk(y) definice dat a manipulace s daty
- znovuvyužitelnost dat
- bezespornost dat
- snížení objemu dat (odstranění redundance)

Mapa předmětu Databázové systémy (v1.1)

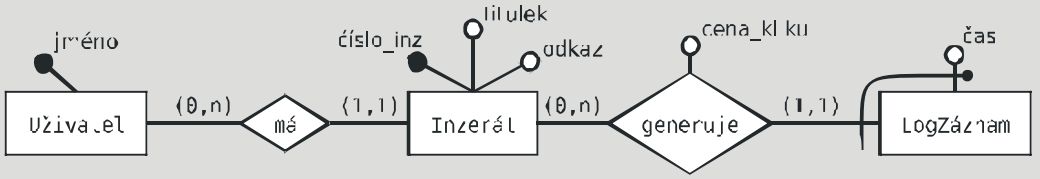
Obecný postup návrhu DB:

Prvotní formulace DB úlohy

Konceptuální datové modelování

Příklad: Aplikace "Webový inzertní server". Uživatel vkládá inzeráty, ke kterým se eviduje počet zobrazení inzerátu (kliknutí na inzerát).

ER schéma



Transformace ER schématu

objektové, objektově-relační schéma, ...

Relační schéma databáze

UŽIVATEL(jméno),
INZERÁT(číslo_inz, titulek, odkaz, jméno),
LOGZÁZNAM(číslo_inz, čas, cena_kliku)

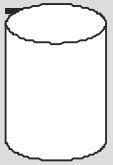
IO: INZERÁT.číslo_inz je cizí klíč v LOGZÁZNAM

funkční závislosti, normální formy relací, dekompozice, syntéza

Definice dat (SQL)

```
...
CREATE TABLE LOGZÁZNAM (
    číslo_inz, INTEGER,
    čas DATETIME,
    cena_kliku DOUBLE,
    KEY(číslo_inz,čas),
    INDEX(číslo_inz),
    INDEX(čas),
    FOREIGN KEY (číslo_inz) REFERENCES
    INZERÁT(číslo_inz) ON DELETE CASCADE);
...
```

Fyzické schéma (organizace souborů, implementace indexů, ...)



Manipulace s daty - modifikace a dotazování (SQL)

```
INSERT INTO UŽIVATEL (jméno) VALUES ("Kovář");

SELECT SUM(cena_kliku)
FROM UŽIVATEL, INZERÁT, LOGZÁZNAM
WHERE UŽIVATEL.jmeno = 'Novotný' AND
UŽIVATEL.jmeno = INZERÁT.jmeno AND
INZERÁT.číslo_inz = LOGZÁZNAM.číslo_inz;
```

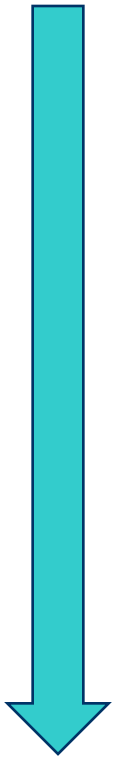
relační algebra, relační kalkul

připravili Tomáš Skopal a Leo Galamboš

Zdroj příkladu:
http://kocour.ms.mff.cuni.cz/~galambos/

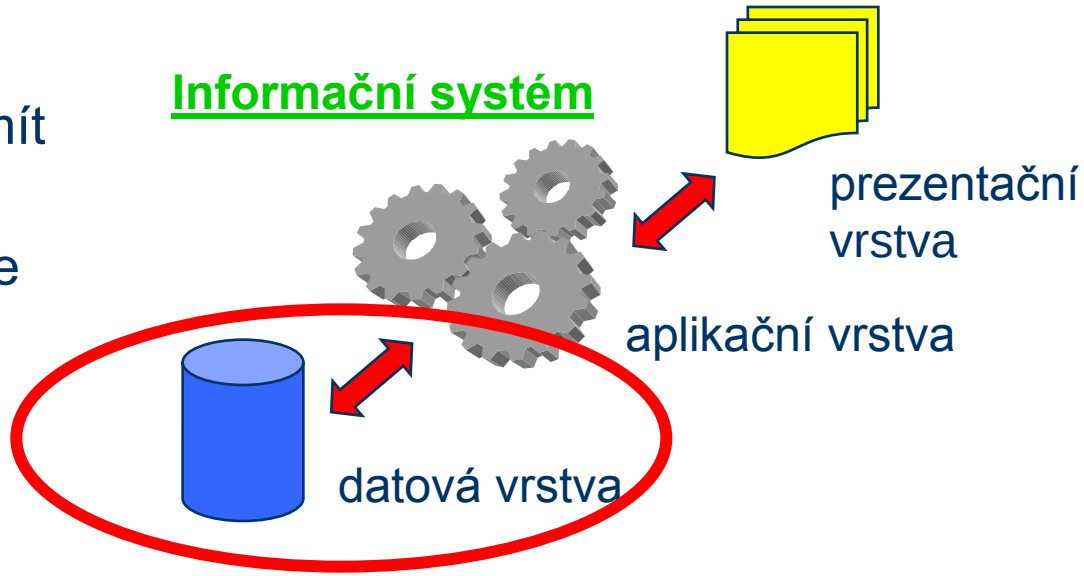
Tři vrstvy modelování databáze

- **Konceptuální model**
  - Nejvyšší úroveň abstrakce, modelujeme datové entity jako objekty reálného světa, nestaráme se o implementaci a architekturu
  - Příklad: ER modelování, UML modelování
- **Logický model**
  - Prostřední úroveň abstrakce, modelujeme entity jako struktury v konkrétním logickém modelu, pojmy konceptuálního modelování dostávají konkrétní podobu (stále se nezajímáme o efektivní implementaci)
  - Příklad: relační, objektově-relační, objektový model
- **Fyzický model**
  - Nízká úroveň abstrakce, implementace logického modelu ve specifických technických podmínkách. Optimalizace výkonu SŘBD a uložení dat tak, aby manipulace s nimi byla rychlá, zabezpečená a škálovatelná.
  - Příklad: management datových souborů, indexování, transakční zpracování



Konceptuální datové modelování

- datová analýza (ne funkční analýza)
  - zpravidla následuje po funkční specifikaci a analýze informačního systému (ta řeší funkcionalitu systému)
  - modelování schématu databáze
  - modelování „datové reality“ (jaká budeme mít v IS data)
  - pohled uživatele (analytika)

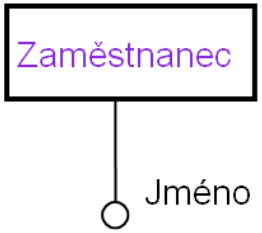


ER modelování

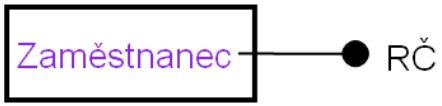
- de-facto standard pro datové modelování
- pro „plochá“ formátovaná (strukturovaná) data
  - objektové, relační a objektově-relační databáze
  - nevhodné pro multimediální data, XML, text
- E-R (entity-relationship) modelování
  - dva typy „objektů“ – entity a vztahy (mezi entitami)
  - ER model databáze definuje její **konceptuální schéma**
  - ER modelování v DB je obdobou UML v OOP
    - UML se stále více prosazuje i v DB, ale zde je to „kanón na vrabce“ (komplexní jazyk)

Entitní typ

entitní typ



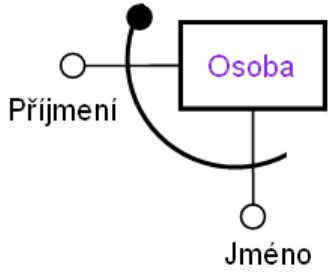
entitní typ s atributem
(atribut je dílčí datový typ náležící entitě)



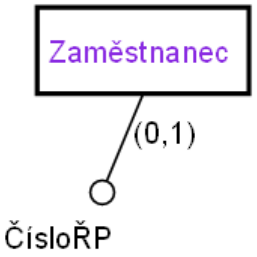
entitní typ s identifikátorem (identifikátor je atribut jednoznačně identifikující instanci entitního typu)

Instance entitního typu je jednoznačně určena, tj. vždy existuje identifikátor (pokud není explicitně označen, jsou složeným identifikátorem všechny atributy)

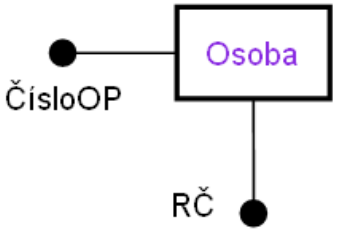
Entitní typ



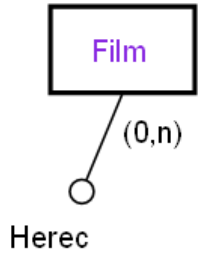
entitní typ s víceatributovým identifikátorem (instanci entity identifikuje kombinace hodnot atributů)



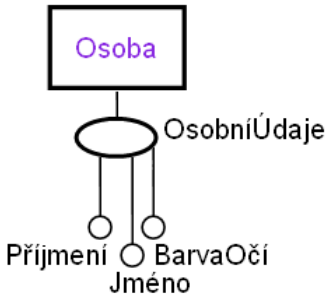
entitní typ s nepovinným atributem



entitní typ s více jednoatributovými identifikátory (každý je identifikátor nezávisle na ostatních)



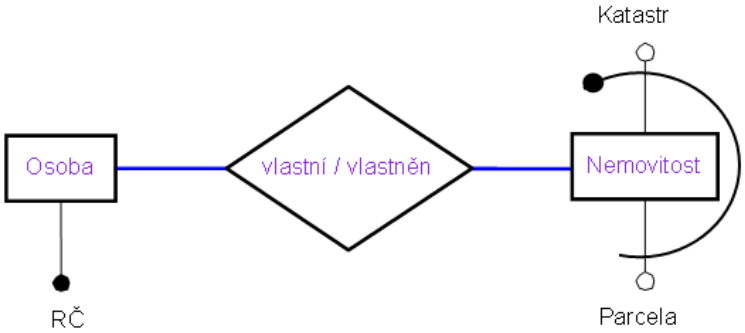
entitní typ s vícehodnotovým atributem



entitní typ se složeným (strukturovaným) atributem

Vztahový typ

vztahový typ (obecně)



binární vztah

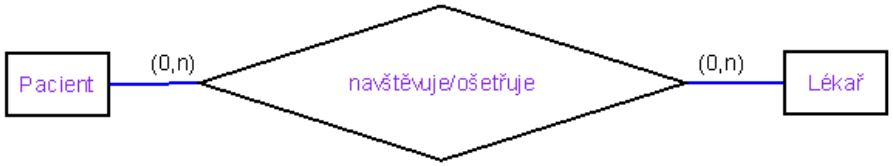
Vztahový typ

vztahový typ s kardinalitami
1:N (one-to-many)

Vztah interpretujeme jako:
“Kniha je půjčena maximálně jednomu studentovi.”
a
„Student má vypůjčeno několik (nebo žádnou) knihu.“

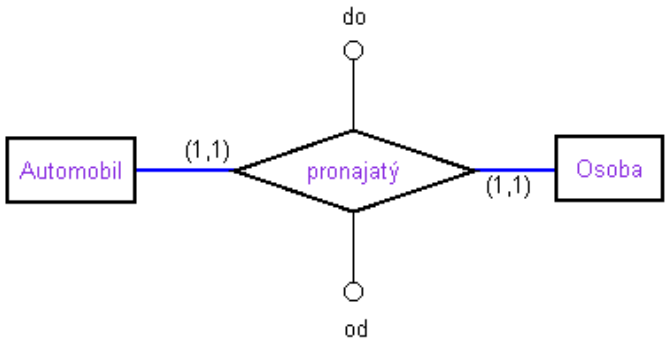
Vztahový typ

vztahový typ s kardinalitami 1:1 (one-to-one)



vztahový typ s kardinalitami M:N (many-to-many)

Vztahový typ

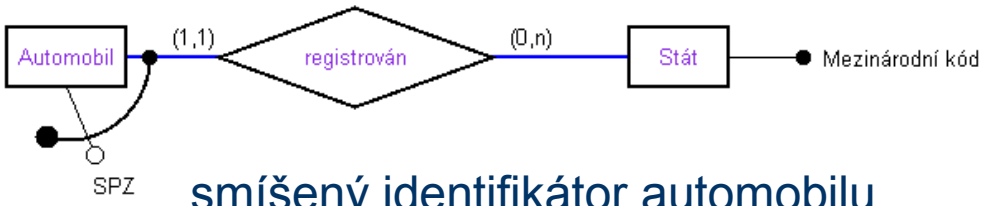


vztahový typ s atributy (nesmí být identikátor, ani jeho součást)

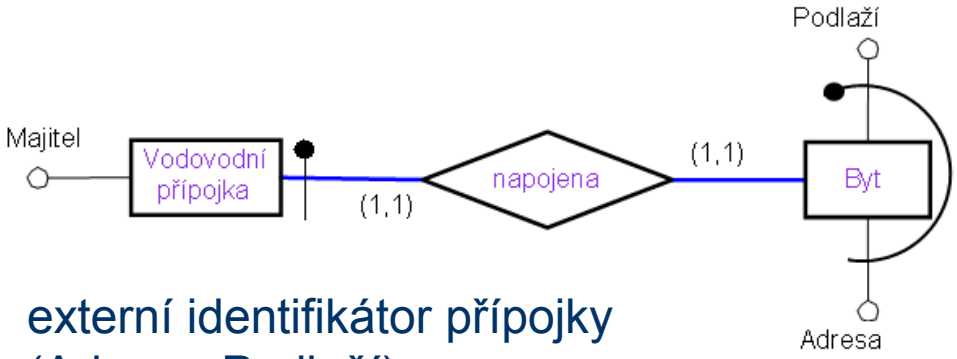
Instance vztahového typu je jednoznačně určena identifikátory entit ve vztahu.

Povinné členství ve vztahu (1,*), nepovinné (0,*)

Slabý entitní typ



smíšený identifikátor automobilu (SPZ, Mezinárodní kód)



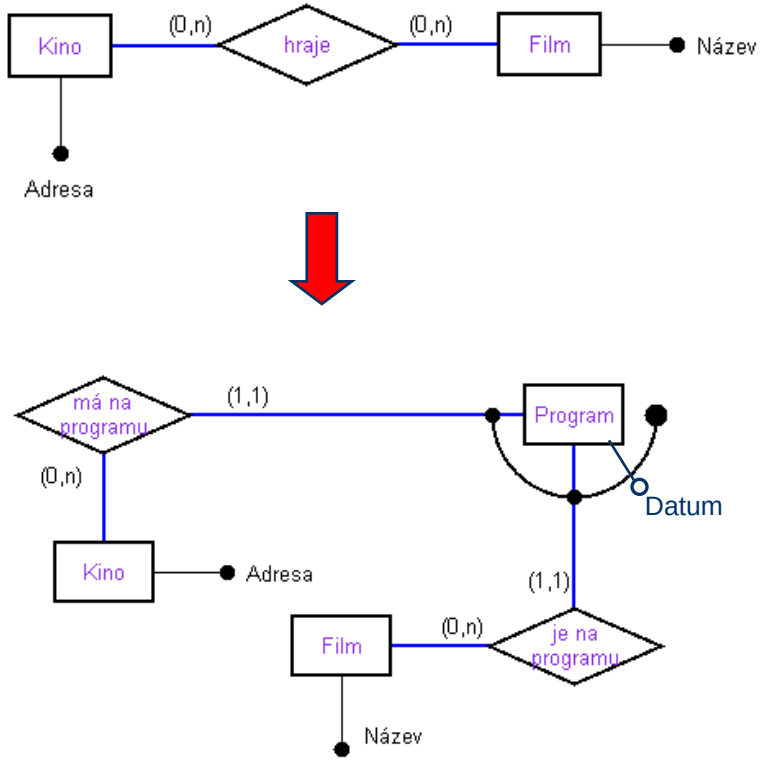
externí identifikátor přípojky (Adresa, Podlaží)

slabý entitní typ – je (spolu)identifikován zvenčí – všemi identifikátory entit vstupujících do vztahu

vstupuje do vztahu **vždy s kardinalitou (1,1)**

tzv. identifikační závislost (implikuje existenční závislost, což je integritní omezení zajišťující existenci identifikačního vlastníka)

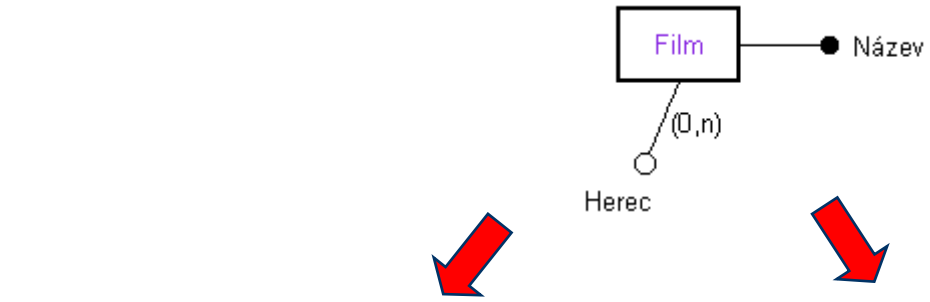
Průnikový entitní typ

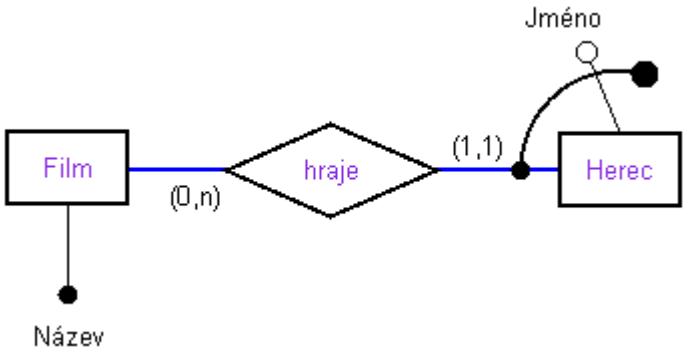


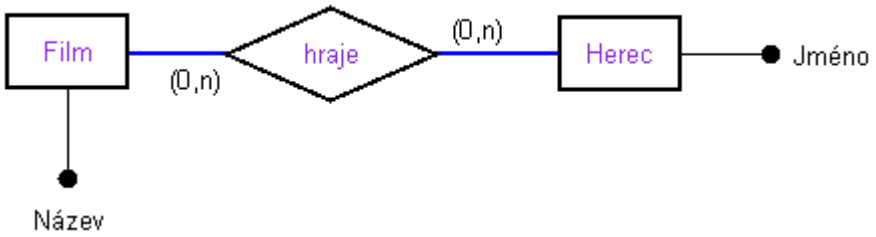
vztah s kardinalitami M:N lze jednoduše převést na dva vztahy s kardinalitami 1:N + tzv. **průnikový entitní typ** (který je slabý)

Umožňuje zobecňovat model, např. zavedením spoluidentifikátoru do vzniklého průnikového typu (zde například Datum)

Vícehodnotové atributy



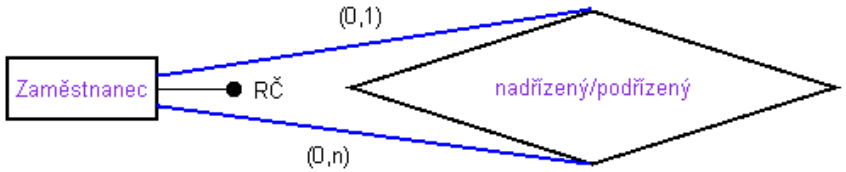




nahrazení vícehodnotového atributu entitou a vztahem

(více způsobů – zobecnění původní situace)

Rekurzivní binární vztah

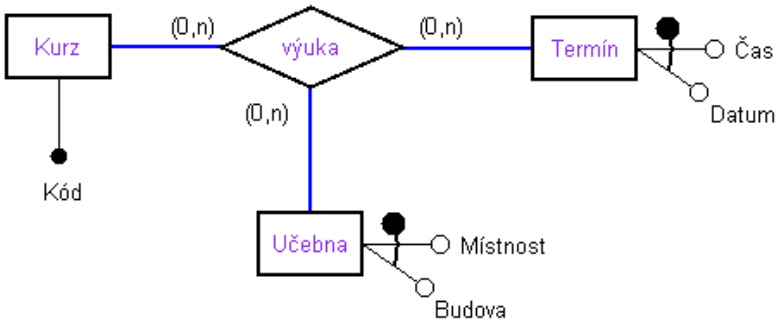


rekurzivní vztah – vstupují do něj entity stejného typu

důležité **rozlišovat role**

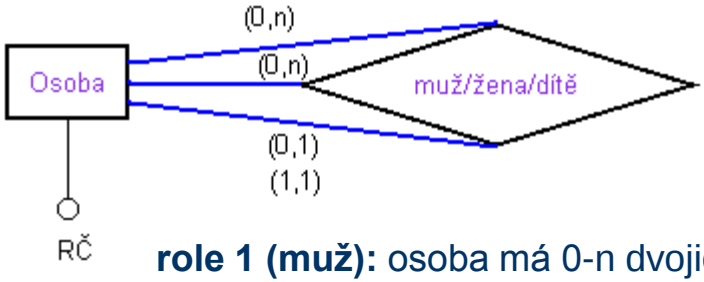
role 1 (podřízený): zaměstnanec má jednoho nebo žádného nadřízeného zaměstnance
role 2 (nadřízený): zaměstnanec má několik (nebo žádného) podřízeného zaměstnance

Ternární a více-ární vztahy



ternární vztah – entita vstupuje do vztahu s **dvojicí** entit

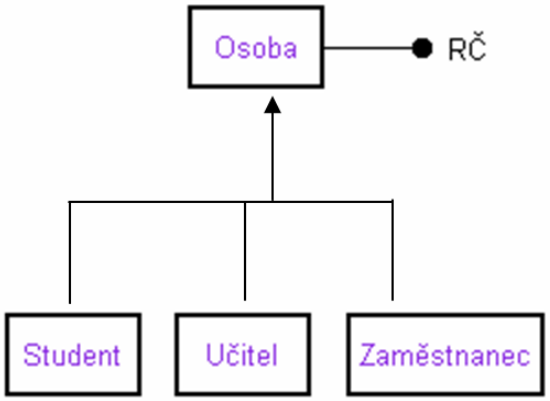
Slabá entita v ternárním vztahu je identifikována oběma zbývajícími entitami zároveň.



rekurzivní ternární vztah

role 1 (muž): osoba má 0-n dvojic [osoba, osoba] (žena, dítě)
role 2 (žena): osoba má 0-n dvojic [osoba, osoba] (muž, dítě)
role 3 (dítě): osoba má právě jednu dvojici [osoba, osoba] (otec, matka)

ISA hierarchie



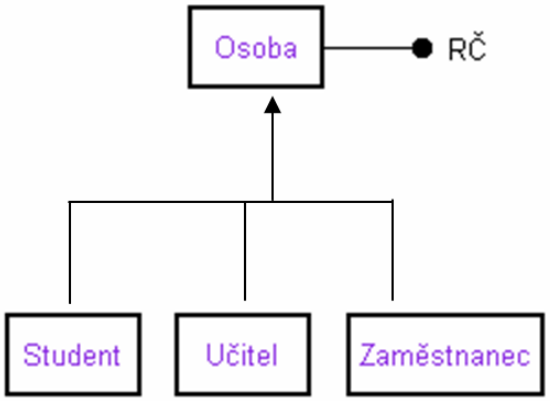
rozšíření ER o dědičnost nadentity / podentity

podentity dědí jak **atributy**, tak **vztahy** (případně integritní omezení) nadentity

pouze jednonásobná dědičnost

podentity jsou identifikovány výhradně předkem (tj. všechny entity v ISA hierarchy sdílí jediný identifikátor)

ISA hierarchie

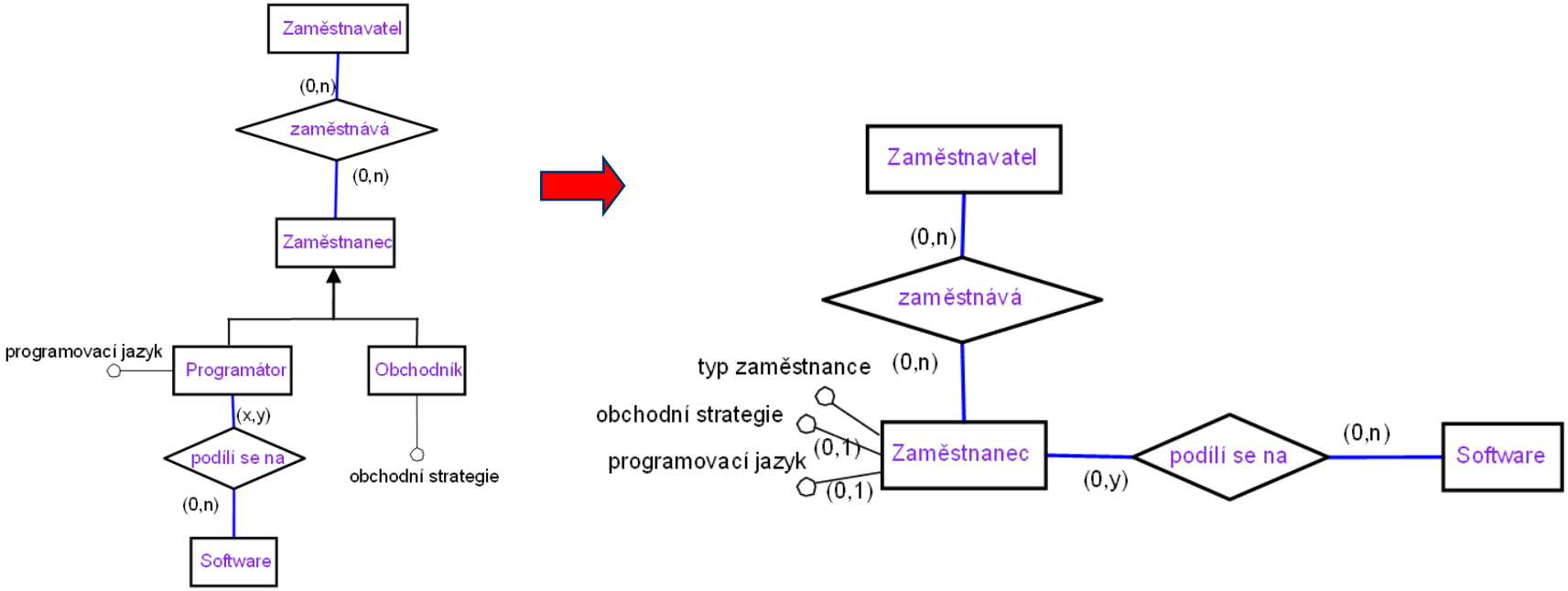


Další vlastnosti ISA vztahu:

1) Specifikace pokrytí (covering constraint)
   - (ne)vynucená specializace entity
   - př. musí/nemusí být Osoba vždy Student/Učitel/Zaměstanec?
2) Specifikace překrytí (overlap constraint)
   - dovolena/zákázána vícenásobná specializace entity
   - př. může/nesmí mít Osoba zároveň více specializací

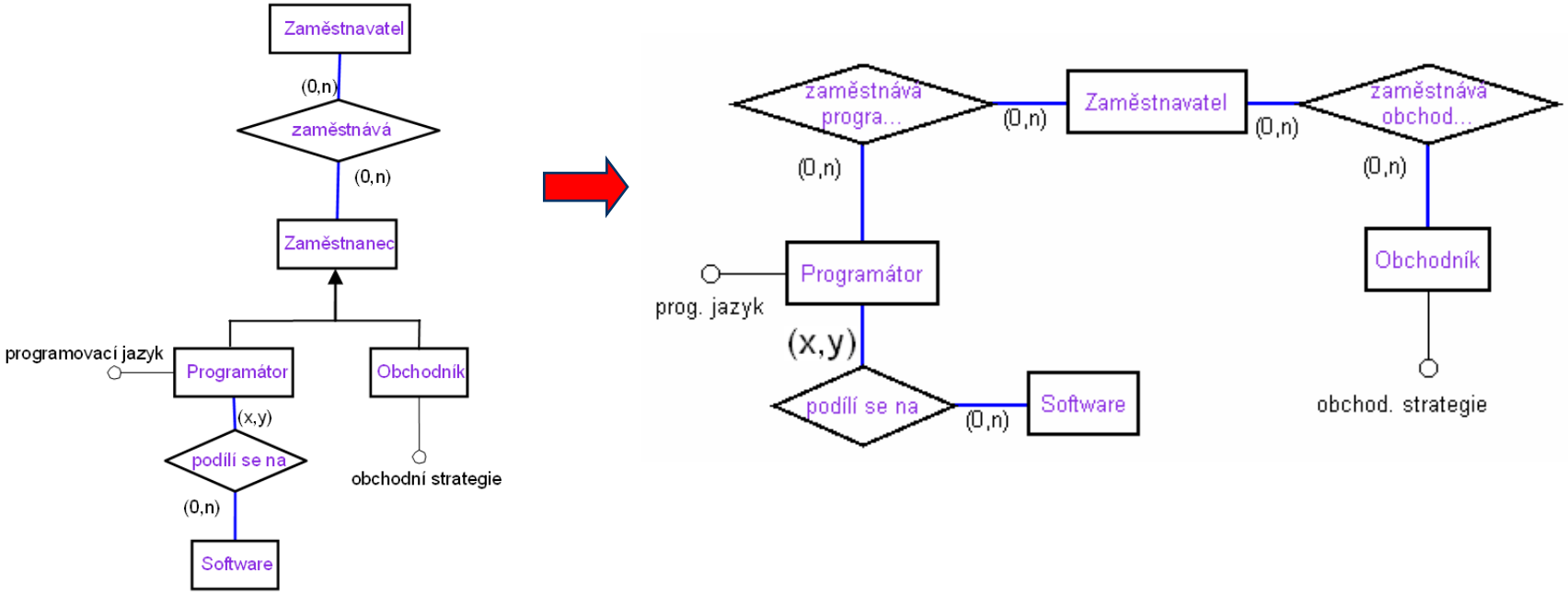
Eliminace ISA hierarchie

- různé možnosti (žádná obecně univerzální)
  - agregace pod entit ISA hierarchie do entity předka + úprava kardinalit vztahů a atributů



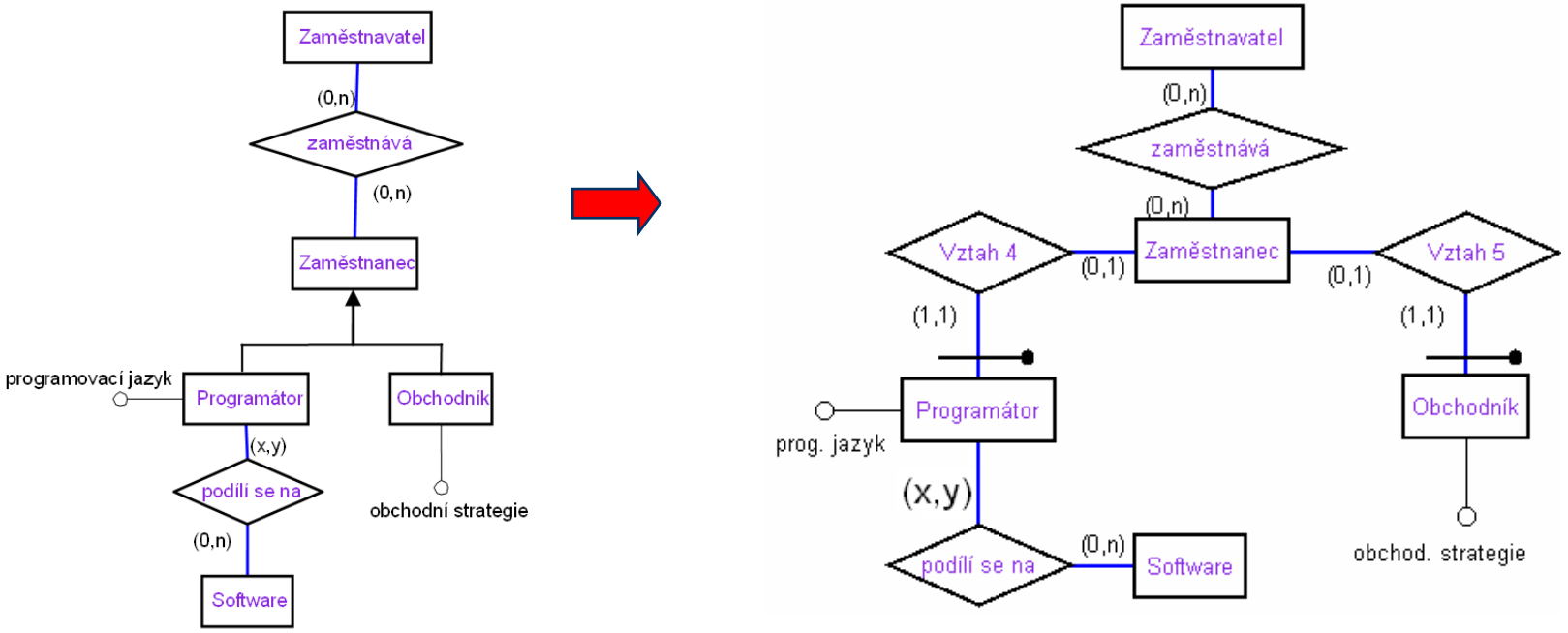
Eliminace ISA hierarchie

- různé možnosti (žádná obecně univerzální)
  - odstranění předka, agregace jeho atributů a vztahů ve všech potomcích



Eliminace ISA hierarchie

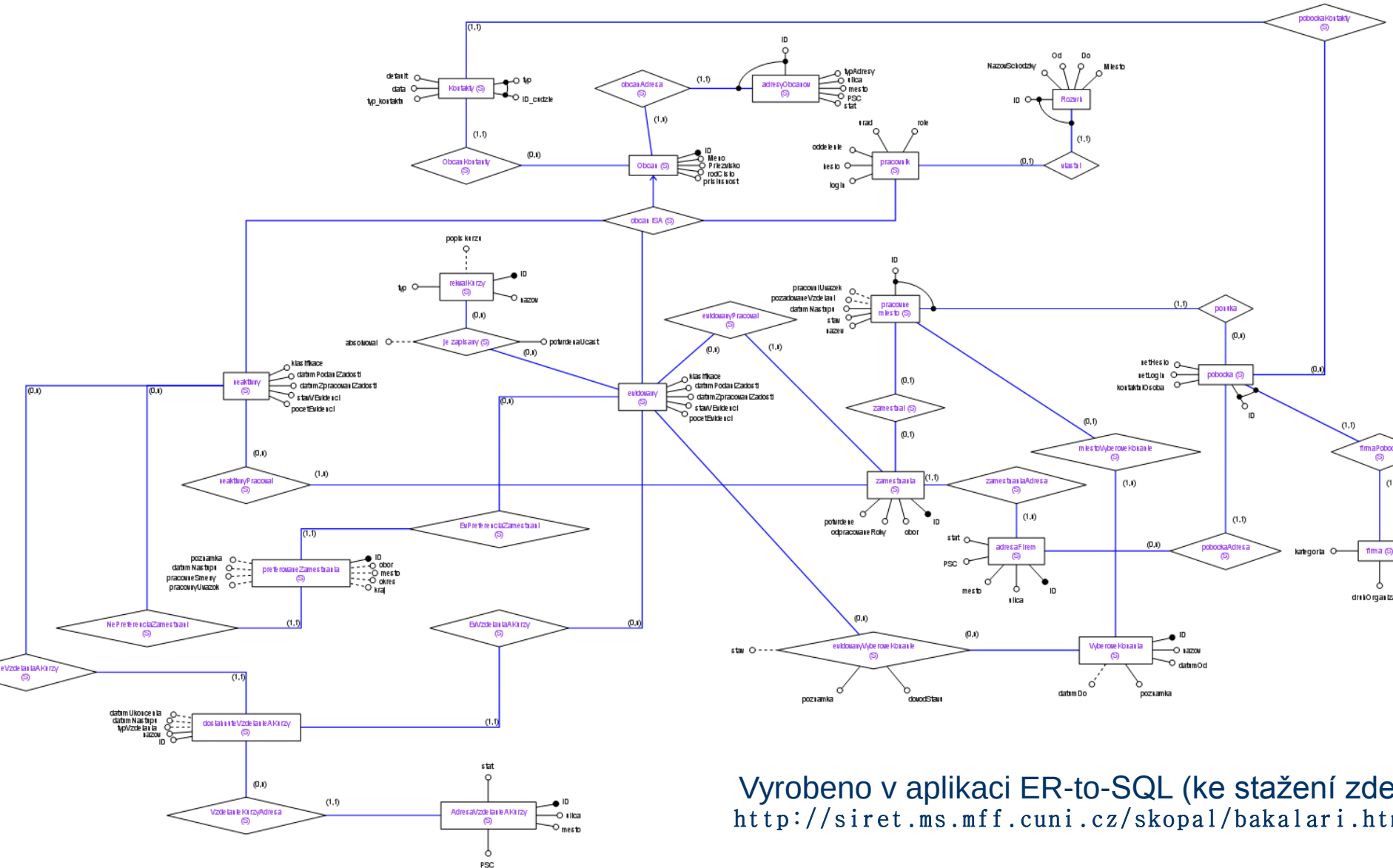
- různé možnosti (žádná obecně univerzální)
  - nahrazení ISA vztahu klasickým vztahem (z potomků vzniknou slabé entitní typy)



Korektní ER schéma

- žádný entitní typ nemá více než jednoho ISA předka
- ISA vztahy netvoří orientovaný cyklus
- identifikační vztahy netvoří orientovaný cyklus
- potomek v ISA hierarchii není identifikačně závislý na žádném entitním typu (je již identifikován předkem)
- jména entitních a vztahových typů jsou jednoznačná

Příklad – IS úřadu práce



Vyrobeno v aplikaci ER-to-SQL (ke stažení zde)
http://siret.ms.mff.cuni.cz/skopal/bakalari.htm

Logické databázové modely

Aktuální modely

- relační databáze
- objektové databáze
- objektově-relační databáze

Překonané (starší) modely (nicméně stále užívané na mainframech)

- hierarchické databáze
  - stromová struktura dat, datový záznam může mít jednoho předka a více potomků
  - nyní nahrazují XML databáze a obecněji objektové databáze
- síťové databáze
  - narozdíl od hierarchického umožňuje více předků pro záznam (tj. svazové uspořádání), opět lze nahradit XML a objektovými DB

Relační databáze (RDBMS)

- nejstarší z moderních DB (Edgar Codd, 1969)
- data se modelují v tabulkách/relacích
  - řádky jsou entity, sloupce atributy (dané entity)
  - jednoduché datové typy (fixní velikost a jejich nestrukturovanost) zajišťují plochost struktury (tzv. 1. normální formu)
- „obdélníková plochost“ struktury tabulky umožňuje jednoduché dotazování pomocí deklarativního jazyka SQL
- umožňuje normalizaci relačních schémat, čímž lze dosáhnout optimálního návrhu eliminujícího redundanci dat a aktualizační anomálie
- jednoduchá implementace → vysoký výkon

Objektové databáze (ODBMS)

- data modelována třídami - jejich instancemi jsou objekty
- výhody jsou podobné jako u OOP
  - zapouzdření datové entity do objektu
  - konceptuální model splývá s logickým modelem (ER a UML)
  - přímé asociace mezi objekty (pointery),
    tj. možnost nativně modelovat grafy objektů
  - model přímo použitelný v OOP, tj. k DB objektu lze přistupovat přímo jako k objektu programovacího jazyka (např. Java, C#)
    - DB lze chápat jako obohacení objektů OOP o perzistenci
- nevýhody
  - perzistence grafu objektů a implementace operací na něm jsou netriviální a u obvyklých transakcí nedosahuje takového výkonu jako relační a objektově-relační DB
  - výkonější pro navigační dotazování (podobně jako DOM u XML) než pro deklarativní SQL (které je ale široce rozšířené a jednoduché)

Objektově-relační databáze (ORDBMS)

- relační databáze obohacená o objektové prvky (definice se liší na konkrétních komečních platformách)
- nejčastěji
  - relace (tabulky) jsou základ, stejně jako u RDBMS
  - povolují se objektové datové typy, tj. atribut typu třída, která má vlastní metody a může agregovat další třídy (tj. tabulka není v tzv. 1. normální formě)
  - vnořené tabulky (atribut typu tabulka)
- od normy SQL:1999 se objektově-relační DB standardizují
- v současné době nejpopulárnější kompromis
  - produkty MS SQL Server, Oracle, DB2 a další

Databázové systémy

Tomáš Skopal

- úvod do relačního modelu
- převod konceptuálního schématu do relačního

Osnova přednášky

- relační model
- převod ER diagramu do relačního modelu
- tvorba univerzálního relačního schématu

Relační model – neformálně v kostce

- zakladatel: E.F Codd – článek „*A relational model of data for large shared data banks*“, Communications of ACM, 1970
- model uchovávání entit/vztahů v **tabulkách**
  - max 1 tabulka na entitní/vztahový typ
- datová **entita/vztah** E je reprezentována **řádkem** tabulky
- **atribut** A entity je reprezentován **sloupcem** tabulky
- buňka tabulky na pozici (x,y) uchovává hodnotu atributu A(x) v kontextu entity/vztahu E(y)
- **schéma tabulky** – popis struktury tabulky (všeho kromě dat v ní)
  - $S(A_1{:}T_1, A_2{:}T_2, ...)$ – kde S je název tabulky, A_i jsou atributy a T_i jejich typy
- **schéma relační databáze** – množina schémat tabulek (+ další věci, např. integritní omezení, atd.)

Relační model – neformálně v kostce

- základní integritní omezení
  - neexistují 2 stejné řádky (unikátní identifikace entit)
  - rozsah hodnot atributu je dán typem (jednoduché typované atributy)
  - tabulka není „děravá“, tj. všechny hodnoty jsou definované (hodnota NULL se chápe jako speciální „metahodnota“)
- každý řádek je identifikovatelný (odlišitelný od ostatních) jedním nebo více pevně určenými atributy v rámci tabulky
  - taková skupina atributů se nazývá **nadklíč** tabulky
  - nadklíč s minimálním počtem atributů se nazývá **klíč** tabulky (může existovat víc klíčů)
- speciální „mezitabulkové“ integritní omezení – **cizí klíč**
  - skupina atributů, která existuje i v jiné (tzv. referenční) tabulce a tam tvoří klíč
  - důsledky: není to zpravidla klíč v tabulce referující, mohou existovat duplicitní hodnoty

Příklad 1 – tabulky, cizí klíč

Zboží (Název: string, Výrobce:string, Cena: float, Skladem: int),
IO(Název je klíč, Výrobce je cizí klíč do tabulky Výrobce(Název))

Název	Výrobce	Cena	Skladem
Mouka	Odkolek	5,80	100
Chleba	Odkolek	12,50	56
Ariel	P&G	325,-	15
Koště	Kartáčovny	135,-	32
Hřebík	Ferona	0,70	9000
Šroub	Ferona	0,90	3600

klíč

cizí klíč
(klíč v tabulce Výrobce)

Výrobce (Název: string, Adresa:string, Dlužíme: bool)
IO(Název je klíč)

Název	Adresa	Dlužíme
Odkolek	Praha	ANO
Kartáčovny	Liberec	NE
Ferona	Olomouc	NE
P&G	USA	ANO
VelkoMasna	Beroun	NE
Papírna	Šumperk	NE

klíč

Příklad 2 – tabulky, cizí klíč

Řidič (Jméno: string, Příjmení:string, MK: string, SPZ: string),
IO(Jméno a Příjmení je klíč, MK a SPZ je cizí klíč do tabulky Auto(Mezinárodní kód, SPZ))

Jméno	Příjmení	MK	SPZ
Radek	Rachota	CZ	2543
František	Dobrota	CZ	5461
Jiří	Loukota	CZ	2345
Václav	Levota	SK	6658
Josef	Drahota	PL	0124

složený klíč

složený cizí klíč
(klíč v tabulce Auto)

Auto (Mezinárodní kód: string, SPZ:string, Značka: strin
IO(Mezinárodní kód a SPZ je klíč)

Mezinárodní kód	SPZ	Značka
CZ	2543	Škoda
CZ	5461	VW
CZ	2345	Volvo
SK	6658	Kia
PL	0124	Fiat (polski)
D	2456	Trabant

složený klíč

Základy relačního modelu – formálně

- algebraický pojem relace $R \subseteq D_1 \times D_2 \times ... \times D_n$ je „databázově“ rozšířen
  - množiny D_i jsou tzv. atributové domény – tj. typy
  - schéma relace je určeno výčtem atributů (případně jejich typů)
  - $R(A_1 : D_1, A_2 : D_2, ..., A_n : D_n)$, také $R(A)$, kde $A = \{A_1, A_2, ... , A_n\}$
- názvosloví – tabulky vs. relace
  - **relace** odpovídá konkrétním **datům** v tabulce
  - **prvek relace** odpovídá konkrétnímu **řádku** (**záznamu**) v tabulce
  - **schéma relace** odpovídá **schématu tabulky**
  - **domény** relace odpovídají **typům atributů** tabulky
- rozlišují se názvy atributů, tj. ve schématu relace lze rozlišit dva atributy stejného typu

Přínosy relačního modelu

- oddělení dat od implementace (přes pojem relace) – při manipulacích s daty se nezajímáme o konkrétní přístupové mechanismy
- dva silné prostředky pro dotazování relačních dat
  - relační kalkul a relační algebra
- lze vhodně navrhovat relační schémata pomocí normalizací relačních schémat (algoritmicky)
  - dekompozice a syntéza

Motivace k normalizaci

- problémy návrhu relačního schématu
  - redundance – ukládání stejné informace víckrát na různých místech (zvyšuje prostorové nároky)
  - mohou nastat aktualizační anomálie (insert/delete/update)
    - při vložení dat příslušejících jedné entitě je potřeba zároveň vložit data i o jiné entitě
    - při vymazání dat příslušejících jedné entitě je potřeba vymazat data patřící jiné entitě
    - pokud se změní jedna kopie redundantních dat, je třeba změnit i ostatní kopie, jinak se databáze stane nekonzistentní
- řešení – normalizace DB schématu využitím funkčních závislostí

Příklad „nenormálního“ schématu

ZamId	Jméno	Funkce	Hodinová mzda	Odpracoval hodin
1	Radek Rachota	účetní	200	50
2	František Dobrota	obchodník	500	30
3	Jiří Loukota	obchodník	500	45
4	Václav Levota	účetní	200	70
5	Josef Drahota	účetní	200	66
6	Pavel Měkota	lektor	300	10

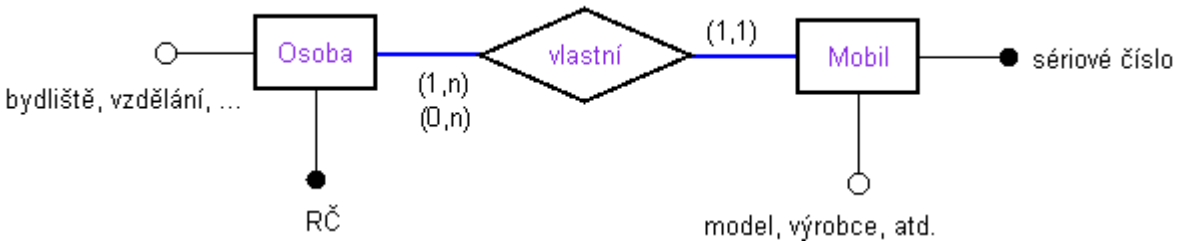
1) Z funkční analýzy víme, že pracovní funkce určuje hodinovou mzdu – přesto se mzda opakuje – redundance.

2) Smažeme-li záznam o zaměstnanci 6, ztratíme rovněž informaci o mzdě lektora.

3) Změníme-li mzdu funkce „účetní“, musíme tak učinit na třech místech.

Jak se to může stát?

- jednoduše
  - ručním návrhem tabulky (ruční návrh se obecně nedoporučuje)
  - špatně/neoptimálně navržený ER diagram
    - např. zbytečně mnoho atributů v entitě



vede na 2 tabulky

Osoba(**RČ**, bydliště, vzdělání, ...)
Mobil(**sériové číslo**, model, výrobce, ..., **RČ**)

Jak se to může stát?

Sériové číslo	Výrobce	Model	Made in	Atest
13458	Nokia	6110	Finsko	EU, USA
34654	Nokia	6110	Finsko	EU, USA
65454	Nokia	6110	Finsko	EU, USA
45464	Siemens	SX1	Německo	EU
64654	Samsung	E720	Taiwan	Asie, USA
65787	Samsung	E720	Taiwan	Asie, USA

Redundance hodnot atributu Výrobce, Model, Made in, Atest

Kde se stala chyba?
Entita Mobil skrývá další entity – Výrobce, Model, případně další...
Jak to spravit? Dvě možnosti – 1) upravit přímo ER diagram (více entit)
2) upravit už hotová schémata relací (viz dále)

Modelování pomocí funkčních závislostí (1)

- typ atributových integritních omezení definovaných uživatelem (např. DB administrátorem)
- do jisté míry relační alternativa ke konceptuálnímu modelování (ER modelování vzniklo později)

- funkční závislost (FZ) $X \rightarrow Y$ nad schématem R(A)
  - parciální zobrazení $f_i : X_i \rightarrow Y_i$, kde $X_i, Y_i \subseteq A$ (kde i = 1..počet závislostí pro R(A))
  - *n*-tice z X_i **funkčně určuje** *m*-tici z Y_i
  - *m*-tice z Y_i **funkčně závisí** na *n*-tici z X_i
  - parciální je proto, že je definováno pouze pro hodnoty atributů přítomné v relacích (tj. pro data v tabulkách), ne pro celé univerzum

- neboli hodnoty atributů X **společně** určují hodnoty atributů Y
- zobecnění principu klíče (identifikace), resp. klíč bude speciální případ, viz dále

Modelování pomocí funkčních závislostí (2)

- pokud $X \rightarrow Y$ a $Y \rightarrow X$, potom X i Y jsou **funkčně ekvivalentní** a lze psát $X \leftrightarrow Y$
- pokud $X \rightarrow a$, kde $a \in A$, potom $X \rightarrow a$ je **elementární funkční závislost**

Příklad – špatná interpretace

ZamId	Jméno	Funkce	Hodinová mzda	Odpracoval hodin
1	Radek Rachota	účetní	200	50
2	František Dobrota	obchodník	500	30
3	Jiří Loukota	obchodník	500	45
4	Václav Levota	účetní	200	70
5	Josef Drahota	účetní	200	66
6	Pavel Měkota	lektor	300	10

Z konkrétních dat v příkladu lze vypozorovat, že **by mohlo** platit:
Funkce → Hodinová mzda stejně jako **Hodinová mzda → Funkce**
ZamId → všechno
Odpracoval hodin → všechno
Jméno → *všechno*

(**což je ovšem nesmysl** vzhledem k přirozenému smyslu uvedených atributů)

Příklad – špatná interpretace

ZamId	Jméno	Funkce	Hodinová mzda	Odpracoval hodin
1	Radek Rachota	účetní	200	50
2	František Dobrota	obchodník	500	30
3	Jiří Loukota	obchodník	500	45
4	Václav Levota	účetní	200	70
5	Josef Drahota	účetní	200	66
6	Pavel Měkota	lektor	300	10
7	Leoš Mrákota	poradce	300	70
8	Josef Drahota	obchodník	500	55

Funkce → Hodinová mzda
ZamId → všechno

~~**Hodinová mzda → Funkce**~~
~~**Odpracoval hodin → všechno**~~
~~**Jméno → *všechno***~~

Příklad – správná interpretace

- nejprve se analyticky (tj. datovou analýzou) definují **„jednou provždy“** funkční závislosti a ty **omezí** „naplň“ tabulek (resp. přípustné prvky v relacích)
- např. **Funkce → Hodinová mzda**
  ZamId → všechno
- pátý řádek v níže uvedené tabulce nemůže nastat (nepovolí se jeho vložení)

ZamId	Jméno	Funkce	Hodinová mzda	Odpracoval hodin
1	Radek Rachota	účetní	200	50
2	František Dobrota	obchodník	500	30
3	Jiří Loukota	obchodník	500	45
4	Václav Levota	účetní	200	70
4	Jaroslav Klokota	obchodník	300	23

Rozšíření relačního schématu

- o funkční závislosti lze rozšířit schéma relace
  - R(A, F), kde F = $\cup_i \{f_i\}$
  - v F jsou dovoleny pouze atributy z A (nemusí tam být všechny)
- klíč $K \subseteq A$ je definován pomocí funkčních závislostí jako (tato redefinice klíče je jiným vyjádřením předchozí „identifikační“ definice)
  - $K \rightarrow A$ (K je alespoň nadklíč)
  - neexistuje $K' \subset K$ takové, že $K' \rightarrow A$ (K je minimální)
- klíčů může existovat více
  (v krajním případě i všechny atributy v A)
- **klíčový atribut** – je obsažen v klíči (nebo je sám klíčem)
- **neklíčový atribut** – není obsažen v klíči

Převod ER schématu

- relační a objektově-relační databáze
  - do tabulek, viz dále
- objektové databáze
  - do tříd, asociací, dědičnosti (v případě ISA vztahů)
  - v podstatě převod 1:1, resp. ER $\rightarrow$ UML

Převod ER diagramu do schématu relačního databáze

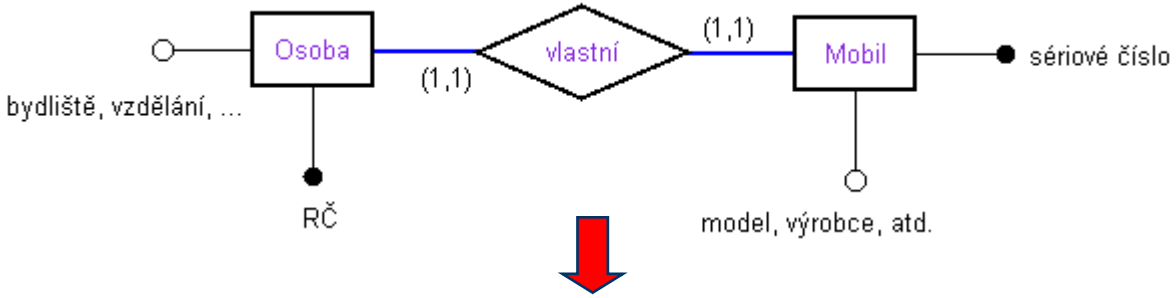
- implementace ER konceptu v relační databázi
- schéma konceptuálního (sémantického) modelu $\rightarrow$ schéma logického modelu (schéma relační databáze)
- zachování integritních omezení (pokud možno bez ztráty)

Převod obecně

- silný entitní typ – tabulka
  - klíčem je identifikátor entity
- slabý entitní typ – tabulka
  - klíčem je smíšený (externí) identifikátor
- vztahový typ – tabulka
  - počet tabulek a jejich sdílení vztahy a entitami je řídí kardinalitami vztahu, do něhož tyto entity vstupují
- integritní omezení ve formě klíčů tabulek + externě
- n-ární vztahy se převádějí v principu stejně jako binární

Kardinality (1,1) : (1,1)

- jediná tabulka – klíčem může být identifikátor libovolné z entit (nebo oba)

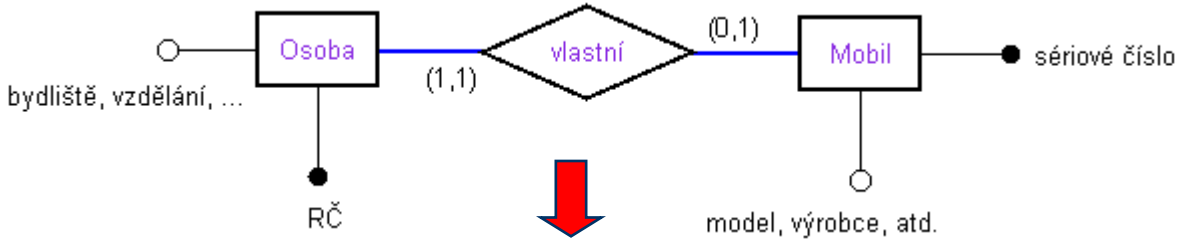


Relační schéma:

OsobaMobil(**RČ**, bydliště, vzdělání, ..., **sériové číslo**, model, výrobce, ...)

Kardinality (1,1) : (0,1)

- dvě tabulky T1 a T2
  - T1 existuje nezávisle na T2
  - T2 obsahuje identifikátor/klíč z T1, který je zde cizím klíčem do T1
  - reprezentace vztahu je „ukrytá“ v tabulce T2

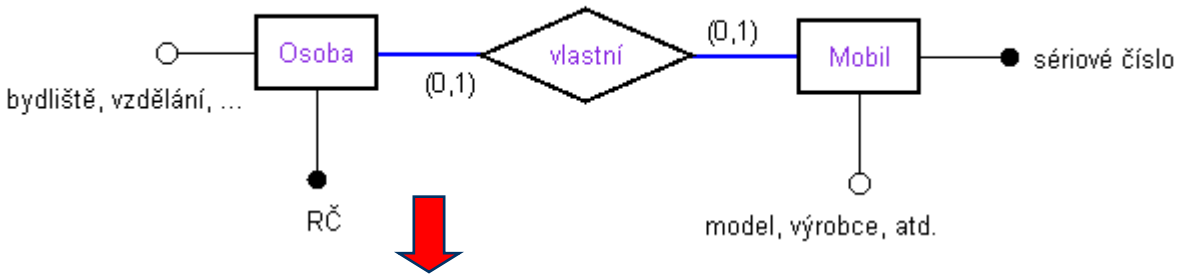


Relační schéma:

Osoba(**RČ**, bydliště, vzdělání, ..., **sériové číslo**)
Mobil(**sériové číslo**, model, výrobce, ...)

Kardinality (0,1) : (0,1)

- tři tabulky – dvě entitní a jedna vztahová
  - vztahová tabulka má dva klíče odpovídající identifikátorům obou entitních typů
  - klíče jsou zároveň cizí klíče do entitních tabulek

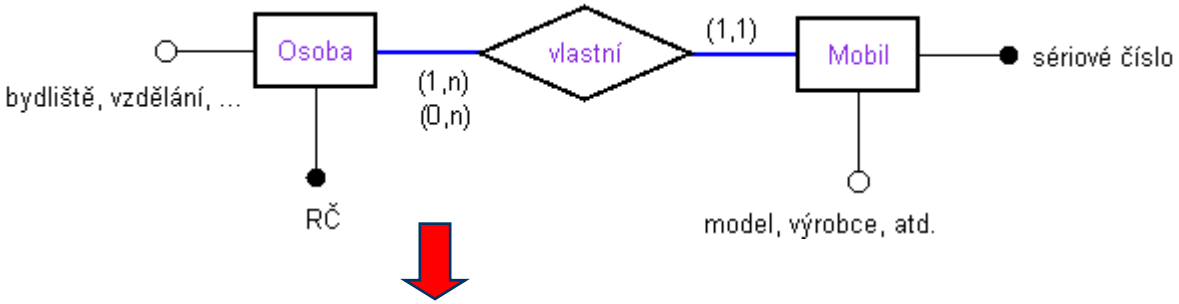


Relační schéma:

Osoba(**RČ**, bydliště, vzdělání, ...)
Mobil(**sériové číslo**, model, výrobce, ...)
Vlastní(**RČ, sériové číslo**)

Kardinality (0/1,n) : (1,1)

- podobně jako u (1,1) : (0,1) – dvě tabulky T1 a T2
  - T1 existuje nezávisle na T2
  - T2 obsahuje identifikátor T1, který je zde cizím klíčem do T1
  - reprezentace vztahu je „ukrytá“ v tabulce T2

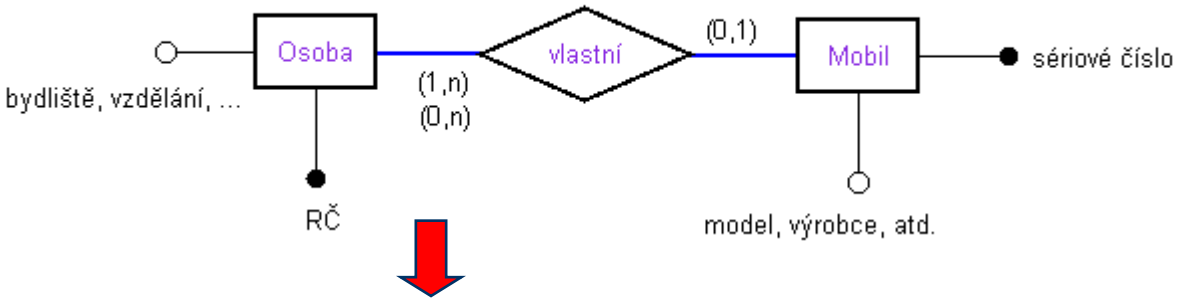


Relační schéma:

Osoba(**RČ**, bydliště, vzdělání, ...)
Mobil(**sériové číslo,** model, výrobce, ..., **RČ**)

Kardinality (0/1,n) : (0,1)

- podobně jako (0,1) : (0,1) – tři tabulky – dvě entitní a jedna vztahová
  - vztahová tabulka má jeden klíč odpovídající identifikátoru jednoho z entitních typů, identifikátor druhého entitního typu je pouze cizím klíčem
  - klíč je zároveň cizí klíč do jedné z entitních tabulek

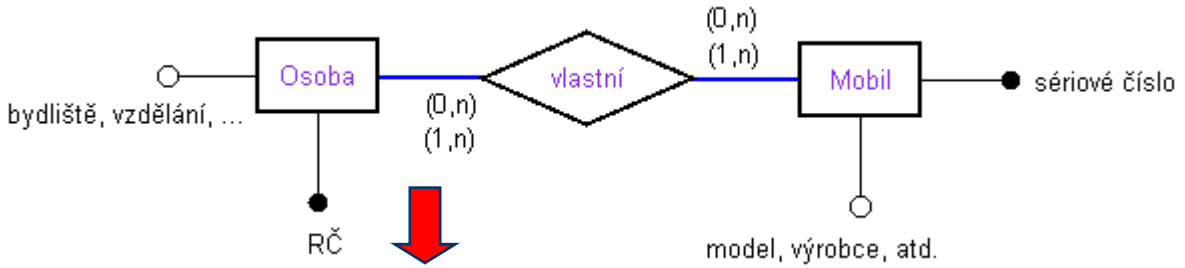


Relační schéma:

Osoba(**RČ**, bydliště, vzdělání, ...)
Mobil(**sériové číslo**, model, výrobce, ...)
Vlastní(**RČ, sériové číslo**)

Kardinality (0/1,n) : (0/1,n)

- nejobecnější situace – tři tabulky
  - dvě entitní a jedna vztahová
  - klíč ve vztahové tabulce je složený z identifikátorů obou entitních typů
  - všechny části klíče vztahové tabulky jsou cizími klíči do tabulek vázaných



Relační schéma:

Osoba(**RČ**, bydliště, vzdělání, ...)
Mobil(**sériové číslo**, model, výrobce, ...)
Vlastní(**RČ, sériové číslo**)

Příklad konverze do SQL

CREATE TABLE ZDROJE_FINANCOVANI (ID Integer NOT NULL, CLASSID Integer NOT NULL, ZFIN_IDENT Varchar(255) NOT NULL, ZFIN_STAV Integer, ZFIN_STR Integer, CONSTRAINT pk_ZDROJE_FINANCOVANI PRIMARY KEY (ID));

CREATE TABLE URI (ID Integer NOT NULL, CLASSID Integer NOT NULL, URI_IDENT Varchar(255) NOT NULL, URI_STAV Integer, URI_STR Integer, URI_USTAVVYUZ Integer, CONSTRAINT pk_URI PRIMARY KEY (ID));

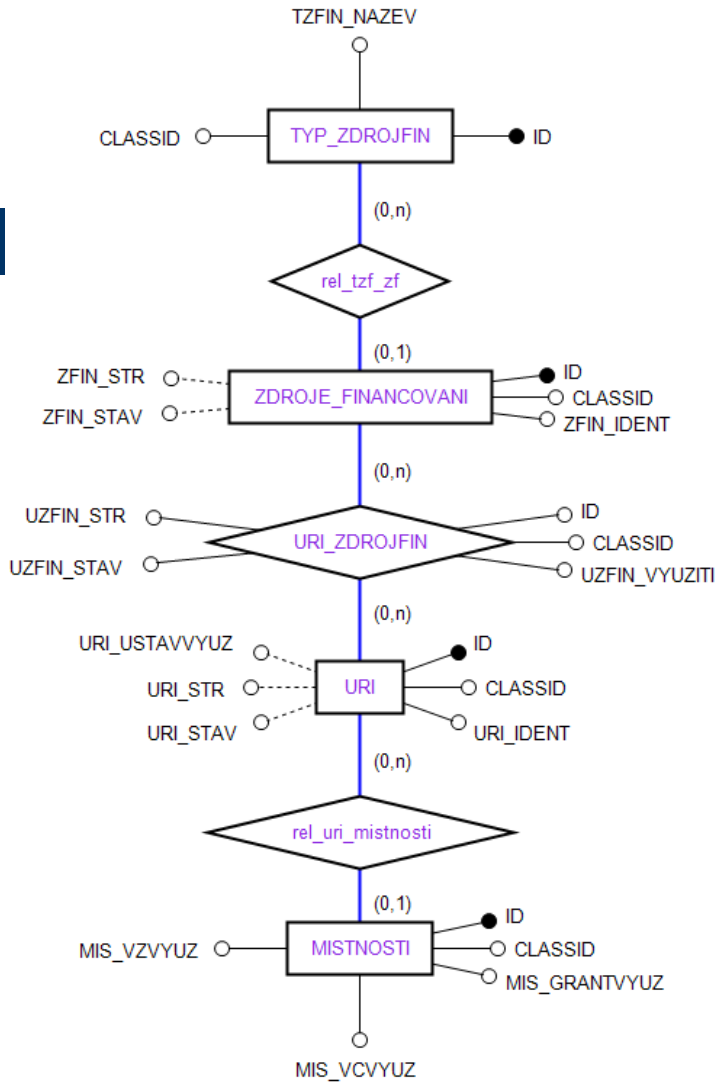
CREATE TABLE MISTNOSTI (ID Integer NOT NULL, CLASSID Integer NOT NULL, MIS_GRANTVYUZ Integer NOT NULL, MIS_VCVYUZ Integer NOT NULL, MIS_VZVYUZ Integer NOT NULL, CONSTRAINT pk_MISTNOSTI PRIMARY KEY (ID));

CREATE TABLE TYP_ZDROJFIN (ID Integer NOT NULL, CLASSID Integer NOT NULL, TZFIN_NAZEV Varchar(255) NOT NULL UNIQUE, CONSTRAINT pk_TYP_ZDROJFIN PRIMARY KEY (ID));

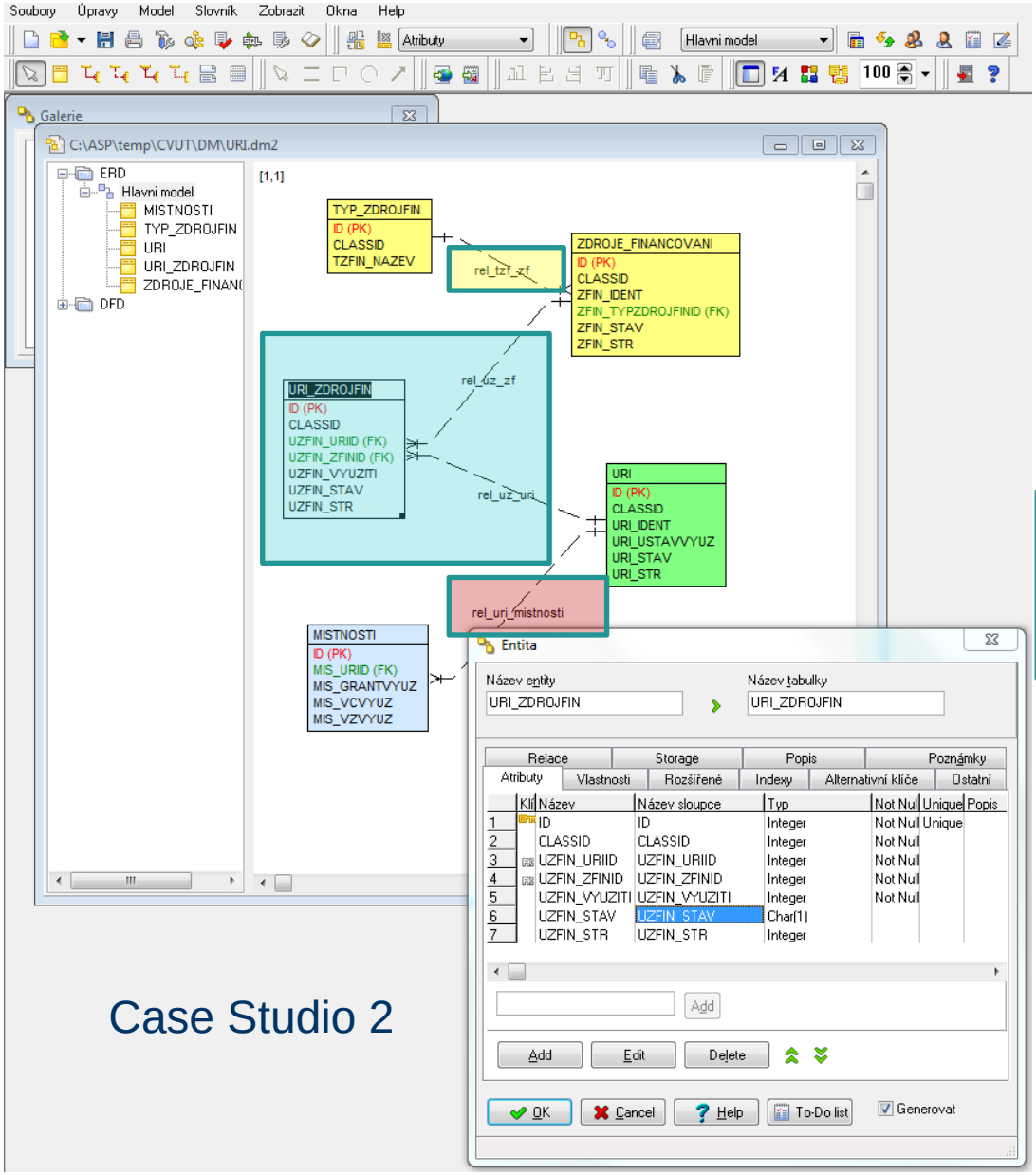
CREATE TABLE URI_ZDROJFIN (ID Integer NOT NULL, CLASSID Integer NOT NULL, UZFIN_VYUZITI Integer NOT NULL, UZFIN_STAV Integer NOT NULL, UZFIN_STR Integer NOT NULL, ZDROJE_FINANCOVANI_ID Integer NOT NULL, URI_ID Integer NOT NULL, CONSTRAINT pk_URI_ZDROJFIN PRIMARY KEY (ZDROJE_FINANCOVANI_ID, URI_ID), CONSTRAINT fk_URI_ZDROJFIN_ZDROJE_FINANCOVANI FOREIGN KEY (ZDROJE_FINANCOVANI_ID) REFERENCES ZDROJE_FINANCOVANI(ID), CONSTRAINT fk_URI_ZDROJFIN_URI FOREIGN KEY (URI_ID) REFERENCES URI(ID));

CREATE TABLE Vztah_2 (MISTNOSTI_ID Integer NOT NULL, URI_ID Integer NOT NULL, CONSTRAINT pk_Vztah_2 PRIMARY KEY (MISTNOSTI_ID), CONSTRAINT fk_Vztah_2_MISTNOSTI FOREIGN KEY (MISTNOSTI_ID) REFERENCES MISTNOSTI(ID), CONSTRAINT fk_Vztah_2_URI FOREIGN KEY (URI_ID) REFERENCES URI(ID));

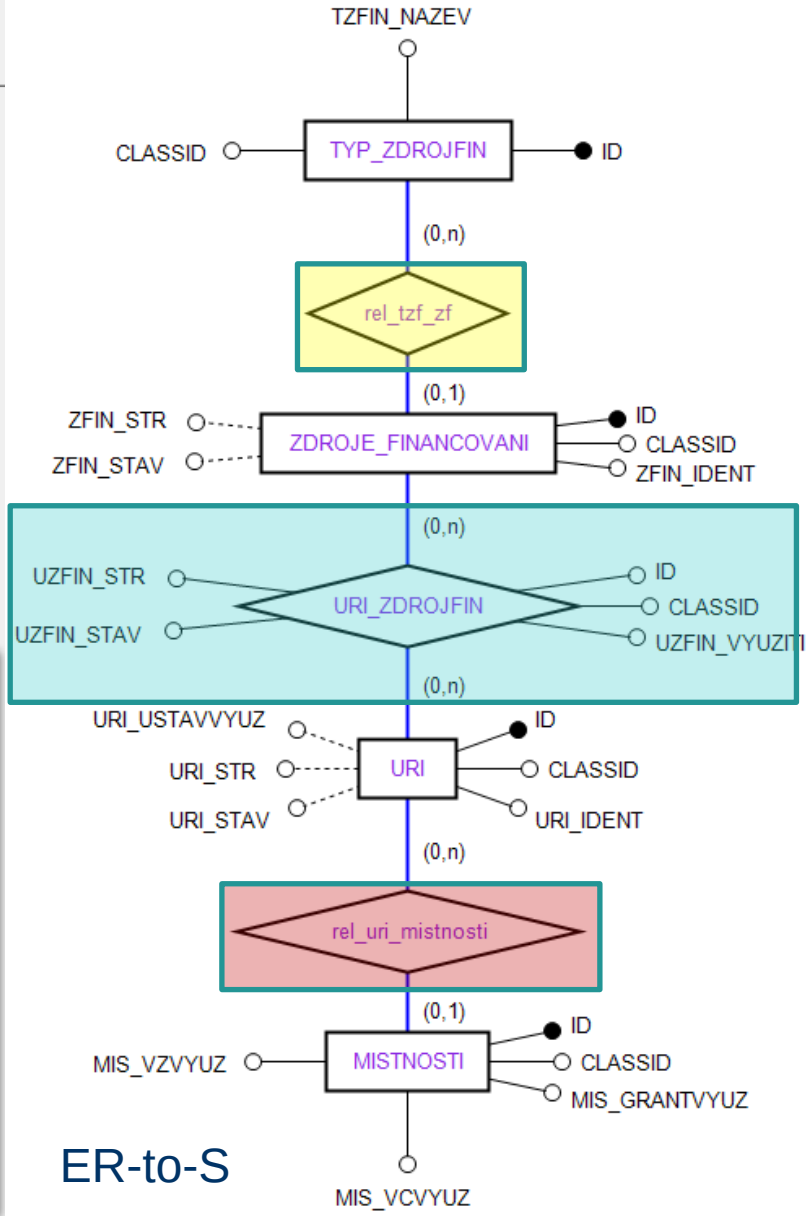
CREATE TABLE Vztah_4 (ZDROJE_FINANCOVANI_ID Integer NOT NULL, TYP_ZDROJFIN_ID Integer NOT NULL, CONSTRAINT pk_Vztah_4 PRIMARY KEY (ZDROJE_FINANCOVANI_ID), CONSTRAINT fk_Vztah_4_ZDROJE_FINANCOVANI FOREIGN KEY (ZDROJE_FINANCOVANI_ID) REFERENCES ZDROJE_FINANCOVANI(ID), CONSTRAINT fk_Vztah_4_TYP_ZDROJFIN FOREIGN KEY (TYP_ZDROJFIN_ID) REFERENCES TYP_ZDROJFIN(ID));



Srovnání s komerčním editorem



Case Studio 2



ER-to-S

Univerzální schéma rel. databáze

- vzniklé tabulky(např. konverzí z ER modelu) lze rozbít do univerzální tabulky (univerzální schéma databáze)
- zajištění unikátních názvů atributů
  - např. prefixem zdrojové tabulky
  - jména vazebních atributů jsou sdílena anebo se zavede funkční ekvivalence
- automatické vytvoření FZ pro dílčí identifikátory

Univerzální schéma rel. databáze - příklad

Dílčí relační schémata (s identifikátory):

Osoba(**RČ**, bydliště, vzdělání)

Mobil(**sériové číslo**, model, výrobce)

Vlastní(**RČ, sériové číslo**)

Univerzální schéma

U(OsobaRČ, OsobaBydliště, OsobaVzdělání,
MobilSériovéČíslo, MobilModel, MobilVýrobce)

Funkční závislosti (identifikátory)

F = { OsobaRČ → OsobaBydliště, OsobaVzdělání;
MobilSériovéČíslo → MobilModel, MobilVýrobce;
MobilSériovéČíslo → OsobaRČ}

Cizí klíče jsou skryty tím, že vazební atributy sdílí názvy.

Jemnější vztahy mezi atributy

K automaticky vygenerovaným FZ můžeme dospecifikovat ještě závislosti mezi atributy, které nešly postihnout v ER modelu.

Univerzální schéma

U(OsobaRČ, OsobaBydliště, OsobaVzdělání, MobilSériovéČíslo, MobilModel, MobilVýrobce)

Funkční závislosti

F = { OsobaRČ → OsobaBydliště, OsobaVzdělání;
MobilSériovéČíslo → MobilModel, MobilVýrobce;
MobilSériovéČíslo → OsobaRČ;
MobilModel → MobilVýrobce}

Reinženýring – „pytel atributů“

- převodem dílčích tabulek do univerzálního schématu sestupujeme na úroveň konceptuálního „atributového“ modelování
- entitami se stávají přímo atributym na kterých jsou definovány funkční závislosti
- později nám zpětným reinženýringem vzniknou nové tabulky
  - pokud byl původní návrh správný, vzniknou stejné, pokud ne, normalizace nám vytvoří jiné (reprezentující logické entity dané FZ)

Databázové systémy

Tomáš Skopal

- relační model
  * funkční závislosti, odvozování
  * normální formy

Osnova přednášky

- Armstrongova pravidla
- atributové a funkční uzávěry
- normální formy relačních schémat

Armstrongova pravidla

Mějme R(A,F). Nechť X, Y, Z $\subseteq$ A a množinu funkčních závislostí F

1) jestliže Y $\subseteq$ X, potom X $\rightarrow$ Y (triviální FZ, axiom)

2) jestliže X $\rightarrow$ Y a Y $\rightarrow$ Z, potom X $\rightarrow$ Z (tranzitivita, pravidlo)

3) jestliže X $\rightarrow$ Y a X $\rightarrow$ Z, pak X $\rightarrow$ YZ (kompozice, pravidlo)

4) jestliže pak X $\rightarrow$ YZ, pak X $\rightarrow$ Y a X $\rightarrow$ Z (dekompozice,pravidlo)

Armstrongova pravidla

Armstrongova pravidla:

- **jsou korektní**
  - co z množiny F odvodíme, platí pro libovolnou instanci R
- **jsou úplná**
  - lze jimi odvodit všechny FZ platné ve všech instancích R (vzhledem k danému F, samozřejmě)
- **1,2,3 (triviální, tranzitivita, kompozice) jsou nezávislá**
  - odstraněním jakéhokoliv z nich porušíme úplnost (dekompozice lze odvodit z triviální FZ a tranzitivity)

Důkaz vyplývá z definice FZ, resp. z vlastností zobrazení.

Příklad – odvozování FZ

R(A,F)

A = {a,b,c,d,e}

F = {ab $\rightarrow$ c, ac $\rightarrow$ d, cd $\rightarrow$ ed, e $\rightarrow$ f}

Lze odvodit např. FZ:

ab $\rightarrow$ a	(triviální)
ab $\rightarrow$ ac	(kompozice s ab $\rightarrow$ c)
ab $\rightarrow$ d	(tranzitivita s ac $\rightarrow$ d)
ab $\rightarrow$ cd	(kompozice s ab $\rightarrow$ c)
ab $\rightarrow$ ed	(tranzitivita s cd $\rightarrow$ ed)
ab $\rightarrow$ e	(dekompozice)
ab $\rightarrow$ f	(tranzitivita)

Příklad – odvození dekompozice

R(A,F)

$A = \{a,b,c\}$

$F = \{a \rightarrow bc\}$

Odvodíme:

$a \rightarrow bc$	(předpoklad dekompozice)
$bc \rightarrow b$	(triviální FZ)
$bc \rightarrow c$	(triviální FZ)
$a \rightarrow b$	(tranzitivita)
$a \rightarrow c$	(tranzitivita), tj. $\mathbf{a \rightarrow bc \Rightarrow a \rightarrow b \wedge a \rightarrow c}$

Funkční uzávěr

- **uzávěr F^+** množiny FZ F (funkční uzávěr) je množina všech FZ odvoditelných Armstrongovými pravidly z FZ v F
  - obecně exponenciální velikost vůči |F|

- závislost f je **redundantní** v F, jestliže platí $(F – \{f\})^+ = F^+$, tj. f lze odvodit

Příklad – funkční uzávěr

$R(A,F)$, $A = \{a,b,c,d\}$, $F = \{ab \rightarrow c, cd \rightarrow b, ad \rightarrow c\}$

$F^+ =$

$\{a \rightarrow a, b \rightarrow b, c \rightarrow c,$
$ab \rightarrow a, ab \rightarrow b, ab \rightarrow c,$
$cd \rightarrow b, cd \rightarrow c, cd \rightarrow d,$
$ad \rightarrow a, ad \rightarrow c, ad \rightarrow d,$
$abd \rightarrow a, abd \rightarrow b, abd \rightarrow c, abd \rightarrow d,$
$abd \rightarrow abcd, atd...\}$

Pokrytí

- **pokrytí** množiny FZ F je libovolná množina FZ G taková, že $F^+ = G^+$
- **kanonické pokrytí** = pokrytí tvořené elementárními FZ (dekomponujeme aby na pravé straně byly jednotlivé atributy)
- **neredundantní pokrytí** F je pokrytí F po odstranění všech redundantních závislostí
  - pozor, záleží na pořadí odebírání – původně redundantní FZ se může stát po odebrání nějaké jiné FZ neredundantní

Příklad – pokrytí

R1(A,F), R2(A,G),

A = {a,b,c,d},

F = {a $\rightarrow$ c, b $\rightarrow$ ac, d $\rightarrow$ abc},

G = {a $\rightarrow$ c, b $\rightarrow$ a, d $\rightarrow$ b}

Pro ověření $G^+ = F^+$ nemusím počítat celé uzávěry,
stačí z FZ v F odvodit FZ v G a naopak, tj.

F' = {a $\rightarrow$ c, b $\rightarrow$ a, d $\rightarrow$ b} – dekompozice

G' = {a $\rightarrow$ c, b $\rightarrow$ ac, d $\rightarrow$ abc } – tranzitivita a kompozice

$\Rightarrow G^+ = F^+$

Schémata R1 a R2 jsou ekvivalentní, protože G je pokrytím F a sdílejí stejnou množinu atributů A.

G je navíc **minimální** pokrytí, F není (minimální pokrytí viz dále).

Příklad – redundantní FZ

R1(A,F), R2(A,G),

A = {a,b,c,d},

F = {a $\rightarrow$ c, b $\rightarrow$ a, b $\rightarrow$ c, d $\rightarrow$ a, d $\rightarrow$ b, d $\rightarrow$ c}

závislosti b $\rightarrow$ c, d $\rightarrow$ a, d $\rightarrow$ c jsou redundantní

po jejich odebrání se nezmění F^+, tj. lze je odvodit ze zbylých FZ

b $\rightarrow$ c vznikne tranzitivitou a $\rightarrow$ c, b $\rightarrow$ a

d $\rightarrow$ a vznikne tranzitivitou d $\rightarrow$ b, b $\rightarrow$ a

d $\rightarrow$ c vznikne tranzitivitou d $\rightarrow$ b, b $\rightarrow$ a, a $\rightarrow$ c

Atributový uzávěr, klíč

- **uzávěr množiny atributů X^+** vzhledem k F je množina všech atributů **funkčně závislých** na X
- důsledek – pokud $X^+ = A$, potom X je **nadklíč**
- pokud obsahuje F závislost $X \rightarrow Y$ a v X existuje atribut *a* takový, že $Y \subseteq (X - a)^+$,
  nazýváme *a* **atributem redundantním v $X \rightarrow Y$**
- **redukovaná FZ** je taková, která na levé straně neobsahuje žádné redundantní atributy
  (jinak je **částečná FZ**)
- **klíč** je množina $K \subseteq A$ taková, že je nadklíč (tj. platí $K \rightarrow A$) a závislost $K \rightarrow A$ je zároveň redukovaná
  - klíčů může existovat více, vždy minimálně jeden
  - pokud není v F žádná FZ, triviálně platí $A \rightarrow A$, tj. klíčem je celá množina A

Příklad – atributový uzávěr

R(A,F), A = {a,b,c,d}, F = {a $\rightarrow$ c, cd $\rightarrow$ b, ad $\rightarrow$ c}

{a}+ = {a,c}	jinými slovy platí	a $\rightarrow$ c	(+ triviální a $\rightarrow$ a)
{b}+ = {b}			(triviální b $\rightarrow$ b)
{c}+ = {c}			(triviální c $\rightarrow$ c)
{d}+ = {d}			(triviální d $\rightarrow$ d)
{a,b}+ = {a,b,c}		ab $\rightarrow$ c	(+ triviální)
{a,d}+ = {a,b,c,d}		ad $\rightarrow$ bc	(+ triviální)
{c,d}+ = {b,c,d}		cd $\rightarrow$ b	(+ triviální)

Příklad – redundantní atribut

R(A,F), A = {i,j,k,l,m},
F = {m → k, lm → j, **ijk → l**, j → m, l → i, l → k}

Hypotéza:

v **ijk → l** je redundantní **k**, tj. platí **ij → l**

Důkaz:

1. podle hypotézy konstruujeme FZ **ij → ?** (všechno co lze odvodit)
2. při této konstrukci **ijk → l** zůstává v F, protože my vlastně **PŘIDÁVÁME** novou FZ **ij → ?** a tudíž můžeme **ijk → l** použít pro konstrukci atributového uzávěru $\{i,j\}^+$
3. obdržíme $\{i,j\}^+ = \{i, j, m, k, l\}$, tj. platí i **ij → l**, kterou přidáme do F (můžeme, protože je to prvek uzávěru F^+)
4. nyní zapomeneme, jak se do F dostala **ij → l**, prostě tam patří
5. a protože **ijk → l** lze odvodit triviálně z **ij → l**, je to v F redundantní FZ a můžeme ji z F odstranit
6. jinými slovy, zlikvidovali jsme redundantní atribut **k** v **ijk → l**

Jinými slovy, převedli jsme redukci redundantního atributu ve FZ na problém redukce redundantní FZ v F.

Minimální pokrytí

- neredundantní kanonické pokrytí, které se skládá pouze z redukovaných FZ
  - konstruuje se odstraněním nejprve redundantních atributů v závislostech **a až potom** redundantních závislostí (tj. záleží na pořadí)

Příklad: abcd $\rightarrow$ e, e $\rightarrow$ d, a $\rightarrow$ b, ac $\rightarrow$ d

Správné pořadí redukce:

1. b,d jsou redundantní v abcd $\rightarrow$ e, tj. odstraníme je
2. ac $\rightarrow$ d je redundantní

Špatné pořadí redukce:

1. žádná redundantní závislost
2. redundantní b,d v abcd $\rightarrow$ e

(nyní ale není minimální pokrytí, protože ac $\rightarrow$ d je redundantní)

První normální forma (1NF)

Každý atribut schématu relace je elementárního typu a je nestrukturovaný.

(1NF je základní podmínka „plochosti databáze“ – tabulka je opravdu dvourozměrné pole, ne např. skrytý strom nebo graf)

Příklad – 1NF

Osoba(Id: **Integer**, Jméno: **String**, Narozen: **Date**)

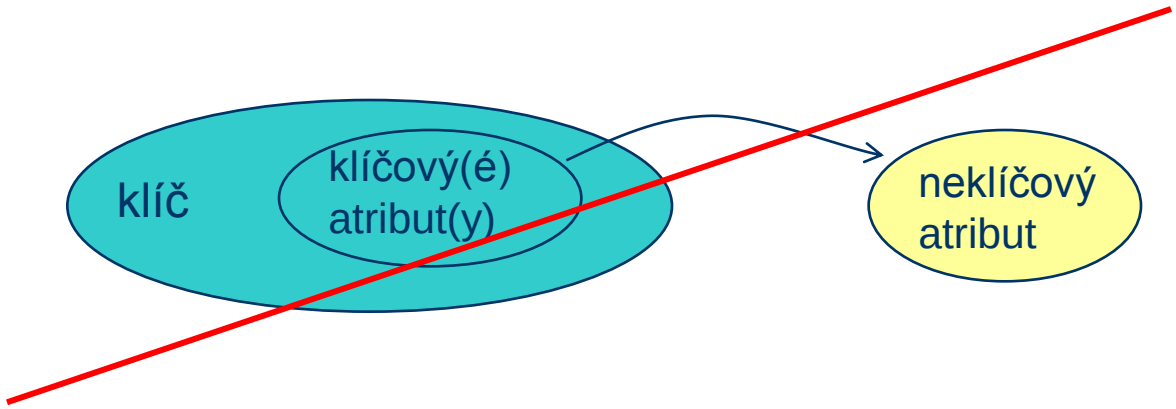
je v 1NF

Zaměstnanec(Id: **Integer**, Podřízení : **Osoba[ ]**, Nadřízený : **Osoba**)

není v 1NF
(vnořená tabulka typu Osoba v atributu Podřízení a atribut Nadřízený je strukturovaný)

Druhá normální forma (2NF)

- neexistují částečné závislosti neklíčových atributů na (libovolném) klíči, tj. platí
  $\forall x \in NK \nexists KK : KK \rightarrow x$
  (kde NK je množina neklíčových atributů a KK je podmnožina nějakého klíče)



Příklad – 2NF (1)

Firma	DB server	Sídlo	Datum zakoupení
Frantova firma	Oracle	Praha	1995
Frantova firma	MS SQL	Praha	2001
Pepova firma	IBM DB2	Brno	2004
Pepova firma	MS SQL	Brno	2002
Pepova firma	Oracle	Brno	2005

Firma, DB Server $\rightarrow$ *všechno*

Firma $\rightarrow$ Sídlo

není ve 2NF, protože Sídlo závisí na části klíče (Firma)

důsledek: redundance hodnot Sídla

Příklad – 2NF (2)

Firma	DB server	Datum zakoupení
Frantova firma	Oracle	1995
Frantova firma	MS SQL	2001
Pepova firma	IBM DB2	2004
Pepova firma	MS SQL	2002
Pepova firma	Oracle	2005

Firma, DB Server → *všechno*

Firma	Sídlo
Frantova firma	Praha
Pepova firma	Brno

Firma → Sídlo

obě schémata jsou ve 2NF

Tranzitivní závislost na klíči

- závislost A → B taková, že A $\not\rightarrow$ *nějaký klíč*
  (A není klíč ani nadklíč), tj. obdržíme tranzitivitu *klíč* → A → B

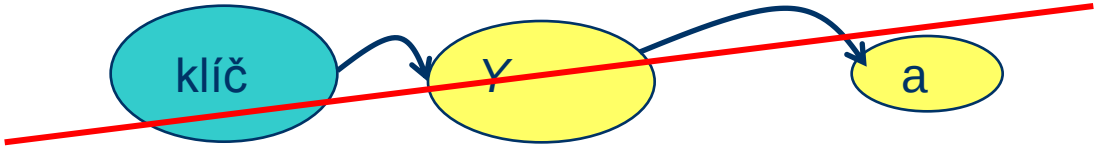
- Samo o sobě poukazuje na hrozbu redundance, neboť z definice FZ jako zobrazení:
  - unikátní hodnoty ***klíč*** se zobrazí na stejně nebo **méně** unikátních hodnot **A** a ty se zobrazí na stejně nebo **méně** unikátních hodnot **B**

Příklad v 2NF:
PSČ → Město → Stát

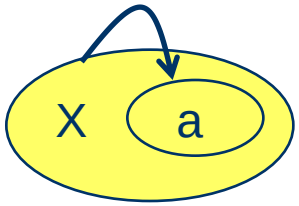
PSČ	Město	Stát
CZ 118 00	Praha	ČR
CZ 190 00	Praha	ČR
CZ 772 00	Olomouc	ČR
CZ 783 71	Olomouc	ČR
SK 911 01	Trenčín	SR

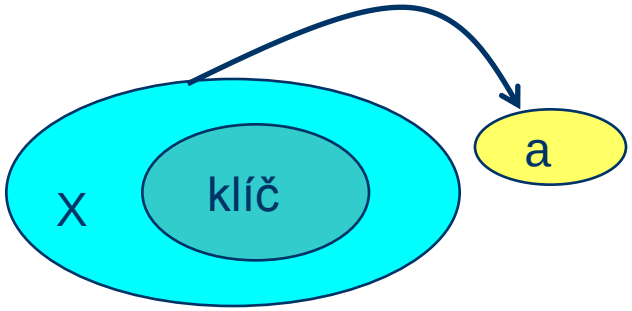
Třetí normální forma (3NF)

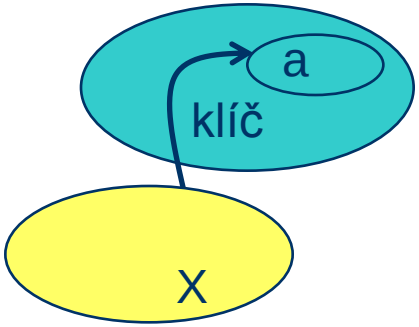
- žádný neklíčový atribut není tranzitivně závislý na žádném klíči



- protože výše uvedená definice není ověřitelná bez konstrukce F+, použijeme definici přímo pro efektivní ověření se znalostí pouze R(A, F)
  - platí alespoň jedna z podmínek pro každou závislost $X \rightarrow a$ (kde $X \subseteq A$, $a \in A$)
    - **závislost je triviální**
    - **X je nadklíč**
    - ***a* je částí klíče**







Příklad – 3NF (1)

Firma	Sídlo	PSČ
Frantova firma	Praha	11800
Martinova firma	Ostrava	70833
Pavlova firma	Brno	22012
Viktorova firma	Praha	11000
Pepova firma	Brno	22012

Firma $\rightarrow$ *všechno*

PSČ $\rightarrow$ Sídlo

je ve 2NF, není ve 3NF (tranzitivní závislost Sídla na klíči přes PSČ)

důsledek: redundance hodnot Sídla

Příklad – 3NF (2)

Firma	PSČ
Frantova firma	11800
Martinova firma	70833
Pavlova firma	22012
Viktorova firma	11000
Pepova firma	22012

PSČ	Sídlo
11800	Praha
70833	Ostrava
22012	Brno
11000	Praha

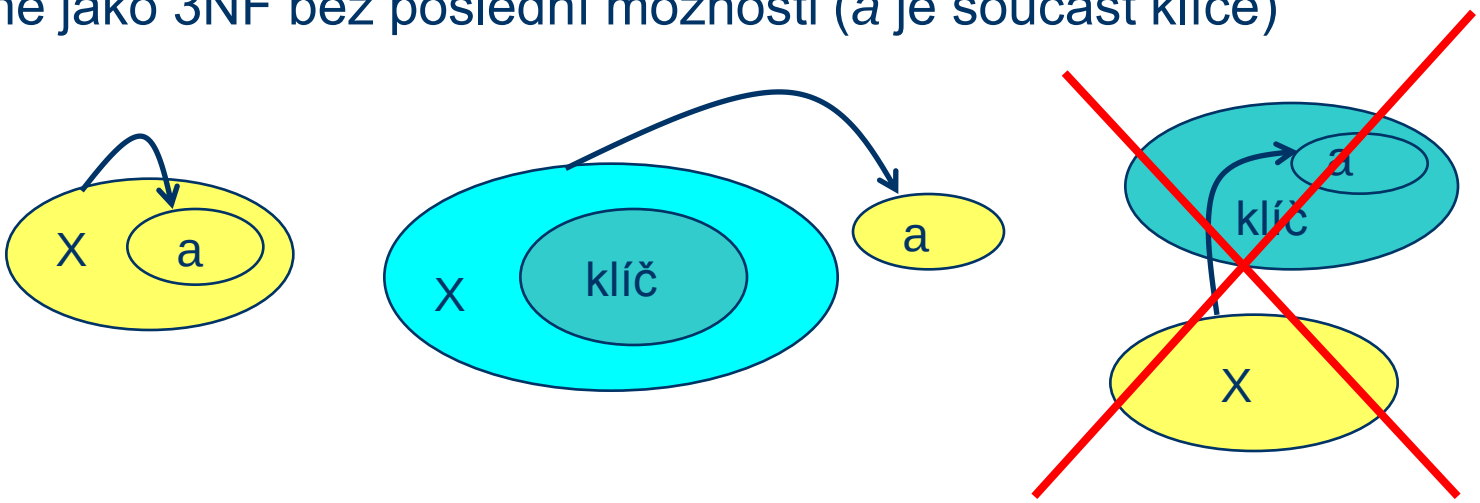
Firma → *všechno*

PSČ → *všechno*

obě schémata jsou ve 3NF

Boyce-Coddova normální forma (BCFN)

- každý atribut je netranzitivně závislý na klíči
- přesněji, v daném schématu R(A, F) platí alespoň jedna z podmínek pro každou závislost $X \rightarrow a$ (kde $X \subseteq A$, $a \in A$)
  - závislost je triviální
  - X je nadklíč
- stejně jako 3NF bez poslední možnosti (*a* je součást klíče)



Příklad – BCNF (1)

Destinace	Pilot	Letadlo	Den
Paříž	kpt. Ptáček	Boeing #1	pondělí
Paříž	kpt. Ptáček	Boeing #2	úterý
Berlín	kpt. Vogel	Airbus #1	pondělí

Pilot, Den $\rightarrow$ *všechno*

Letadlo, Den $\rightarrow$ *všechno*

Destinace $\rightarrow$ Pilot

je ve 3NF, není v BCNF (Pilot závisí na Destinaci, což není nadklíč)

důsledek: redundance hodnot Pilot

Příklad – BCNF (2)

Destinace	Pilot
Paříž	kpt. Ptáček
Berlín	kpt. Vogel

Destinace	Letadlo	Den
Paříž	Boeing #1	pondělí
Paříž	Boeing #2	úterý
Berlín	Airbus #1	pondělí

Destinace → Pilot

Letadlo, Den → *všechno*

nyní jsou obě schémata je v BCNF

Další normální formy

- 4NF – multizávislosti
- 5NF

Databázové systémy

Tomáš Skopal

- relační model
  * základní algoritmy
  * hledání klíčů
  * dekompozice a syntéza

Osnova přednášky

- algoritmy
  - pro analýzu schémat
    - základní algoritmy (atributový uzávěr, příslušnost a redundance FZ)
    - hledání klíčů
    - testování normálních forem
  - pro normalizaci univerzálního schématu
    - dekompozice
    - syntéza

Algoritmus atributového uzávěru

- atributový uzávěr množiny atributů X vůči množině závislostí F
  - princip: postupně odvozujeme všechny atributy „F-určené“ atributy z X
  - polynomiální složitost (O(m*n), kde **n** je počet atributů a **m** počet závislostí)

algorithm ***AttributeClosure***(set of dependencies F, set of attributes X) : **returns set** X^+

```
ClosureX := X; DONE := false; m = |F|;
while not DONE do
        DONE := true;
        for i := 1 to m do
                if (LS[i] ⊆ ClosureX and RS[i] ⊄ ClosureX) then
                        ClosureX := ClosureX ∪ RS[i];
                        DONE := false;
                endif
        endfor
endwhile
return ClosureX;
```

Poznámka: výraz LS[i] (resp. RS[i]) představuje levou (pravou) stranu i-té závislosti v F
Využije se triviální FZ (inicializace algoritmu) a potom tranzitivity (test levé strany v uzávěru).
Využití kompozice a dekompozice je skrytá v testu inkluze.

Příklad – atributový uzávěr

$F = \{a \rightarrow b,\ bc \rightarrow d,\ bd \rightarrow a\}$

$\{b,c\}^+ = ?$

1. ClosureX := {b,c} (inicializace)
2. ClosureX := ClosureX $\cup$ {d} = {b,c,d} ($bc \rightarrow d$)
3. ClosureX := ClosureX $\cup$ {a} = {a,b,c,d} ($bd \rightarrow a$)

$\{b,c\}^+ = \{a,b,c,d\}$

Algoritmus příslušnosti

- často potřebujeme zjistit příslušnost nějaké závislosti $X \rightarrow Y$ do F^+, tj. vyřešit problém $\{X \rightarrow Y\} \in F^+$
- počítat celý F^+ je nepraktické, lze použít algoritmus atributového uzávěru

algorithm ***IsDependencyInClosure***(set of dependencies F, dependency $X \rightarrow Y$)

return $Y \subseteq$ *AttributeClosure*(F, X);

Testování redundancí

Algoritmus příslušnosti lze jednoduše použít k testu redundance

- závislosti $X \rightarrow Y$ v F.
- atributu v X (vzhledem k F a $X \rightarrow Y$).

algorithm ***IsDependencyRedundant***(set of dependencies F, dependency $X \rightarrow Y \in F$)
    return *IsDependencyInClosure*(F – {$X \rightarrow Y$}, $X \rightarrow Y$);

algorithm ***IsAttributeRedundant***(set of deps. F, dep. $X \rightarrow Y \in F$, attribute $a \in X$)
    return *IsDependencyInClosure(*F, X – {a} $\rightarrow$ Y*);*

V dalším výkladu nám bude užitečný algoritmus vracející k FZ redukovanou levou stranu:

algorithm ***GetReducedAttributes***(set of deps. F, dep. $X \rightarrow Y \in F$)
    X' := X;
    for each $a \in X$ **do**
        if *IsAttributeRedundant*(F, X' $\rightarrow$ Y, a) **then** X' := X' – {a};
    endfor
return X';

Minimální pokrytí

- použijeme postupně na všechny FZ testy redundance a ty odstraníme

algorithm ***GetMinimumCover***(set of dependencies F): returns minimal cover G

decompose each dependency in F into elementary ones

for each $X \rightarrow Y$ **in** F **do**

F := (F – {$X \rightarrow Y$}) $\cup$ {*GetReducedAttributes*(F, $X \rightarrow Y$) $\rightarrow$ Y};

endfor

for each $X \rightarrow Y$ **in** F **do**

if *IsDependencyRedundant*(F, $X \rightarrow Y$) **then** F := F – {$X \rightarrow Y$};

endfor

return F;

Nalezení (prvního) klíče

- algoritmus testu redundance atributu lze přímo použít při hledání klíčů
- postupně se odstraňují redundantní atributy z $A \rightarrow A$

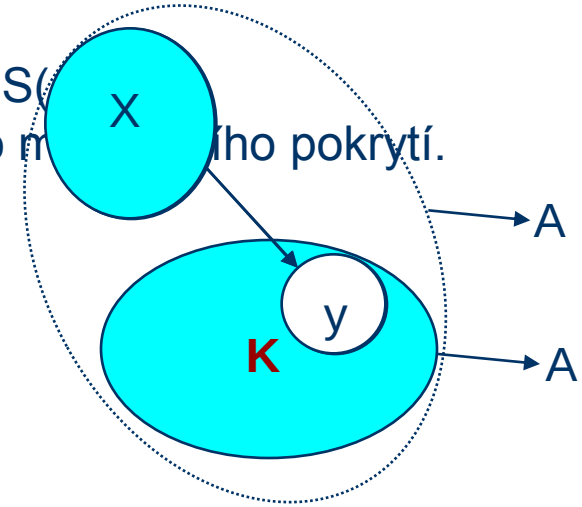
algorithm ***GetFirstKey***(set of deps. F, set of attributes A) : **returns a key** K;

return *GetReducedAttributes*(F, $A \rightarrow A$);

Poznámka: Klíčů samozřejmě může být víc, algoritmus najde jen jeden (který – to záleží na pořadí procházení množiny atributů uvnitř algoritmu GetReducedAttributes).

Nalezení všech klíčů, princip

Máme schéma S(
Převeďme F do n ího pokrytí.



1. Nalezněme libovolný klíč **K** (viz předchozí slajd).
2. V F nalezněme funkční závislost **X** → **y** takovou, že **y** ∈ **K**. (pokud neexistuje, končíme, další klíč není)
3. Protože **X** → **y** a **K** → **A**, platí tranzitivně i **X**{**K** – **y**} → **A**, tj. **X**{**K** – **y**} je nadklíč.
4. Zredukujeme závislost **X**{**K** – **y**} → **A** a tím na levé straně dostaneme klíč **K'**.
   Tento klíč je nutně různý od **K**, protože jsme z něj odstranili **y**.
5. Pokud **K'** zatím není mezi nalezenými klíči, přidáme jej,
   prohlásíme **K**=**K'** a celý postup opakujeme od kroku 2. V opačné případě končíme.

Nalezení všech klíčů, algoritmus

- Lucchesi-Osborn algoritmus
- k již nalezenému klíči hledáme ekvivalentní množiny atributů, tj. jiné klíče
- NP-úplný problém (teoreticky exponenciální počet klíčů/závislostí)

algorithm ***GetAllKeys***(set of deps. F, set of attributes A) : **returns set of all keys** Keys;
    let all dependencies in F be non-trivial, i.e. replace every $X \rightarrow Y$ by $X \rightarrow (Y - X)$
    K := *GetFirstKey*(F, A);
    Keys := {K};
    Done := **false**;
    while Done = **false do**
        Done := **true**;
        for each $X \rightarrow Y \in F$ **do**
            if $(Y \cap K \neq \emptyset$ **and** $\neg \exists K' \in$ Keys : $K' \subseteq (K \cup X) - Y)$ **then**
                K := *GetReducedAttributes*(F, $((K \cup X) - Y) \rightarrow A$);
                Keys := Keys $\cup$ {K};
                Done := **false**;
        endfor
    endwhile
return Keys;

Příklad – nalezení všech klíčů

Contracts(A, F)

A = {c = ContractId, s = SupplierId, j = ProjectId, d = DeptId,
p = PartId, q = Quantity, v = Value}

F = {**c** → *all*, **sd** → **p**, **p** → **d**, **jp** → **c**, **j** → **s**}

1. Najdu první klíč – Keys = {**c**}
2. Iterace 1: najdu **jp** → **c**, která má na pravé straně kus posledního klíče (v tomto případě celý klíč – **c**) a zároveň **jp** není nadmnožinou již nalezeného klíče
3. **jp** → *all* je redukovaná (žádný redundantní atribut), tj.
4. Keys = {**c, jp**}
5. Iterace 2: najdu **sd** → **p**, má na pravé straně kus posledního klíče (**jp**), {**jsd**} není nadmnožinou ani **c** ani **jp**, tj. je to kandidát na klíč
6. v **jsd** → *all* je redundantní atribut **s**, tj.
7. Keys = {**c, jp, jd**}
8. Iterace 3: najdu ještě **p** → **d**, nicméně **jp** už jsme našli, takže nepřidávám nic
9. končím, iterace 3 proběhla naprázdno

Testování normálních forem

- NP-úplný problém
  - buď musím znát všechny klíče – pak stačí otestovat jen FZ z F, nemusím testovat celý F^+
  - nebo musím znát jeden klíč, ale zase potřebuji F rozšířit na celé F^+
- naštěstí v praxi je nalezení všech klíčů rychlé
  - díky omezené velikosti F a „separovanosti“ závislostí v F

Návrh schématu databáze

Dva způsoby modelování relační databáze:

- získám množinu relačních schémat (ručně nebo např. převodem z ER diagramu)
  - normalizaci pro dodržení NF provádím pro každou tabulku zvlášť
  - riziko nadbytečného „rozdrobení“ databáze na příliš mnoho tabulek
- chápu modelování celé databáze na úrovni globálních atributů a navrhnu tzv. univerzální schéma databáze – tj. jednu velkou tabulku – včetně množiny globálně platných funkčních závislostí
  - normalizaci pro dodržení NF provádím najednou pro celou databázi
  - menší riziko „rozdrobení“
  - „entity“ jsou vygenerovány (rozpoznány) jako důsledky FZ
  - modelování na úrovni atributů je méně intuitivní než např. ER modelování
- můžu zkombinovat oba přístupy – tj. nejprve vytvořit ER model databáze, ten převést do schémat a postupně některé sloučit (v krajním případě všechny)

Normalizace relačního schématu

- jediný způsob – dekompozice na více schémat
  - případně nejdříve sloučení více „nenormálních“ schémat a pak dekompozice
- přístupy podle různých kritérií
  - **zachování integrity dat**
    - tzv. **bezztrátovost**
    - tzv. **pokrytí závislostí**
  - požadavek na NF (3NF nebo BCNF)
- ručně nebo algoritmicky

Proč zachovávat integritu?

Pokud dekompozici nijak neomezíme, můžeme rozložit tabulku na několik jednosloupcových, které jistě všechny splňují BCNF.

Firma	Sídlo	Nadmořská výška
Sun	Santa Clara	25 mnm
Oracle	Redwood	20 mnm
Microsoft	Redmond	10 mnm
IBM	New York	15 mnm

Firma
Sun
Oracle
Microsoft
IBM

Firma

Sídlo
Santa Clara
Redwood
Redmond
New York

Sídlo

Nadmořská výška
25 mnm
20 mnm
10 mnm
15 mnm

Nadmořská výška

Firma,
Sídlo → Nadmořská výška

Evidentně je ale s takovouto dekompozicí něco špatně...

...je **ztrátová** a nezachovává **pokrytí závislostí**

Bezztrátovost

- vlastnost dekompozice, která zaručuje korektní rekonstrukci univerzální relace z dekomponovaných relací

- Definice 1:
  Nechť $R(\{X \cup Y \cup Z\}, F)$ je univerzální schéma, kde $Y \rightarrow Z \in F$. Potom dekompozice $R_1(\{Y \cup Z\}, F_1)$, $R_2(\{Y \cup X\}, F_2)$ je bezztrátová.

- Alternativní Definice 2:
  Dekompozice $R(A, F)$ do $R_1(A_1, F_1)$, $R_2(A_2, F_2)$ je bezztrátová, jestliže platí $A_1 \cap A_2 \rightarrow A_1$ nebo $A_2 \cap A_1 \rightarrow A_2$

- **Alternativní Definice 3:**
  Dekompozice $R(A, F)$ na $R_1(A_1, F_1)$, ..., $R_n(A_n, F_n)$ je bezztrátová, pokud platí $R' = *_{i=1..n} R'[A_i]$.

  Poznámka: R' je instance schématu R (tj. konkrétní relace/tabulka s daty). Operace * je přirozené spojení relací a $R'[A_i]$ je projekce relace R' na podmnožinu atributů $A_i \subseteq A$. (operace budou blíže vysvětleny na příští přednášce)

Příklad – ztrátová dekompozice

Firma	Používá DBMS	Spravuje dat
Sun	Oracle	50 TB
Sun	DB2	10 GB
Microsoft	MSSQL	30 TB
Microsoft	Oracle	30 TB

Firma, Používá DBMS

Firma	Používá DBMS
Sun	Oracle
Sun	DB2
Microsoft	MSSQL
Microsoft	Oracle

Firma, Používá DBMS

Firma	Spravuje dat
Sun	50 TB
Sun	10 GB
Microsoft	30 TB

Firma, Spravuje dat

„rekonstrukce“
(přirozené spojení)

Firma	Používá DBMS	Spravuje dat
Sun	Oracle	50 TB
Sun	Oracle	10 GB
Sun	DB2	10 GB
Sun	DB2	50 TB
Microsoft	MSSQL	30 TB
Microsoft	Oracle	30 TB

Firma, Používá DBMS, Spravuje dat

Příklad – bezztrátová dekompozice

Firma	Sídlo	Nadmořská výška
Sun	Santa Clara	25 mnm
Oracle	Redwood	20 mnm
Microsoft	Redmond	10 mnm
IBM	New York	15 mnm

Firma,
Sídlo → Nadmořská výška

Firma	Sídlo
Sun	Santa Clara
Oracle	Redwood
Microsoft	Redmond
IBM	New York

Firma

Sídlo	Nadmořská výška
Santa Clara	25 mnm
Redwood	20 mnm
Redmond	10 mnm
New York	15 mnm

Sídlo

„rekonstrukce“
(přirozené spojení)

Pokrytí závislostí

- vlastnost dekompozice, která zaručuje zachování všech funkčních závislostí
- Definice:
  Nechť $R_1(A_1, F_1)$, $R_2(A_2, F_2)$ je dekompozicí $R(A, F)$, potom dekompozice zachovává pokrytí závislostí, pokud
  $F^+ = (\cup_{i=1..n} F_i)^+$.
- Pokrytí závislostí může být narušeno dvěma způsoby
  - při dekompozici F neodvodíme všechny FZ platné v Fi – ztratíme FZ, která má přímo platit v jednou dílčím schématu
  - i když odvodíme všechny platné (tj. provedeme projekci F^+), můžeme v důsledku ztratit FZ, která platí **napříč** schématy

Příklad – pokrytí závislostí

Firma	Sídlo	Nadmořská výška
Sun	Santa Clara	25 mnm
Oracle	Redwood	20 mnm
Microsoft	Redmond	10 mnm
IBM	New York	15 mnm

pokrytí porušeno, ztratili jsme
Sídlo → Nadmořská výška

Firma	Nadmořská výška
Sun	25 mnm
Oracle	20 mnm
Microsoft	10 mnm
IBM	15 mnm

Firma

Firma	Sídlo
Sun	Santa Clara
Oracle	Redwood
Microsoft	Redmond
IBM	New York

Sídlo

Firma,
Sídlo → Nadmořská výška

pokrytí zachováno

Firma	Sídlo
Sun	Santa Clara
Oracle	Redwood
Microsoft	Redmond
IBM	New York

Firma

Sídlo	Nadmořská výška
Santa Clara	25 mnm
Redwood	20 mnm
Redmond	10 mnm
New York	15 mnm

Sídlo

Algoritmus „Dekompozice“

- algoritmus pro dekompozici do BCNF, zachovávající bezztrátovost
- nezachovává pokrytí závislostí
  - nezávisí na algoritmu – někdy prostě nelze dekomponovat do BCNF a zároveň pokrýt závislosti

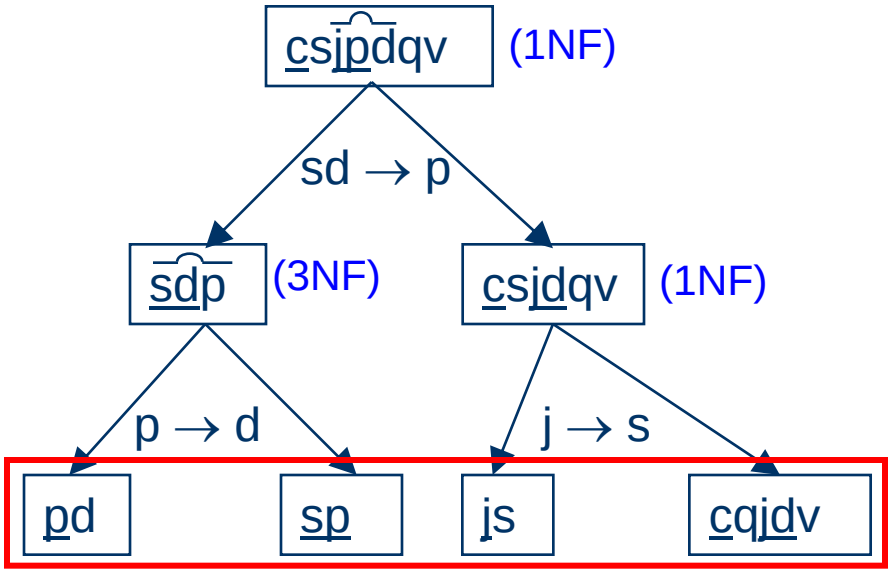
algorithm ***Decomposition***(set of elem. deps. F, set of attributes A) : **returns set** $\{R_i(A_i, F_i)\}$
 Result := {R(A, F)};
 Done := **false**;
 Create F^+;
 while not Done **do**
  if $\exists R_i(F_i, A_i) \in$ Result not being in BCNF **then** *// pokud ve výsledku je schéma porušující BCNF*
   Let $X \rightarrow Y \in F_i$ such that $X \rightarrow A_i \notin F^+$. *// X není (nad)klíč a $X \rightarrow Y$ tedy narušuje BCNF*
   Result := (Result – $\{R_i(A_i, F_i)\}$) ∪ *// odebereme rozkládané schéma z výsledku*
   $\{R_i(A_i – Y, \text{cover}(F, A_i – Y))\} \cup$ *// přidáme rozkládané schéma bez atrib. Y*
   $\{R_j(X \cup Y, \text{cover}(F, X \cup Y))\}$ *// přidáme schéma s atrib. XY*
  else
   Done := **true**;
  endwhile
 return Result;

Tato dílčí dekompozice na dvě tabulky je bezztrátová, dostaneme dvě schémata, která obě obsahují X, druhé navíc pouze Y a platí $X \rightarrow Y$. X je nyní v druhé tab. nadklíčem a $X \rightarrow Y$ tedy již neporušuje BCNF (v první tab. už není Y).

Poznámka: Funkce cover(X, F) vrátí všechny závislosti platné na atributech z X, tj. podmnožinu z F^+ takovou, která obsahuje pouze atributy z X. Proto je nutné počítat explicitně F^+.

Příklad – dekompozice

Contracts(A, F)
A = {c = ContractId, s = SupplierId, j = ProjectId, d = DeptId, p = PartId, q = Quantity, v = Value}
F = {c → all, sd → p, p → d, jp → c, j → s}



Nepokryté závislosti:
c → p
sd → p

cp (BCNF)

sdp (3NF)

Algoritmus „Syntéza“

- algoritmus pro dekompozici do 3NF, zachovávající pokrytí závislostí
- základní verze nezachovává bezztrátovost

algorithm ***Synthesis***(set of elem. deps. F, set of attributes A) : **returns set** $\{R_i(F_i, A_i)\}$
    create minimal cover from F into G
    compose FDs having equal left side into a single FD
    every composed FD forms a scheme R_i (A_i, F_i) of decomposition
return $\cup_{i=1..n}\{R_i (A_i, F_i)\}$

- bezztrátovost lze zajistit přidáním dalšího schématu do dekompozice, které obsahuje univerzální klíč (tj. nějaký klíč původního univerzálního schématu)
- schéma v dekompozici, které je podmnožinou jiného můžu vypustit
- můžu se pokusit sloučit schémata s funkčně ekvivalentními klíči, ale tato operace může obecně porušit 3NF!! (nebo BCNF pokud jí bylo dosaženo)

Příklad – syntéza

Contracts(A, F)
A = {c = ContractId, s = SupplierId, j = ProjectId, d = DeptId, p = PartId, q = Quantity, v = Value}
F = {c → sjdpqv, sd → p, p → d, jp → c, j → s}

Minimální pokrytí:
V závislostech z F nejsou redundantní atributy. Byly vyřazeny redundantní FZ c → s a c → p.
G = {c → j, c → d, c → q, c → v, sd → p, p → d, jp → c, j → s}

Kompozice podle levých stran:
G' = {c → jdqv, sd → p, p → d, jp → c, j → s}

Výsledek:
R_1({cqjdv}, {c → jdqv}), R_2({sdp}, {sd → p}), ~~R_3({pd},{p → d})~~ (podmnožina v R_2), R_4({jpc}, {jp → c}), R_5({js}, {j → s}))

Ekvivalentní klíče: {c, jp, jd}
R_1({cqjpdv}, {c → jdqv, jp → c}), R_2({sdp}, {sd → p, p → d}), R_5({js}, {j → s}))

sloučení R_1 a R_4
(nyní ale p → d porušuje BCNF)

Bernsteinovo rozšíření syntézy

- pokud by sloučení schémat podle ekvivalence klíčů K_1, K_2 porušilo 3NF, provedeme dekompozici znovu
  1. $F_{new} = F \cup \{K_1 \rightarrow K_2, K_2 \rightarrow K_1\}$
  2. zjistíme redundantní závislosti v F_{new}, ale odstraníme je z F
  3. tabulky se navrhnou z redukované F a $\{K_1 \cup K_2\}$

Demo

- program Databázové algoritmy
  - stáhnete z mého webu
- příklad 1
- příklad 2

Databázové systémy

Tomáš Skopal

- relační model
  * relační algebra

Osnova přednášky

- relační algebra
  - operace na relacích
  - ekvivalentní dotazy
  - relační úplnost

Dotazování v relačním modelu

- smyslem každé databáze je zejména poskytovat data – ty jsou získávány formulováním dotazů na databázi
  - dotaz je formulován v **dotazovacím jazyce**
  - dotazovací jazyk musí být natolik **silný**, aby umožnil získat libovolnou (smysluplnou) podmnožinu dat z relací (tabulek)
  - každý dotazovací jazyk využívá symbolů popisujících schémata relací jako základní konstrukty jazyka

Databázový dotaz

- dotaz = vymezení konkrétní instance dat
  - jediný dotaz může být vyjádřen několika výrazy v dotazovacím jazyce – **ekvivalentní výrazy**

- rozsah dotazu
  - v **klasických modelech** může být výsledkem dotazu pouze podmnožina dat z databáze
    (tj. hodnoty přímo obsažené v DB)
  - v **rozšířených modelech** může výsledek dotazu obsahovat i odvozená data (tj. výpočty, statistiky, atd. z původních dat)

Dotazovací formalismy

- díky pohledu na data jako na relace můžeme využít dvou matematických formalismů
  - **relační algebra**
    (využívající množiny operací na DB relacích)
  - **relační kalkuly**
    (databázové rozšíření predikátové logiky 1. řádu)

Relační algebra (RA)

- RA je množina operací (unárních, či binárních) na relacích se schématy, jejichž výsledkem je opět relace (a její schéma)
  - pro úplnost budeme k relaci **R*** vždy uvažovat i její schéma **R(A)** (název a (typované) atributy), tj. dvojici **<R*, R(A)>**
- schéma $\mathbf{R_x}$ budeme označovat libovolným unikátním jménem (resp. uživatelem zvoleným)
  - pro výsledek operace většinou nepotřebujeme konstruovat nový název relace (ani schématu) – buď relace opět vstupuje do jiné operace anebo je již výsledkem, který
  - pokud budeme chtít výsledek uložit, např. pro zjednodušení zápisu dotazu, použijeme konstrukci
    NazevVysledku := <*operace na relacích*>

Relační algebra (RA)

- pokud je schéma jasné z kontextu, budeme místo značení $<R_1, R_1(A_1)>$ ***operace*** $<R_2, R_2(A_2)>$ užívat zjednodušené značení R_1 ***operace*** R_2
- pro binární operace používáme infixové značení, pro unární operace postfixové značení
- výsledek operace lze opět použít jako operand jiné operace, tj. takto lze zkonstruovat složitější operaci/dotaz
  (<R1*, R1(A)> ***op1*** <R1*, R1(A)>) ***op2***

RA – přejmenování atributů

- přejmenování atributů – unární operace

$$\mathbf{R^*<a_i \rightarrow b_i,\ a_j \rightarrow b_j,\ \ldots >} = <R^*\ ,\ R_x((A - \{a_i,\ a_j,\ \ldots\}) \cup \{b_i,\ b_j,\ \ldots\})>$$

  - pouze se přejmenují atributy schématu, s daty se nic neděje (tj. výslekem je stejná relace a stejné schéma, pouze příslušné atributy mají jiná jména)

RA – množinové operace

- množinové operace (binární, infixová notace)
  - sjednocení – $\mathbf{\langle R_1, R_1(A)\rangle \cup \langle R_2, R_2(A)\rangle} = \langle R_1 \cup R_2 , R_x(A)\rangle$
  - průnik – $\mathbf{\langle R_1, R_1(A)\rangle \cap \langle R_2, R_2(A)\rangle} = \langle R_1 \cap R_2 , R_x(A)\rangle$
  - rozdíl – $\mathbf{\langle R_1, R_1(A)\rangle - \langle R_2, R_2(A)\rangle} = \langle R_1 - R_2 , R_x(A)\rangle$
  - kartézský součin – $\mathbf{\langle R_1, R_1(A)\rangle \times \langle R_2, R_2(B)\rangle}$
    $= \langle R_1 \times R_2 , R_x(\{R_1\} \times A \cup \{R_2\} \times B)\rangle$

- sjednocení, průnik a rozdíl vyžadují **kompatibilní schémata** obou operandů, které je rovněž schématem výsledku

RA – kartézský součin

- kartézský součin konstruuje nové schéma, složené z atributů obou schémat – pokud existují stejná jména atributů, pro rozlišení použije se tečková notace např. $\mathbf{R_1.a}$, resp. $\mathbf{R_2.a}$

- pokud jsou oba operandy kartézského součinu totožné, musíme nejprve provést přejmenování atributů, tj.
  $\mathbf{<R_1, R_1 (\{a,b,c\})> \times R_1<a \rightarrow d, b \rightarrow e, c \rightarrow f>}$

Příklad – množinové operace

FILM(JMENO_FILMU, JMENO_HERCE)

AMERICKE_FILMY = {(‘Titanic’, ‘DiCaprio’), (‘Titanic’, ‘Winslet’), (‘Top Gun’, ‘Cruise’)}

NOVE_FILMY = {(‘Titanic’, ‘DiCaprio’), (‘Titanic’, ‘Winslet’), (‘Samotáři’, ‘Macháček’)}

CESKE_FILMY = {(‘Vesničko má, středisková’, ‘Labuda’), (‘Samotáři’, ‘Macháček’)}

VSECHNY_FILMY := **AMERICKE_FILMY ∪ CESKE_FILMY** =
{(‘Titanic’, ‘DiCaprio’), (‘Titanic’, ‘Winslet’), (‘Top Gun’, ‘Cruise’),
(‘Pelíšky’, ‘Donutil’), (‘Samotáři’, ‘Macháček’)}

STARE_AMERICKE_A_CESKE_FILMY :=
(AMERICKE_FILMY ∪ CESKE_FILMY) – NOVE_FILMY =
{(‘Top Gun’, ‘Cruise’), (‘Vesničko má, středisková’, ‘Labuda’)}

NOVE_CESKE_FILMY := **NOVE_FILMY ∩ CESKE_FILMY** = {(‘Samotáři’, ‘Macháček’)}

Relační algebra

- projekce (unární)

  <R*[C], R(A)> = <{u[C] $\in$ R*}, R(C)>, kde C $\subseteq$ A

  - u[C] je prvek relace zbavený hodnot atributů A – C
  - případné duplicitní prvky jsou odstraněny

RA – selekce

- selekce (unární)

  <R*(φ), R(A)> = <{u | u ∈ R* a φ(u)}, R(A)>

  - výběr těch prvků u relace z R*, které splňují logickou podmínku φ(u)
  - podmínka je zadána Boolským výrazem (tj. pomocí spojek **and, or, not**) atomických formulí $\mathbf{t_1}\ \Theta\ \mathbf{t_2}$ nebo $\mathbf{t_1}\ \Theta\ \mathbf{a}$, kde Θ ∈ {<, >, =, ≥, ≤, ≠} a $\mathbf{t_i}$ jsou jména atributů

RA – přirozené spojení

- přirozené spojení (binární) natural join

  <R*, R(A)> * <S*, S(B)> =
  $<\{u \mid u[A] \in R^* \text{ a } u[B] \in S^*\}, R_x(A \cup B)>$

  - spojení prvků relace přes stejné hodnoty **všech atributů** sdílených mezi A a B
  - pokud $A \cap B = \emptyset$, přirozené spojení je vlastně kartézský součin (spojuje se přes prázdnou množinu, tj. libovolně – „všechno se vším“)
  - lze vyjádřit pomocí kartézského součinu, selekce a projekce

Příklad – selekce, projekce, přir.spojení

FILM(JMENO_FILMU, JMENO_HERCE)
HEREC(JMENO_HERCE, ROK_NAROZENI)
FILMY = {('Titanic', 'DiCaprio'), ('Titanic', 'Winslet'), ('Top Gun', 'Cruise')}
HERCI = {('DiCaprio',1974), ('Winslet',1975), ('Cruise', 1962), ('Jolie', 1975)}

HERECKE_ROCNIKY := **FILMY_HERCI[ROK_NAROZENI]** =
{(1974), (1975), (1962)}

MLADI_HERCI := **HERCI(ROK_NAROZENI > 1970) [JMENO_HERCE]** =
{('DiCaprio'), ('Winslet'), ('Jolie')}

FILMY_S_HERCI := **FILMY * HERCI** =
{('Titanic', 'DiCaprio', 1974), ('Titanic', 'Winslet', 1975), ('Top Gun', 'Cruise', 1962)}

RA – Θ-spojení

- vnitřní Θ-spojení (binární) inner join

$$\mathbf{<R^*, R(A)>[t_1 \Theta t_2]<S^*, S(B)>} = <\{u \mid u[A] \in R^*, u[B] \in S^*, u.t_1 \Theta u.t_2\}, A \cup B>$$

  - zobecnění přirozeného spojení
  - spojuje se přes predikát Θ aplikovaný na jednotlivých atributech (schémat vystupujících v operaci)

RA – levé Θ-spojení

- levé vnitřní Θ-polospojení (binární)
  left inner semi-join

  $<R^*, R(A)><t_1 \Theta t_2]<S^*, S(B)>$ = $(R[t_1 \Theta t_2]S)[A]$

  - spojení omezené na levou stranu (ve spojení nás zajímají pouze atributy A)
  - podobně se definuje pravá varianta (projekce na B)
    - right inner semi-join

RA – dělení

- dělení relací (binární)

  $\langle R^*, R(A) \rangle \div \langle S^*, S(B \subset A) \rangle$ =
  $\langle \{t \mid \forall s \in S^* \; (t \oplus s) \in R^*\}, A - B \}$

  - $\oplus$ je operace zřetězení (z prvků $\langle a_1, a_2, ... \rangle$ a $\langle b_1, b_2, ... \rangle$ se stane $\langle a_1, a_2, ..., b_1, b_2, ... \rangle$)
  - vrací ty prvky z R*,
    které mají na A stejné atributy a na B obsahují **všechny** prvky z S*
  - alternativní definice: **R* ÷ S*** = R*[A–B] – ((R*[A–B] × S*) – R*)[A–B]
  - využívá se tam, kde je potřeba vybrat ty řádky tabulky, jejichž projekce jsou obsaženy ve všech řádcích jiné tabulky

Příklad – dělení

FILM(JMENO_FILMU, JMENO_HERCE) HEREC(JMENO_HERCE, ROK_NAROZENI)

*Ve kterých filmech hráli **všichni** herci?*

HRALI_VSICHNI := **FILMY ÷ HERCI[JMENO_HERCE])** = {('Titanic')}

JMENO_FILMU	JMENO_HERCE
Titanic	DiCaprio
Titanic	Winslet
The Beach	DiCaprio
Enigma	Winslet
The Kiss	Zane
Titanic	Zane

JMENO_HERCE	DATUM_NAROZENI
DiCaprio	1974
Zane	1966
Winslet	1975

Vnitřní vs. vnější spojení

- dosud uvedené operace spojení se nazývají **vnitřní spojení** (inner join)
- v praxi je užitečné zavést **nulové metahodnoty** (NULL) atributů
- **vnější spojení** využívá doplnění nulových hodnot k těm prvkům, které nebylo možno „normálně“ spojit (tj. neobjevily se ve vnitřním spojení)
  - levé vnější spojení (left outer join)

    $\mathbf{R} *_L \mathbf{S} = (R * S) \cup (\underline{R} \times (\text{NULL}, \text{NULL}, \ldots))$
  - pravé vnější spojení (right outer join)

    $\mathbf{R} *_R \mathbf{S} = (R * S) \cup ((\text{NULL}, \text{NULL}, \ldots) \times \underline{S})$

    kde $\underline{R}$, resp. $\underline{S}$ obsahují n-tice nespojitelné s S, resp. R
  - plné vnější spojení (full outer join)

    $\mathbf{R} *_F \mathbf{S} = (R *_L S) \cup (R *_R S)$
  - uvedená vnější spojení představují „přirozenou“ variantu, podobně se definují i vnější Θ-spojení

Příklad – všechny typy spojení

tabulka **Lety**

Let	Společnost	Destinace	Počet cestujících
OK251	CSA	New York	276
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
AC906	Air Canada	Torronto	116
KL1245	KLM	Amsterdam	130

tabulka **Letadla**

Letadlo	Kapacita
Boeing 717	106
Airbus A380	555
Airbus A350	253

Dotaz: Kterými letadly mohou letět všichni cestující v daném letu tak, aby počet neobsazených míst v letadle byl menší než 200?

Vnitřní Θ-spojení – chceme **právě** ty lety a letadla, která vyhovují danému kritériu:
Lety [Lety.PocetCestujich ≤ Letadla.Kapacita AND Lety.PocetCestujich + 200 > Letadla.Kapacita] Letadla

Levé/pravé/plné vnější Θ-spojení – chceme vedle letů-letadel specifikovaných vnitřním spojením taky ty lety a letadla, která danému kritériu nevyhovují **ani v jednom** případě.

Let	Společnost	Destinace	Počet cest.	Letadlo	Kapacita
OK251	CSA	New York	276	NULL	NULL
KL1245	KLM	Amsterdam	130	Airbus A350	253
AC906	Air Canada	Torronto	116	Airbus A350	253
LH438	Lufthansa	Stuttgart	68	Airbus A350	253
LH438	Lufthansa	Stuttgart	68	Boeing 717	106
OK012	CSA	Milano	37	Boeing 717	106
NULL	NULL	NULL	NULL	Airbus A380	555

plné vnější spojení

vnitřní spojení

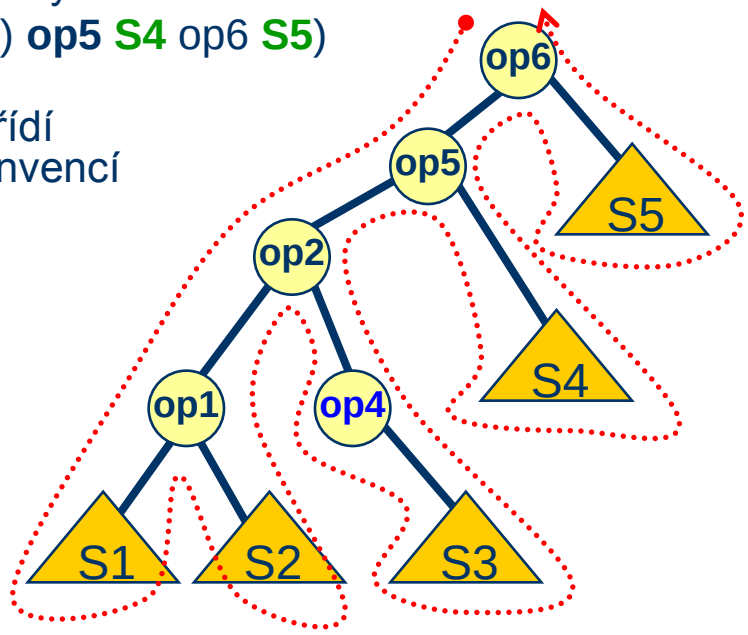
levé vnější spojení

pravé vnější spojení

levé/pravé polospojení (bez prvního a posledního řádku + po odstranění duplicit)

Vyhodnocení dotazu v relační algebře

- logické pořadí vyhodnocení operací v dotazu
  - pro vyhodnocení operace se musí nejprve vyhodnotit její operandy – vede k průchodu syntaktickým stromem do hloubky
  - např. (...((**S1** **op1** **S2**) **op2** (**op4** **S3**)) **op5** **S4** op6 **S5**)
  - konstrukce stromu (parsování dotazu) se řídí prioritou operací, závorkami, případně konvencí levé/pravé asociativity

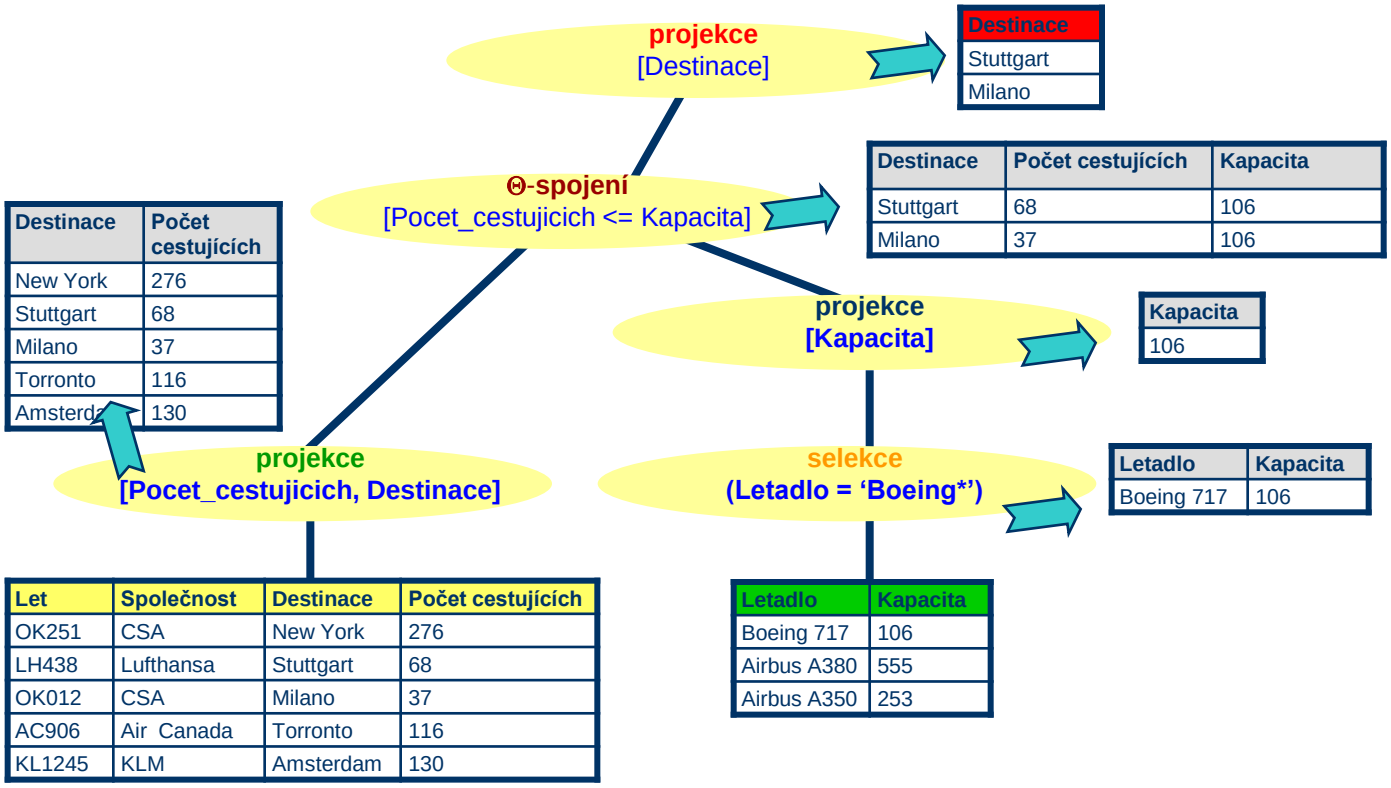


- precedence operací (priorita)
  1. projekce R[] (nejvyšší)
  2. selekce R()
  3. kart. součin ×
  4. spojení, dělení *, ÷
  5. rozdíl –
  6. sjednocení, průnik ∪, ∩ (nejnižší)

Příklad – vyhodnocení dotazu

Do kterých destinací může přiletět Boeing? (tak, aby se cestující daného letu vešli všichni do letadla)

(Lety[Pocet_cestujicich, Destinace] [Pocet_cestujicich <= Kapacita] (Letadla(Letadlo = 'Boeing*')[Kapacita]))[Destinace]



Ekvivalentní výrazy

- tentýž dotaz lze vyjádřit různými výrazy
  - nahradíme-li „redundantní“ operace základními (např. dělení, přirozené spojení)
  - využijeme-li komutativity, distributivity a asociativity (některých) operací
- selekce
  - kaskáda selekcí: $(\ldots((R(\varphi_1))(\varphi_2))\ldots)(\varphi_n) \equiv R(\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n)$
  - komutativita selekcí: $(R(\varphi_1))(\varphi_2) \equiv (R(\varphi_2))(\varphi_1)$
- projekce
  - kaskáda projekcí: $(\ldots(R[A_1])[A_2])\ldots)[A_n] \equiv R[A_n]$, kde $A_n \subseteq A_{n-1} \subseteq \ldots \subseteq A_2 \subseteq A_1$
- spojení a kart. součin
  - komutativita: $R \times S \equiv S \times R$, $R\ [\Theta]\ S \equiv S\ [\Theta]\ R$, atd.
  - asociativita: $R \times (S \times T) \equiv (R \times S) \times T$, $R\ [\Theta]\ (S\ [\Theta]\ T) \equiv (R\ [\Theta]\ S)\ [\Theta]\ T$, atd.
  - kombinace, např: $R\ [\Theta]\ (S\ [\Theta]\ T) \equiv (R\ [\Theta]\ T)\ [\Theta]\ S$

Ekvivalentní výrazy

- složené ekvivalence pro selekci, projekci a spojení
  - výměna selekce a projekce:
    $\mathbf{(R[A_i])(\varphi) \equiv (R(\varphi))[A_i]}$, pokud $\forall a \in \varphi \Rightarrow a \in A_i$
  - kombinace selekce a kartézského součinu (definice spojení):
    $\mathbf{R\ [\Theta]\ S \equiv (R \times S)(\Theta)}$
  - výměna selekce a kart. součinu (nebo spojení)
    $\mathbf{(R \times S)(\varphi) \equiv R(\varphi) \times S}$, pokud $\forall a \in \varphi \Rightarrow a \in A_R \wedge a \notin A_S$
  - výměna projekce a kart. součinu (nebo spojení)
    $\mathbf{(R \times S)[A_1] \equiv R[A_2] \times S[A_3]}$,
    pokud $A_2 \subseteq A_1 \wedge A_2 \subseteq R_A$ a $A_3 \subseteq A_1 \wedge A_3 \subseteq S_A$
    Podobně pro spojení, $\mathbf{(R\ [\Theta]\ S)[A_1] \equiv R[A_2]\ [\Theta]\ S[A_3]}$,
    kde navíc $\forall a \in \Theta \Rightarrow a \in A_1$
- další ekvivalence dostaneme zapojením množinových operací

Příklad – přirozené spojení

$$\langle R, A_R \rangle * \langle S, A_S \rangle \equiv (R \times S)(\forall a \in A_R \cap A_S \Rightarrow R.a = S.a)[(\{R\} \times A_R) \cup (\{S\} \times (A_S - A_R))]$$

projekce
$[(\{R\} \times A_R) \cup (\{S\} \times (A_S - A_R))]$

R.Let	R.Společnost	R.Destinace	R.Kapacita	S.Letadlo
OK251	CSA	New York	276	Boeing 717
KL1245	KLM	Amsterdam	130	Airbus A350

selekce
$[(\forall a \in A_R \cap A_S \Rightarrow R.a = S.a)]$

R.Let	R.Společnost	R.Destinace	R.Kapacita	S.Společnost	S.Letadlo	S.Kapacita
OK251	CSA	New York	276	CSA	Boeing 717	276
KL1245	KLM	Amsterdam	130	KLM	Airbus A350	130

kart. součin

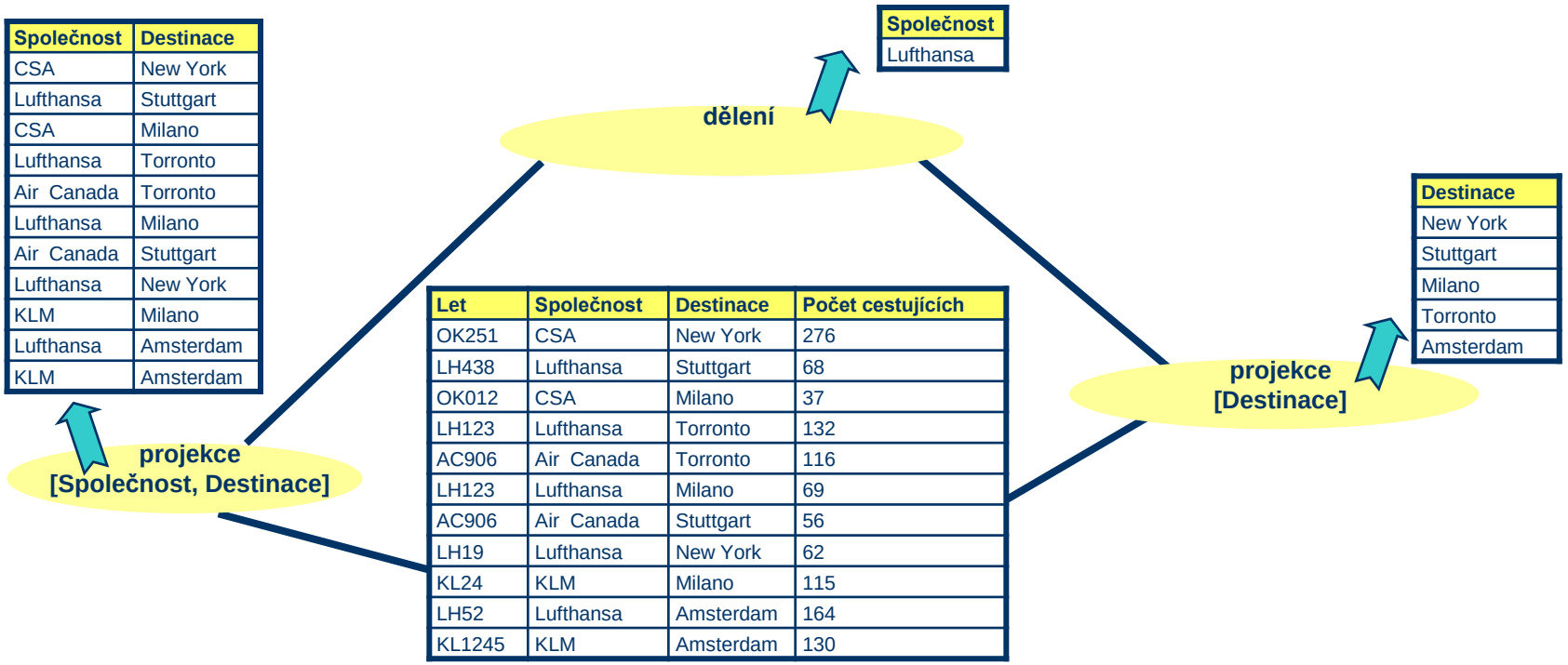
R.Let	R.Společnost	R.Destinace	R.Kapacita	S.Společnost	S.Letadlo	S.Kapacita
OK251	CSA	New York	276	CSA	Boeing 717	276
OK251	CSA	New York	276	CSA	Airbus A380	555
OK251	CSA	New York	276	KLM	Airbus A350	130
AC906	Air Canada	Torronto	116	CSA	Boeing 717	276
AC906	Air Canada	Torronto	116	CSA	Airbus A380	555
AC906	Air Canada	Torronto	116	KLM	Airbus A350	130
KL1245	KLM	Amsterdam	130	CSA	Boeing 717	276
KL1245	KLM	Amsterdam	130	CSA	Airbus A380	555
KL1245	KLM	Amsterdam	130	KLM	Airbus A350	130

Let	Společnost	Destinace	Kapacita
OK251	CSA	New York	276
AC906	Air Canada	Torronto	116
KL1245	KLM	Amsterdam	130

Společnost	Letadlo	Kapacita
CSA	Boeing 717	276
CSA	Airbus A380	555
KLM	Airbus A350	130

Příklad – dělení relací

Které společnosti létají do všech destinací?
Lety[Společnost, Destinace] ÷ Lety[Destinace]

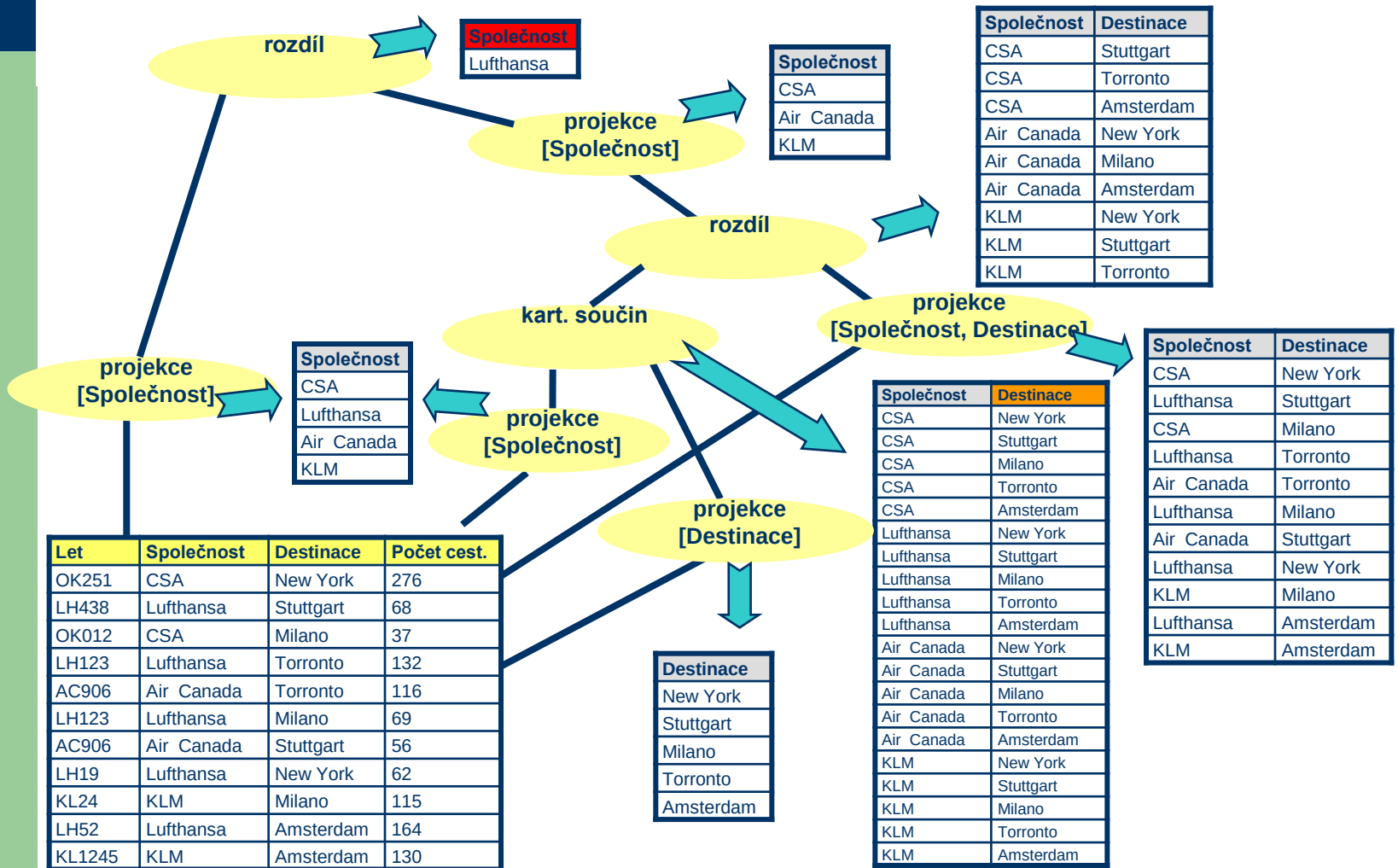


Příklad – dělení relací „bez dělení“

Které společnosti létají do všech destinací? (**R* ÷ S*** = R*[A–B] – ((R*[A–B] × S*) – R*)[A–B])

Lety[Společnost, Destinace] ÷ Lety[Destinace]

Lety[Společnost] – ((Lety[Společnost] × Lety[Destinace]) – Lety[Společnost, Destinace])[Společnost]



Společnost
Lufthansa

Společnost
CSA
Air Canada
KLM

Společnost	Destinace
CSA	Stuttgart
CSA	Torronto
CSA	Amsterdam
Air Canada	New York
Air Canada	Milano
Air Canada	Amsterdam
KLM	New York
KLM	Stuttgart
KLM	Torronto

Společnost
CSA
Lufthansa
Air Canada
KLM

Let	Společnost	Destinace	Počet cest.
OK251	CSA	New York	276
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
LH123	Lufthansa	Torronto	132
AC906	Air Canada	Torronto	116
LH123	Lufthansa	Milano	69
AC906	Air Canada	Stuttgart	56
LH19	Lufthansa	New York	62
KL24	KLM	Milano	115
LH52	Lufthansa	Amsterdam	164
KL1245	KLM	Amsterdam	130

Destinace
New York
Stuttgart
Milano
Torronto
Amsterdam

Společnost	Destinace
CSA	New York
CSA	Stuttgart
CSA	Milano
CSA	Torronto
CSA	Amsterdam
Lufthansa	New York
Lufthansa	Stuttgart
Lufthansa	Milano
Lufthansa	Torronto
Lufthansa	Amsterdam
Air Canada	New York
Air Canada	Stuttgart
Air Canada	Milano
Air Canada	Torronto
Air Canada	Amsterdam
KLM	New York
KLM	Stuttgart
KLM	Milano
KLM	Torronto
KLM	Amsterdam

Společnost	Destinace
CSA	New York
Lufthansa	Stuttgart
CSA	Milano
Lufthansa	Torronto
Air Canada	Torronto
Lufthansa	Milano
Air Canada	Stuttgart
Lufthansa	New York
KLM	Milano
Lufthansa	Amsterdam
KLM	Amsterdam

Relační úplnost

- ne všechny uvedené operace jsou nezbytně nutné pro vyjádření všech dotazů
  - minimální množina je tvořena operacemi
    B = {sjednocení, kartézský součin, rozdíl, selekce, projekce, přejmenování}
- dotazovací jazyk relační algebra je množina výrazů, které vzniknou vnořením operací B nad relacemi danými schématem databáze
- jestliže dva výrazy označují stejný dotaz, jsou oba výrazy ekvivalentní
- dotazovací jazyk, kterým lze vyjádřit všechny konstrukce relační algebry (tj. všechny dotazy, které lze popsat relační algebrou) se nazývá **relačně úplný**

Relační algebra – vlastnosti

- deklarativní dotazovací jazyk
  - tj. neprocedurální, nicméně struktura výrazu navádí na pořadí a způsob vyhodnocení
- výsledkem je vždy konečná relace
  - „bezpečně“ definované operace
- vlastnosti operací
  - asociativita, komutativita - kart. součin, spojení

Databázové systémy

Tomáš Skopal

- relační model
  * relační kalkuly

Osnova přednášky

- relační kalkuly
  - doménový
  - n-ticový

Relační kalkuly

- využití aparátu predikátové logiky 1. řádu pro dotazování
- rozšíření o „databázové“ predikáty, jejichž dvojí forma definuje
  - doménový kalkul (DRK) – pracuje s daty na úrovni atributů
  - n-ticový kalkul (NRK) – pracuje s daty na úrovni n-tic (prvků relace/řádků)
- výsledkem dotazu v DRK / NRK je opět relace (a její schéma)

Relační kalkuly

- prvky jazyka
  - termy – proměnné a konstanty
  - predikátové symboly
    - standardní binární predikáty {<, >, =, ≥, ≤, ≠}
    - „databázové“ predikáty (rozšiřující logiku 1. řádu)
  - formule
    - atomické – $R(t_1, t_2, ...)$, kde R predikátový symbol a t_i je term
    - složené – výrazy, kterými lze kombinovat atomické/složené formule logickými spojkami $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$
  - kvantifikátory $\exists$ (existenční), $\forall$ (všeobecný)

Doménový kalkul

- proměnné zastupují atributy, resp. jejich hodnoty
- databázový predikát R(x, y, ...)
  - R je zároveň názvem tabulky, ke které se predikát vztahuje
  - schéma predikátu (očekávané vstupní parametry) je shodné se schématem relace, tj. za každý atribut je třeba dosadit (ohodnocenou) proměnnou nebo konstantu
  - predikát pro konkrétní hodnoty atributů x, y, ... (ať jsou to proměnné nebo konstanty) nabývá hodnoty *true*, pokud v relaci R existuje prvek (řádek tabulky) s těmito hodnotami

Doménový kalkul

- databázový predikát
  - proměnné a konstanty v predikátu mají přiřazen název atributu, přes který proběhne ohodnocování, např:
    KINO(*NÁZEV_KINA* **: x ,** *FILM* **: y)**
    Potom schéma relace stačí definovat (neuspořádanou) množinou {*NÁZEV_KINA*, *FILM*}.

  - pokud proměnné, resp. konstanty nemají přiřazen název atributu, předpokládá se, že jejich jednoznačné určení k atributům je popsáno pořadím, přičemž schéma relace je zadáno uspořádanou množinou atributů, např:
    KINO(x, y), kde <*NÁZEV_KINA, FILM*> je schéma relace KINO – v dalším budeme uvažovat tento úsporný zápis

Doménový kalkul

- výstupem dotazu v DRK jsou všechna ohodnocení proměnných (a konstant) ve tvaru uspořádané n-tice, pro které platí formule dotazu
  - **{(t_1, t_2, ...) | formule dotazu obsahující proměnné t_1, t_2, ... }**
    - t_i je buď konstanta, anebo **volná proměnná**, tj. tyto proměnné nejsou uvnitř formule kvantifikovány
    - schéma výsledné relace (odpovědi na dotaz) je definováno přímo názvy volných proměnných
  - např. dotaz **{(x, y) | KINO(x, y)}** vrátí relaci sestávající ze všech prvků relace KINO
  - dotaz **{(x) | KINO(x, 'Titanic')}** vrátí názvy všech kin, kde se hraje film Titanic

Doménový kalkul

- kvantifikátory umožňují svázat proměnnou s výskytem v nějaké formuli
  - formule ∃**x R(t_1, t_2, ..., x, ...)** je vyhodnocena jako pravdivá, pokud **existuje** doménové ohodnocení x takové, že n-tice (t_1, t_2, ..., x, ...) je prvkem R
  - formule ∀**x R(t_1, t_2, ..., x, ...)** je vyhodnocena jako pravdivá, pokud pro **všechna** doménová ohodnocení x jsou n-tice (t_1, t_2, ..., x, ...) prvky R
  - např. dotaz **{(film) | ∃nazev_kina KINO(nazev_kina, film) }** vrátí názvy všech filmů hraných **v alespoň** jednom kině

Doménový kalkul

- důležité je určit, z jaké domény probíhá ohodnocování proměnných při kvantifikaci
  1. doména může být nespecifikovaná (tj. ohodnocení není omezeno žádnou doménou) – ohodnocení typu **universum**
  2. doména je typ příslušného atributu – ohodnocení podle **domény**
  3. doména je množina hodnot daného atributu přítomných v relaci, ke které se ohodnocení vztahuje – ohodnocení podle **aktuální domény**

Doménový kalkul

- např. dotaz **{(film) | ∀nazev_kina KINO(nazev_kina, film) }** se podle způsobu ohodnocování proměnné **nazev_kina** (typu/domény string) může lišit
  - pokud ohodnocujeme podle **univerza**, dotaz nevrátí nic, protože v relaci KINO určitě nebudou prvky nabývající všech možných hodnot v atributu NAZEV_KINA (např. hodnoty ‘kůň’, 125, ‘bflmpsvz’ tam jistě nebudou)
  - pokud ohodnocujeme podle **domény**, dotaz také pravděpodobně nevrátí nic, protože v relaci KINO nebudou všechny hodnoty z domény string, např. ‘kůň’, ‘bflmpsvz’, ...
  - pokud ohodnocujeme podle **aktuální domény**, dotaz vrátí názvy všech filmů, které se hrají ve všech kinech (přítomných v relaci KINO)

Doménový kalkul

- pokud se implicitně uvažuje ohodnocování v kvantifikátorech podle **aktuální domény,** nazývá se takto omezený DRK **DRK s omezenou interpretací**

- protože schémata často obsahují mnoho atributů, zápis kvantifikace lze zjednodušit tak, že výraz $R(t_1,.., t_i, t_{i+2}, ...)$ – tj. kde chybí proměnná t_{i+1} – chápeme jako $\exists t_{i+1}\ R(..., t_{i+1}, ...)$
  - např. dotaz **{(x) | KINO(x)}** chápeme jako **{(x) | ∃y KINO(x, y)}**

Příklady – DRK

FILM(JMENO_FILMU, JMENO_HERCE) HEREC(JMENO_HERCE, ROK_NAROZENI)

*Ve kterých filmech hráli **všichni** herci?*

$\{(f) \mid FILM(f) \wedge \forall h\ (HEREC(h) \Rightarrow FILM(f, h))\}$

*Který herec je **nejmladší**?*

$\{(h,r) \mid HEREC(h,r) \wedge \forall h2\ \forall r2\ (HEREC(h2,r2) \wedge h \neq h2) \Rightarrow r2 > r\}$

nebo

$\{(h,r) \mid HEREC(h,r) \wedge \forall h2\ (HEREC(h2) \Rightarrow \neg \exists r2 (HEREC(h2,r2) \wedge h \neq h2 \wedge r2 > r))\}$

*Které dvojice herců se sešly **alespoň** v jednom filmu?*

$\{(h1, h2) \mid HEREC(h1) \wedge HEREC(h2) \wedge h1 \neq h2 \wedge$
$\exists f, fh1\ FILM(f, fh1) \wedge (\exists fh2\ FILM(f, fh2) \wedge h1 = fh1 \wedge h2 = fh2)\}$

Vyhodnocení dotazu v DRK

*Který herec je **nejmladší**?*

$\{(h,r) \mid HEREC(h,r) \wedge \forall h2(\, HEREC(h2) \Rightarrow \neg\exists r2\, (HEREC(h2,r2) \wedge h \neq h2 \wedge r2 > r))\ \}$

```
$result = ∅
for each (h,r) do
      if (HEREC(h,r) and
            (for each h2 do
                  if (not HEREC(h2) or not (for each r2 do
                        if  (HEREC(h2,r2) ∧ h ≠ h2 ∧ r2 > r) = true then return true
                  end for
                  return false)) = false then return false

            end for
            return true) ) = true then Add (h,r) into $result
end for
```

univerzální kvantifikátor = řetězec konjunkcí

existenční kvantifikátor = řetězec disjunkcí

N-ticový kalkul

- téměř vše stejné jako u DRK, pouze proměnné/konstanty jsou celé prvky relace (řádky), tj. predikát R(t) je ohodnocen pravdivě, pokud ohodnocení prvku t náleží do R
  - výsledné schéma je tvořeno konkatenací schémat volných proměnných (n-tic)

- pro možnost přístupu k hodnotám atributů v rámci termu se používá tečkové notace
  - např. dotaz **{t | KINO(t) ∧ t.FILM = 'Titanic'}** vrátí všechna kina, která hrají film Titanic

- navíc lze výsledek promítnout pouze na určité atributy
  - např. dotaz **{t[NAZEV_KINA] | KINO(t)}**

Příklady – NRK

FILM(JMENO_FILMU, JMENO_HERCE) HEREC(JMENO_HERCE, ROK_NAROZENI)

Dvojice stejně starých herců hrajících ve stejném filmu.

$$\{h1, h2 \mid HEREC(h1) \wedge HEREC(h2) \wedge h1.ROK\_NAROZENI = h2.ROK\_NAROZENI \wedge \exists f1, f2\ FILM(f1) \wedge FILM(f2) \wedge f1.JMENO\_FILMU = f2.JMENO\_FILMU \wedge f1.JMENO\_HERCE = h1.JMENO\_HERCE \wedge f2.JMENO\_HERCE = h2.JMENO\_HERCE\}$$

*Ve kterých filmech hráli **všichni** herci?*

$$\{film[JMENO\_FILMU] \mid \forall herec(HEREC(herec) \Rightarrow \exists f(FILM(f) \wedge f.JMENO\_HERCE = herec.JMENO\_HERCE \wedge f.JMENO\_FILMU = film.JMENO\_FILMU))\}$$

Bezpečné formule DRK

- u neomezené interpretace proměnných (resp. u doménově závislých formulí) mohou nastat problémy vedoucí k nekonečným odpovědím
  - negace: **{x | ¬R(x)}**
    - např. {j | ¬Zaměstnanec(Jméno: j)}
  - disjunkce: **{x,y | R(.., x, ...) ∨ S(.., y, ...)}**
    - např. {i, j | Zaměstnanec(Jméno: i) ∨ Student(Jméno: j)}
- naopak k prázdné odpovědi vedou (díky neomezené doméně) univerzální kvantifikace {x | ∀y R(x, ..., y)}, a obecně dotazy **{x | ∀y φ(x, ..., y)}**, kde φ neobsahuje disjunkce (resp. implikace)
- problém s nekonečnými kvantifikacemi – jak je vyhodnocovat v konečném čase?
- řešením je omezit množinu formulí DRK

Bezpečné formule DRK

- abychom předešli nekonečné kvantifikaci, je dobré zavést omezené kvantifikátory, které **omezují interpretaci/ohodnocení vázaných proměnných**
  - místo $\exists x\,(\varphi(x))$ budeme používat $\exists x\,(R(x) \wedge \varphi(x))$
  - místo $\forall x\,(\varphi(x))$ budeme používat $\forall x\,(R(x) \Rightarrow \varphi(x))$
    - vyhodnocení lze potom implementovat jako

```
for each x in R      // konečná enumerace
    místo
for each x           // nekonečná enumerace
```

- volné proměnné v $\varphi(x)$ lze rovněž omezit – konjunkcí
  - $R(x) \wedge \varphi(x)$

Bezpečné formule DRK

- pro bezpečné formule DRK platí:
  1. dotaz neobsahuje $\forall$ (není problém, $\forall x\, \varphi(x)$ lze nahradit $\neg\exists x\, (\neg\varphi(x))$)
  2. pro každou disjunkci $\varphi_1 \vee \varphi_2$ platí, že φ_1, φ_2 sdílí stejné volné proměnné
     (předpokladáme všechny implikace $\varphi_1 \Rightarrow \varphi_2$ převedené na disjunkce $\neg\varphi_1 \vee \varphi_2$, totéž pro ekvivalence)
  3. všechny volné proměnné v každé maximální konjunkci $\varphi_1 \wedge \varphi_2 \wedge \ldots \varphi_n$ **jsou omezené**,
     tj. pro každou volnou proměnnou **x** platí alespoň jedna z podmínek:
     1. existuje nějaká φ_j kde se proměnná vyskytuje, která není negací ani binárním porovnáním
        (tj. φ_j je nenegovaná složená formule anebo nenegovaný „databázový“ predikát)
     2. existuje $\varphi_i \equiv x = a$, kde a je konstanta
     3. existuje $\varphi_i \equiv x = v$, kde v je omezená
  4. negaci lze aplikovat pouze v konjunkcích bodu 3

Příklad – bezpečné výrazy

$\{x,y \mid x = y\}$	není bezpečná (x ani y není omezená)
$\{x,y \mid x = y \vee R(x,y)\}$	není bezpečná (disjunkce sice sdílí obě volné proměnné, ale první maximální konjunkce (x=y) opět obsahuje porovnávání neomezených proměnných)
$\{x,y \mid x = y \wedge R(x,y)\}$	je bezpečná
$\{x,y,z \mid R(x,y) \wedge \neg(P(x,y) \vee \neg Q(y,z))\}$	není bezpečná (*z* neomezeno v konjunkci + navíc disjunkce nesdílí tytéž proměnné)
$\{x,y,z \mid R(x,y) \wedge \neg P(x,y) \wedge Q(y,z)\}$	ekvivalentní úprava předchozí f. – už bezpečná

Relační kalkuly – vlastnosti

- ještě více deklarativní než relační algebra (tam jistou strukturu vyhodnocení dotazu naznačuje vnoření operací)
  - specifikuje se pouze to, „co má výsledek splňovat“
- DRK i NRK jsou relačně úplné
  - lze je navíc rozšířit tak, že zahrnují větší třídu dotazů
- kromě jiného formy zápisu dotazu lze relační algebru a kalkuly vnímat jako různě „jemný“ přístup k datům
  - relační algebra pracuje s relacemi (celými tabulkami)
  - NRK pracuje s prvky relace (celými řádky)
  - DRK pracuje s prvky atributů (prvky řádků)

Příklady – srovnání DRK, NRK, rel.alg.

FILM(JMENO_FILMU, JMENO_HERCE) HEREC(JMENO_HERCE, ROK_NAROZENI)

*Ve kterých filmech hráli **všichni** herci?*

RA:

FILM % HEREC[JMENO_HERCE]

DRK:

$\{(f) \mid FILM(f) \wedge \forall h\ (HEREC(h) \Rightarrow FILM(f, h))\}$

NRK:

$\{film[JMENO\_FILMU] \mid \forall herec(HEREC(herec) \Rightarrow$
$\exists f(FILM(f) \wedge f.JMENO\_HERCE = herec.JMENO\_HERCE \wedge$
$f.JMENO\_FILMU = film.JMENO\_FILMU))\}$

Příklady – srovnání DRK, NRK, rel.alg.

ZAMESTNANEC(jmeno, prijmeni, stav, pocet_deti, kvalifikace, delka_praxe, zdravotni_stav, trestni_rejstrik, plat) – klíčem jsou všechny atributy kromě platu

Dvojice zaměstnanců pobírajících podobný plat (rozdíl max. o 1000Kč)?

DRK:

$\{(z1, z2) \mid \exists p1, s1, pd1, k1, dp1, zs1, tr1, pl1, p2, s2, pd2, k2, dp2, zs2\ tr2, pl2$
$ZAMESTNANEC(z1, p1, s1, pd1, k1, dp1, zs1, tr1, pl1) \wedge ZAMESTNANEC(z2, p2, s2, pd2, k2, dp2, zs2\ tr2, pl2) \wedge |pl1-pl2| \leq 1000 \wedge$
$(z1 \neq z2 \vee s1 \neq s2 \vee pd1 \neq pd2 \vee k1 \neq k2 \vee dp1 \neq dp2 \vee zs1 \neq zs2 \vee tr1 \neq tr2)\}$

NRK:

$\{z1[jmeno], z2[jmeno] \mid ZAMESTNANEC(z1) \wedge ZAMESTNANEC(z2) \wedge$
$z1 \neq z2 \wedge |z1.plat - z2.plat| \leq 1000\}$

Rozšíření relačních kalkulů

- relační úplnost často nestačí – umožňuje pouze data existující v tabulkách
- chtěli bychom data i odvozovat, resp. v dotazech používat tato odvozená data
- např. dotazy typu
  „Kteří zaměstnanci mají plat o 10% vyšší než je průměrný plat?“
- řešení – zavedení agregačních funkcí do kalkulů
- více o agregačních funkcích v dalších přednáškách (SQL)

Databázové systémy

Tomáš Skopal

- SQL
  * úvod
  * dotazování – SELECT

Osnova přednášky

- úvod do SQL
- dotazování v SQL
  - příkaz SELECT
  - třídění
  - množinové operace

SQL

- structured query language
  - standardní jazyk pro přístup k relačním databázím
  - původně snaha o co nejpřirozenější formulace DB požadavků (proto je např. příkaz SELECT tak složitý – je to v podstatě „věta“)
- je zároveň jazykem pro
  - definici dat (DDL)
    - vytváření/modifikace schémat, resp. tabulek
  - manipulaci s daty (DML)
    - dotazování
    - vkládání, aktualizace, mazání dat
  - řízení transakce
  - moduly (programovací jazyk)
  - definici integritní omezení
  - atd.

SQL

- standardy ANSI/ISO SQL 86, 89, 92, 1999, 2003 (zpětně kompatibilní)
- komerční systémy implementují SQL podle různých norem (nyní nejčasteji SQL 99), bohužel ne striktně (zpravidla něco nestd. navíc a zároveň něco std. neimplementují)
  - specifická rozšíření pro procedurální, transakční a další funkcionalitu např. TRANSACT-SQL (Microsoft SQL Server), PL-SQL (Oracle)

Vývoj SQL standardů

- **SQL 86** – první „nástřel“, průnik implementací SQL firmy IBM
  - **SQL 89** – malá revize motivovaná komerční sférou, mnoho detailů ponecháno implementaci
- **SQL 92** – silnější jazyk, specifikace 6x delší než u SQL 86/89
  - modifikace schémat, tabulky s metadaty, vnější spojení, kaskádové mazání/aktualizace podle cizích klíčů, množinové operace, transakce, kurzory, výjimky
  - čtyři stádia – Entry, Transitional, Intermediate, Full
- **SQL 1999** – mnoho nových vlastností, např.
  - objektově-relační rozšíření
  - typy STRING, BOOLEAN, REF, ARRAY, typy pro full-text, obrázky, prostorová data,
  - triggery, role, programovací jazyk, regulární výrazy, rekurzivní dotazy, atd...
- **SQL 2003** – další rozšíření, např. XML management, autočísla, std. sekvence, nicméně zmizel např. typ BIT

Dotazy v SQL

- dotaz v SQL vs. kalkuly a algebra
  - příkaz SELECT sdílí prvky obou aparátů
    - rozšířený DRK (práce se sloupci, kvantifikátory, agregační funkce)
    - algebra (některé operace – projekce, selekce, spojení, kart.součin, množinové operace)
  - narozdíl od striktní formulace relačního modelu jsou povoleny duplikátní řádky a nulové hodnoty atributů
- validátor syntaxe pro SQL 92, 1999, 2003
  - umožnuje zkontrolovat dotaz (nebo jiný SQL příkaz) podle normy
  - **http://developer.mimer.com/validator/index.htm**

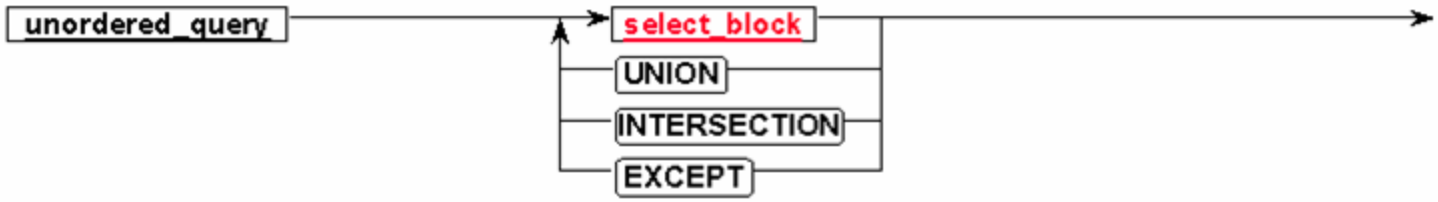
Dotazy v SQL

- pro jednoduchost vystačíme se syntaxí SQL 86 pomocí diagramů

  (zdroj: prof. H.J. Schek (ETH Zurich))
  - orientovaný graf (lze chápat jako DFA automat rozpoznávající SQL)
  - rozlišování termů v diagramu
    - malá písmena, podtržení – podvýraz v rámci dané konstrukce
    - velká písmena – klíčové slovo SQL
    - malá písmena, kurzíva – jméno (tabulky/sloupce/...)
  - pro odlišení budeme místo atribut/doména, relace používat označení sloupec, tabulka
  - diagramy neobsahují ANSI SQL 92 syntaxi pro spojení (klauzule CROSS JOIN, NATURAL JOIN, INNER JOIN, LEFT | RIGHT | FULL OUTER JOIN, UNION JOIN) – uvedeme později

Základní konstrukce dotazu

- netříděný dotaz sestává
  - vždy příkaz **SELECT** (hlavní logika dotazování)
  - případně z příkazů **UNION**, **INTERSECTION**, **EXCEPT** (sjednocení/průnik/rozdíl dvou nebo více výsledků získaných dotazem popsaným v příkazu **SELECT**)
  - výsledky nemají definované uspořádání (resp. jejich pořadí je určeno implementací vyhodnocení dotazu)



Tříděný dotaz

- výsledek netříděného dotazu lze setřídit
  - klauzule **ORDER BY**, třídění podle
    - sloupce (*column*)
  - třídit lze vzestupně (**ASC**) nebo sestupně (**DESC**) podle definovaného uspořádání
  - lze definovat více sekundárních třídících kritérií, která se uplatní v případě nedefinovaného lokálního pořadí (shodné primární tříděné hodnoty)

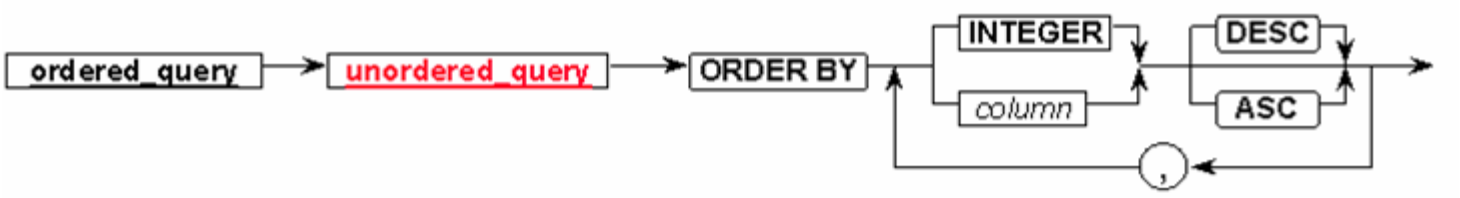


Schéma příkazu SELECT

Příkaz **SELECT** se skládá z 2 až 5 klauzulí (+ případně ještě z klauzule **ORDER BY**, ta není specifická pouze pro **SELECT**)

klauzule **SELECT** – projekce výstupního schématu, případně definice nových odvozených a agregačních sloupců

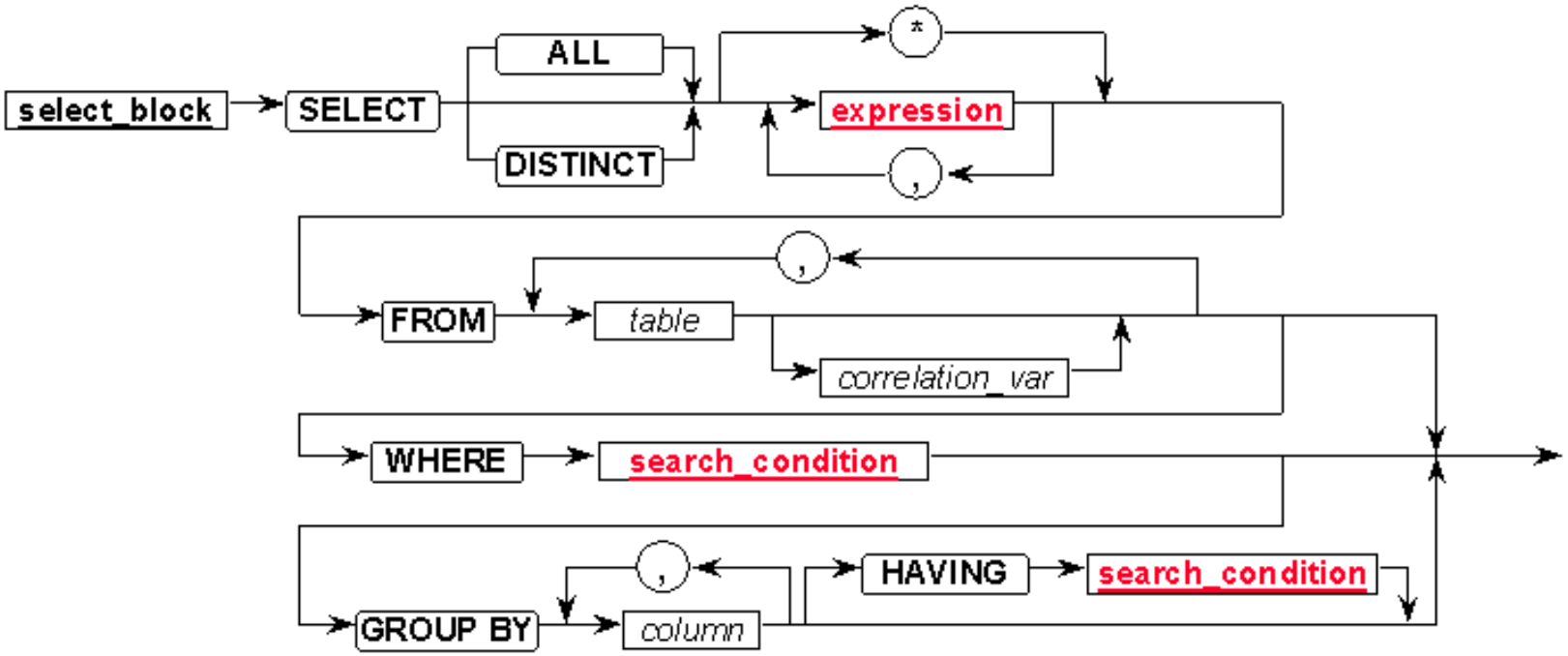
klauz. **FROM** – na které tabulky (v případě SQL ≥ 99 i vnořené dotazy, pohledy) se dotazujeme

klauz. **WHERE** – podmínka, kterou musí záznam (řádek) splňovat, aby se dostal do výsledku (logicky patří ke klauzuli **WHERE**)

kl. **GROUP BY** – přes které atributy se má výsledek popsaný pouze předchozími klauzulemi agregovat

klauz. **HAVING** – podmínka, kterou musí agregovaný záznam splňovat, aby se dostal do výsledku (patří ke klauzuli **GROUP BY**)

SELECT – schéma



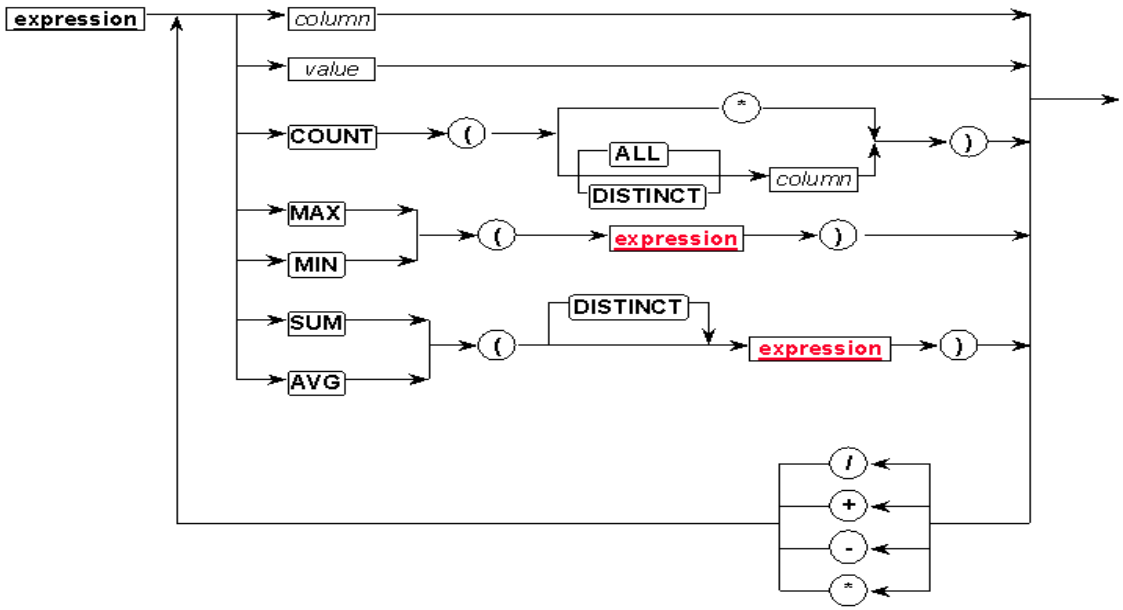
Logické pořadí vyhodnocení (resp. asociativita SELECT klauzulí):

FROM → **WHERE** → **GROUP BY** → **HAVING** → projekce **SELECT** (→ **ORDER BY**)

SELECT – expression

Kontext: SELECT ALL | DISTINCT **expression** FROM ...

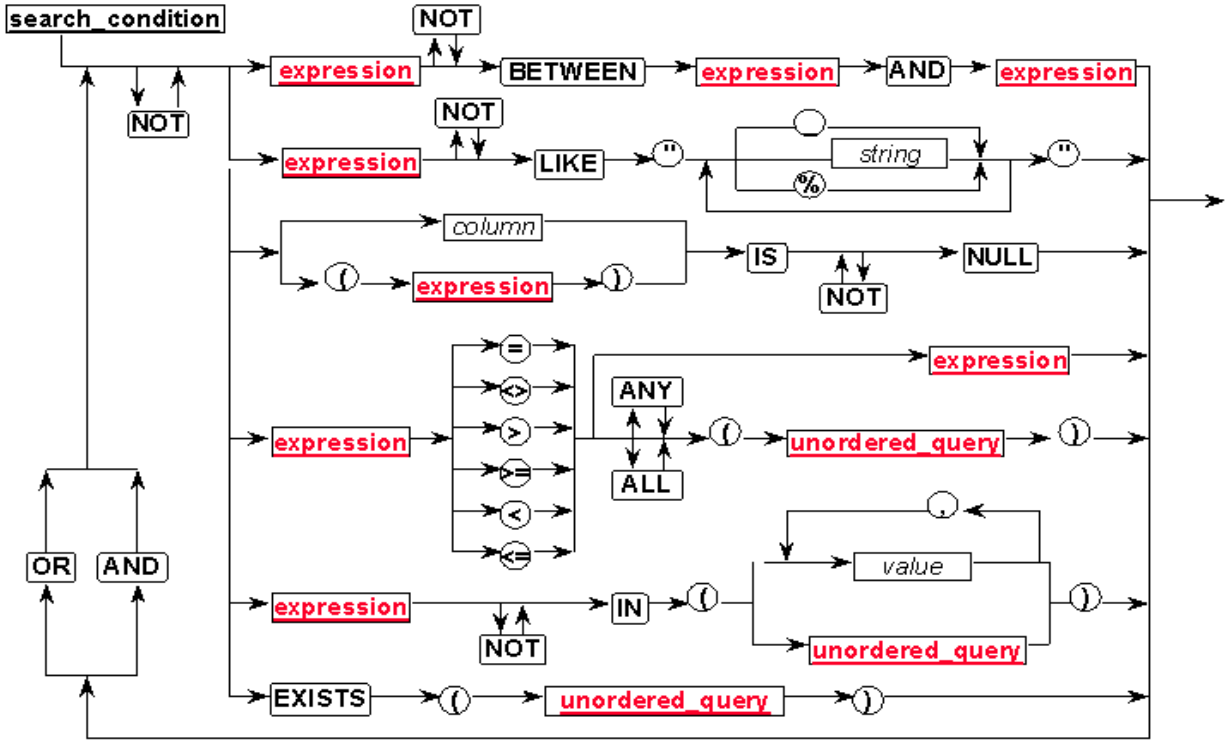
mnohokrát uvnitř **search_condition**



SELECT – search condition

Kontext: SELECT ... FROM ... WHERE **search_condition**

SELECT ... FROM ... WHERE ... GROUP BY ... HAVING **search_condition**



Tabulky užívané v příkladech

tabulka
Lety

Let	Společnost	Destinace	Počet cestujících
OK251	CSA	New York	276
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
OK321	CSA	London	156
AC906	Air Canada	Torronto	116
KL7621	KLM	Rotterdam	75
KL1245	KLM	Amsterdam	130

tabulka
Letadla

Letadlo	Společnost	Kapacita
Boeing 717	CSA	106
Airbus A380	KLM	555
Airbus A350	KLM	253

SELECT ... FROM ...

- nejjednodušší forma dotazu:
  SELECT [ALL] | DISTINCT expression **FROM** table1, table2, ...
- výraz může obsahovat
  - sloupce (hvězdička * je zástupce pro všechny neuvedené sloupce)
  - konstanty
  - agregace na výrazech
    - pokud se vyskytne alespoň jedna agregace, lze ve výrazu použít samostatně pouze agregované sloupce (ty uvedené v klauzuli **GROUP BY**), ostatní sloupce lze pouze „zabalit“ do agregačních funkcí
    - pokud není definována klauzule **GROUP BY**, seskupuje se do jediné skupiny (celý zdroj určený klauzulí FROM se agreguje do jediného řádku) – tj. odpovídá agregaci podle prázdné množiny
  - **DISTINCT** eliminuje duplikátní řádky ve výstupu, **ALL** (resp. bez specifikace) povoluje ve výstupu i duplikátní řádky (pozor, má vliv na agregační funkce – u **DISTINCT** vstupuje do agregačních funkcí méně hodnot)
- **FROM** obsahuje jednu nebo více tabulek, na kterých se dotaz provádí
  - pokud je specifikováno více tabulek, provede se kartézský součin

Příklady – SELECT ... FROM ...

Které společnosti přepravují cestující?

SELECT DISTINCT ‘Spol.:’, Společnost **FROM** Lety

‘Spol.’	Společnost
Spol.:	CSA
Spol.:	Lufthansa
Spol.:	Air Canada
Spol.:	KLM

Jaké páry letadel mohu vytvořit (bez ohledu na vlastníka) a jaká bude celková kapacita párů:

SELECT L1.Letadlo, L2.Letadlo,
L1.Kapacita + L2.Kapacita
FROM Letadla **AS L1**, Letadla **AS L2**

L1.Letadlo	L2.Letadlo	L1.Kapacita + L2.Kapacita
Boeing 717	Boeing 717	212
Airbus A380	Boeing 717	661
Airbus A350	Boeing 717	359
Boeing 717	Airbus A380	661
Airbus A380	Airbus A380	1110
Airbus A350	Airbus A380	803
Boeing 717	Airbus A350	359
Airbus A380	Airbus A350	803
Airbus A350	Airbus A350	506

Příklady – SELECT ... FROM ...

Kolik společností přepravuje cestující?

SELECT COUNT(**DISTINCT** Společnost) **FROM** Lety

COUNT(Společnost)
4

Kolika lety se přepravují cestující?

SELECT COUNT(Společnost) **FROM** Lety

SELECT COUNT(*) **FROM** Lety

COUNT(Společnost), resp COUNT(*)
7

Kolik je letadel, jakou mají maximální, minimální, průměrnou a celkovou kapacitu?

SELECT COUNT(*), **MAX**(Kapacita), **MIN**(Kapacita), **AVG**(Kapacita), **SUM**(Kapacita)
FROM Letadla

COUNT(*)	MAX(Kapacita)	MIN(Kapacita)	AVG(Kapacita)	SUM(Kapacita)
3	555	106	304,666	914

SELECT ... FROM ... WHERE ...

- logická podmínka selekce, tj. řádek tabulky (nebo kart. součinu, případně joinu), který podmínku splňuje, se dostane do výsledku
  - jednoduché podmínky lze kombinovat logickými spojkami **AND**, **OR**, **NOT**
- lze se ptát
  - srovnávacím predikátem (=, <>, <, >, <=, >=) na hodnoty dvou atributů
  - na interval expr1 **[NOT] BETWEEN** (expr2 **AND** expr3)
  - řetězcovým predikátem **[NOT] LIKE** "*maska*", kde maska je řetězec obsahující speciální znaky **%** (reprezentující libovolný podřetězec) a _ (reprezentující libovolný znak)
  - testem na nedefinovanou hodnotu, **(expr1) IS [NOT] NULL**
  - predikátem příslušnosti do množiny **expr1 [NOT] IN** (**unordered_query**)
  - jednoduchým existenčním kvantifikátorem **EXISTS** (**unordered_query**) testující prázdnost
  - rozšířenými kvantifikátory
    - existenčním **expr1** = | <> | < | > | <= | >= **ANY** (**unordered_query**)
      - tj. platí, že **alespoň jeden** prvek/řádek z **unordered_query** splňuje daný srovnávací predikát aplikovaný na **expr1**
    - všeobecným **expr1** = | <> | < | > | <= | >= **ALL** (**unordered_query**)
      - tj. platí, že **všechny** prvky/řádky z **unordered_query** splňují daný srovnávací predikát aplikovaný na **expr1**

Příklady – SELECT ... FROM ... WHERE ...

Kterými lety cestuje více než 100 cestujících?

```
SELECT Let, Počet_cestujících FROM Lety
    WHERE Počet_cestujících > 100
```

Let	Počet cestujících
OK251	276
OK321	156
AC906	116
KL1245	130

Kterými letadly by mohli letět cestující dané společnosti, pokud chceme mít naplněnost letadla alespoň 30%?

```
SELECT Let, Letadlo,
    (100 * Lety.Počet_cestujících / Letadla.Kapacita) AS Naplnenost  FROM Lety, Letadla
    WHERE          Lety.Společnost = Letadla.Společnost AND
                   Lety.Počet_cestujících <= Letadla.Kapacita AND
                   Naplnenost >= 30
```

Let	Letadlo	Naplnenost
OK012	Boeing 717	35
KL1245	Airbus A350	51
KL7621	Airbus A350	30

Příklady – SELECT ... FROM ... WHERE ...

Do kterých destinací se dá letět Airbusem nějaké společnosti (bez ohledu na naplněnost)?

Destinace
Rotterdam
Amsterdam

SELECT DISTINCT Destinace **FROM** Lety
WHERE Lety.Společnost **IN** (**SELECT** Společnost **FROM** Letadla
WHERE Letadlo **LIKE** “Airbus%”)

nebo např.

SELECT DISTINCT Destinace **FROM** Lety, Letadla
WHERE Lety.Společnost = Letadla.Společnost **AND** Letadlo **LIKE** “Airbus%”

Cestující kterých letů se vejdou do libovolného letadla (bez ohledu na vlastníka letadla)?

SELECT * **FROM** Lety **WHERE** Počet_cestujících <= **ALL** (**SELECT** Kapacita **FROM** Letadla)

Let	Společnost	Destinace	Počet cestujících
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
KL7621	KLM	Rotterdam	75

Spojení všeho druhu

Jak lze spojovat bez speciálních konstrukcí (SQL 86):

- (vnitřní) spojení na podmínku a přirozené spojení lze realizovat jako omezený kartézský součin, tj.
  SELECT ... **FROM** table1, table2 **WHERE** table1.A = table2.B
- levé/pravé polospojení se upřesní v klauzuli **SELECT**, tj. projekcí

Nové konstrukce SQL 92:

- **kartézský součin** – **SELECT** ... **FROM** table1 **CROSS JOIN** table2 ...
- **přirozené spojení** – **SELECT** ... **FROM** table1 **NATURAL JOIN** table2 **WHERE** ...
- **vnitřní spojení** –
  SELECT ... **FROM** table1 **INNER JOIN** table2 **ON search_condition WHERE** ...
- **sjednocení spojení** – vrací řádky první tabulky doplněné NULL hodnotami v atributech druhé tabulky + řádky druhé tabulky doplněné o NULL hodnoty v atributech první tabulky
  SELECT ... **FROM** table1 **UNION JOIN** table2 **WHERE** ...
- **levé, pravé, plné vnější spojení** –
  SELECT ... **FROM** table1 **LEFT** | **RIGHT** | **FULL OUTER JOIN** table2 **ON search_condition** ... **WHERE** ...

Příklady – spojení, ORDER BY

Vrať dvojice let-letadlo uspořádané vzestupně podle volného místa v letadle po obsazení cestujícími (uvažujeme pouze ta letadla, kam se cestující z letu vejdou)?

```
SELECT Let, Letadlo, (Kapacita – Počet_cestujících) AS Volnych_mist FROM Lety INNER JOIN Letadla
    ON (Lety.Společnost = Letadla.Společnost AND Počet_cestujících <= Kapacita)
    ORDER BY Volnych_mist
```

Let	Letadlo	Volnych_mist
OK012	Boeing 717	69
KL1245	Airbus A350	123
KL7621	Airbus A350	178
KL1245	Airbus A380	425
KL7621	Airbus A380	480

Které lety nemohou být uskutečněny (protože společnost nevlastní vhodné/žádné letadlo)?

```
SELECT Let, Destinace FROM Lety LEFT OUTER JOIN Letadla
    ON (Lety.Společnost = Letadla.Společnost AND
            Počet_cestujících <= Kapacita)
    WHERE Letadla.Společnost IS NULL
```

Let	Destinace
OK251	New York
LH438	Stuttgart
OK321	London
AC906	Torronto

SELECT /.../ GROUP BY ... HAVING ...

- agregační klauzule, díky které se řádky dosavadní „tabulky“ výsledku dotazu (tj. po fázi **FROM** a **WHERE**) poslučují podle totožných hodnot v definovaných sloupcích do skupin „superřádků“

- výstupem je potom tabulka „superřádků“, kde slučující sloupce mají definované hodnoty původních řádků, ze kterých vznikly (protože byly pro všechny řádky v superřádku stejné), kdežto v ostatních sloupcích by hodnota superřádku byla nejednoznačná (různé hodnoty v původních řádcích), takže pro tyto řádky jsou dvě možnosti
  - buď se ve výsledku dotazu vůbec nebudou nevyskytovat
  - anebo se jim jedinečná hodnota vyrobí nějakou agregací z hodnot původních

SELECT /.../ GROUP BY ... HAVING ...

- zobecnění použití agregačních funkcí uvedených dříve (**COUNT**, **MAX**, **MIN**, **AVG**, **SUM**), kde výsledkem není jednořádková tabulka (jako v případě nepoužití klauzule **GROUP BY**), ale tabulka s tolika řádky, kolik je superřádků vzniklých po fázi **GROUP BY**

- z této „superřádkové“ tabulky lze pomocí klauzule **HAVING** odfiltrovat nezajímavé řádky (zde tedy nezajímavé superřádky), podobně jako se pomocí **WHERE** filtrovaly výsledky pocházející z „FROM-fáze“
  - pozor, lze používat pouze agregované hodnoty sloupců

Příklady – SELECT /.../ GROUP BY ... HAVING ...

Jakou (kladnou) přepravní kapacitu mají jednotlivé společnosti?

```
SELECT Společnost, SUM(Kapacita) FROM Letadla GROUP BY Společnost
```

Společnost	SUM(Kapacita)
CSA	106
KLM	803

Kterým společnostem se vejdou najednou všichni cestující do letadel (bez ohledu na destinaci, tj. v jednom letadle mohou být cestující více letů)?

```
SELECT Společnost, SUM(Počet_cestujících) FROM Lety
    GROUP BY Společnost HAVING SUM(Počet_cestujících) <=
        (SELECT SUM(Kapacita) FROM Letadla WHERE Lety.Společnost = Letadla.Společnost )
```

Společnost	SUM(Počet_cestujících)
KLM	205

Vnořené dotazy

- standard SQL92 (full) rozšiřuje možnost použití vnořených dotazů také v klauzuli **FROM**
  - zatímco v SQL 86 bylo dovoleno jejich užití pouze v predikátech **ANY**, **ALL**, **IN**, **EXISTS**
- dva druhy použití
  - **SELECT** ... **FROM** (**unordered_query**) **AS** q1 **WHERE** ...
    - umožňuje výběr přímo z výsledku jiného dotazu (místo tabulky)
    - poddotaz je pojmenován q1 a s tímto identifikátorem se pracuje v dalších klauzulích jako s identifikátorem tabulky
  - **SELECT** ... **FROM** ((**unordered_query**) **AS** q1 **CROSS** | **NATURAL** | **INNER** | **OUTER** | **LEFT** | **RIGHT JOIN** (**unordered_query**) **AS** q2 **ON** (**expression**)
    - použití ve spojení všeho druhu

Příklad – vnořené dotazy

Pro společnosti vlastnící letadla vrať součet všech cestujících a kapacit letadel.

```
SELECT Lety.Společnost, SUM(Lety.Počet_cestujících), MIN(Q1. CelkováKapacita)
FROM Lety, (SELECT SUM(Kapacita) AS CelkováKapacita, Společnost  FROM Letadla GROUP BY Společnost) AS Q1
    WHERE Q1.Společnost = Lety.Společnost
    GROUP BY Lety.Společnost;
```

Společnost	SUM(Počet_cestujících)	CelkováKapacita
CSA	469	106
KLM	205	808

Jaké jsou trojice různých letů, v rámci nichž se počty cestujících liší max. o 50?

```
SELECT Lety1.Let, Lety1.Počet_cestujích, Lety2.Let, Lety2.Počet_cestujích,
        Lety3.Let, Lety3.Počet_cestujích
    FROM Lety AS Lety1
    INNER JOIN (Lety AS Lety2 INNER JOIN Lety AS Lety3 ON Lety2.Let < Lety3.Let)
        ON Lety1.Let < Lety2.Let
    WHERE           abs(Lety1.Počet_cestujících – Lety2.Počet_cestujících) <=50 AND
                    abs(Lety2.Počet_cestujících – Lety3.Počet_cestujících) <=50 AND
                    abs(Lety1.Počet_cestujících – Lety3.Počet_cestujících) <=50
    ORDER BY Lety1.Let, Lety2.Let, Lety3.Let;
```

Lety1.Let	Lety1.Počet...	Lety2.Let	Lety2.Počet...	Lety3.Let	Lety3.Počet...
AC906	116	KL1245	130	OK321	156
AC906	116	KL7621	75	LH438	68
KL7621	75	LH438	68	OK012	37

Dotazy s omezením na k řádků

- často chceme vrátit prvních **k** výsledků (podle nějakého uspořádání v ORDER BY)
- pomalá verze (nicméně SQL 92)
  - **SELECT** ... **FROM** tab1 **AS** a
    WHERE (**SELECT COUNT**(*)
    FROM tab1 **AS** b
    WHERE a.**<třídící atribut>** **<** b.**<třídící atribut>**) < **k**;

- rychlá verze (SQL:1999 non-core, implementuje Oracle, DB2)
  - **SELECT** ... **FROM** (
    SELECT ROW_NUMBER() **OVER** (**ORDER BY <třídící atribut(y)>**
    ASC | **DESC**) **AS** rownumber
    FROM tablename) **WHERE** rownumber <= **k**

- proprietární implementace (nestandardní)
  - **SELECT TOP k** ... **FROM** ... **ORDER BY <třídící atribut(y)> ASC** | **DESC**
    - MS SQL Server
  - **SELECT** ... **FROM** ... **ORDER BY <třídící atribut(y)> ASC** | **DESC LIMIT k**
    - MySQL, PostgreSQL

Příklad (MS SQL notace, analogicky ostatní)

Jak se jmenuje největší letadlo?

SELECT TOP 1 Letadlo **FROM** Letadla **ORDER BY** Kapacita **DESC**

Letadlo
Airbus A380

Kteří jsou první dva největší dopravci (podle součtu jejich cestujících)?

SELECT TOP 2 Společnost, **SUM**(Počet_cestujících) **AS** Počet **FROM** Lety
GROUP BY Společnost **ORDER BY** Počet **DESC**

Společnost	Počet
CSA	469
KLM	205

Databázové systémy

Tomáš Skopal

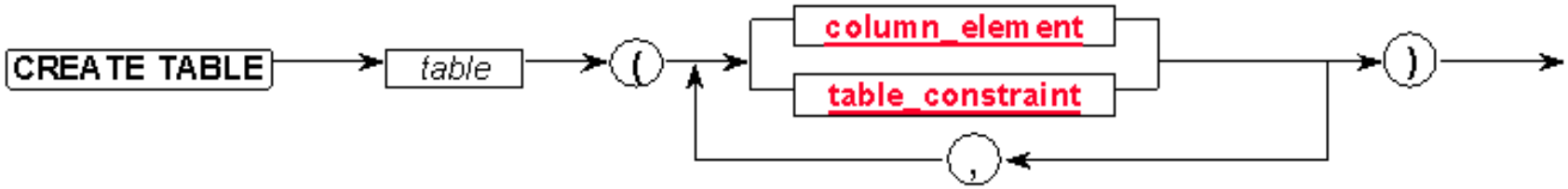
- SQL
  * definice dat
  * aktualizace
  * pohledy

Osnova přednášky

- definice dat
  - definice (schémat) tabulek a integritních omezení – CREATE TABLE
  - změna definice schématu – ALTER TABLE
- aktualizace
  - vkládání, modifikace, mazání (INSERT INTO, UPDATE, DELETE)
- pohledy

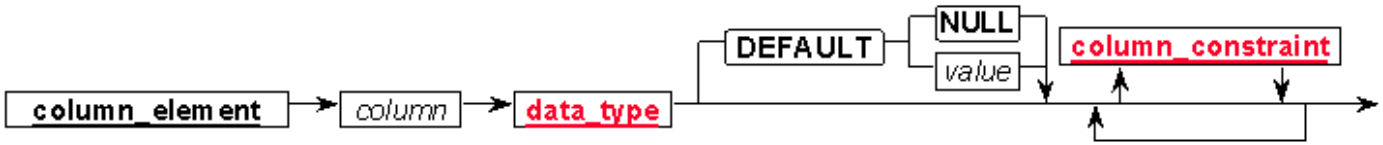
CREATE TABLE – základní konstrukce

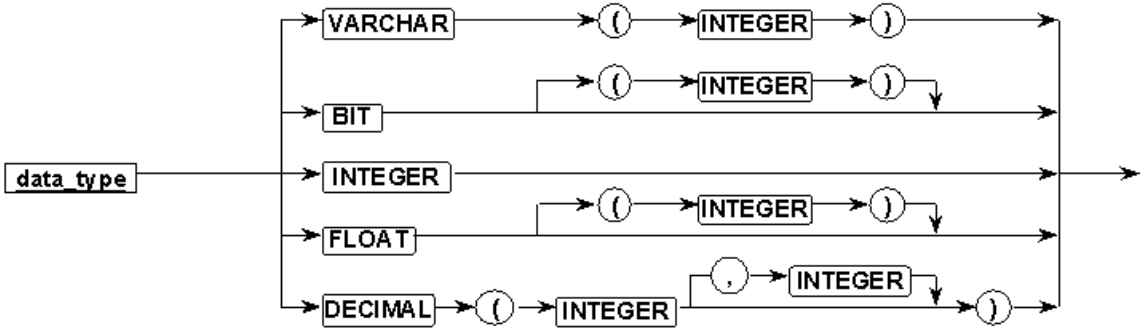
- vytvoření schématu a prázdné tabulky (pokud již není tabulka s daným jménem založena)
- tabulka má definovány jednak sloupce (atributy a související atributová IO) a jednak tabulková IO



CREATE TABLE – definice sloupce (atributu)

- každý sloupec má vždy přiřazen jednak datový typ data_type
- nepovinně je možno specifikovat
  - implicitní hodnotu v nově vytvořeném záznamu (**DEFAULT NULL** | ***value***)
  - sloupcové integritní omezení



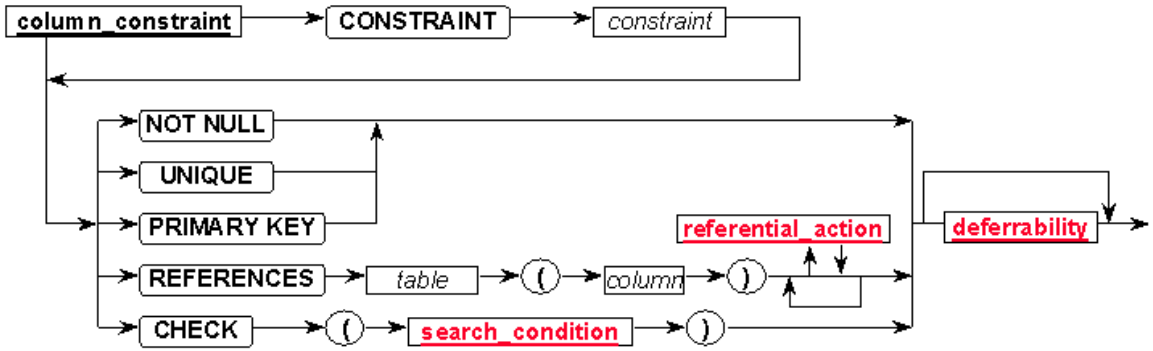


Příklad – jednoduchá tabulka (bez IO)

```
CREATE TABLE Výrobek
    (Id INTEGER, Název VARCHAR(128), Cena DECIMAL(6,2),
    DatumVýroby DATE, JeNaSkladě BIT DEFAULT TRUE, Hmotnost FLOAT)
```

CREATE TABLE – integritní omezení sloupce/atributu (lokální)

- sloupcové IO umožňuje omezit množinu platných hodnot daného atributu v rámci záznamu (nového nebo modifikovaného)
  - pojmenované **CREATE TABLE** ... (..., **CONSTRAINT** constraint ...)
  - nepojmenované
- 5 typů omezení na platnou hodnotu atributu
  - **NOT NULL** – hodnota musí být nenulová
  - **UNIQUE** – hodnota musí být unikátní (v rámci všech řádků v tabulce)
  - **PRIMARY KEY** – definuje primární klíč (sémanticky totéž jako **NOT NULL** + **UNIQUE**)
  - **REFERENCES** – definuje jednoatributový cizí klíč (oba atributy musí být kompatibilní)
  - **CHECK** – obecná podmínka, stejně jako u příkazu **SELECT** ... **WHERE**
    - s tím rozdílem, že se vyhodnotí pouze na vkládaném řádku (řádcích)
    - při vyhodnocení podmínky na TRUE je hodnota atributu platná
- při pokusu o aktualizaci řádků s neplatnou hodnotou atributu se aktualizace celého řádku neprovede



Příklad – tabulka s atributovými IO

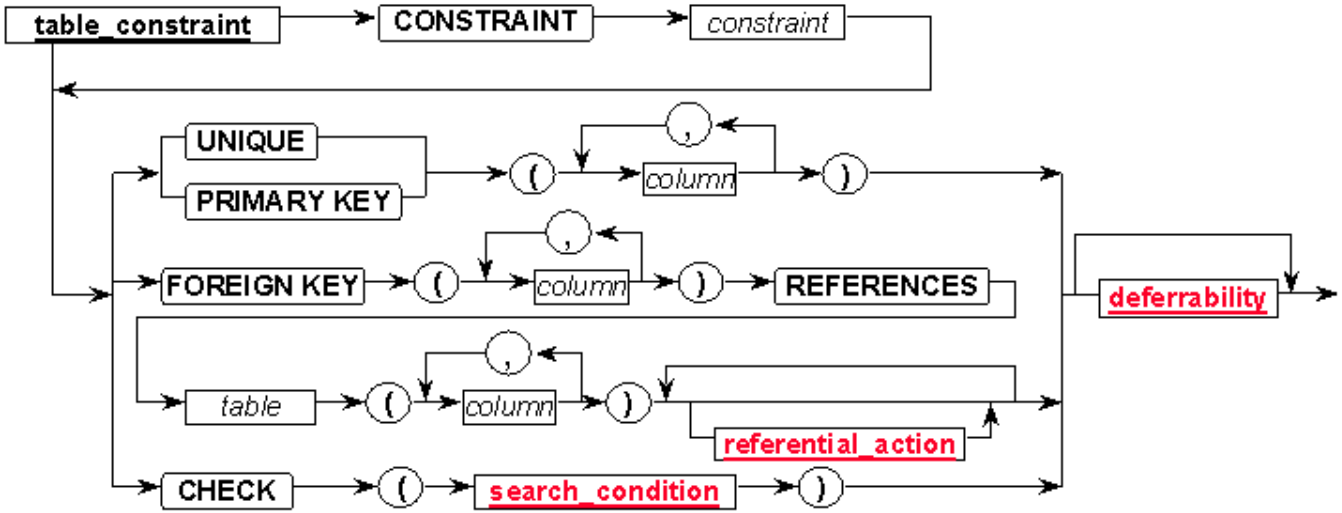
```
CREATE TABLE Výrobek
    (Id INTEGER CONSTRAINT pk PRIMARY KEY, Název
    VARCHAR(128) UNIQUE, Cena DECIMAL(6,2) NOT NULL,
    DatumVýroby DATE, JeNaSkladě BIT DEFAULT TRUE, Hmotnost
    FLOAT,
    Výrobce INTEGER REFERENCES Výrobce (Id))

CREATE TABLE Výrobce
    (Id INTEGER PRIMARY KEY, JménoVýrobce VARCHAR(128),
    Sídlo VARCHAR(256))
```

Příklad – cizí klíč na jedné tabulce

```
CREATE TABLE Zaměstnanec
    (IdZam INTEGER PRIMARY KEY, Jméno VARCHAR(128),
    Šéf INTEGER REFERENCES Zaměstnanec (IdZam))
```

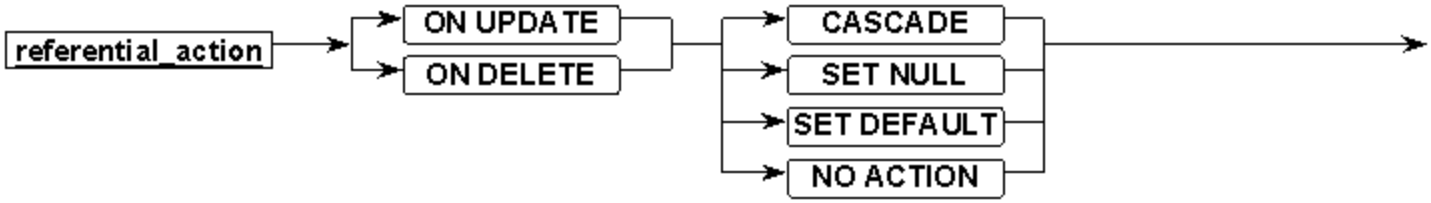
CREATE TABLE – integritní omezení celé tabulky (globální)

- zobecnění atributových IO pro kombinace hodnot více sloupců
  - kromě NOT NULL, to má smysl pouze u jednotlivých atributů
- **UNIQUE** – n-tice hodnot je unikátní
- **FOREIGN KEY** – stejně jako **REFERENCES** u atributu
- **CHECK**



Referenční integrita

- při aktualizaci tabulky **referující** (ta, pro kterou je IO definováno) nebo **referované** může nastat porušení integrity cizích klíčů
  - pokus o vložení/aktualizaci záznamu s hodnotou cizího klíče, která se nevyskytuje v sloupci referované tabulky
  - pokus o smazání záznamu z referované tabulky, když pro mazanou hodnotu klíče existuje reference
- při porušení integrity cizích klíčů mohou nastat dva případy chování
  - hlášení chyby aktualizace, pokud není definována referenční akce (SQL 89)
  - vykonání referenční akce **referential_action** (SQL 92)
    - **ON UPDATE**, **ON DELETE** – podmínka spuštění akce
      - při modifikaci referované hodnoty nebo smazání řádku v referované tabulce
    - **CASCADE** – dotčený záznam s referující hodnotou se také smaže, resp. aktualizuje novou hodnotou
    - **SET NULL** – referující hodnotou dotčeného záznamu se nastaví na NULL
    - **SET DEFAULT** – referující hodnotou dotčeného záznamu se nastaví implicitní hodnotu definovanou v CREATE TABLE
    - **NO ACTION** – implicitní, neprovede se nic, resp. SŘBD ohlásí chybu, tj. chování v rámci SQL 89



Příklad – tabulka s globálními IO

```
CREATE TABLE Výrobek
    (Id INTEGER PRIMARY KEY, Název VARCHAR(128) UNIQUE, Cena
    DECIMAL(6,2) NOT NULL, DatumVýroby DATE, JeNaSkladě BIT
    DEFAULT TRUE, Hmotnost FLOAT, Výrobce VARCHAR(128),
    SídloVýrobce VARCHAR(256),
    CONSTRAINT fk FOREIGN KEY (Výrobce, SídloVýrobce)
    REFERENCES Výrobce (JménoVýrobce, Sídlo))

CREATE TABLE Výrobce
    (JménoVýrobce VARCHAR(128), Sídlo VARCHAR(256),
    OborČinnosti VARCHAR(64),
    CONSTRAINT pk PRIMARY KEY(JménoVýrobce, Sídlo))
```

Příklad – CHECK

```
CREATE TABLE Výrobek
    (Id INTEGER PRIMARY KEY, Název VARCHAR(128) UNIQUE, Cena
    DECIMAL(6,2) NOT NULL, DatumVýroby DATE, JeNaSkladě BIT
    DEFAULT TRUE, Hmotnost FLOAT,
    CONSTRAINT chk CHECK
    ((Id = 0 OR Id > ALL (SELECT Id FROM Výrobek)) AND Hmotnost > 0))
```

Příklad – ON DELETE, ON UPDATE

```
CREATE TABLE Výrobek
    (Id INTEGER CONSTRAINT pk PRIMARY KEY, Název
    VARCHAR(128) UNIQUE, Cena DECIMAL(6,2) NOT NULL,
    DatumVýroby DATE, JeNaSkladě BIT DEFAULT TRUE, Hmotnost
    FLOAT,
    Výrobce INTEGER REFERENCES Výrobce (Id)
    ON DELETE CASCADE)

CREATE TABLE Výrobce
    (Id INTEGER PRIMARY KEY, JménoVýrobce VARCHAR(128),
    Sídlo VARCHAR(256))
```

ALTER TABLE

- změna definice schématu
  - atributy – přidání/odebrání atributu, změna DEFAULT hodnoty
  - IO – přidání/odebrání IO
- pozor, v tabulce už mohou být data, která nedovolí změnit IO (např. zavést IO primární klíč)

ALTER TABLE table-name

... **ADD** [**COLUMN**] column-name column-definition

... **ADD** constraint-definition

... **ALTER** [**COLUMN**] column-name SET

... **ALTER** [**COLUMN**] column-name DROP

... **DROP COLUMN** column-name

... **DROP CONSTRAINT** constraint-name

Příklad – ALTER TABLE

```
CREATE TABLE Výrobek
    (Id INTEGER PRIMARY KEY, Název VARCHAR(128) UNIQUE, Cena
    DECIMAL(6,2) NOT NULL, DatumVýroby DATE, JeNaSkladě BIT
    DEFAULT TRUE, Hmotnost FLOAT,
    CONSTRAINT chk CHECK
    ((Id = 0 OR Id > ALL (SELECT Id FROM Výrobek)) AND Hmotnost > 0))

ALTER TABLE Výrobek DROP CONSTRAINT chk

ALTER TABLE Výrobek ADD CONSTRAINT chk CHECK
    ((Id = 0 OR Id > ALL (SELECT Id FROM Výrobek)) AND Hmotnost > 10))
```

DROP TABLE

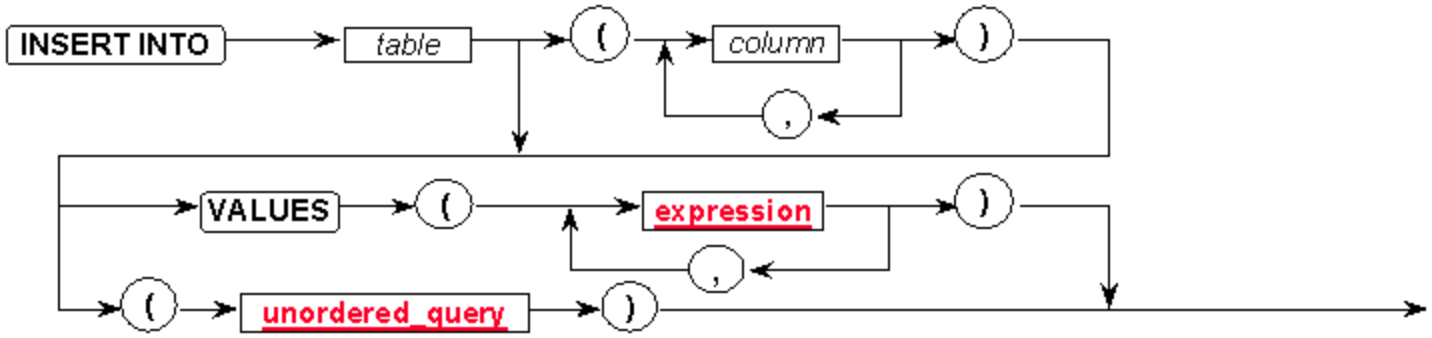
- **DROP TABLE *table***
- komplementární k příkazu **CREATE TABLE *table***
- smaže se jak obsah tabulky, tak i její definice, tj. schéma tabulky
  - pokud chceme vymazat pouze obsah tabulky (ne záhlaví – resp. schéma), použijeme příkaz **DELETE FROM *table*** (viz dále)

Modifikace dat

- SQL obsahuje kromě SELECT tři příkazy pro manipulaci s daty
  - INSERT INTO – vložení řádků
  - DELETE FROM – vymazání řádků
  - UPDATE – aktualizace hodnot v řádcích

INSERT INTO

- vkládání řádku výčtem hodnot, dvě možnosti
  - **INSERT INTO** table (col1, col3, col5) **VALUES** (val1, val3, val5)
  - **INSERT INTO** table **VALUES** (val1, val2, val3, val4, val5)
- vkládání více řádků, jejichž hodnoty vzniknou jako výsledek dotazu
  - **INSERT INTO** table1 | (výčet atributů) | (**SELECT** ... **FROM** ...)



Příklad – INSERT INTO

```
CREATE TABLE Výrobek
    (Id INTEGER CONSTRAINT pk PRIMARY KEY, Název VARCHAR(128)
    UNIQUE, Cena DECIMAL(6,2) NOT NULL, DatumVýroby DATE,
    JeNaSkladě BIT DEFAULT TRUE, Hmotnost FLOAT,
    Výrobce INTEGER REFERENCES Výrobce (Id))
```

```
INSERT INTO Výrobek VALUES (0, ‘Koště’, 86, ‘2005-5-6’, TRUE, 3, 123456)
```

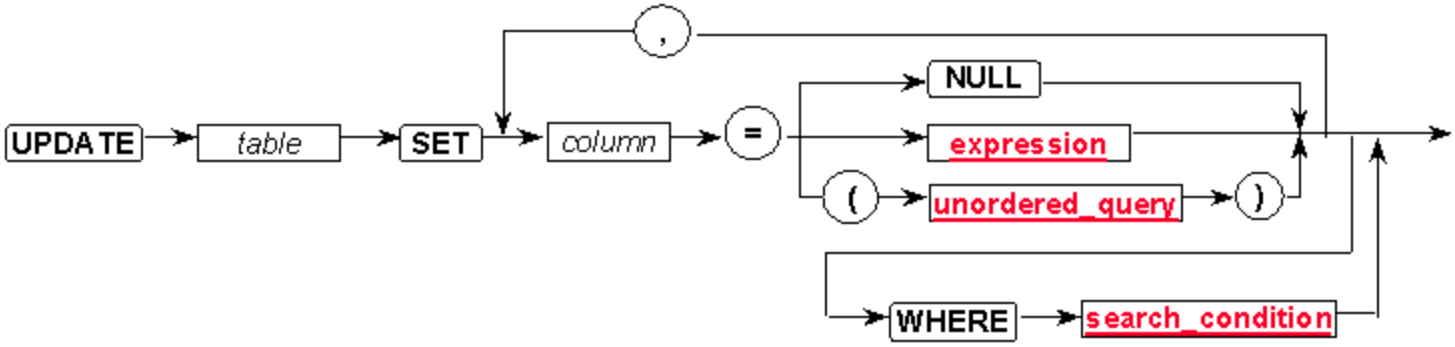
```
INSERT INTO Výrobek (Id, Název, Cena, DatumVýroby, Hmotnost, Výrobce)
    VALUES (0, ‘Koště’, 86, ‘2005-5-6’, 3, 123456)
```

Příklad – INSERT INTO

```
CREATE TABLE VýrobekNaSkladě
    (Id INTEGER PRIMARY KEY, Název VARCHAR(128) UNIQUE, Cena
    DECIMAL(6,2))

INSERT INTO VýrobekNaSkladě VALUES
    (SELECT Id, Název, Cena FROM Výrobek WHERE JeNaSkladě = TRUE)
```

UPDATE

- aktualizace záznamů splňujících podmínku
- hodnoty zvolených atributů vybraných záznamů jsou zvlášť nastaveny na
  - NULL
  - hodnotu expression (např. konstanta)
  - výsledek dotazu



Příklad – UPDATE

```
UPDATE Výrobek SET Název = 'Notebook' WHERE Název = 'Laptop'

UPDATE Výrobek SET Cena = Cena * 0.9 WHERE
    CAST(DatumVýroby AS VARCHAR(32)) <  '2003-01-01'

UPDATE Výrobek AS V1 SET Hmotnost = (SELECT AVG(Hmotnost)
    FROM Výrobek AS V2 WHERE V1.Název = V2.Název)
```

DELETE FROM

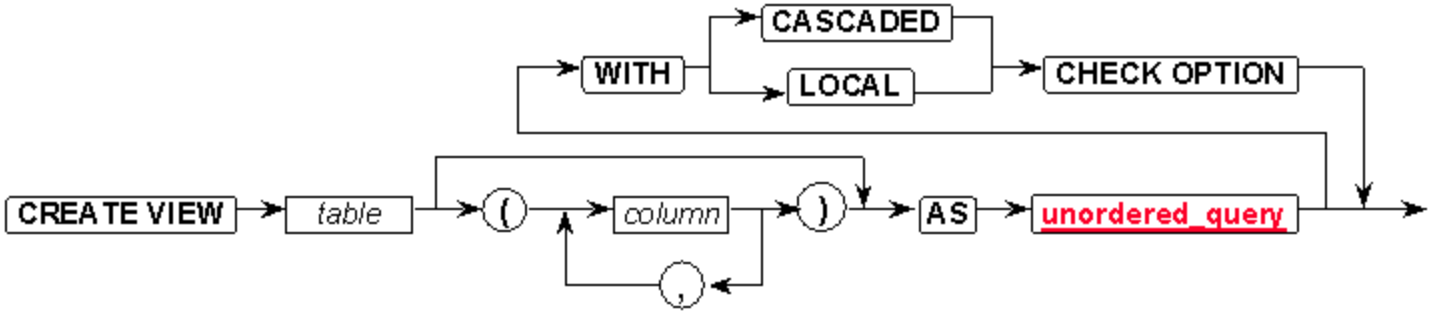
- vymaže záznamy, které splňují podmínku
- **DELETE FROM *table*** vymaže všechny záznamy

Příklad:

DELETE FROM Výrobek **WHERE** Cena > 100

Pohledy

- pojmenovaný dotaz, který lze dále použít jako tabulku
- generuje se dynamicky, podle dat vypočtených v okamžiku použití
  - lze do něj vkládat, mazat data
- CHECK OPTION zajistí, že po vložení/modifikaci záznamu do pohledu bude tato změna „vidět“
  - alespoň v tomto pohledu
  - ve všech závislých pohledech



Příklad – pohledy

```
CREATE VIEW NovéVýrobkyNaSkladě AS
    SELECT * FROM Výrobky WHERE JeNaSkladě = TRUE AND
    CAST(DatumVýroby AS VARCHAR(32)) > '2003-01-01'
    WITH LOCAL CHECK OPTION
```

```
INSERT INTO NovéVýrobkyNaSkladě VALUES
    (0, 'Hrábě', 135, '2004-05-06', TRUE, 4, 3215)
```

- vloží se (tranzitivně do tabulky Výrobky)

```
INSERT INTO NovéVýrobkyNaSkladě VALUES
    (0, 'Lopata', 135, '1999-11-07', TRUE, 4, 3215)
```

!! Chyba – takovýto záznam se nemůže vložit, protože by nebyl „vidět“ v pohledu (moc stará lopata)

Databázové systémy

Tomáš Skopal

– transakce

* vlastnosti transakcí
* rozvrhy

Osnova

- motivace – co je a proč je transakce
- vlastnosti transakcí
- rozvrhy („prokládané“ zpracování transakcí)
  - uspořádatelnost
  - konflikty
  - (ne)zotavitelný rozvrh

Motivace

- potřeba složitějších databázových operací, navíc v multitaskovém prostředí (běh více operací současně)
  - nelze (korektně) vyřešit základními „jednotkami“ přístupu/manipulací s daty, jako jsou SQL příkazy **SELECT**, **INSERT**, **UPDATE**, atd.
  - chceme vykonat sérii operací v přesně daném pořadí, používat mezivýsledky
- pojem transakce
  - lze chápat jako „databázový program“ (narozdíl od C++)
  - předpis sekvence tzv. **akcí**
    - operací s databází
    - dalších operací (aritmetické, pomocné, atd.)

Příklad – neformálně

Mějme bankovní databázi, v ní tabulku ***Účty*** a následující transakci pro převod sumy z účtu na účet:

transakce **PříkazKÚhradě**(***kolik***, ***zÚčtu***, ***naÚčet***)

{

1. SELECT Zůstatek **INTO** X **FROM** Účty **WHERE** čÚčtu = ***zÚčtu***

2. if (X < ***kolik***) ***ZrušTransakci***(“Malý zůstatek na účtu!!”);

3. UPDATE Účty **SET** Zůstatek = Zůstatek – ***kolik*** **WHERE** čÚčtu = ***zÚčtu***;

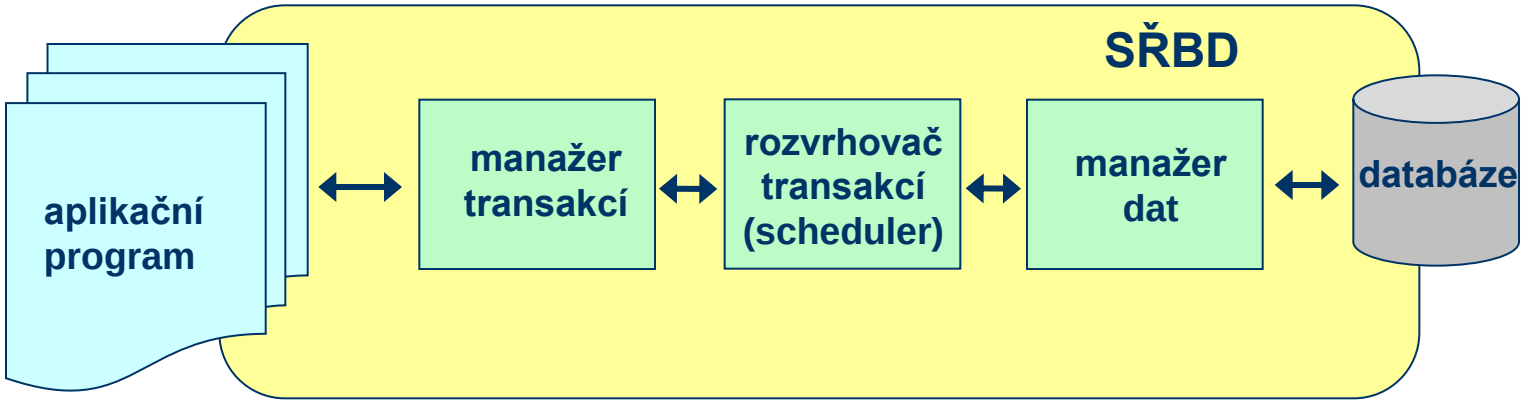
4. UPDATE Účty **SET** Zůstatek = Zůstatek + ***kolik*** **WHERE** čÚčtu = ***naÚčet***;

5. ***PotvrďTransakci***;

}

Architektura transakčního zpracování v SŘBD

- aplikační program vysílá požadavky na provedení transakcí
- manažer transakcí příjmá požadavky a vrací výsledky aplikacím
- rozvrhovač (dynamicky) vytváří/přizpůsobuje **rozvrh zpracování transakcí** podle požadavků manažeru transakcí
- manažer dat vykonává dílčí požadavky rozvrhovače na čtení/zápis dat z databáze (příslušející prováděným dílčím operacím transakcí)



Požadované vlastnosti transakčního zpracování

- tzv. ACID vlastnosti
  - **Atomicity** – transakce je provedena buď celá nebo vůbec
    - všechny akce transakce se provedou, nebo žádná z nich
      - přesněji databáze je v takovém stavu, že buď proběhla celá transakce nebo vůbec nezačala
    - např. nechceme, aby se z účtu odečetla částka a vinou „spadnutí systému“ se už nepřičetla na cílový účet
    - zajišťuje SŘBD (manažer transakcí)
  - **Consistency** – transakce musí zachovat databázi v konzistentním stavu
    - databáze je před provedením transakce i po jejím provedení v konzistentním stavu
    - za dodržení této vlastnosti je zodpovědný uživatel (programátor transakce), nazvěme ji **sémantická konzistence**
    - např. nechceme, aby transakce převodu peněz odečetla zůstatek do mínusové hodnoty (pokud existuje integritní omezení, které nedovoluje debetní zůstatek)
    - atomicita je vlastně také typ udržení konzistence databáze, nicméně o tu se stará SŘBD

Požadované vlastnosti transakčního zpracování

- další ACID vlastnosti
  - **Isolation** – více (souběžně běžících) transakcí o sobě „neví“, tj. SŘBD zařídí, že, pokud běží současně, se navzájem neovlivňují, tj. jsou nezávislé
    - není zaručeno pořadí provádění transakcí, tj. transakce se na sebe nemohou odkazovat
      - pokud je potřeba zřetězit více transakcí (tj. stanovit konkrétní pořadí vykonánání), musí se skombinovat do jediné transakce
    - např. pokud současně spustím transakce PříkazKÚhradě a SpočítejÚrok, pak první instance „nevidí“ změny provedené druhou instancí, dokud první instance úspěšně neskončí (a naopak) – vlastně ani „neví“, že existují i jiné transakce než ona
  - **Durability** – zaručuje, že operace provedené (úspěšně ukončenou) transakcí jsou trvalé, tj. uložené v databázi
    - zvlášť důležité, když transakce úspěšně skončí, ale z různých důvodů se data neobjevily v DB (např. spadne systém a data jsou v bufferu) – potom SŘBD musí zajistit korektní zapsání změn
- způsob transakčního zpracování
  - co nejvyšší propustnost dat (throughoutput) – taky transakce za sekundu
  - současné zpracování více transakcí

Požadované vlastnosti transakčního zpracování – shrnutí

- způsoby ukončení - transakce může být ukončena
  - úspěšně (**COMMIT**) – potvrdí se změny provedené akcemi transakce
  - neúspěšně – tři důvody
    - uživatelské přerušení (**ABORT**) – nastala situace, kdy sama transakce rozhodne o svém přerušení (viz příkaz k úhradě, když není dostatečný zůstatek)
    - systémové přerušení (**vynucený ABORT**) – SŘBD přeruší transakci z nějakého interního důvodu (např. došlo k porušení integritního omezení, které nebylo ošetřeno v transakci)
      - např. transakce se pokouší provést UPDATE neexistující tabulky
    - „spadne systém“ – HW porucha (disk, sběrnice), přerušení dodávky proudu, ...
- po neúspěšném ukončení musí SŘBD dostat databázi do stavu, v jakém byla před začátkem transakce (případně spustí transakce znovu, např. po restartu systému)
- dále popisované mechanismy transakčního zpracování „mají za úkol“
  - respektovat ACID vlastnosti
  - zároveň umožnit co nejvýkonnější a souběžné zpracování transakcí

Základní akce transakce

- omezíme se pouze na databázové operace
  - nechť A je nějaká databázová entita (tabulka, záznam, pole)
  - **READ**(A) – načte A z databáze
  - **WRITE**(A) – zapíše A do databáze
  - **COMMIT** – potvrzení vykonaných změn a ukončení transakce
  - **ABORT** – stornování změn a ukončení transakce
- prozatím uvažujeme **statickou databázi**
  (neexistuje vkládání a mazání objektům pouze čtení a aktualizace)
- transakce samozřejmě může obsahovat i další operace a řídící konstrukce, např. aritmetické operace, cykly, větvení, atd.
  - z hlediska dalšího výkladu nás nezajímají (až na výjimku – další slajdy)
- SQL příkazy **SELECT**, **INSERT**, **UPDATE**, atd. budeme uvažovat jako transakce implementované pomocí těchto základních operací (DB primitiv)
  - v SQL se místo termínu **ABORT** používá **ROLLBACK**

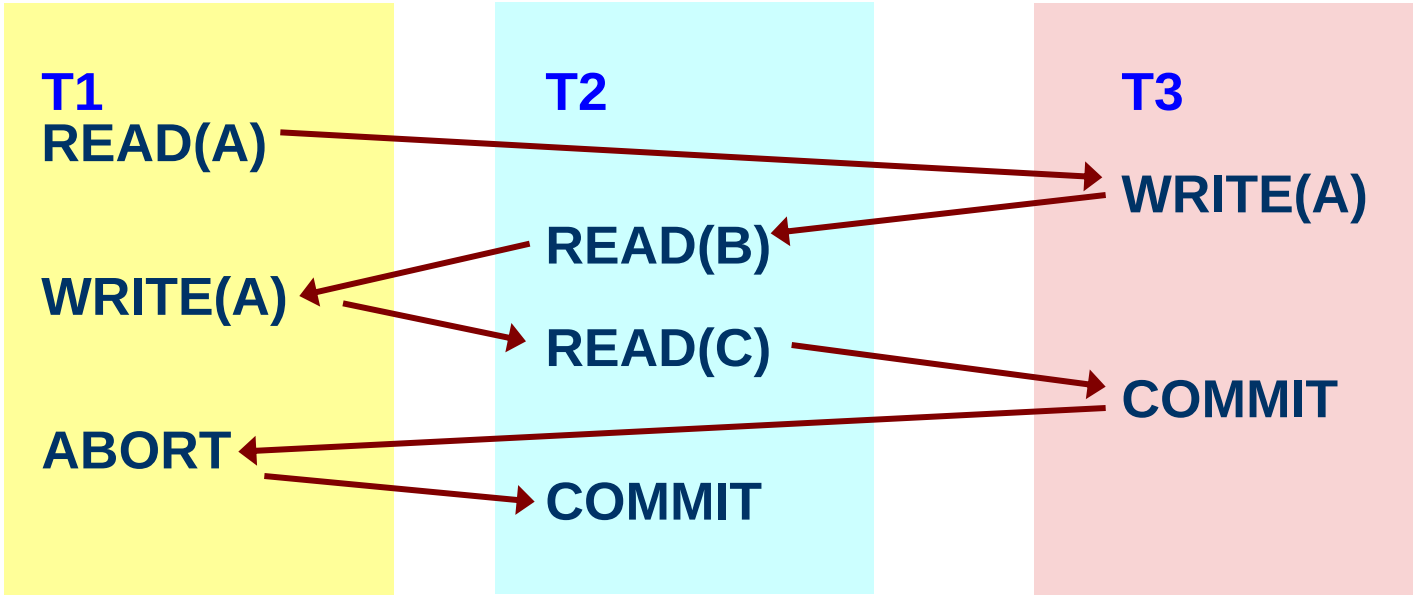
Transakce

- transakce je posloupnost akcí
  - T = <A_T^1, A_T^2, ... , **COMMIT** nebo **ABORT**>
- vykonání transakce může dostat databázi do (dočasně) nekonzistentního stavu, ale je zodpovědná za uvedení do konzistentního stavu před úspěšném ukončením (**COMMIT**)
- Příklad:
  Odečti z A (nějakého pole nějakého záznamu v nějaké tabulce) hodnotu 5 tak, aby A zůstala kladná.

```
T1 = <READ(A),                                          // akce 1
      if (A ≤ 5) then ABORT else WRITE(A – 5),          // akce 2
      COMMIT>                                           // akce 3
```

Transakční rozvrhy

- transakční rozvrh je setříděný seznam akcí několika transakcí tak, že akce různých transakcí jsou navzájem různě proloženy
  - uspořádání akcí každé transakce v rozvrhu je stejné (jako v samotné transakci)
- pro přehlednost budeme zapisovat akce dané transakce do sloupce



Transakční rozvrhy

Je třeba rozlišovat:

- **program transakce**
  - "design-time" (neběžící) kus kódu jedné transakce
  - tj. nelineární – větvení, cykly, skoky

- **transakční rozvrh**
  - „runtime" historie **již výkonaných** akcí několika transakcí
  - tj. lineární – sekvence volání primitiv, bez řídících částí (podmínky, cykly, apod.)

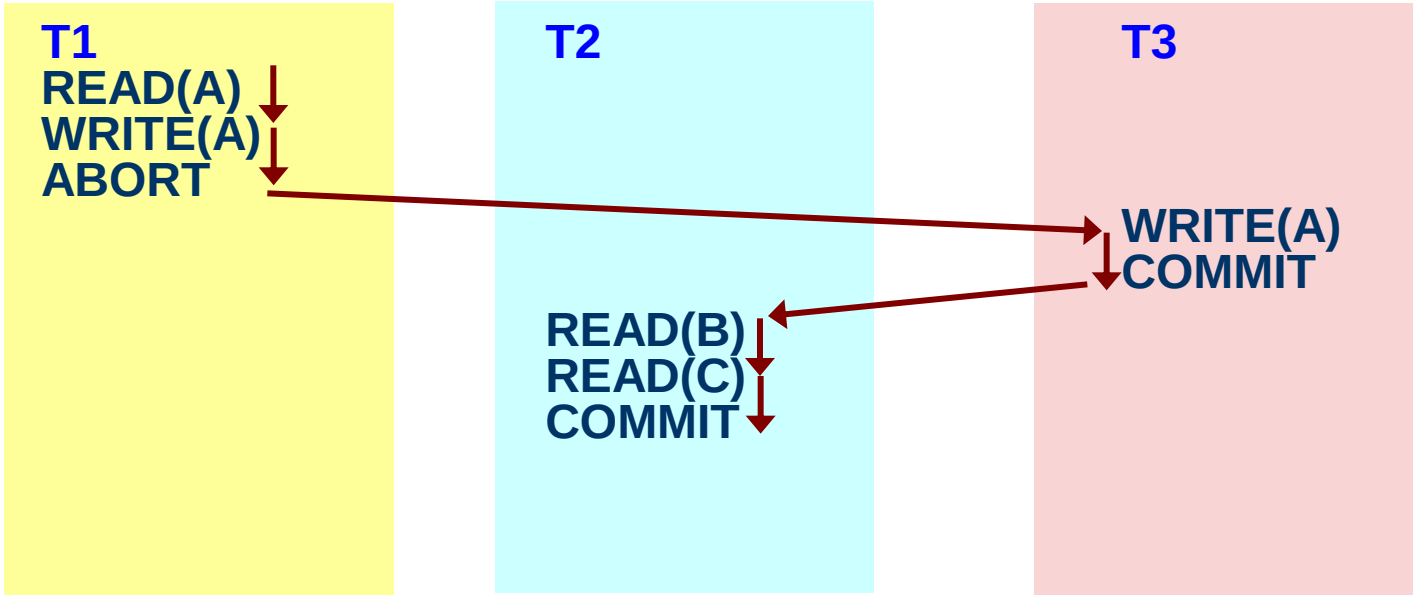
Rozvrhovač (scheduler)

- slouží k vytváření rozvrhů pro danou množinu transakcí, tj. rozvrhy nevytváří uživatel, ale SŘBD
- rozvrh je vytvářen dynamicky, tj. již existující rozvrh s transakcemi T1, ..., Tn může být proložen další transakcí (prokládá se od místa v rozvrhu, k jehož zpracování ještě nedošlo)
- stav databáze se mění zpracováváním rozvrhu – je jen jeden
  - hrozba – dočasně nekonzistentní DB „vidí“ i ostatní transakce
  - rozvrhovač musí zajistit „iluzi nezávislosti“ a předcházet konfliktům správnými typy rozvrhů (viz dále)
  - příklad:

T1	T2	
READ(A)		
A := A + 1		// A = 5
WRITE(A)	**READ(B)**	// B = 3
	READ(A)	// A = 6 !!!
COMMIT		
	COMMIT	

Sériové rozvrhy

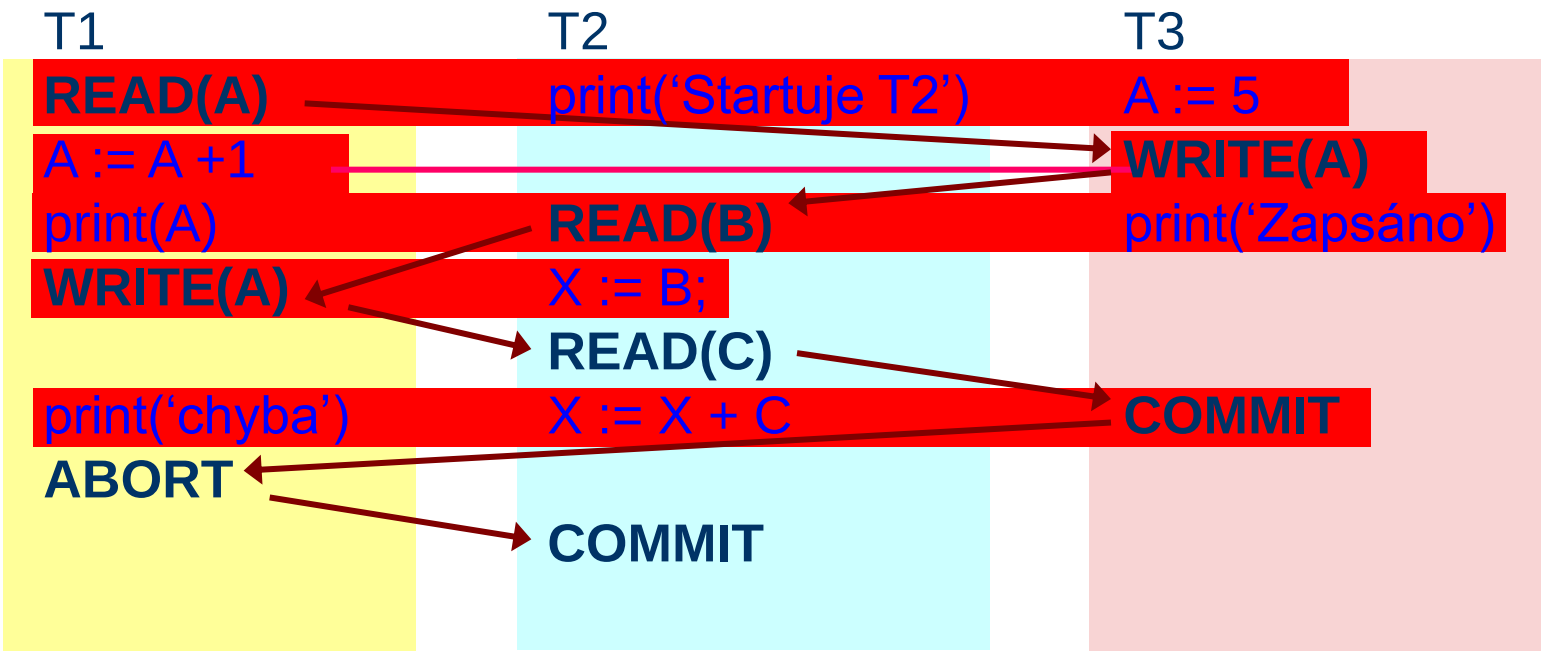
- speciální typ rozvrhu, kdy **všechny** akce **každé** transakce jsou uspořádány před (nebo po) **všech** akcích **každé** jiné transakce v rozvrhu
- počet možných sériových rozvrhů pro danou množinu transakcí S určuje počet permutací zúčastněných transakcí, tj. |S|!
- jednoduše:



Proč prokládat akce transakcí?

- každé proložení DB akcí určuje **sekvenční pořadí** jejich vyhodnocení, tj. nelze zároveň provádět dvě DB akce
- proč má tedy rozvrhovač vyrábět “prokládané” rozvrhy a ne sériové?
- dva hlavní důvody
  - paralelizace „nedatabázových“ akcí – zvýšení výkonu a propustnosti
    - zatímco jedna transakce čeká na načtení stránky z disku, jiná transakce počítá databázově nezávislou úlohu, např. aritmetickou operaci na již získaných datech
  - potřeba rychle vykonat „rychlé“ transakce tak, aby je nebrzdily „pomalé“ transakce
    - např. pokud už probíhá pomalá transakce **SpočítejVýplaty**, která trvá 1 hodinu, tak nechceme, aby náš dotaz **KolikMámeZaměstnanců** (který za normálních okolností trvá 1 sekundu) byl zařazen do fronty za pomalou transakci

Příklad – paralelismus DB a neDB akcí



V dalším výkladu už budeme přikládat význam pouze DB akcím, korektní sdílení jiných médií transakcemi (např. textové konzole, kam zapisuje funkce print) není v kompetenci SŘBD.

Uspořádatelnost

- rozvrh Sch(T1, T2, ... Tn) je **uspořádatelný** (přesněji řečeno serializovatelný, z angl. serializable), pokud jeho vykonání vede ke konzistentnímu stavu DB, tj. k témuž stavu databáze, který obdržíme **libovolným** sériovým rozvrhem na stejných transakcích
  - **pozor, nyní uvažujeme**
    - pouze potvrzené (committed) transakce
    - statickou databázi (neexistuje vkládání a mazání objektům pouze čtení a aktualizace)
  - „tentýž stav“ se vztahuje pouze k databázi, pochopitelně neDB akce typu print zde nejsou zohledněny a tudíž stav textové konzole se může lišit pro různé uspořádatelné rozvrhy
  - zeslabení požadavku na **libovolný** (a ne konkrétní) sériový rozvrh podstatu věci nenarušuje, protože množina transakcí k vykonání stejně nemá definováno pořadí
- silná vlastnost, která zaručuje **izolaci** (nezávislost) transakcí a **konzistenci** databáze (pokud samotné transakce jsou konzistentní)
- později zcela obecně (včetně akce abort a dynamické povahy DB) definujeme konzistenci zachovávající uspořádatelnost jako **pohledovou uspořádatelnost**

„Nebezpečí“ způsobená prokládáním

- abychom dosáhli uspořádatelnosti (tj. nezávislosti a konzistence), nelze jednotlivé akce transakcí v rozvrhu prokládat libovolně, existují 3 typy konfliktních situací, které se mohou v rozvrhu vyskytovat
  - konflikty pramení z pořadí dvojic akcí na stejném objektu dvou různých transakcí v rozvrhu
  - čtyři typy dvojic
    - read-read – jediná nekonfliktní dvojice
      - je jedno jestli nejprve přečte objekt A transakce T1 a pak T2 nebo naopak, navzájem je to nijak neovlivní
    - write-read – konflikt, čtení nepotvrzených dat
    - read-write – konflikt, neopakovatelné čtení
    - write-write – konflikt, přepsání nepotvrzených dat

Konflikty (WR)

- **čtení nepotvrzených dat** (write-read conflict)
  - transakce T2 přečte z DB hodnotu objektu A, který předtím zapsala transakce T1, ale ještě nepotvrdila, tj. čtou se potenciálně nekonzistentní data
  - tzv. **dirty read**

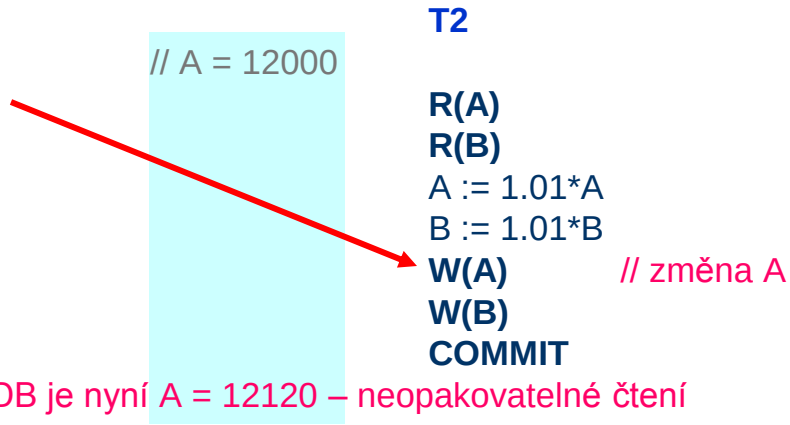
Příklad: T1 převádí 1000 Kč z účtu A na účet B (A = 12000, B = 10000)
T2 provádí roční úročení účtů (připíše 1% na každý účet)

T1	T2
R(A) // A = 12000	
A := A – 1000	
W(A) // DB v nekonzistentním stavu – na účtu B je pořád starý zůstatek	
	R(A) // zde se čtou nepotvrzená data
	R(B)
	A := 1.01*A
	B := 1.01*B
	W(A)
	W(B)
	COMMIT
R(B) // B = 10100	
B := B + 1000	
W(B)	
COMMIT	// nekonzistentní DB, A = 11110, B = 11100

Konflikty (RW)

- **neopakovatelné čtení** (read-write conflict)
  - transakce T2 zapíše objekt A, který předtím přečetla T1, která ještě neskončila
    - T1 už „nezopakuje čtení“ tj. přečte dvě různé hodnoty A

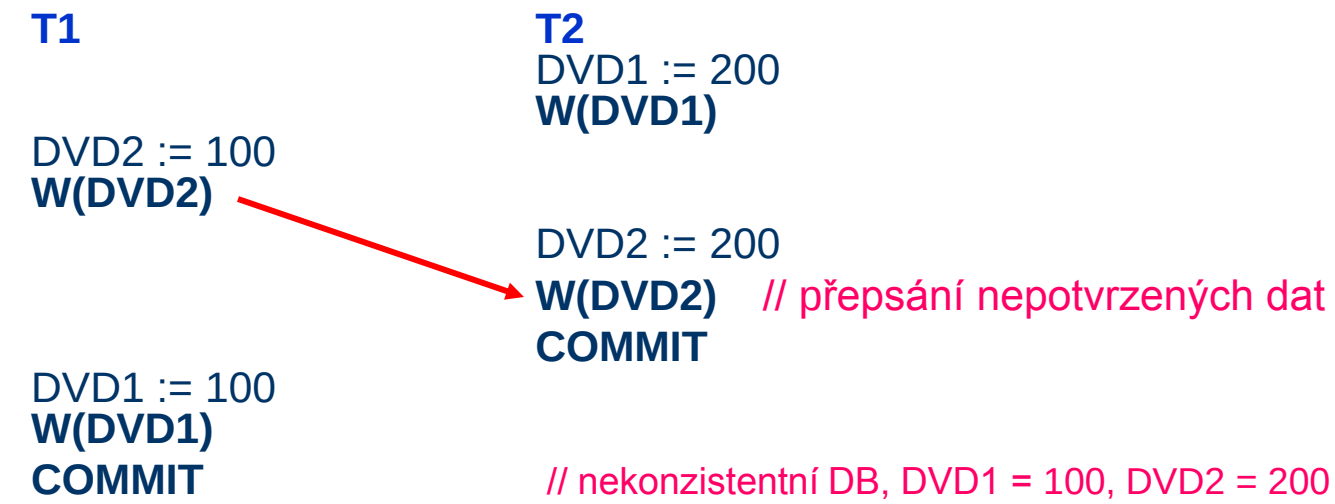
Příklad: T1 převádí 1000 Kč z účtu A na účet B (A = 12000, B = 10000)
T2 provádí roční úročení účtů (připíše 1% na každý účet)



Konflikty (WW)

- **přepsání nepotvrzených dat** (write-write conflict)
  - transakce T2 přepíše hodnotu A, která byla předtím přepsána transakcí T1 a ta stále běží
  - ztráta aktualizace (původní hodnota A zapsaná T1 je ztracená)
    - nekonzistence se projeví při tzv. **blind write** – zapsání hodnoty, aniž předtím byla přečtena

Příklad: Nastav totožnou cenu všem DVD. (mějme dvě instance této transakce, jedna nastavuje cenu 100 Kč, druhá 200Kč)



Konfliktová uspořádatelnost

- dva rozvrhy jsou **konfliktově ekvivalentní**, když všechny „konfliktní“ páry operací v jednom rozvrhu existují (stejně konfliktní) i ve druhém rozvrhu
- rozvrh je **konfliktově uspořádatelný**, pokud je konfliktově ekvivalentní nějakému sériovému rozvrhu (na stejné množině transakcí)
  - tj. nejsou v něm konflikty (protože v sériovém rozvrhu konflikty neexistují)

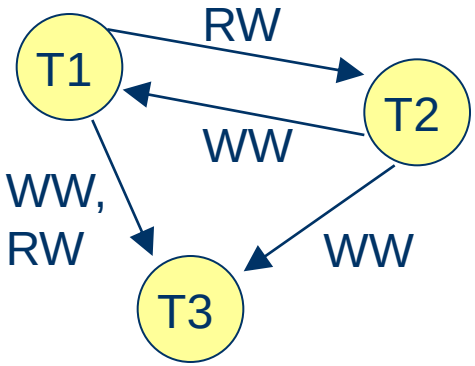
Příklad: rozvrh, který **je uspořádatelný** (sériový rozvrh <T1, T2, T3>), ale **není konfliktově uspořádatelný** (zápisy v T1 a T2 jsou v opačném pořadí)

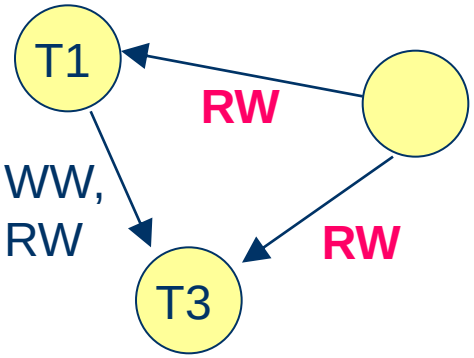
Detekce konfliktové uspořádatelnosti

- precedenční graf (také graf uspořádatelnosti) na rozvrhu
  - uzly Ti představují **potvrzené** transakce
  - orientované hrany představují konflikty RW, WR, WW mezi transakcemi
- rozvrh je konfliktově uspořádatelný,
  pokud je jeho precedenční graf **acyklický**

Příklad:

T1	T2	T3
R(A)		
	R(A)	
	COMMIT	
W(A)		
COMMIT		
		W(A)
		COMMIT





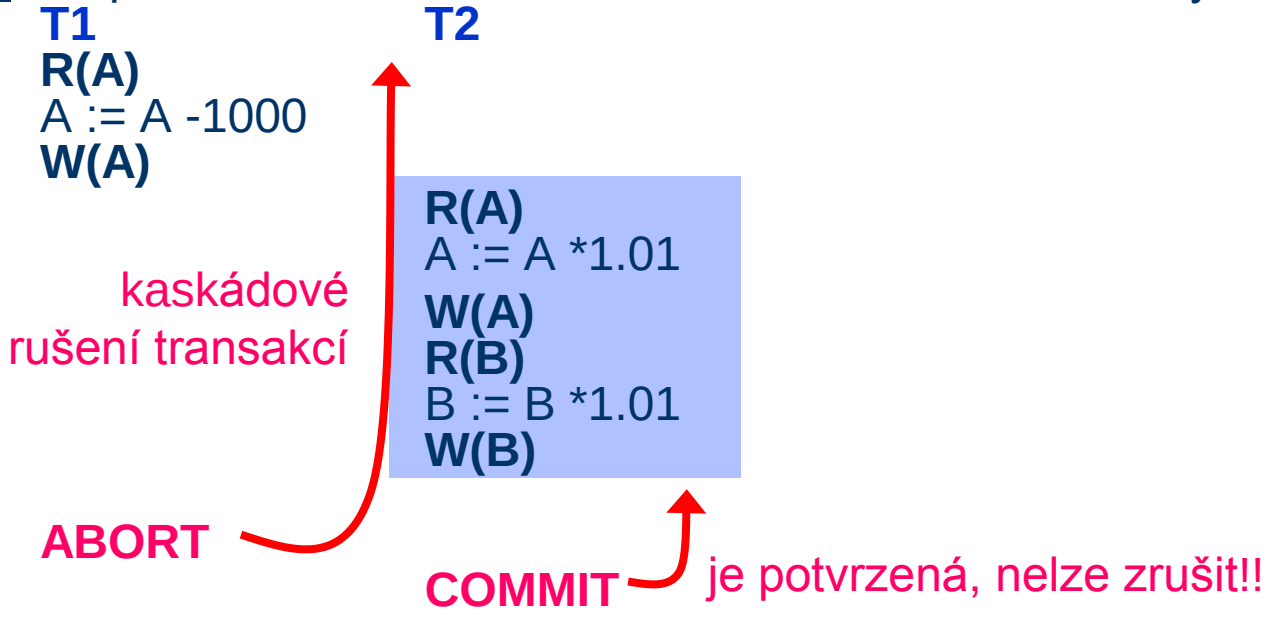
Konfliktová uspořádatelnost

- konfliktová uspořádatelnost zaručuje eliminaci rozvrhů s **WR**, **RW** a **WW** konflikty
- konfliktová uspořádatelnost (definovaná množinou párů konfliktních akcí) **restriktivnější** než uspořádatelnost (definovaná pomocí zachování konzistence DB)
- nicméně samotná konfliktová uspořádatelnost **nezohledňuje**
  - zrušení transakce – akci ABORT
    - rozvrh může být **nezotavitelný**
  - dynamickou povahu databáze (vkládání a mazání DB objektů)
    - může se vyskytnout tzv. **fantom** (příští přednáška)
  - z tohoto pohledu je konfliktová uspořádatelnost **nepostačující podmínka**, postačující je tzv. pohledová uspořádatelnost (viz dále)

Nezotavitelný rozvrh

- nyní navíc dovolíme ukončení transakce akcí ABORT, tj. zrušením/stornováním transakce
- z toho pramení další „nebezpečí“ – nezotavitelný rozvrh
  - pokud nastane ABORT v transakci, který má vliv na potvrzenou transakci

Příklad: T1 převádí 1000 Kč z účtu A na účet B, T2 úročí vklady



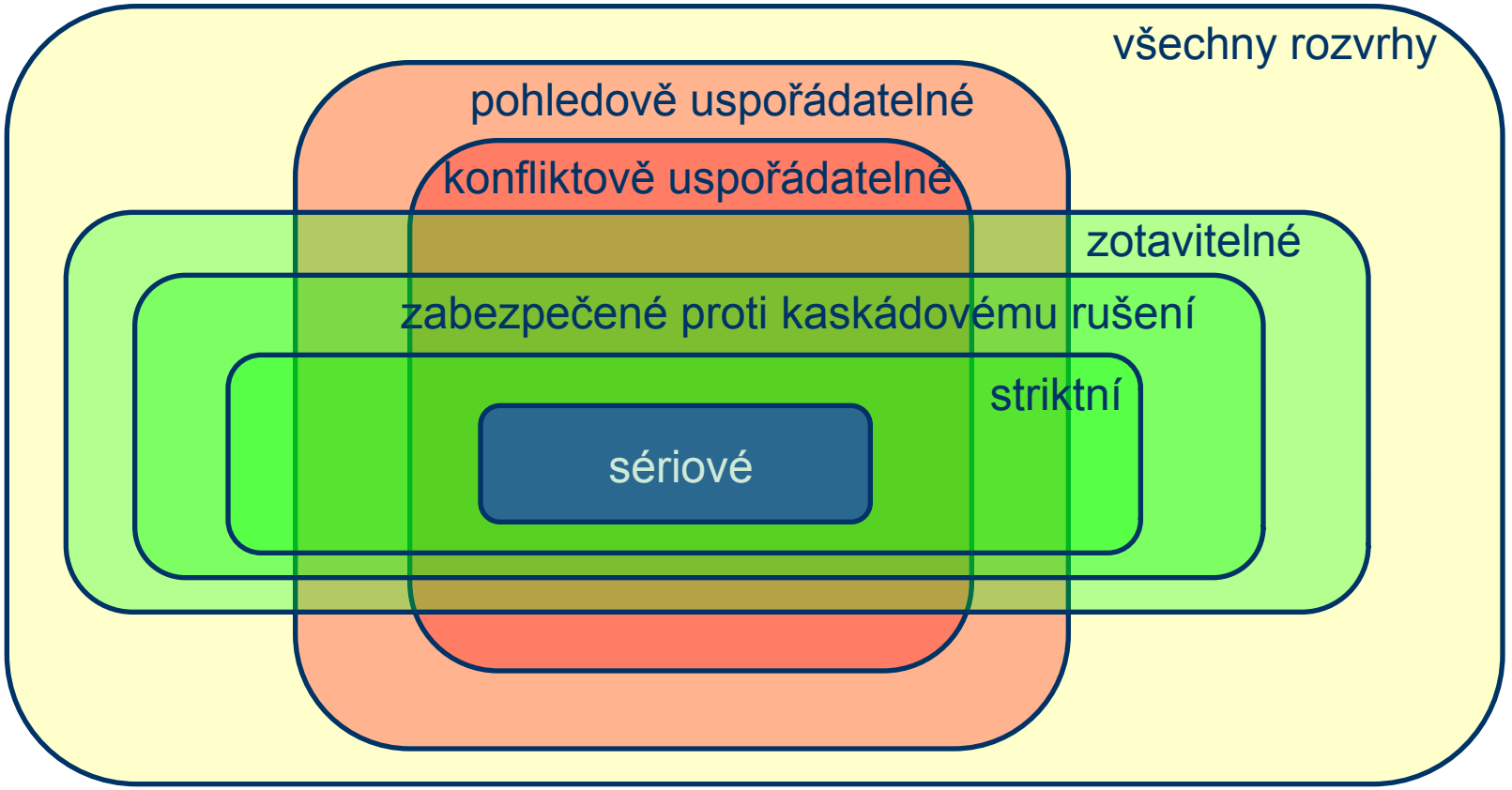
Zotavitelný rozvrh

- **v zotavitelném rozvrhu** je transakce T potvrzena až poté, co potvrdí všechny ostatní transakce, které změnily data později čtené (a zapsané) v T
  - tj. v předchozím příkladu by musel být poslední akcí COMMIT transakce T2
- pokud v zotavitelném rozvrhu navíc dochází ke čtení změn pouze potvrzených transakcí, nemůže dojít ani ke kaskádovému rušení transakcí (tzv. **transakce zabezpečené proti kaskádovému rušení**)
  - tj. v předchozím příkladu by T2 mohla začít číst R(A) až po ABORTu transakce T1

Pohledová uspořádatelnost

- rozvrhy S1 a S2 jsou **pohledově ekvivalentní**, když splňují tyto podmínky
  - pokud Ti čte počáteční hodnotu A v S1,
    musí číst počáteční hodnotu A i v S2
  - pokud Ti čte hodnotu A zapsanou Tj v S1,
    musí číst tuto hodnotu zapsanou Tj i v S2
  - transakce, která provede poslední zápis A v S1,
    musí provést tento poslední zápis i v S2
- rozvrh je **pohledově uspořádatelný**, pokud je pohledově ekvivalentní s nějakým sériovým rozvrhem
- testování uspořádatelnosti pro pohledovou ekvivalenci je **NP-úplný problém**, takže se v praxi nepoužívá
  - lze nahradit konfliktově uspořádatelnými a zotavitelnými rozvrhy

Přehled typů rozvrhů



Databázové systémy

Tomáš Skopal

– transakce

* uzamykací protokoly
* alternativní protokoly
* zotavení

Osnova

- uzamykací protokoly
  - 2PL
  - striktní 2PL
  - uváznutí, prevence
  - fantom
- alternativní protokoly
  - optimistické řízení
  - časová razítka
- zotavení po havárii systému

Protokoly současného vykonávání transakcí (concurrency control protocols)

- rozvrhovač transakcí volí nějaký protokol (soubor pravidel pro rozvrhování) tak, aby byly zachovány požadované vlastnosti zpracování, jako
  - vysoký stupeň paralelizace (konkurence)
  - uspořádatelnost (konzistence, izolace)
  - zotavitelnost
  - atd.
- uzamykací protokoly (pesimistické řízení)
  - využívá se zámků, aby se předešlo konfliktům, nezotavitelnosti, ...
- alternativní protokoly
  - optimistické řízení
  - časová razítka

Proč protokol?

Rozvrhovač „nic dopředu neví“:
- požadavky na spuštění transakcí přicházejí různě v čase
- jelikož jedna transakce obsahuje řídící (programové) konstrukce, neví se dopředu ani sekvence skutečně vykonaných operací v rámci jediné transakce
- tj. nelze připravit rozvrh „dopředu“ a pak ho jen vykonat

Rozvrhovač tvoří rozvrh dynamicky, tj. může se řídit pouze lokálními podmínkami → PROTOKOL

DB klient(i)	T1	T2	T3
	R(A)	**R(C)**	**R(A)**
	W(B)	**W(C)**	**R(B)**
	W(A)		**R(C)**

Rozvrh

Proč uzamykat?

- zamykání entit může rozvrhovači sloužit jako nástroj pro zajištění konfliktové uspořádatelnosti tím, že potenciálně konfliktním dvojicí akcí je zámky nastaveno pevné pořadí vykonání
  - krátce, transakce nelze prokládat (vytvářet rozvrh) libovolně
  - uzamykací protokoly zjednodušují úlohu zajistit konfliktově uspořádatelný rozvrh

Uzamykání databázových entit

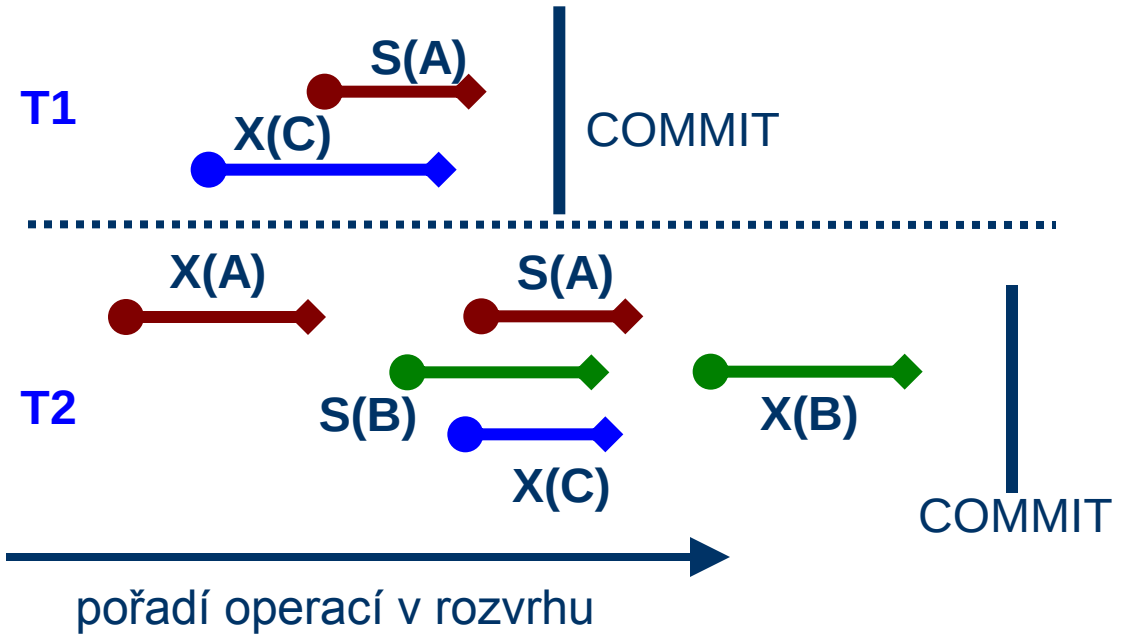
- **exkluzivní (exclusive) zámky**
  - **X(A)**, zcela uzamknou entitu A – číst i zapisovat do A může pouze vlastník zámku (ten, kdo uzamknul)
  - může být přidělen pouze jedné transakci
- **sdílené (shared) zámky**
  - **S(A)**, uzamknou entitu pro zápis – číst A může vlastník zámku – ten má jistotu, že do A nikdo nezapisuje
  - může být přidělen (sdílen) více transakcím (pokud již na A neexistuje exkluzivní zámek)
- odemyká se požadavkem **U(A)**
- pokud transakce požaduje zámek, který není k dispozici (je přidělen jiné transakci), je suspendována a čeká na přidělení
- uzamykání je uživateli (kódu transakce) skryto, používá ho rozvrhovač podle zvoleného uzamykacího protokolu
  - tj. použití zámků se objevuje pouze v rozvrhu (kam jsou přidány), ne v samotných (neproložených) transakcích

Implementace uzamykání

- akce X(A), S(A) lze chápat volněji – jako vznesení požadavku na zamknutí, tj. na těchto akcích „kurzor“ v rozvrhu „nečeká“, nicméně nelze pokračovat v rozvrhování následujících akcích dané transakce, dokud není zámek přidělen tzv. správcem zámků
- správce zámků (lock manager)
  - tabulka zámků
  - záznam v tabulce pro daný zámek: počet transakcí sdílejících zámek, typ zámku, ukazatel do fronty uzamykacích požadavků
- atomicita
  - je zcela nezbytné, aby zamykání a odemykání byly atomické operace (prostředky systému)

Příklad: rozvrh s uzamykáním

T1	T2
	X(A)
X(C)	
	W(A)
R(C)	
W(C)	
	U(A)
S(A)	
	S(B)
R(A)	
	R(B)
U(C)	
	X(C)
U(A)	
	S(A)
COMMIT	
	W(C)
	R(A)
	U(B)
	U(C)
	U(A)
	X(B)
	W(B)
	U(B)
	COMMIT



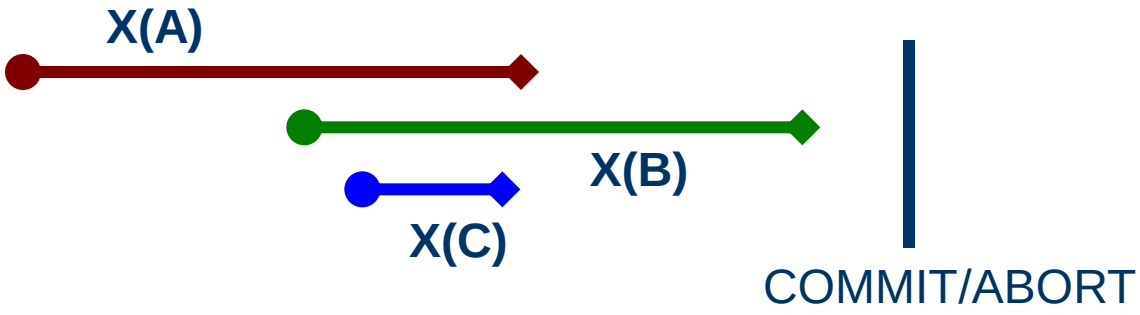
Dvoufázový uzamykací protokol (2PL)

2PL uplatňuje dvě pravidla pro sestavení rozvrhu:

1) pokud chce transakce číst (resp. modifikovat) entitu A, nejprve musí obdržet sdílený (resp. exkluzivní) zámek na A
2) transakce nemůže požadovat **žádný zámek**, pokud už nějaký zámek měla a uvolnila ho – na libovolné entitě

Jsou zřejmé dvě fáze – zamykání a odemykání

Příklad: upravení druhé transakce v předchozím rozvrhu na **2PL**

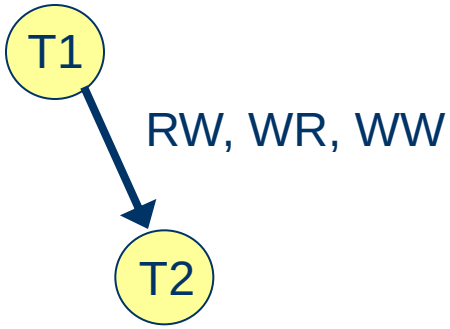


Vlastnosti 2PL

- omezení rozvrhu tak, aby splňoval 2PL zaručuje, že precedenční graf rozvrhu je **acyklický**, tj. rozvrh je **konfliktově uspořádatelný**
- 2PL **negarantuje zotavitelnost** rozvrhu

Příklad: rozvrh splňující 2PL, ale nezotavitelný, pokud T1 transakci zruší

T1	T2
X(A)	
R(A)	
W(A)	
U(A)	
	X(A)
	R(A)
	A := A *1.01
	W(A)
	S(B)
	U(A)
	R(B)
	B := B *1.01
	W(B)
	U(B)
	COMMIT
ABORT / COMMIT	

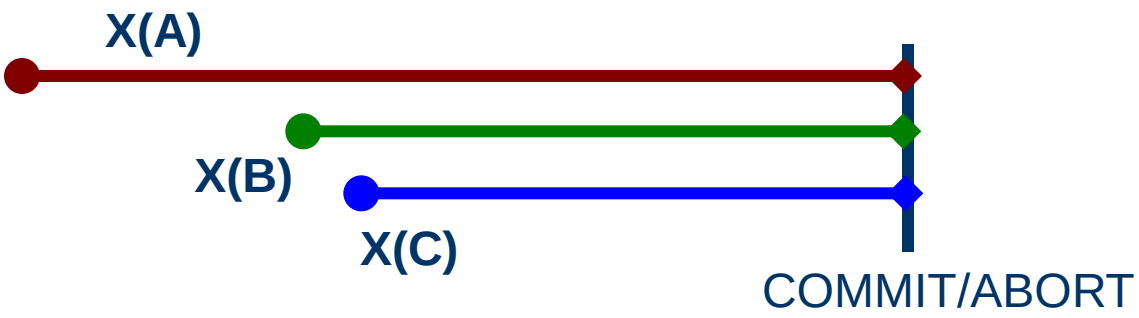


Striktní 2PL

Striktní 2PL „zostřuje“ druhé pravidlo 2PL, obě tedy zní:

1) pokud chce transakce číst (resp. modifikovat) entitu A, nejprve musí obdržet sdílený (resp. exkluzivní) zámek na A
2) **všechny zámky jsou uvolněny při ukončení transakce**

Příklad: upravení druhé transakce v předchozím rozvrhu na **striktní 2PL**



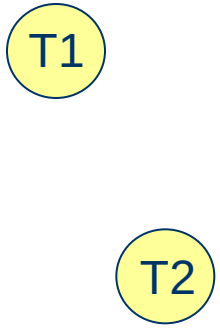
Vkládat požadavky U(A) do rozvrhu není potřeba, provedou se implicitně při COMMIT nebo ABORT.

Vlastnosti striktního 2PL

- omezení rozvrhu tak, aby splňoval 2PL zaručuje, že precedenční graf rozvrhu je **acyklický**, tj. rozvrh je **konfliktově uspořádatelný**
- striktní 2PL navíc **garantuje**
  - **zotavitelnost** rozvrhu (transakci lze stornovat a obnovit původní hodnoty)
  - zabezpečení proti **kaskádovému rušení** transakcí

Příklad: rozvrh splňující striktní 2PL

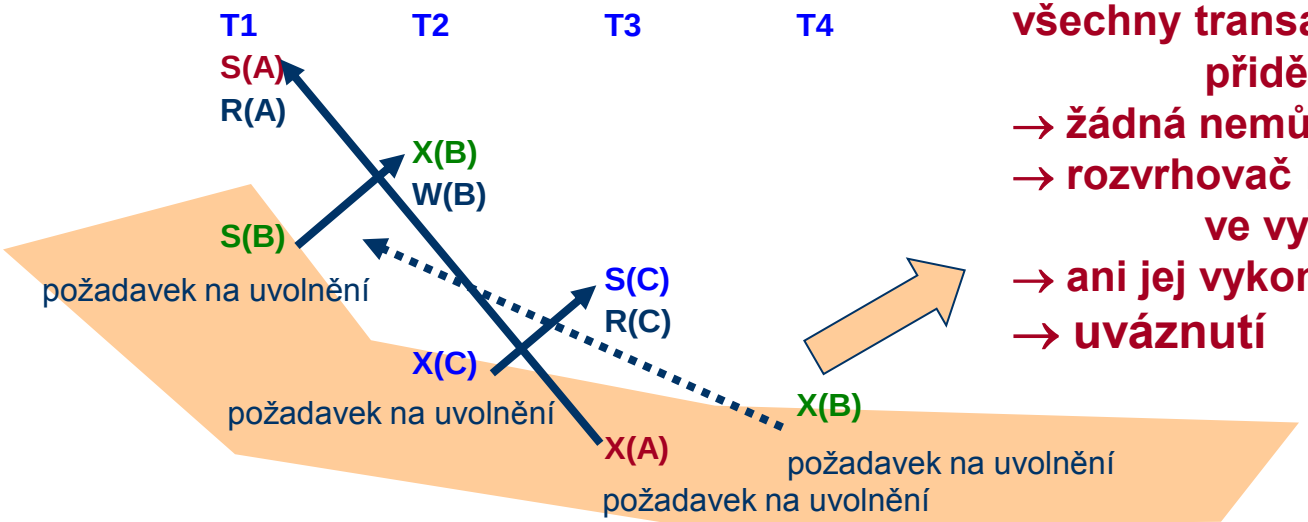
T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
	X(B)
X(C)	
R(C)	
	R(B)
	W(B)
	COMMIT
W(C)	
ABORT / COMMIT	



Uváznutí (deadlock)

- při zpracovávání rozvrhu může dojít k situaci, kdy transakce T1 žádá zámek, který má přidělený T2, ale ta ho nemohla uvolnit, protože zase čeká na zámek neuvolněný T1
  - situaci lze zobecnit na více transakcí,
    T1 čeká na T2, T2 čeká na T3, ..., Tn čeká na T1
- může nastat i při použití striktního 2PL (nemluvě o slabších protokolech)

Příklad:



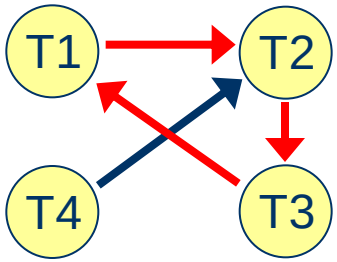
všechny transakce čekají na přidělení zámku
→ žádná nemůže zámek uvolnit
→ rozvrhovač nemůže pokračovat ve vytváření rozvrhu
→ ani jej vykonávat
→ uváznutí

Detekce uváznutí

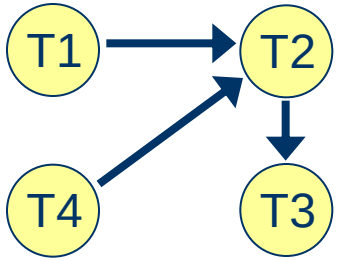
- uváznutí lze detekovat pravidelným zkoumáním tzv. waits-for grafu
- **waits-for** graf je dynamicky se měnící graf, podle toho jak jsou přidělovány/uvolňovány zámky
  - uzly tvoří aktivní transakce rozvrhu
  - orientovaná hrana znázorňuje čekání transakce T1 na (nějaký) zámek v současnosti přidělený transakci T2
  - pokud je v grafu cyklus, došlo k uváznutí (alespoň dvou transakcí)

Příklad: graf k předchozímu příkladu

(a) s požadavkem T3 na X(A)



(b) bez požadavku T3 na X(A)



Konverze zámků

- **lock upgrade** – změna již uděleného sdíleného zámku na exkluzivní
  - může pomoci zmenšit riziko uváznutí
  - Příklad: když transakce T1 již má S(A) a později požaduje i X(A), měla by dostat přednost před T2, která rovněž požaduje X(A), ale stejně ho nemůže dostat, protože S(A) vlastní T1

- **lock downgrade** – změna již uděleného exkluzivného zámku na sdílený
  - rovněž zmenšuje riziko uváznutí, i když snižuje paralelismus díky (potenciálně bezúčelně) přidělovaným exkluzivním zámkům
  - 2PL lze rozšířit tak, aby zahrnoval i lock downgrade
  - používá většina komerčních systémů

Řešení uváznutí

- k uváznutí zpravidla dochází zřídka, proto lze případné uváznutí řešit jednoduše
  - pokud transakce čeká na zámek moc dlouho, asi uvázla – transakci zrušíme
- pro lepší diagnostiku se použije waits-for graf, který je periodicky testován na cyklus
  - pokud nastane uváznutí, zrušíme jednu transakci v cyklu
  - zruší se ta transakce v cyklu, která (jedna z možností)
    - má nejmenší počtu zámků
    - zatím vykonala nejméně práce
    - má nejdále k dokončení
  - zrušená transakce může být znovu restartována a při dalším případném uváznutí upřednostněna (aby nebyla zrušena znovu)

Prevence uváznutí

- **prioritní upřednostňování**
  - každá transakce má prioritu (danou např. časovým razítkem, čím starší transakce, tím vyšší priorita)
  - pokud transakce T1 požaduje zámek a T2 již tento zámek drží, správce zámků volí mezi dvěma strategiemi
    - **wait-die** – pokud T1 má vyšší prioritu, může čekat, pokud ne, je zrušena
    - **wound-wait** – pokud T1 má vyšší prioritu, zruší se T2, jinak T1 čeká
  - restartované (zrušené) transakci je potřeba přiřadit původní prioritu, aby měla vyšší šanci na dokončení při případném dalším uváznutí
- **konzervativní 2PL protokol**
  - protokol, který požaduje, aby **všechny zámky**, které transakce bude v celém svém průběhu potřebovat, byly žádány na začátku transakce
  - pokud nelze přidělit všechny, je alespoň požádáno o jejich blokování (rezervaci) tak, že jakmile dojde k jejich uvolnění, dostane je transakce najednou a začne pracovat
  - protokol není v praxi používán, protože
    - se těžko předem odhaduje, které zámky budou potřeba v celém průběhu transakce (dynamicita a větvení)
    - vysoká režie uzamykání

Fantom

- nyní uvažujme dynamickou databázi, tj. do databáze lze vkládat a z databáze mazat (tj. ne pouze číst a aktualizovat stávající data)
- pokud jedna transakce pracuje s množinou entit a druhá tuto množinu LOGICKY mění (přidává nebo odebírá), může to mít za následek nekonzistenci databáze (neuspořádatelný rozvrh)
  - proč: T1 uzamkne všechny entity, které v dané chvíli odpovídají jisté vlastnosti (např. podmínce WHERE příkazu SELECT)
  - T2 v průběhu zpracování T1 může tuto množinu LOGICKY rozšířit (tj. v této chvíli by množina zámků definovaná podmínkou WHERE byla větší), což má za následek, že některé nově přidané entity budou uzamknuté (a tudíž i zpracované), kdežto jiné ne
- platí i pro striktní 2PL

Příklad – fantom

T1: najdi nejstaršího zaměstnance – muže a nejstarší zaměstnankyni – ženu
(**SELECT * FROM** Zaměstnanci ...) + **INSERT INTO** Statistika ...

T2: vlož zaměstnance Františka a smaž zaměstnankyni Evu (náhrada zaměstnance)
(**INSERT INTO** Zaměstnanci ..., **DELETE FROM** Zaměstnanci ...)

Výchozí stav databáze: **{[Pepa, 52, m], [Jaroslav, 46, m], [Eva, 55, ž], [Dáša, 30, ž]}**

T1	T2
uzamkni muže, tj.	
S(Pepa)	
S(Jaroslav)	
M = max{**R(Pepa)**, **R(Jaroslav)**}	
	Insert(František, 72, m)
	uzamkni ženy, tj.
	X(Eva)
	X(Dáša)
	Delete(Eva)
	COMMIT
uzamkni ženy, tj.	
S(Dáša)	
Ž = max{**R(Dáša)**}	
Insert(M, Ž) // výsledek se vloží do tabulky Statistika	
COMMIT	

fantom
lze vložit zaměstance muže, ačkoliv by (logicky) měli být uzamknuti **všichni** muži během celé transakce T1

Ačkoliv rozvrh splňuje **striktní 2PL**, výsledek **[Pepa, Dáša]** je nekorektní, protože neodpovídá ani sériovému rozvrhu T1, T2, který vrátí **[Pepa, Eva]**, ani T2, T1, který vrátí **[František, Dáša]**.

Fantom – prevence

- pokud neexistují indexy (např. B+-stromy) na entitách „uzamykací podmínky“, je potřeba zamknout vše, co by mohlo být fantomem negativně ovlivněno
  - např. celou tabulku, nebo i více tabulek
- pokud existují indexy na entitách „uzamykací podmíky“, je možné „hlídat fantoma“ na úrovni indexu (**index locking**) tak, že při vnějším pokusu o modifikaci množiny vyhovujících záznamů je „potenciálně fantomová transakce“ pozdržena stejně, jako by byla nová/modifikovaná/mazaná entita uzamčena
- zobecněním indexového uzamykání je predikátové uzamykání (predicate locking), kdy se požadují zámky přímo na úrovni logické podmínky a ne na úrovni fyzické
  - těžko se implementuje, proto se používá zřídka

Úrovně izolace

- čím striktnější uzamykací protokol, tím horší možnosti souběžného zpracování transakcí
  - pro různé účely jsou vhodné různé protokoly tak, aby se dosáhlo co nejvyššího výkonu a dostatečné „iluze“ izolace transakcí
- proto SQL-92 charakterizuje úrovně izolace pro různé účely

Úroveň	Protokol	WR	RW	Fantom
READ UNCOMMITTED (read only transakce)	žádný	možná	možná	možná
READ COMMITTED	S2PL na X() + 2PL na S()	Ne	možná	možná
REPEATABLE READ	S2PL	Ne	Ne	možná
SERIALIZABLE	S2PL + prevence fantoma	Ne	Ne	Ne

Alternativní („nezámkové“) protokoly

- mějme DB situaci, v níž transakce mohou přijít do konfliktu jen zřídka, tehdy
  - jsou uzamykací protokoly „kanón na vrabce“ – zbytečně pesimistické řízení – pořád se něco zamyká/odemyká
  - režie uzamykání neúměrně vysoká skutečnému využití zámků při prevenci konfliktů
- lepší použít
  - optimistické řešení
  - řízení pomocí časových razítek

Optimistické řízení

- základní předpoklad je, že ke souběžně vykonávané transakce mohou přijít do konfliktu jen zřídka (tj. průnik dat, na kterých každé dvě transakce pracují, je malý nebo nulový)
- třífázový optimistický protokol
  - **Read**: transakce reálně čte data z DB, nicméně zapisuje do svého lokálního datového prostoru
  - **Validation**: jakmile chce transakce potvrdit, předloží SŘBD svůj datový prostor (tj. požadavek na změnu databáze)
    - SŘBD rozhodne, zda takový požadavek není v konfliktu s jinou transakcí – pokud ano, transakce je zrušena a restartována
    - pokud je vše nekonfliktní, nastance poslední fáze:
  - **Write**: obsah lokálních dat se zkopíruje do databáze
- pokud naopak nastává mnoho konfliktů (transakce hodně „soutěží“ o společná data),
  - je optimistické řízení nevýhodné, často dochází k rušení/restartu transakcí
  - lépe je použít uzamykací protokol

Časová razítka

- uzamykací protokoly tím, že vynucují čekání transakce na zámek, konfliktově uspořádávají akce (a tím vlastně i celé transakce), aby odpovídaly nějakému sériovému rozvrhu
- využití uspořádání konfliktů lze zařídit i bez uzamykání, pomocí **časových razítek**
  - každá transakce Ti obdrží na začátku vykonávání časové razítko TS(Ti) (logický nebo fyzický čas v okamžiku startu transakce)
  - během zpracování/rozvrhování je kontrolováno pořadí konfliktních operací, tj. pokud akce A1 transakce T1 je v konfliktu s akcí A2 transakce T2, musí A1 nastat před A2 pokud TS(T1) < TS(T2)
    - pokud tato podmínka neplatí, je transakce T1 zrušena a restartována
  - implementuje se pomocí časových razítek pro čtení i zápis každé entity v databázi
  - pro dosažení zotavitelnosti rozvrhu se musí navíc implementovat „bufferování“ všech zápisů dokud transakce nepotvrdí

Zotavení po havárii systému

- správce zotavení (recovery manager) zajišťuje
  - **atomicitu** – odvolání (undoing) všech akcí, pokud je transakce zrušena
  - **trvanlivost** – promítnutí všech potvrzený akcí do databáze, i když systém zhavaruje
- jedna z nesložitějších a nejhůře implementovatelných součástí SŘBD
- ARIES – algoritmus pro zotavení
  - používaný mnoha systémy (např. IBM DB2, MS SQL, Informix, Sybase, Oracle)
  - algoritmus se spustí po restartu systému, který předtím zhavaroval)
- tři fáze ARIES:
  - **Analysis**: identifikují se „dirty pages“ (modifikované, ale nezapsané stránky) v bufferu a transakce, které byly aktivní v okamžiku havárie
  - **Redo**: zopakují se všechny akce od příslušného místa zapsané v tzv. **logu** a databáze se obnoví do stavu před havárií
  - **Undo**: odvolají se všechny akce těch transakcí, které nestačily potvrdit, tj. databáze je nakonec ve stavu, který odráží pouze potvrzené transakce

Principy algoritmu ARIES

- Write-Ahead Logging (WAL)
  - každá změna databáze je nejdříve zaznamenána do logu (který musí být na stabilním médiu)
- Repeating History During Redo
  - při restartu systému ARIES pomocí záznamů v logu vystopuje všechny akce před havárií a znovu je promítne (redo) tak, aby databáze byla ve stavu před havárií
  - potom jsou všechny nepotvrzené transakce zrušeny
- Logging Changes During Undo
  - při procesu rušení transakce (undo) jsou změny rovněž logovány, pro případ, že by systém (opět) zhavaroval ve fázi ABORT

Log

- někdy také nazývaný, **trail** nebo **journal**
- uchovává **historii akcí** provedených SŘBD
  - např. na úrovni stránek
- záznam v logu je identifikován (log sequence number - LSN)

Příklad – zotavení pomocí ARIES

Databázové systémy

Tomáš Skopal

– fyzická implementace
relačních databází

Osnova

- správa disku, stránkování, buffer manager
- organizace databázových souborů
- indexování
  - jednoatributové indexy
    - B^+-strom, bitové mapy, hašování
  - víceatributové indexy

Úvod

- relace/tabulky uloženy v souboru(rech) na disku
- potřeba organizace záznamů uvnitř souboru pro jejich efektivní uložení, modifikaci a přístup k nim

Příklad:
Zaměstnanec(jmeno char(20), vek integer, mzda integer)

Stránkování

- záznamy fyzicky organizovány ve stránkách pevné velikosti (blocích o několika kB) na disku
- důvod je HW, disk obsahuje rotační plotny a čtecí hlavy, data je potřeba přizpůsobit tomuto mechanismu
- HW je schopen přistupovat k celým stránkám (I/O operace – čtení, zápis)
- čas pro I/O operaci =
  = seek time + rotational delay + data transfer time
- sekvenční přístup ke stránkám je mnohonásobně rychlejší než náhodný přístup, odpadá seek time a rotational delay

  Příklad: načtení 4 KB může trvat typicky 8 + 4 + 0,5 ms = 12,5 ms; tj. samotné čtení trvá pouhých 0,5 ms = 4% celkového času!!!

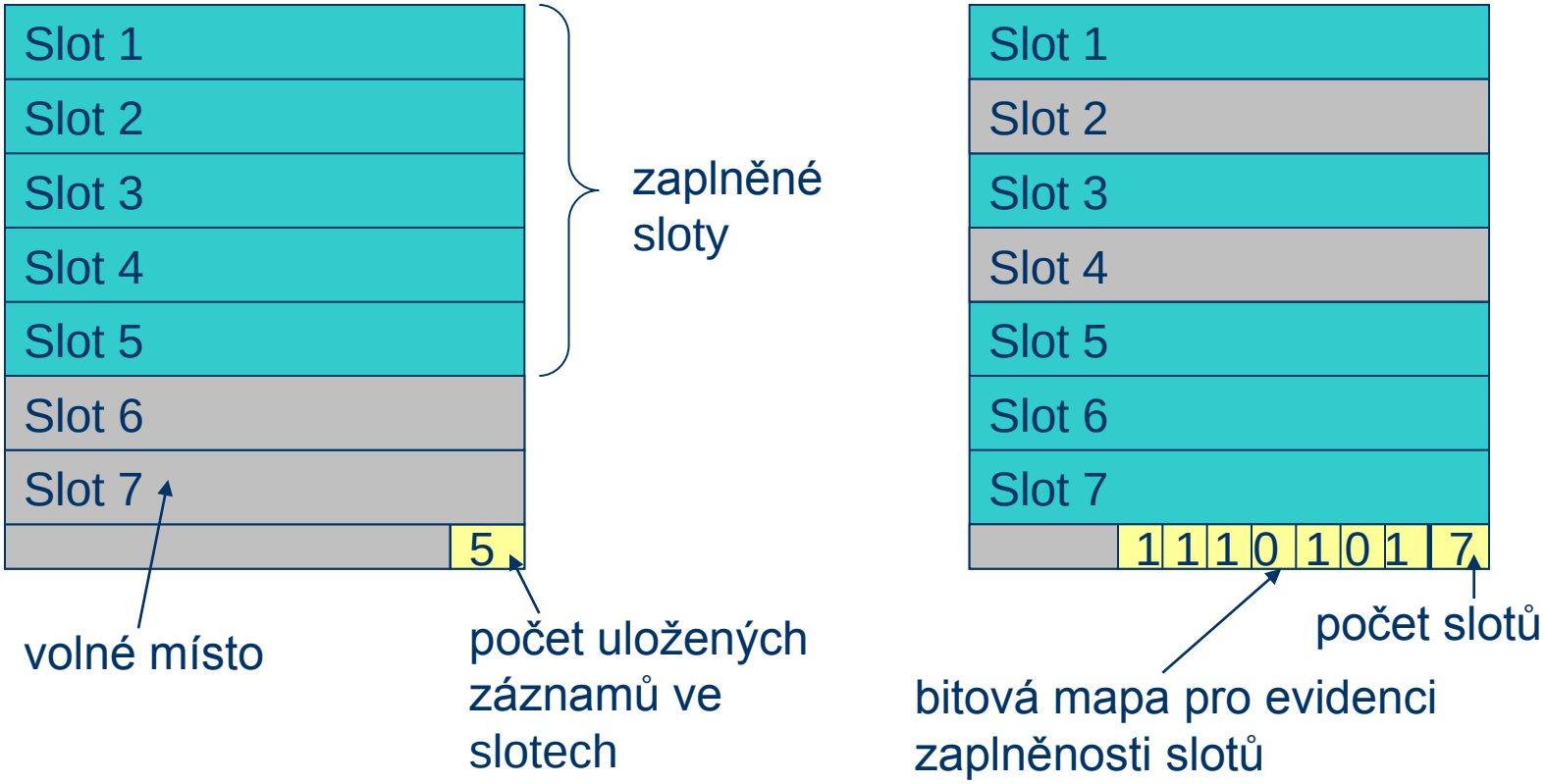
Stránkování, pokr.

- I/O jako jednotka časových nákladů
- stránka je rozdělena na sloty, do kterých se ukládají záznamy, identifikována před *page id*
- záznam může být uložen
  - přes více stránek = lepší využití místa, ale potřeba více I/O pro manipulaci se záznamem
  - nebo jen v jedné stránce (za předpokladu že se tam vejde) = příp. nevyužití celé stránky, méně I/O
  - v ideálním případě záznamy bezezbytku vyplňují stránku
- záznam identifikován pomocí **rid** (record id), což je dvojice *page id* a *slot it*

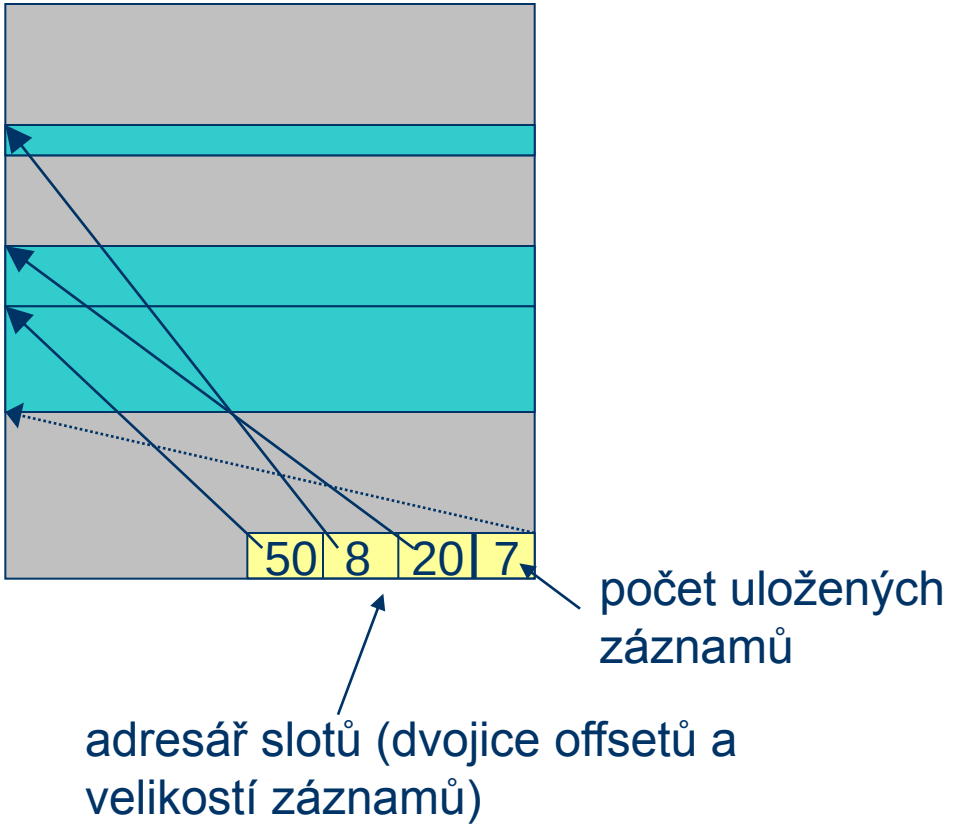
Stránkování, pokr.

- záznam je tvořen hodnotami datových typů pevné velikosti → pevná velikost záznamu
- přítomnost datových typů proměnlivé velikosti → proměnlivá velikost záznamu např. typy varchar(X), blob, ...
- záznamy pevné délky = sloty pevné délky
- záznamy proměnlivé délky = potřeba adresáře slotů v hlavičce každé stránky

Organizace stránky pro záznamy pevné velikosti, příklad



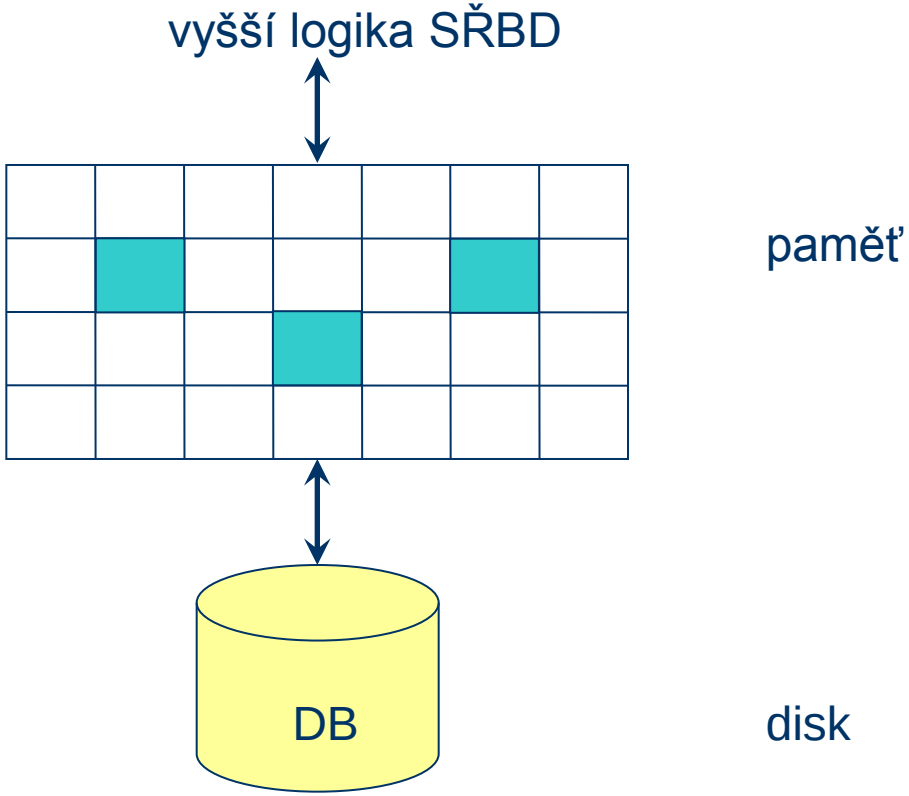
Organizace stránky pro záznamy proměnlivé velikosti, příklad



Buffer a jeho správa

- buffer = kus hlavní paměti pro dočasné uchování diskových stránek, diskové stránky se mapují do rámců v paměti 1:1
- každý rámec má 2 příznaky: **pin_count** (počet referencí na stránku v rámci) a **dirty** (příznaky modifikace záznamů)
- slouží k urychlení opakovaného přístupu ke stránkám - správce bufferu implementuje operace **read** a **write**
- odstínění vyšší logiky SŘBD od diskového managementu
- implementace **read** provede načtení stránky z bufferu, pokud tam není, provede se nejdříve načtení z disku (fetch), zvýšení **pin_count**
- implementace **write** zapíše stránku do bufferu, nastaví se **dirty**
- pokud v bufferu není místo (během read nebo write), uvolní se nějaká jiná stránka → různé politiky uvolňování, např. LRU (least-recently-used), pokud má uvolňovaná stránka nastaveno **dirty**, uloží se (store)

Buffer a jeho správa, schéma



Organizace databáze

- datové soubory
  (obsahující veškerá data tabulek)
- indexové soubory
- systémový katalog – obsahuje metadata
  - schémata tabulek
  - jména indexů
  - integritní omezení, klíče, atd.

Datové soubory

- halda
- uspořádaný soubor
- hašovaný soubor

Sledujeme průměrné I/O náklady jednoduchých operací:

1) sekvenční načtení záznamů
2) vyhledání záznamů na rovnost (podle vyhledávacího klíče)
3) vyhledání záznamů na rozsah (podle vyhledávacího klíče)
4) vložení záznamu
5) vymazání záznamu

Cost model:

N = počet stránek, R = počet záznamů na stránku

Jednoduché operace, SQL příklady

- sekvenční načtení
  SELECT * FROM Zaměstnanci
- vyhledání na rovnost
  SELECT * FROM Zaměstnanci WHERE věk = 40
- vyhledání na rozsah
  SELECT * FROM Zaměstnanci
  WHERE mzda > 10000 AND mzda < 20000
- vložení záznamu
  INSERT INTO Zaměstnanci VALUES (...)
- vymazání záznamu podle **rid**
  DELETE FROM Zaměstnanci WHERE rid = 1234
- DELETE FROM Zaměstnanci WHERE mzda < 5000

Halda (heap file)

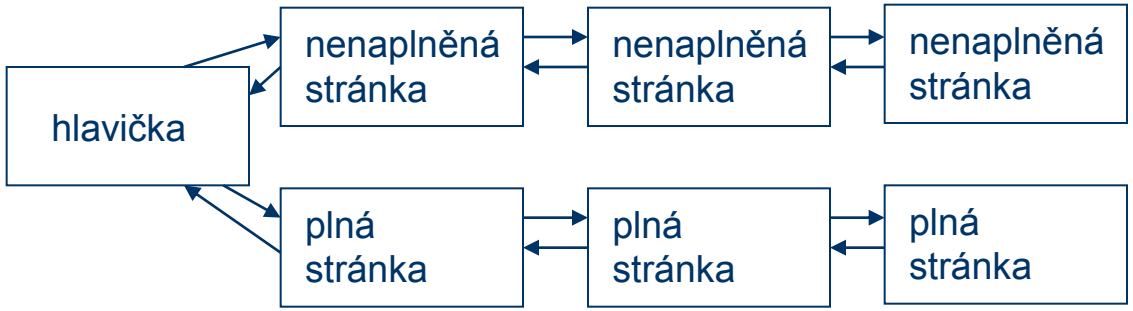
- záznamy ve stránkách uloženy neuspořádaně sekvenčně za sebou, resp. jsou ukládány tak, jak přicházejí požadavky *insert*
- vyhledání stránky možné pouze sekvenčním průchodem (a operace GetNext)
- rychlé vkládání záznamů na konec souboru
- problémy s mazáním → „díry“ (kusy prázdného prostoru)

Údržba prázdných stránek haldy

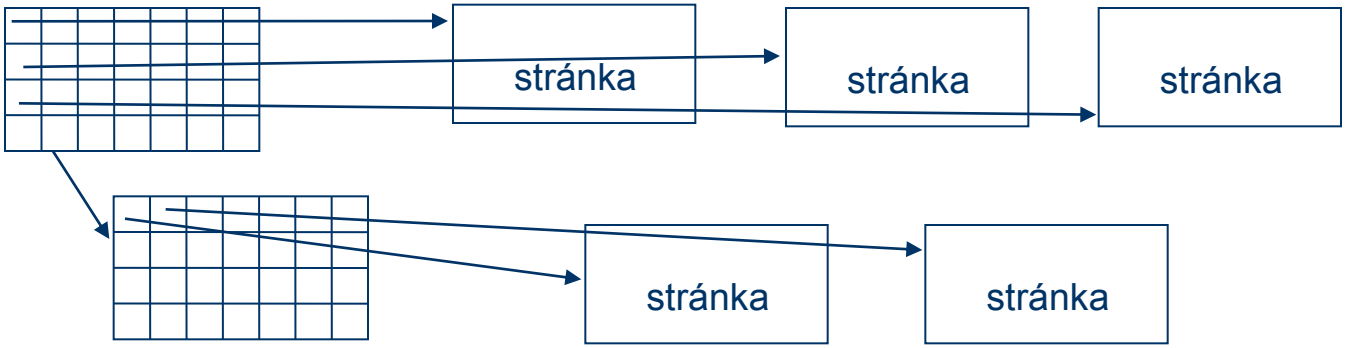
- dvojitě spojový seznam
  - hlavička + seznamy zaplněných a nezaplněných stránek
- adresář stránek
  - spojový seznam adresářových stránek
  - každá položka v adresáři ukazuje na datovou stránku
  - příznakový bit zaplněnosti stránky pro každou položku

Údržba prázdných stránek haldy, pokr.

dvojitě spojový seznam



adresář stránek



Halda, náklady jednoduchých operací

- sekvenční načtení = N
- vyhledávání na rovnost = 0,5*N nebo N
- vyhledávání na rozsah = N
- vložení záznamu = 1
- vymazání záznamu = 2 za předpokladu, že vyhledávání podle **rid** stojí 1 I/O,
  pokud se maže na shodu nebo na rozsah, náklady jsou N nebo 2*N

Setříděný soubor (sorted file)

- záznamy ve stránkách uloženy uspořádaně podle vyhledávacího klíče (jeden nebo více atributů)
- stránky souboru jsou udržovány spojitě,
  tj. neexistují „díry“ prázdného prostoru
- umožňuje rychlé vyhledávání podle klíče a to jak na rovnost, tak na rozsah
- pomalé vkládání a mazání, „hýbání“ se zbytkem stránek
- v praxi se používá kompromis – za začátku je setříděný soubor, každá stránka má „volnou rezervu“, kam se vkládá; pokud je rezerva zaplněna, využívají se aktualizační stránky (spojový seznam). Jednou za čas je třeba provést reorganizaci, tj. setřídění

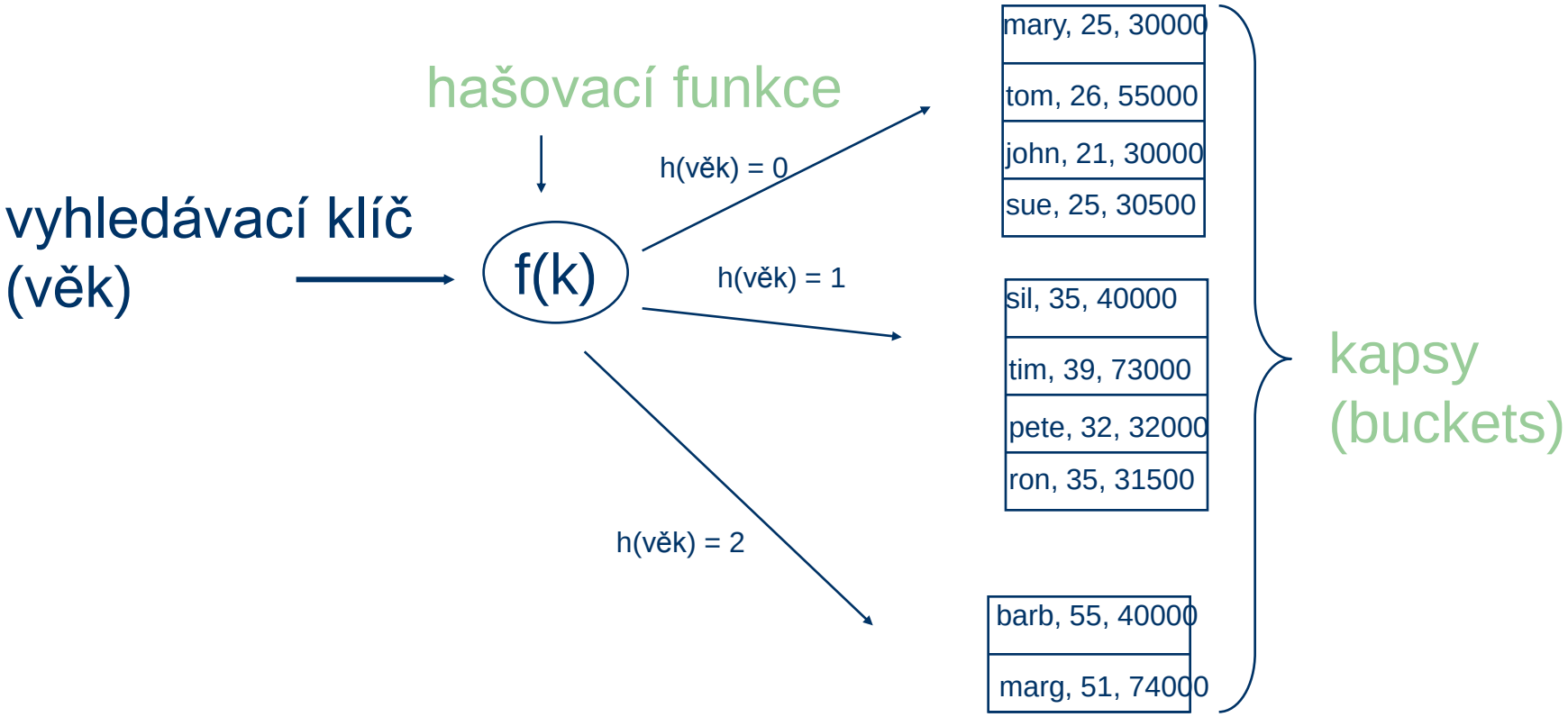
Setříděný soubor, náklady jednoduchých operací

- sekvenční načtení = N
- vyhledávání na rovnost = $\log_2 N$ nebo N
- vyhledávání na rozsah = $\log_2 N + M$
  (kde M je počet relevantních stránek)
- vložení záznamu = N
- vymazání záznamu = $\log_2 N + N$ podle klíče, jinak 1,5*N

Hašovaný soubor (hashed file)

- organizován do skupiny K kapes (buckets), kapsa může sestávat z několika stránek
- záznam je vložen/čten do/z kapsy, která je určena hašovací funkcí a klíčem pro vyhledání; id kapsy = f(klíč)
- pokud není v kapse místo, vytvoří se nové stránky, které se na kapsu napojí (spojový seznam)
- rychlé dotazy na shodu a mazání na shodu
- vyšší prostorová režie, komplikace se zřetězenými stránkami (řeší dynamické hašovací techniky)

Hašovaný soubor



Hašovaný soubor, náklady jednoduchých operací

- sekvenční načtení = N
- vyhledávání na rovnost = N/K (v ideálním případě)
- vyhledávání na rozsah = N
- vložení záznamu = N/K (v ideálním případě)
- vymazání záznamu na shodu = N/K + 1 (v ideálním případě), jinak N

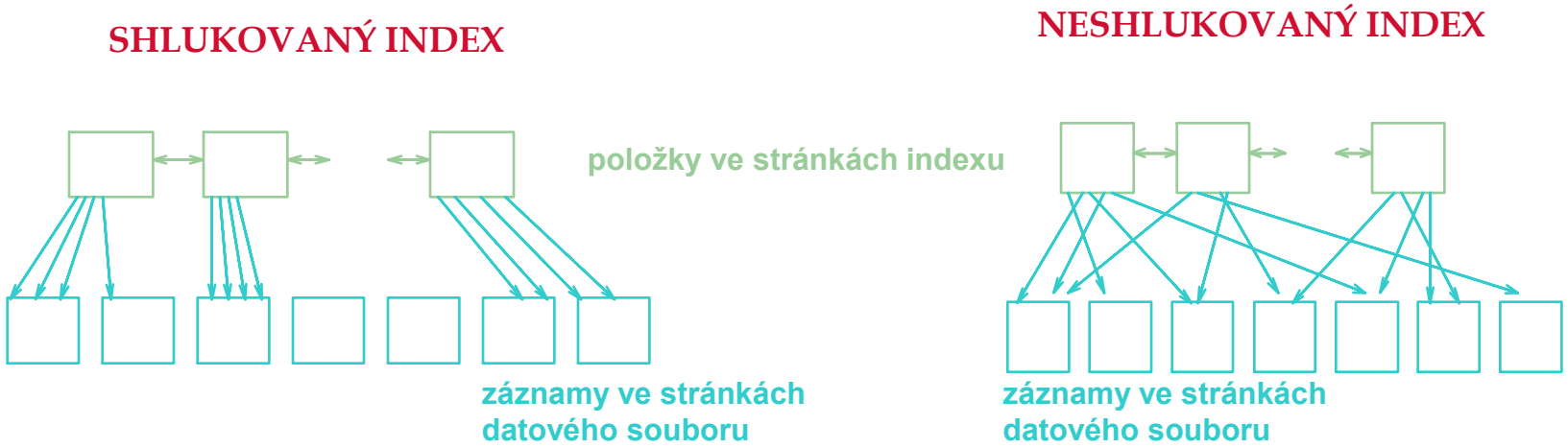
Indexování

- index je pomocná struktura umožňující rychle vyhledávat podle vyhledávacího klíče (klíčů)
- organizována do stránek podobně jako datové soubory
- zpravidla v jiném souboru
- obsahuje pouze (některé) hodnoty klíčů a odkazy k příslušným záznamům (tj. rid)
- spotřebují daleko menší velikost prostoru (např. 100x méně) než datové soubory

Indexování, principy

- položka indexu může obsahovat
  - celý záznam (index a datový soubor splývají)
  - dvojici <klíč, rid>
  - dvojici <klíč, rid-list>, kde rid-list obsahuje seznam odkazů na záznamy se stejným klíčem
- shlukované vs. neshlukované indexy
  - **shlukované:** uspořádání položek ve stránkách indexu je (téměř) stejné jako uspořádání záznamů ve stránkách datového souboru, tuto vlastnost mají pouze stromové indexy + indexy obsahující celé záznamy (i hašované)
  - **neshlukované:** pořadí klíčů v obou strukturách není dodrženo

Indexování, principy



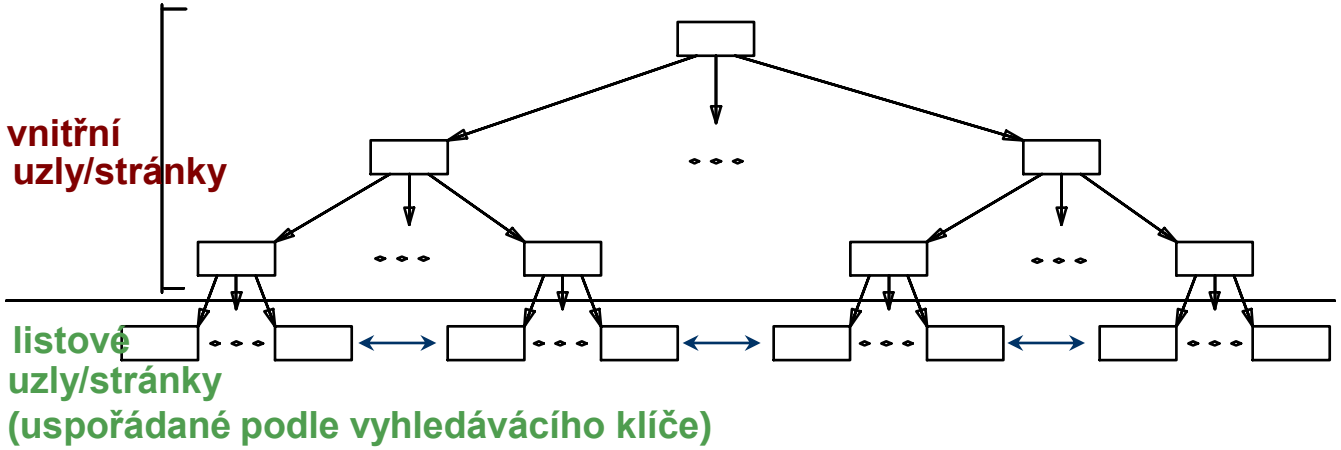
Výhodou shlukovaného indexu je velké zrychlení při vyhledávání na rozsah (rozsahový/intervalový dotaz), neboť stránky se záznamy jsou čteny sekvenčně. U neshlukovaného (a navíc stromového) indexu se sekvenčně čtou pouze stránky indexu.

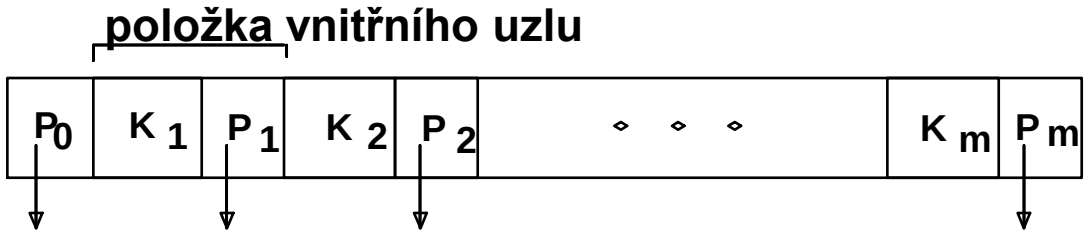
Nevýhody: velká režie při udržování uspořádání datového souboru, zvlášť pokud existují další indexy

B+-strom

- vychází z B-stromu, což je stránkovaný, vyvážený stromový index (Rudolf Bayer, 1972).
- poskytuje logaritmické složitosti pro vkládání, dotaz na shodu, mazání na shodu
- zaručuje 50% zaplněnost uzlů (stránek)
- B+-strom rozšiřuje B-strom o
  - provázání listových stránek pro efektivní rozsahové dotazy
  - vnitřní uzly obsahují indexované intervaly, tj. všechny klíče jsou v listech

B+-strom, schéma





Demo: http://slady.net/java/bt/

Hašovaný index

- podobně jako hašovaný soubor využívá kapsy a hašovací funkci
- v kapsách jsou pouze hodnoty klíčů spolu s odkazy na záznamy **rid**
- stejné výhody/nevýhody

Bitové mapy

- jsou vhodné pro indexování atributů s malou doménou (jednotky až desítky hodnot)
  - vhodné např. pro atribut **RODINNÝ_STAV** = {svobodný, ženatý, rozvedený, vdovec}
  - nevhodné např. pro atribut **CENA_VÝROBKU** (mnoho hodnot), tam bude lepší B-strom
- pro každou HODNOTU ***h*** indexovaného atributů ***a*** se zkonstruuje bitová mapa (binární vektor), kde jednička na pozici i znamená, že hodnota ***h*** se vyskytuje v i-tém záznamu tabulky (jako hodnota atributu ***a***) a platí
  - bitový součet (OR) všech map pro atribut vytvoří samé jedničky (každý záznam nabývá v daném atributu nějaké hodnoty)
  - bitový součin (AND) libovolných dvou map atributu je nula (každý záznam nabývá v atributu nejvýše jedné hodnoty)

Jméno	Adresa	Rodinný stav
František Novák	Liberec	svobodný
Rostislav Drobil	Praha	ženatý
René Vychodil	Ostrava	ženatý
Kamil Svoboda	Beroun	svobodný
Pavel Horák	Cheb	rozvedený

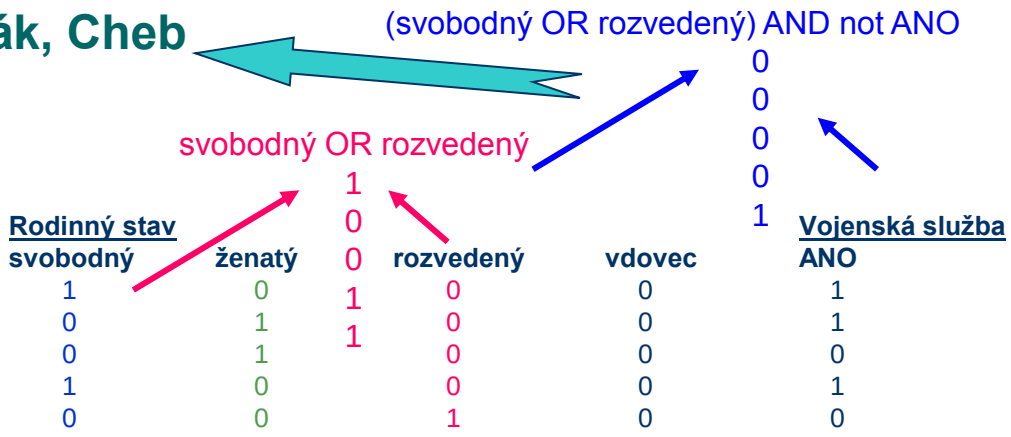
svobodný	ženatý	rozvedený	vdovec
1	0	0	0
0	1	0	0
0	1	0	0
1	0	0	0
0	0	1	0

Bitové mapy

- vyhodnocení dotazu
  - bitové operace s mapami jednotlivých hodnot atributů
  - výsledná bitová mapa označuje záznamy vyhovující dotazu
- příklad
  - **Kteří svobodní nebo rozvedení neabsolvovali vojenskou službu?**
    (bitmap(**svobodný**) **OR** bitmap(**rozvedený**)) **AND not** bitmap(**ANO**)

odpověď: Pavel Horák, Cheb

Jméno	Adresa	Vojenská služba	Rodinný stav
František Novák	Liberec	ANO	svobodný
Rostislav Drobil	Praha	ANO	ženatý
René Vychodil	Ostrava	NE	ženatý
Kamil Svoboda	Beroun	ANO	svobodný
Pavel Horák	Cheb	NE	rozvedený



Bitové mapy

- výhody
  - úspora místa, navíc lze efektivně (de)komprimovat podle potřeby
  - úspora místa souvisí i s rychlostí vyhodnocování dotazu, bitové operace jsou navíc rychlé
  - dotazy nad mapami lze jednoduše paralelizovat
- nevýhody
  - omezeno pouze na atributy s malou doménou
  - intervalové dotazy se zpomalují přímoúměrně s počtem hodnot v intervalu (je potřeba procházet bitové mapy všech hodnot v intervalu, neexistuje uspořádání)

Víceatributové indexování

- uvažujme konjunktivní rozsahový dotaz
  SELECT * FROM Zaměstnanci WHERE
  mzda BETWEEN 10000 AND 30000 AND
  věk < 40 AND
  name BETWEEN ‘Dvořák’ AND ‘Procházka’
- jednoduchá řešení řešitelná pomocí B+-stromu (počet indexovaných atributů M = 3):
  1) tři nezávislé indexy
  2) jeden index M zřetězených atributů

  - obě řešení jsou špatná, druhá varianta stačí pouze pro dotazy na shodu (tedy ne na rozsah), první ani na to

Víceatributové indexování, příklady

Tři samostatné indexy:

...WHERE **mzda BETWEEN 10000 AND 30000** AND **věk < 40** AND **name BETWEEN 'Dvořák' AND 'Procházka'**

r1	Čech Jaroslav	17000	27
r2	Dostál Jan	21000	33
r3	Malý Zdeněk	15000	45
r4	Mrázek František	22000	37
r5	Novák Josef	32000	25
r6	Novák Karel	13000	19
r7	Oplustil Arnošt	9000	36
r8	Papoušek Jindřich	19000	50
r9	Richter Tomáš	26000	41
r10	Zlámal Alois	13000	52

{r3, r4, r5, r6, r7, r8}

r7	Oplustil Arnošt	9000	36
r6	Novák Karel	13000	19
r10	Zlámal Alois	13000	52
r3	Malý Zdeněk	15000	45
r1	Čech Jaroslav	17000	27
r8	Papoušek Jindřich	19000	50
r2	Dostál Jan	21000	33
r4	Mrázek František	22000	37
r9	Richter Tomáš	26000	41
r5	Novák Josef	32000	25

{r6, r10, r3, r1, r8, r2, r4, r9}

r6	Novák Karel	13000	19
r5	Novák Josef	32000	25
r1	Čech Jaroslav	17000	27
r2	Dostál Jan	21000	33
r7	Oplustil Arnošt	9000	36
r4	Mrázek František	22000	37
r9	Richter Tomáš	26000	41
r3	Malý Zdeněk	15000	45
r8	Papoušek Jindřich	19000	50
r10	Zlámal Alois	13000	52

{r6, r5, r1, r2, r7, r4}

průnik = {r4, r6}

Víceatributové indexování, příklady

Index zřetězených atributů:

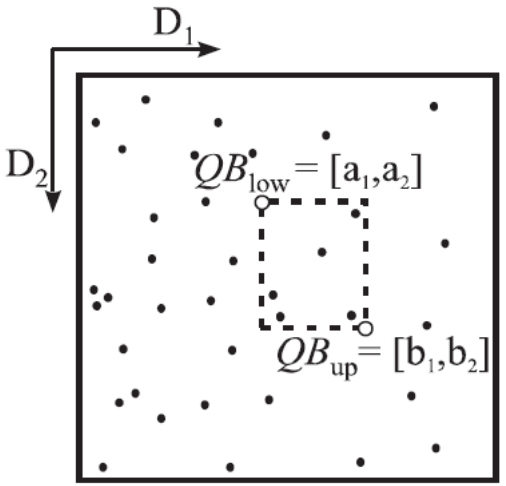
...WHERE **mzda BETWEEN 10000 AND 30000** AND **věk < 40** AND **name BETWEEN 'Dvořák' AND 'Procházka'**

r1	Čech Jaroslav	17000	27
r2	Dostál Jan	21000	33
r3	Malý Zdeněk	15000	45
r4	Mrázek František	22000	37
r5	Novák Josef	32000	25
r6	Novák Karel	13000	19
r7	Oplustil Arnošt	9000	36
r8	Papoušek Jindřich	19000	50
r9	Richter Tomáš	26000	41
r10	Zlámal Alois	13000	52

{r4, r6}

Prostorové indexování

- abstrakce M-tice klíčů jako M-rozměrných vektorů
  <Novák Josef, 32000, 25> ⇒ [sig('Novák Josef'), 32000, 25]
  - musí se zachovat uspořádání klíčů, např. pro sig(*)
- transformace na problém vyhledávání v M-rozměrném prostoru R^M
- konjunktivní rozsahový dotaz = (hyper)-kvádr QB v prostoru R^M vymezen dvěma body, dolní a horní meze rozsahů



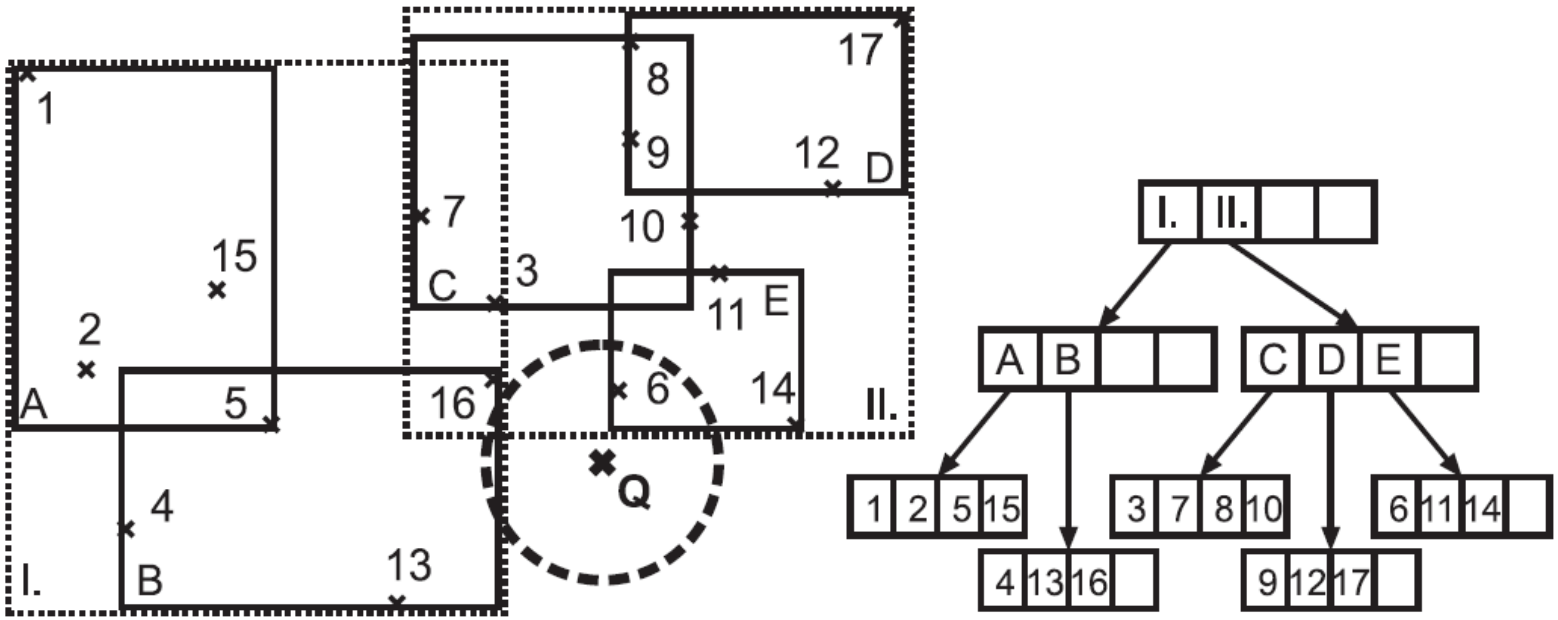
Prostorové indexování

- různé indexy (spatial access methods), založené na stromové struktuře, hašování i sekvenčním průchodu
- společným rysem nesekvenčních indexů je snaha o shlukování těch vektorů blízko ve stejné části indexu, které jsou shlukovány i v prostoru
- během rozsahového dotazu je potom (v ideálním případě) přistupováno jen k těm stránkám, které obsahují klíče uvnitř dotazovacího kvádru
- velmi dobře fungují do dimenze 10, potom přestávají být účinné a lepší jsou sekvenční indexy, ve kterých jsou klíče reprezentovány malým počtem bitů

Prostorové indexování

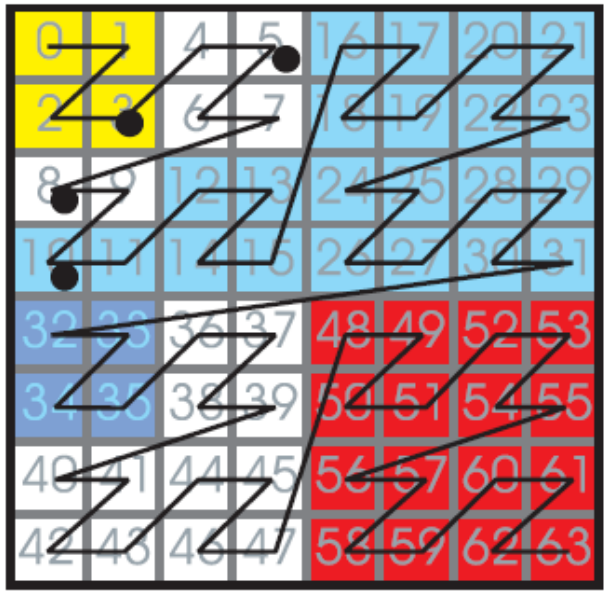
- stromové indexy
  - R-strom, UB-strom
- hašované indexy
  - Grid file
- sekvenční indexy
  - VA-file

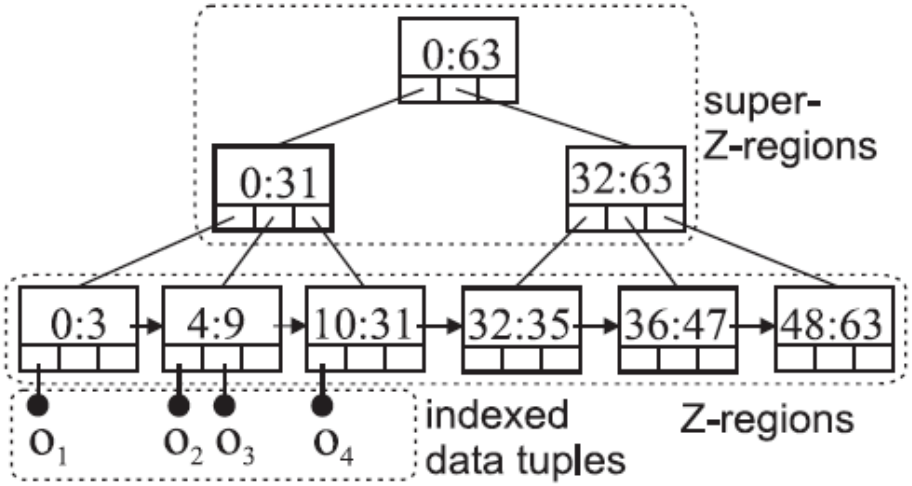
R-strom



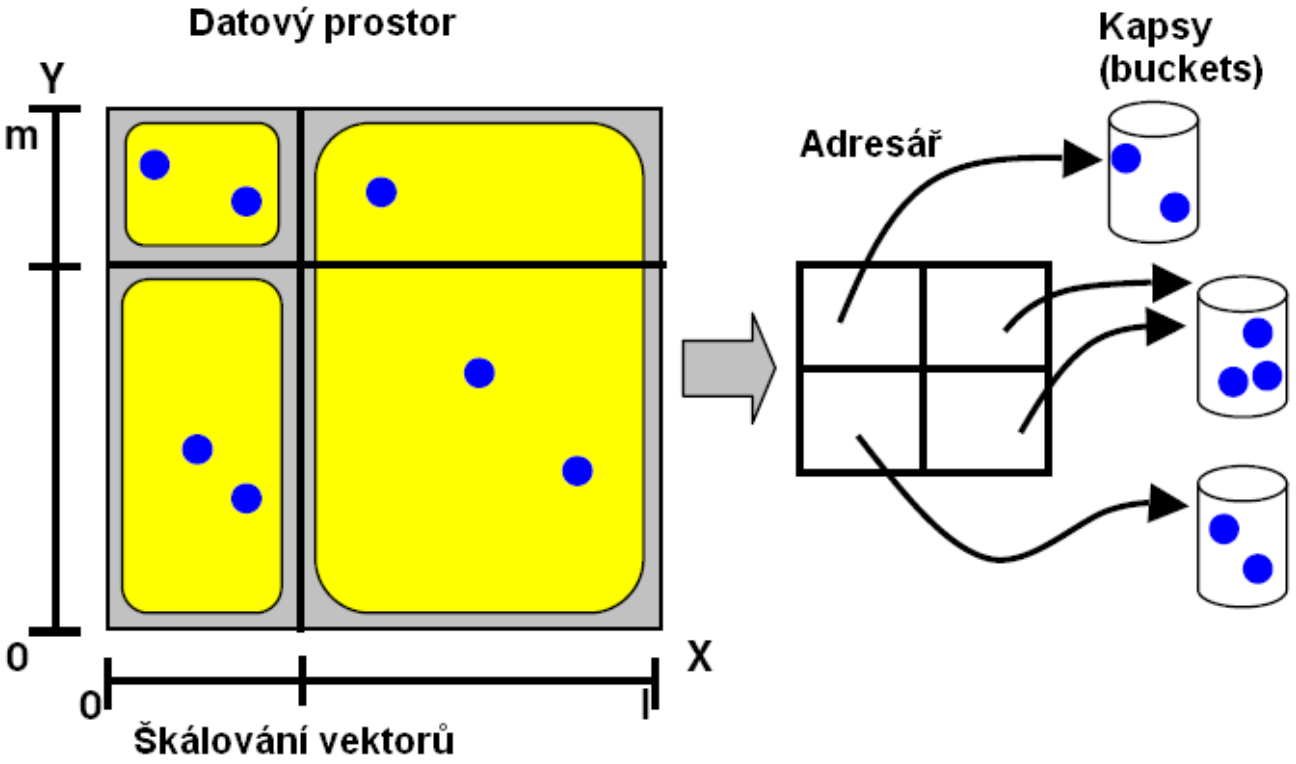
Demo: http://www.dbnet.ece.ntua.gr/~mario/rtree/

UB-strom




0:63
super-
Z-regions
0:31
32:63
0:3
4:9
10:31
32:35
36:47
48:63
o_1
o_2 o_3
o_4
indexed
data tuples
Z-regions

Grid file


Datový prostor
Y
m
0
0
X
l
Škálování vektorů
Kapsy
(buckets)
Adresář

Databázové systémy

Tomáš Skopal

- Databázové aplikace
  * embedded SQL
  * externí aplikace

Osnova přednášky

- „vnitřní“ programovací konstrukce (Embedded SQL)
  - uložené procedury
  - kurzory
  - triggery
- „vnější“ programování (přes rozhraní/knihovny)
  - rozhraní ODBC, JDBC, knihovna ADO.NET
  - rozhraní podporující objektově-relační mapování
    - Microsoft ADO
    - Java Hibernate

Programování v Embedded SQL

- procedurální rozšíření SQL, std. SQL je podjazyk (proto Embedded - vnořené)
  - SQL server: Transact SQL (T-SQL)
  - Oracle: PL/SQL
- přínosy
  - řídící konstrukce (nelze skriptovat), if-then, for/while cykly
  - kurzory (podpora sekvenčního průchodu přes řádky tabulky)
  - menší komunikační režie (kód uložen na serveru, narozdíl od skriptování)
  - zobecnění integritních omezení – triggery
  - vyšší bezpečnost – kód na serveru může mít vyšší práva (skripty ne)
- nevýhody
  - aplikace nepřenositelná mezi různými DB platformami – standardizace až v rámci SQL 1999 – ale nikdo nedodržuje

Struktura (SQL Server)

DECLARE sekce

BEGIN ... **END**

Např.:

```
DECLARE @prum_vek FLOAT
BEGIN
SELECT @prum_vek = AVG(vek) FROM Zamestnanec
END
```

Uložené procedury (SQL Server)

CREATE PROCEDURE *jmproc* [; *číslo*]
[*deklarace_parametru* [, …]]
[WITH RECOMPILE]
AS příkazy [;]

- Deklarace parametru
  - *@jméno typ* [= *výraz*] [OUT[PUT]]
    - OUT[PUT] parametr je výstupní
- *číslo* umožňuje vytvoření více verzí stejné procedury
- Volání procedury
  - EXEC[UTE] jmproc [výraz [, …]]
    - Parametry se předávají podle pořadí
  - EXEC[UTE] jmproc [@jméno=výraz [, …]]
    - Parametry se předávají podle jména

Procedury, příklad

```
CREATE PROCEDURE Platba
      @ucetZdroj INTEGER,
      @ucetCil INTEGER,
      @castka INTEGER = 0
    AS
    BEGIN
          UPDATE Accounts SET zustatek = zustatek - @castka
            WHERE ucet=@ucetZdroj;

          UPDATE Accounts SET zustatek = zustatek + @castka
            WHERE ucet=@ucetCil;
    END


EXEC Platba 21-87526287/0300, 78-9876287/0800, 25000;
```

Kurzory (SQL Server)

- Deklarace
  - C [SCROLL] **CURSOR FOR**
    SELECT …;
- Získání dat
  - **FETCH**
    {NEXT | PRIOR | ABSOLUTE n | RELATIVE n | LAST | FIRST}
    FROM C
    [INTO @proměnná [, …]]
  - Pokud kurzor není deklarovaný s klíčovým slovem SCROLL, je možné použít jen NEXT

Kurzory, příklad (placení daní)

```
DECLARE
     Cur CURSOR FOR
          SELECT *
          FROM Accounts;
    BEGIN
          OPEN Cur
          DECLARE @acc int, @zus int;
          FETCH NEXT FROM Cur INTO @acc, @zus;
          WHILE @@FETCH_STATUS=0
          BEGIN
            EXEC Platba(@acc, CU_URAD, @zus*0.01)
            FETCH NEXT FROM Cur;
          END;
          CLOSE Cur;
          DEALLOCATE Cur;
    END
```

Triggery – DML triggery

- událostí na tabulce/databázi spouštěná uložená procedura
- rozšíření funkcionality integritních omezení
  - ***inserted, deleted*** – logické tabulky

CREATE TRIGGER trigger_name **ON** { table | view }
[WITH ENCRYPTION]
{ FOR | AFTER | INSTEAD OF }
{ [INSERT] [,] [UPDATE] [,] [DELETE] }
[WITH APPEND]
AS
[{IF UPDATE (column) [{ AND | OR } UPDATE (column)] …
| IF (COLUMNS_UPDATED (bitwise_operator
updated_bitmask)}]
sql_statement [...]

DML - Triggery (příklad)

```
CREATE TRIGGER LowCredit ON Purchasing.PurchaseOrderHeader
AFTER INSERT
AS
DECLARE @creditrating tinyint, @vendorid int

SELECT @creditrating = v.CreditRating, @vendorid = p.VendorID FROM
    Purchasing.PurchaseOrderHeader p INNER JOIN inserted i ON p.PurchaseOrderID
    = i.PurchaseOrderID JOIN Purchasing.Vendor v on v.VendorID = i.VendorID

IF @creditrating = 5
BEGIN
    RAISERROR ('This vendor''s credit rating is too low to accept new purchase orders.',
    16, 1)
    ROLLBACK TRANSACTION
END
```

Triggery – DDL triggery

CREATE TRIGGER trigger_name **ON** { ALL SERVER | DATABASE }

[WITH <ddl_trigger_option> [,...n]]

{ FOR | AFTER } { event_type | event_group } [,...n] **AS**

{sql_statement [;] [,...n] | EXTERNAL NAME < method specifier > [;] }

<ddl_trigger_option> ::= [ENCRYPTION] [EXECUTE AS Clause]

<method_specifier> ::= assembly_name.class_name.method_name

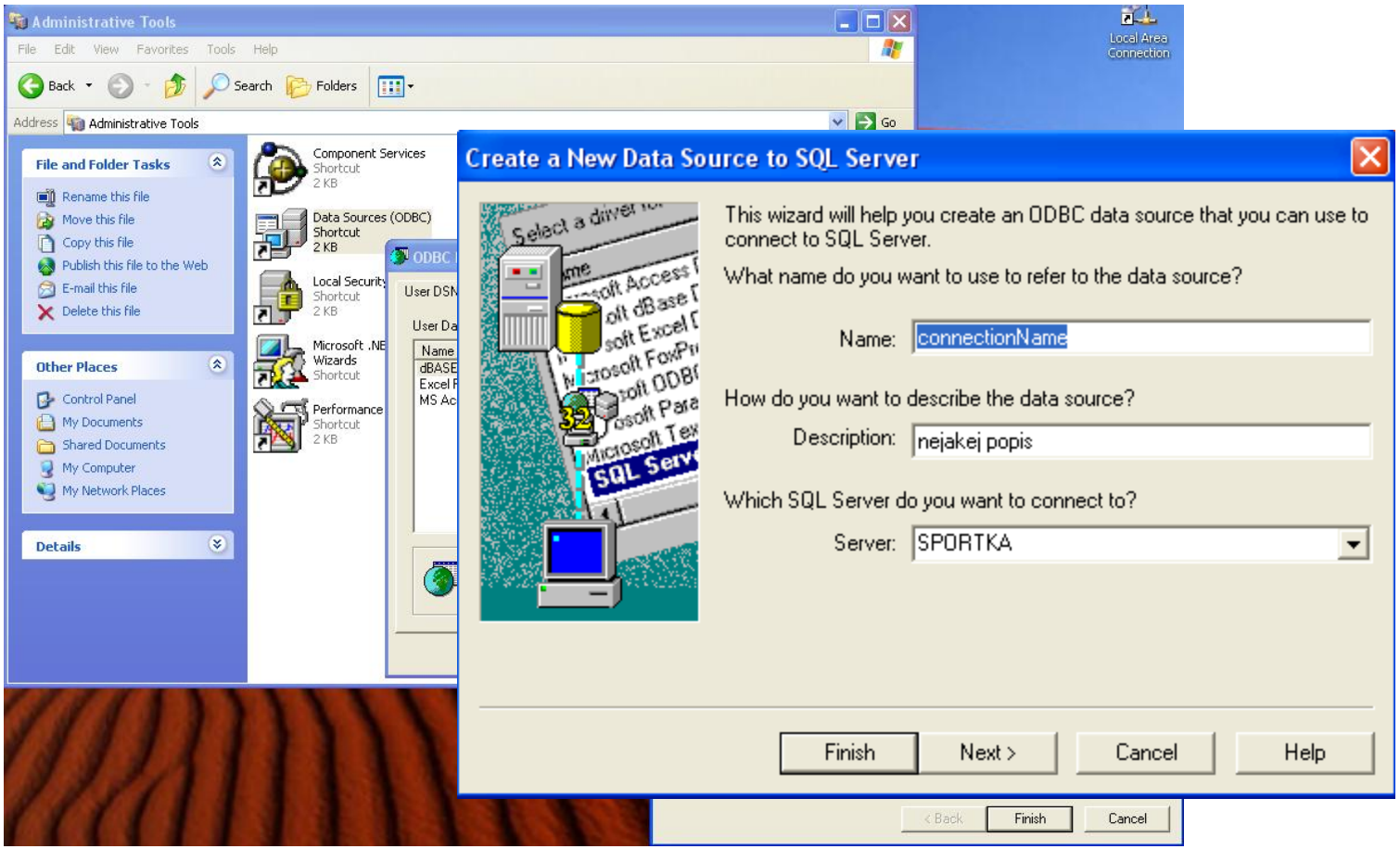
DDL - Triggery (příklad)

```
CREATE TRIGGER safety ON DATABASE
    FOR DROP_SYNONYM
AS
    RAISERROR ('You must disable Trigger
    "safety" to drop synonyms!',10, 1)
    ROLLBACK
```

Externí programování DB aplikace

- externí/samostatné aplikace (tj. mimo aplikační framework DBMS) mohou využívat standardizovaná rozhraní
  - ODBC (Open DataBase Connectivity)
    - 1992, Microsoft
  - JDBC (Java DataBase Connectivity)
    - přes ODBC (většinou), nebo nativní driver/protokol, síťový driver
  - knihovna ADO.NET (Active Data Objects .NET)
    - nad OLE DB, ODBC, případně přímo drivery MS SQL Server, Oracle
    - větší komfort, rychlost a spolehlivost (než ODBC)
- „polonativní“ databázové objektové programování pomocí objektově-relačního mapování mapování
  - Java Hibernate
  - totéž pro Microsoft .NET

ODBC, konfigurace ve Windows



ODBC, aplikace (C#)

```
using System.Data.Odbc;
OdbcConnection DbConnection = new OdbcConnection("DRIVER={SQL
      Server};SERVER=MyServer;Trusted_connection=yes;DATABASE=northwind; ");
DbConnection.Open();
OdbcCommand DbCommand = DbConnection.CreateCommand();
DbCommand.CommandText = "SELECT * FROM Employee";
OdbcDataReader DbReader = DbCommand.ExecuteReader();

int fCount = DbReader.FieldCount;

while( DbReader.Read()) {
      Console.Write( ":" );
      for (int i = 0; i < fCount; i++) {
            String col = DbReader.GetString(i); Console.Write(col + ":");
      }
      Console.WriteLine();
}

DbReader.Close(); DbCommand.Dispose(); DbConnection.Close();
```

JDBC, aplikace (Java)

```
Class.forName( "com.somejdbcvendor.TheirJdbcDriver" );

Connection conn = DriverManager.getConnection( "jdbc:somejdbcvendor:other data
    needed by some jdbc vendor", "myLogin", "myPassword" );

Statement stmt = conn.createStatement();

try {
    stmt.executeUpdate( "INSERT INTO MyTable( name ) VALUES ( 'my name' ) " );
    }
finally { stmt.close(); }
```

ADO.NET, aplikace (C#)

přes ODBC:

```
SqlConnection pripojeni = new
    SqlConnection("server=localhost;database=mojeDatabaze;uid=sa;pwd=");
```

přes OLE DB (sqloledb = SQL Server, msdaora = Oracle):

```
OleDbConnection pripojeni = new OleDbConnection
      ("provider=sqloledb;server=localhost;database="+"mojeDatabaze;uid=sa;pwd=");

pripojeni.Open();

SqlCommand command = new SqlCommand("SELECT * FROM tabulka", pripojeni);
command.ExecuteNonQuery();
```

Framework Java Hibernate

- poskytuje perzistenci klasickým Java objektům, tj. poskytuje „opravdové“ objektové databázové programování
- nutná definice mapování mezi objektem a jeho perzistentní verzí v DB (xml soubor pro každou třídu)
- zjednodušení: manager paměti organizuje objekty rovnou v databázi (+používá hlavní paměť jako cache, když se k objektu přistupuje)
- HQL (Hibernate query language)
  - objektový dotazovací jazyk
  - Hibernate překládá HQL do SQL

Ekvivalenty Java Hibernate pro Microsoft.NET Framework

- ADO.NET Entity Framework
- NHibernate
- Persistor.NET
- a další...

Relační vs. objektový přístup

- mapování „objektů do tabulek“ přináší režii, kterou uživatel nevidí (což může být dobře i špatně)
  - realiazce objektového DBMS pomocí relačního DBMS
- relační DBMS jsou vhodné pro datově intenzivní aplikace
  - objektové DBMS by byly neefektivní díky vytváření spousty malých objektů, se kterými se manipuluje uniformně, tj. není třeba je jednotlivě „zhmotňovat“ do objektů
- objektové DBMS jsou vhodné pro složité „Enterprise aplikace“, kde DB výkon není na prvním místě
  - relační DBMS zde poskytují nízkoúrovňový přístup k datům, tj. neodstíní programátora, který

Kurs Databázové aplikace

- DBI026
  - zaměření Oracle a MS SQL Server (alternativně)
    - embedded SQL, administrace
    - externí aplikace
    - indexování, optimalizace
    - transakce
    - zabezpečení
  - viz http://www.ms.mff.cuni.cz/~kopecky/vyuka/dbapl/