Vysoká škola polytechnická Jihlava

Databázové systémy 2

Studijní opora

Ing. Zbyněk Bureš. Ph.D.

Zbyněk Bureš

DATABÁZOVÉ SYSTÉMY 2

1. vydání
ISBN 978-80-87035-89-4

Vydala Vysoká škola polytechnická Jihlava, Tolstého 16, Jihlava, 2014

Tisk Ediční oddělení VŠPJ, Tolstého 16, Jihlava

Za jazykovou a věcnou správnost obsahu díla odpovídá autor. Text neprošel jazykovou ani redakční úpravou.

© Ing. Zbyněk Bureš, PhD., 2014

Úvod **6**

1 Opakování základů datové analýzy – E-R model ..... **- 7 -**

Motivace ..... *- 7 -*

E-R model ..... *- 7 -*

Identifikační klíč ..... *- 9 -*

Kardinalita (poměr) vztahu ..... *- 9 -*

Parcialita (členství) vztahu ..... *- 10 -*

Slabé entitní typy ..... *- 11 -*

Dekompozice M:N vztahu ..... *- 11 -*

Rekurzivní typ vztahu ..... *- 13 -*

ISA hierarchie ..... *- 14 -*

Speciální typy atributů ..... *- 15 -*

Výlučný typ vztahu ..... *- 15 -*

Častá chyba ..... *- 15 -*

2 Metodika logického návrhu DB v relačním prostředí ..... **- 18 -**

Logický návrh ..... *- 18 -*

ER model ..... *- 19 -*

Přístupy ke konceptuálnímu návrhu ..... *- 21 -*

Integrace pohledů ..... *- 22 -*

3 Transformace E-R schematu do RMD ..... **- 30 -**

Motivace ..... *- 30 -*

Silný entitní typ ..... *- 30 -*

Vztahový typ ..... *- 31 -*

Slabý entitní typ ..... *- 34 -*

Entitní podtyp (ISA hierarchie) ..... *- 34 -*

Výlučné vztahy ..... *- 35 -*

Normální forma výsledných relací ..... *- 36 -*

4 Modelování funkcí ..... **- 38 -**

Úvod ..... *- 38 -*

Co to je funkce ..... *- 39 -*

Hierarchie funkcí ..... *- 39 -*

Funkční analýza ..... *- 39 -*

Data flow diagram ..... *- 40 -*

5 Funkční závislosti ..... - 56 -

Motivace ..... *- 56 -*

Funkční závislost ..... *- 57 -*

Základní vlastnosti FZ ..... *- 58 -*

Armstrongova pravidla ..... *- 59 -*

Minimální pokrytí ..... *- 60 -*

Tranzitivní závislost na klíči ..... *- 61 -*

6 Normální formy relací ..... - 63 -

Úvod ..... *- 63 -*

První normální forma (1NF) ..... *- 63 -*

Druhá normální forma (2NF) ..... *- 64 -*

Třetí normální forma (3NF) ..... *- 64 -*

Boyce-Coddova norm. forma (BCNF) ..... *- 65 -*

Pokrytí závislostí a bezetrátovost ..... *- 66 -*

Návrh schematu databáze ..... *- 68 -*

7 Pokročilejší rysy jazyka SQL (1) ..... - 71 -

Datové typy PostgreSQL ..... *- 71 -*

Dostupné funkce a operátory ..... *- 75 -*

Podmíněné výrazy ..... *- 77 -*

Klauzule LIMIT a OFFSET ..... *- 78 -*

Pohledy ..... *- 78 -*

Dědičnost ..... *- 78 -*

Schemata ..... *- 79 -*

8 Pokročilejší rysy jazyka SQL (2) ..... - 83 -

Transakce ..... *- 83 -*

Indexy ..... *- 86 -*

Oprávnění ..... *- 88 -*

Uživatelské role ..... *- 90 -*

9 Procedurální rozšíření jazyka SQL ..... - 94 -

Procedurální rozšíření - motivace ..... - 94 -

Funkce v PostgreSQL ..... - 94 -

SQL funkce ..... - 97 -

Procedurální funkce ..... - 99 -

10 Kursory ..... - 110 -

Deklarace ..... - 110 -

Otevření kursoru ..... - 111 -

Použití kursoru ..... - 111 -

Další poznámky ..... - 114 -

11 Triggery ..... - 115 -

Triggerové funkce ..... - 117 -

Další poznámky ..... - 119 -

12 Datové sklady a OLAP ..... - 121 -

Business intelligence ..... - 121 -

Datové sklady a OLAP ..... - 121 -

Úvod

Předmět je rozšířeným úvodem do databázových systémů, probírá pokročilejší pojmy z databázových systémů. Studenti budou seznámeni s jazykem PLpg/SQL, naučí se programovat uložené procedury a funkce a používat triggery. Součástí předmětu je vytvoření netriviální databázové aplikace, vypracování průvodní dokumentace a její obhájení. Předmět přímo navazuje na předmět Databázové systémy 1. Předmět dále souvisí s předmětem Softwarové inženýrství.

Studijní opora je členěna do kapitol, v nichž se student postupně seznámí s celou problematikou. V kapitolách je obsažen jednak teoretický a praktický výklad, jednak řada řešených a neřešených příkladů, které umožní studentovi procvičit si zevrubně všechny potřebné dovednosti.

Student by měl zvládnout následující oblasti:

- Metodologie logického návrhu databáze v relačním prostředí
- Integrace pohledů
- Transakční zpracování, paralelismus
- Modelování funkcí
- Funkční závislosti, normální formy relací, minimální pokrytí
- Programování na serveru
- Procedurální rozšíření SQL
- Jazyk PLpg/SQL
- Uložené funkce, triggery

1 Opakování základů datové analýzy – E-R model

Cíl kapitoly

Kapitola opakuje pojmy související s konceptuálním modelem databáze, konkrétně s E-R modelem. Probírají se základní konstrukty E-R modelu a jejich význam při modelování. Cílem je zopakovat si:

- co to je konceptuální model
- použití E-R modelu pro databázové modelování
    - entity, vztahy, atributy
    - identifikační klíč
    - integritní omezení
    - kardinalita a parcialita vztahů
    - dekompozice M:N vztahu
    - další konstrukty E-R modelu

Klíčové pojmy

Konceptuální model, E-R model, entita, vztah, atribut, identifikační klíč, integritní omezení, kardinalita, parcialita – povinné resp. nepovinné členství ve vztahu, ISA hierarchie, výlučný vztah.

Motivace

Konceptuální model umožňuje popis dat v DB nezávisle na fyzické a logické implementaci, má co nejvěrněji vystihnout lidský konceptuální pohled na danou oblast. Konceptuální model představuje vlastně datovou analýzu

- modelování datové reality
- jaká budeme mít v informačním systému data
- pohled uživatele vs. pohled analytika

Konceptuální modely jsou různé, my budeme používat entitně-vztahový (**E-R model**, entity-relationship) model. E-R model obsahuje datové prvky (entity) a vztahy mezi nimi, je to v podstatě standard pro datové modelování. E-R model definuje **konceptuální schéma** databáze.

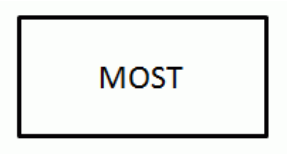
E-R model

- množina pojmů, umožňujících popis struktury DB na uživatelsky srozumitelné úrovni
- nejlépe se vyjadřuje graficky

Základní konstrukty

entita (entity)

- objekt reálného světa, který je schopen nezávislé existence a je jednoznačně identifikovatelný
- např. mostní objekt číslo 221-0608
- je obvykle odpovědí na otázku „co“, je vystižena podstatným jménem



vztah (relationship)

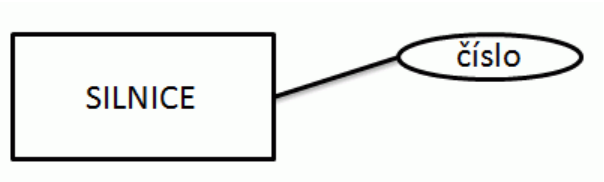
- vazba mezi dvěma či více entitami
- např. mostní objekt č. 221-0608 leží na silnici třetí třídy č. 221, tj. obecně entita „most“ je ve vztahu „leží na“ s entitou „silnice“
- často je vystižen slovesem

hodnota popisného typu (value)

- jednoduchý datový typ (tj. dvojice {množina hodnot, množina operací}), např. „celé číslo“ apod.

atribut (attribute)

- funkce přiřazující entitě či vztahu popisnou hodnotu, např. entita „SILNICE“ má atribut „číslo“



Postup modelování

Při modelování projektant na základě podrobného seznámení s modelovanou realitou

- identifikuje typy entit jako třídy objektů téhož typu, například „SILNICE“, „MOST“ atd.
- identifikuje typy vztahů, do nichž mohou entity vstupovat, např. MOST (entita) LEŽÍ NA (vztah) SILNICI (entita)
- přiřadí typům entit a typům vztahů vhodné atributy, které popisují jejich vlastnosti
- formuluje **integritní omezení**, vyjadřující soulad schématu s modelovanou realitou, např. MOST musí ležet na nějaké SILNICI; SILNICE je identifikována číslem

Měli bychom ještě upozornit na jistou nejednoznačnost v terminologii

- jako označení abstraktního objektu by měl být používán pojem „typ entity“: analogie třídy v OOP
- pojem „entita“ by měl být použit pro označení konkrétního výskytu objektu: analogie instance třídy v OOP

Identifikační klíč

Každá entita musí být jednoznačně identifikovatelná. Atribut nebo skupina atributů, jejichž hodnota slouží k identifikaci konkrétní entity se nazývá **identifikační klíč** (v E-R schematu značíme podtrženě). Příkladem může být atribut „rodné číslo“ entity ZAMĚSTNANEC.

Entitní typ může mít několik kandidátů na klíč, například ZAMĚSTNANEC může být identifikován

- rodným číslem
- číslem zaměstnance
- trojicí „jméno, příjmení, datum narození“

Volba klíče

- podle efektivity
- pokud má entita jednoduchý atribut, který ji může identifikovat, zvolíme právě tento atribut (např. „rodné číslo“ nebo „číslo faktury“ apod.), není třeba přidávat žádný umělý identifikátor
- klíčům složeným z více atributů se spíše vyhýbáme, raději zavedeme pomocný identifikátor

Kardinalita (poměr) vztahu

Kardinalita je integritní omezení pro vztahový typ. Existují tři druhy kardinality

- 1:1
- 1:N
- M:N

Kardinalita každého binárního vztahu je popsána dvěma tvrzeními, která vyjadřují zapojení každého ent. typu do vztahu. Mějme entitní typy KINO a FILM a vztahový typ HRAJE.

Poměr 1:1

1. dané kino dává maximálně jeden film
2. daný film je na programu maximálně v jednom kině

- může zahrnovat vztahy 0:1 a 1:0, tj. kino nic nedává nebo film se nikde nehraje, což je vlastně povinnost resp. nepovinnost členství (viz dále)

Poměr 1:N

1. dané kino může hrát více než jeden film
2. daný film se dává maximálně v jednom kině

- může zahrnovat vztahy 0:1, 1:0, 1:1
- důležitý je směr („jedno kino – víc filmů“ versus „jeden film – víc kin“)

Poměr M:N

1. dané kino může hrát více než jeden film
2. daný film se může dávat ve více než v jednom kině

- může zahrnovat vztahy 0:1, 1:0, 1:1, 1:N, N:1 – některé z nich mohou být vyloučeny přísnějšími pravidly

Kardinalita také odpovídá tvrzení, že jedna entita jednoznačně určuje druhou entitu, resp. je determinantem entity druhého typu:

- pro vztah 1:1 lze např. říct, že název kina determinuje název filmu a také název filmu determinuje název kina
- pro vztah 1:N lze např. říct, že název filmu determinuje název kina, ale název kina nedeterminuje název filmu (tj. známe-li název kina, nemůžeme jednoznačně říct, jaký film se v něm hraje, neboť jich může být více)
- u vztahu M:N není determinující ani jedna entita

Parcialita (členství) vztahu

Účastní-li se entita vztahu, říkáme, že je členem vztahu. Někdy má entita předepsanou účast ve vztahu, tj. členství je povinné, např. zaměstnanec musí být zaměstnán na nějakém oddělení. Jindy se entita do vztahu zapojovat nemusí, členství je nepovinné, např. oddělení může existovat i bez zaměstnanců. Povinné členství ve vztahu je velmi důležité integritní omezení: vyjadřuje, že entita **nemůže existovat** bez zapojení do vztahu s druhou entitou.

V grafickém konceptuálním modelu se parcialita značí různě, například

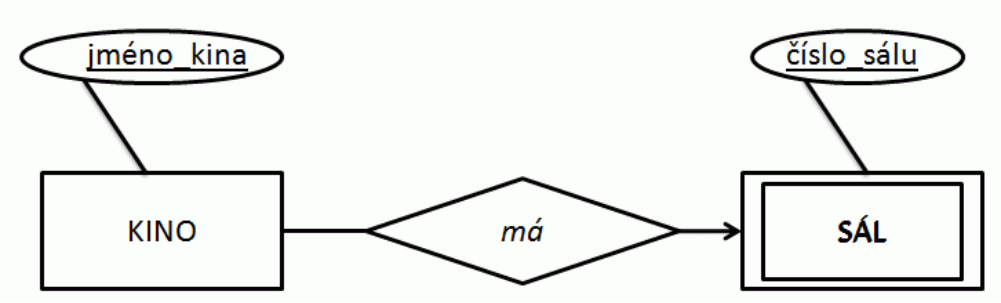
- v kině se může dávat víc filmů anebo žádný
- film se musí dávat právě v jednom kině

Slabé entitní typy

Součástí klíče některých entit nemusí být pouze jejich vlastní atributy. **Slabá entita** je taková entita, která není jednoznačně identifikovatelná pouze pomocí svých atributů - mohou existovat instance, které mají stejné hodnoty atributů. Např. identifikace kinosálu je možná pouze ve spojení s identifikací multikina, v němž se nalézá:

- kino – identifikační vlastník
- kinosál – slabý entitní typ

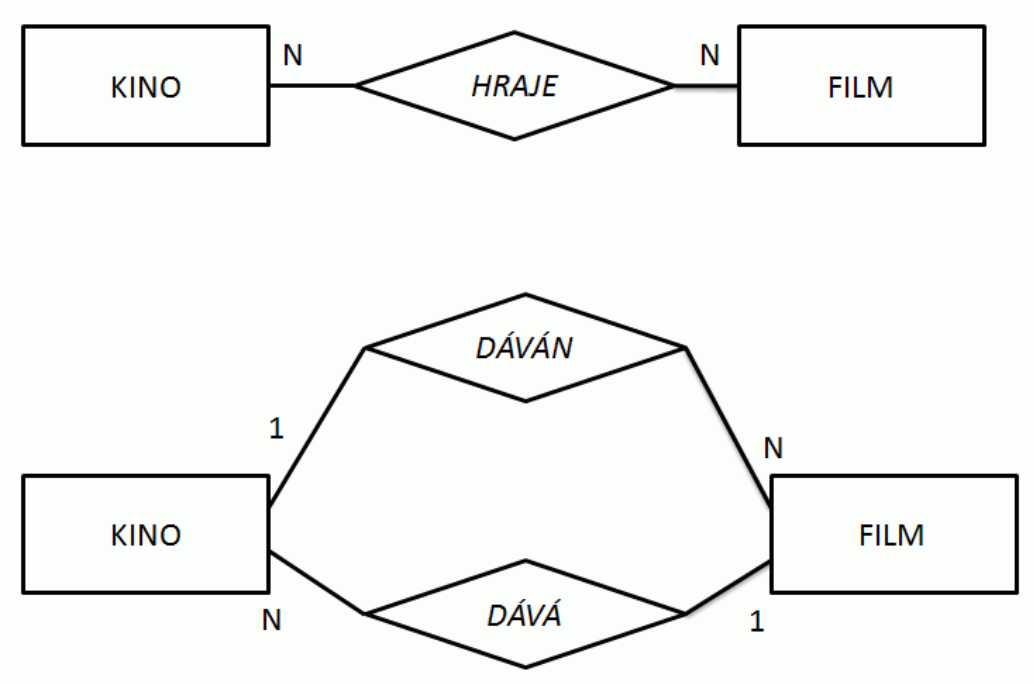
Vztah kino-kinosál se pak nazývá identifikační vztah. Slabý entitní typ má vždy povinné členství v identifikačním vztahu, jinak by nemohl existovat. Výše uvedený příklad vyjadřuje stav, kdy v databázi může existovat více sálů s týmž číslem, které se však nalézají v různých kinech. Identifikační klíč sálu je pak dvojice (jméno_kina, číslo_sálu), vlastní atribut „číslo_sálu“ je jen částečný klíč.



Dekompozice M:N vztahu

Návrh konceptuálního schematu je sice nezávislý na logickém modelu, nicméně SŘBD obvykle neumějí reprezentovat vztahy M:N přímo. Vztahy M:N je určitě vhodné používat při konceptuálním modelování, avšak musíme být schopni je následně rozdělit do dvou vztahů typu 1:N – tzv. dekompozice.

Chybná dekompozice



Proč je tato dekompozice chybná:

- původní schema říká, že ani v jednom směru neexistuje funkční závislost mezi účastníky vztahu, ani jedna entita není determinantem vztahu

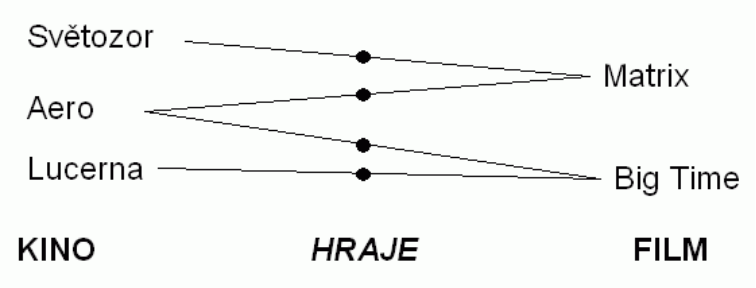
Nové schema vyjadřuje, že

- ve vztahu DÁVÁ je entita KINO determinantem vztahu, určuje tedy jednoznačně promítaný film
- ve vztahu DÁVÁN je entita FILM determinantem vztahu, určuje tedy jednoznačně kino, kde se hraje

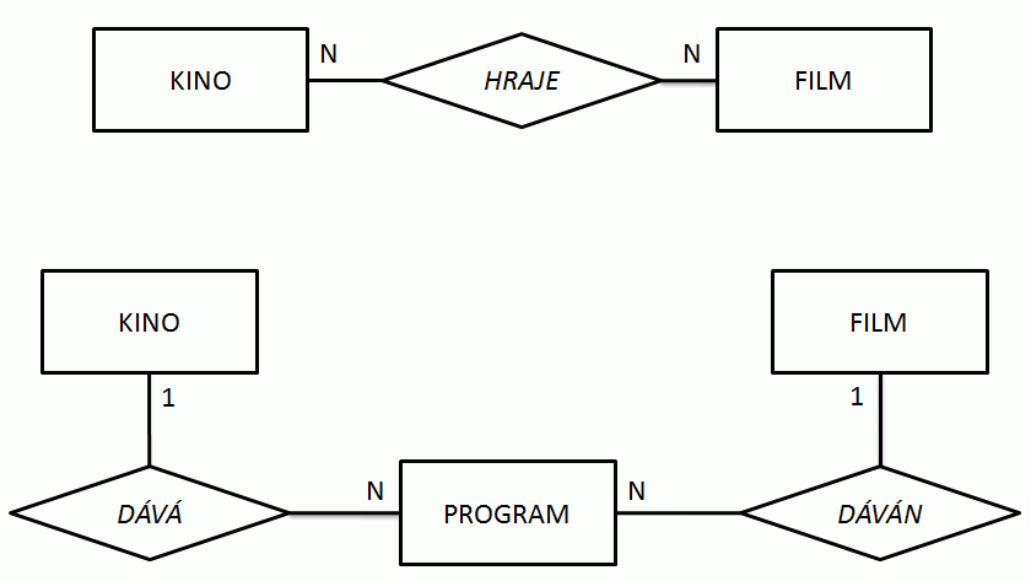
Výše uvedené podmínky jsou ovšem současně splněny pouze pro vztah 1:1, což je jen podmnožina vztahu M:N.

Správná dekompozice M:N vztahu

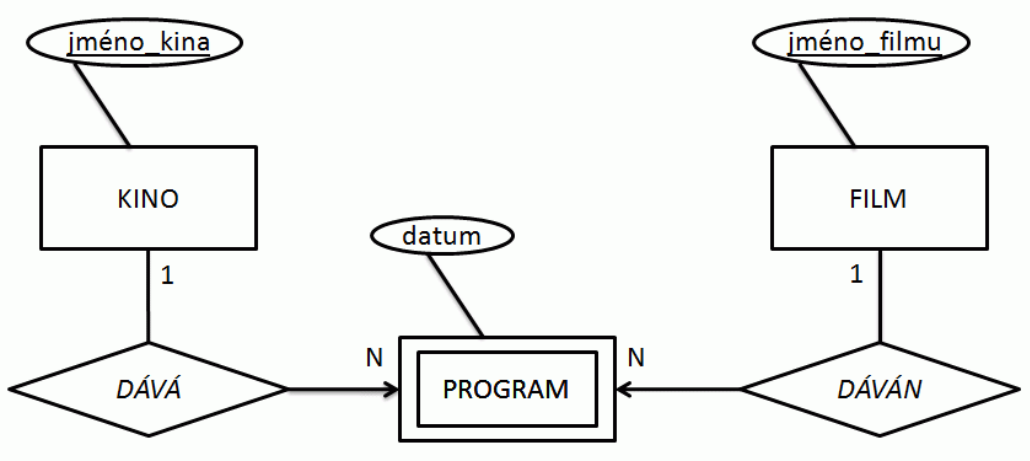
Podívejme se na diagram výskytů



Každý výskyt vztahu odpovídá promítání konkrétního filmu v konkrétním kině. Budeme-li nyní považovat výskyt vztahu za výskyt entitního typu, máme vyhráno. Vztah lze v podstatě vždy považovat za entitu, vzniká tzv. průnikový entitní typ.



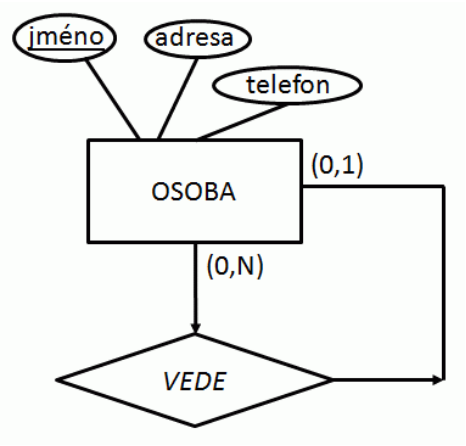
Lze též využít identifikační závislosti



- entita PROGRAM má identifikační klíč (jméno_kina,jméno_filmu)
- pokud není atribut „datum“ klíčový, kino nemůže promítat jeden film v různých dnech

Rekurzivní typ vztahu

Popisuje situaci, kdy typ entity vstupuje do vztahu sám se sebou. Vzniká tak stromová hierarchie – osoba může vést několik osob, ale je vedena pouze jednou osobou. Aby se odlišila úloha entitního typu na jedné a druhé straně vztahu, je vhodné zavést tzv. role (vede, je_veden).

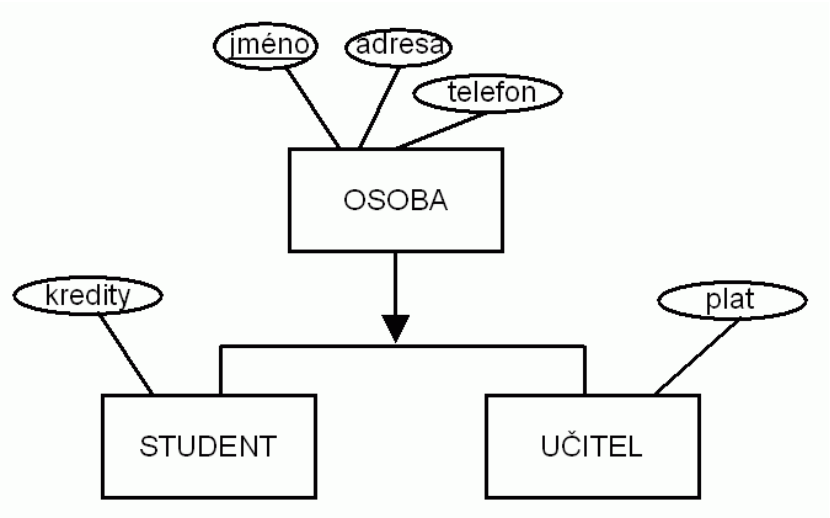


ISA hierarchie

ISA hierarchie je v podstatně analogie dědičnosti v objektově orientovaném programování, např. OSOBA může být buď STUDENT nebo UČITEL – entitní nadtyp má entitní podtypy. Atributy nadtypu se dědí, k nim může mít každý podtyp sadu vlastních atributů.

- STUDENT IS A OSOBA, UČITEL IS A OSOBA
- společné atributy: jméno, RČ, adresa
- atributy vázané pouze na podtyp
    - STUDENT: počet kreditů, ...
    - UČITEL: platová třída, ...

ISA vztah je reflexivní a tranzitivní, zavádí do schematu obvykle stromové struktury.



Důležité je si uvědomit, že entitní podtypy musí tvořit úplné pokrytí entitního nadtypu, tj. v našem případě každá osoba je buď student nebo učitel, ale nemůže být nic jiného. Naopak, platí, že prvek entitního nadtypu může patřit pouze do jednoho ent. podtypu, tj. entitní podtypy jsou vzájemně disjunktní. Tedy když je nějaká osoba studentem, nemůže být zároveň učitelem a naopak. Příslušnost k entitním podtypům je tedy výlučná.

Speciální typy atributů

Konceptuální model se nemusí omezovat pouze na atomické atributy (logický model se na ně obvykle již omezuje). Můžeme zavést speciální typy atributů: skupinový nebo vícehodnotový (multiatribut).

Skupinový atribut

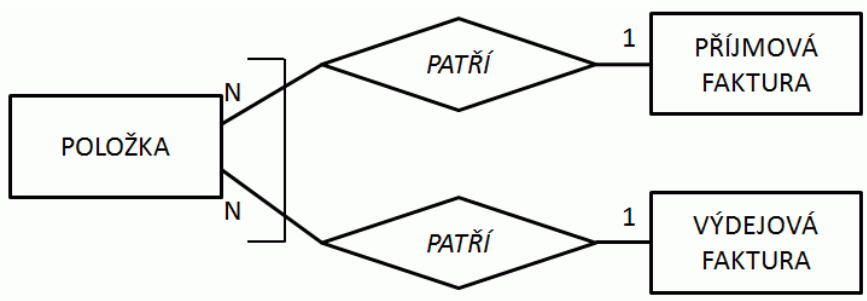
- např. adresa (ulice, číslo, město, PSČ)
- heterogenní strukturovaný datový typ, podobný např. datovému typu *struct* v jazyce C
- může tvořit hierarchie
- užitečný, přistupujeme-li někdy ke složkám a někdy k celku

Vícehodnotový atribut

- např. atribut telefon ent. typu ZAMĚSTNANEC
- homogenní datový typ s více prvky stejného typu, podobný např. poli
- předem nevíme, kolik prvků bude mít

Výlučný typ vztahu

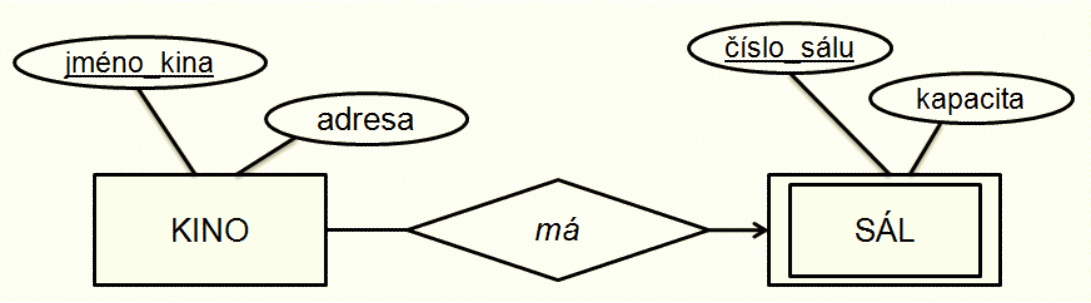
Výlučný typ vztahu se může uplatňovat u vztahů 1:N. Značí, že entitní typ vstupuje právě do jednoho z N vztahů. Například v následujícím fragmentu E-R schematu každá položka patří buď do příjmové faktury nebo do výdejové faktury, nikdy ale do obou. Zároveň každá položka musí být zapojena do některého z uvedených vztahů, neboť členství ve výlučném vztahu má tomto případě povinné.



Častá chyba

Záměna pojmů konceptuálního a logického modelu:

- entitě v *ER modelu* přiřadíme atribut, který slouží k reprezentaci vztahu v *relačním* modelu:

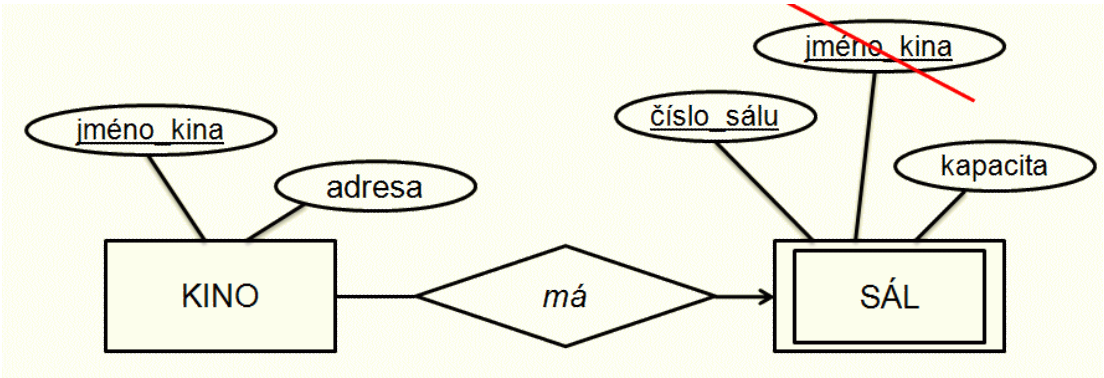


Toto schema bude tranformováno do dvou relací:

KINO(jméno_kina, adresa)

SÁL(číslo_sálu, jméno_kina, kapacita)

To ovšem neznamená, že by ER diagram měl vypadat takto:



- atribut „jméno_kina“ v entitě SÁL je na konceptuální úrovni nesmyslný
- to, že daný sál patří do daného kina je **jednoznačně řečeno vztahem má**

Kontrolní otázky

- Co to je silný a slabý entitní typ?
- Může být entitní typ zapojen do více vztahů?
- Může být vztah zapojen do dalších vztahů?
- Co to je integritní omezení?
- Co to je ISA hierarchie?
- Co to je výlučný typ vztahu?

Úlohy k procvičení

1. Vytvořte E-R model databáze pilotních průkazů. Průkaz má atributy platnost a ev. číslo. Dělí se na: ultralehké, paragliding, motorový paragliding, motorové rogalo, rogalo. Paragliding se dělí na A (std), B (competition), C (profi). Letec může mít víc průkazů (každý na jiný typ letadla), má obvyklé lidské atributy, důležitý je věk, každé dva roky musí být prohlídka u lékaře. Ale pozor, u

paraglidingu je prohlídka do 60ti let věku jen jedna a po 60tém roce věku 1x za rok. Dále máme seznam strojů (revize 1x za rok, ev. číslo, pojištění povinné), pilot má 1 a více strojů.

2 Metodika logického návrhu DB v relačním prostředí

Cíl kapitoly

Kapitola seznamuje čtenáře s metodologií návrhu databáze v relačním prostředí. Opakuje základní kroky návrhu databáze a probírá různé strategie datové analýzy. Dále se zabývá integrací schemat konceptuálního modelu. Cílem je naučit se:

- jaké jsou kroky logického návrhu databáze
- jaké jsou kroky konceptuálního modelování
- jaká jsou kritéria kvality E-R modelu
- strategie konceptuálního modelování
- integrovat konceptuální schemata

Klíčové pojmy

E-R model, strategie shora dolů, strategie zdola nahoru, strategie zevnitř ven, smíšená strategie, integrace pohledů.

Logický návrh

Logický návrh datové části informačního systému se dá rozdělit do tří fází:

1) Analýza požadavků, konceptuální modelování
2) Relační modelování, hrubý návrh tabulek
3) Detailní specifikace tabulek (datové typy, integritní omezení), verifikace návrhu

1. fáze

Identifikace požadavků na IS:

- formulace a shromažďování uživatelských požadavků
- identifikace dat organizace
- objevujeme základní entity

ER modelování

- první krok (iterativně)
    - definice nezávislých entitních typů
    - definice typů vztahů
    - definice typů závislých entit
- druhý krok
    - nalezení atributů
    - určení identifikačních klíčů

2. fáze

Tato fáze zahrnuje převod ER modelu do RMD. Transformace je částečně algoritmizovatelná, je nutná kontrola normálních forem. Dále je proveden návrh tabulek:

- hrubý návrh
- tabulky s atributy, primárními a cizími klíči

3. fáze

V této fázi probíhá detailní návrh tabulek. Specifikujeme

- datové typy
- integritní omezení (IO)

Některá IO ovšem nemusí být přímo podporovaná daným SŘBD, je třeba je ošetřit na úrovni aplikace, popř. jako uložené procedury a triggery.

ER model

Entity a atributy

Nejprve nalezneme entitní typy, popř. jejich kandidátní klíče. Dále identifikujeme atributy. Atributy slouží ke kvalifikaci, identifikaci, klasifikaci a k vyjádření stavu entity. Vícehodnotové atributy obvykle eliminujeme. Atributy je vhodné pojmenovávat tak, aby bylo pokud možno jasné, k jaké entitě náleží, tj. nikoli „číslo“, ale např. „číslo_zaměstnance“. Po transformaci modelu do tabulek v SQL to značně zvýší čitelnost kódu.

Samostatnou entitu pravděpodobně zavedeme, pokud u ní identifikujeme více atributů než jeden. Naopak, říká se, že pokud má entita víc než 8 atributů, je to známka toho, že nám v modelu chybí entity či vztahy.

V této fázi modelování je také třeba vyjasnit rozdíly mezi entitami a vztahy na jedné straně a atributy na straně druhé. Jeden jev lze často modelovat více způsoby, tomu se říká sémantický relativismus. Už zde je možné provádět prvotní kontrolu normálních forem, např. testovat, zda nově identifikovaný atribut závisí pouze na celém klíči apod.

Vztahy

U vztahů je potřeba určit název, kardinalitu, parcialitu, popř. atributy. Je vhodné (a často CASE-podporováno) testovat návrh na chybějící či redundantní vztahy:

- pokud entita není zapojena do vztahu, něco je divně
- redundantní vztah vznikne například tranzitivitou (ISA hierarchie)

Doporučená pravidla pro kreslení E-R diagramu

- vztahy kreslit vodorovně, číst zleva doprava
- u vztahů 1:N dávat N nalevo

- minimalizovat šikmé čáry a křížení
- nekreslit čáry těsně u sebe
- používat podmnožiny ER diagramů
- označit každý diagram jménem a datem

Kontrola E-R diagramu

Entity

- je jméno entity smysluplné a je v jednotném čísle?
- jak je to s výlučností?
- jsou dány alespoň dva atributy?
- není zvláštní, že existuje více než 8 atributů?
- co homonyma a synonyma v názvech typů entit?
- je definice typu entity úplná (nezapomněli jsme na něco)?
- jsou známy skutečné četnosti entit daného typu (tj. kolik řádků bude asi mít výsledná tabulka)?
- je dán klíč entity?
- existuje alespoň jeden typ vztahu, ve kterém je typ entity členem?
- existuje alespoň jeden proces, který používá danou entitu?
- mění se entita v čase (a je třeba evidovat historii)?
- vyhovuje definice atributů normalizaci?
- není typ entity příliš generický (obecný)?
- je typ entity generický postačujícím způsobem (tj. není přílš specializovaný)?

Entitní podtypy

- jsou podtypy vzájemně výlučné?
- mají podtypy vlastní atributy a/nebo participují ve vztazích?
- přijímají identifikátor nadřazeného typu entity?
- je množina podtypů úplná (tj. pokrývá úplně entitní nadtyp)?
- známe atributy a/nebo vztahy a podmínky, které odlišují podtyp entity od jiného typu?

Atributy

- má atribut jméno v jednotném čísle?
- je atribut jednoduchý?
- je dána definice formátu, délky, domény, atd.?
- nejde ve skutečnosti o chybějící typ entity či vztahu (viz sémantický relativismus)?
- není atribut replikován odjinud (např. z nadtypu)?
- není důležité znát jeho hodnoty v čase (tj. evidovat jeho historii)?
- závisí hodnota atributu pouze na entitě?
- je-li hodnota atributu povinná, je vždy známá?
- je hodnota atributu funkčně závislá na části klíče nebo na atributech, které nejsou částí klíče?

Vztahy

- je každý konec spojnice vztahu pojmenovaný?
- má typ vztahu právě dva konce (spojnice)?
- řekneme-li větu popisující vztah "inverzně", je vztah stále korektní?
- má každý člen typu vztahu kardinalitu a typ členství?
- nejde o zřídka se vyskytující konstrukce?
- není typ vztahu redundantní?
- vycházejí konce výlučných vztahů ze stejné entity?
- má tento typ entity ve výlučných typech vztahů vždy stejný typ členství?
- je vztah jen v jedné skupině výlučných vztahů?

Přístupy ke konceptuálnímu návrhu

Konstrukce větších SW celků je mentálně obtížně zvládnutelný problém. Často máme příliš mnoho entit a ještě více vztahů, je tedy nutná týmová práce. Neexistuje pak ovšem nikdo, kdo by měl v hlavě celý model. Pro usnadnění a zpřehlednění práce se používají různé strategie návrhu:

- shora dolů
- zdola nahoru
- zevnitř ven
- smíšená

Strategie shora dolů

Tato strategie začíná nejprve globálním pohledem na doménu aplikace. Poté dochází k postupnému zjemňování pojmů a tvorbě stále podrobnějších ER diagramů. Každý ER diagram tak popisuje celou doménu, ale v různém detailu.

Tato strategie je mentálně velmi náročná, vyžaduje vysokou schopnost abstrakce, nesmí se na nic zapomenout, což je složité zvlášť u rozsáhlých systémů. Kromě toho, chyba v počátku analýzy se obtížně odstraňuje, neboť se propaguje stále do dalších etap návrhu.

Strategie zdola nahoru

U této strategie vyjdeme z jednotlivých atributů, tedy od objektů na nejnižší úrovni, které postupně seskupujeme a vytváříme z nich entitní typy. Dalším postupem je integrace dílčích částí do vyšších celků za použití generalizace a jejich zapojení do vztahů. Často vznikají ISA hierarchie.

Tento postup je jednoduchý, ale může vyžadovat častou restrukturalizaci modelu. Na začátku může být množina dat nepřehledná a řada pojmů se objeví až na konci.

Strategie zevnitř ven

Na počátku stojí jeden jasný a dobře definovaný objekt. Pak hledáme, do jakých vztahů a s jakými dalšími objekty tento prvotní objekt vstupuje. Od jakéhosi „krystalizačního jádra“ tak postupujeme sítí vztahů až k nejvzdálenějším entitám.

Nevýhoda metody: pohybujeme se stále na jedné úrovni abstrakce, tuto úroveň je nutno na začátku dobře odhadnout.

Strategie smíšená

Tato strategie je kombinací strategií zdola nahoru a shora dolů. Nejprve se problém základně rozdělí na podproblémy (skeletální schema), které se následně řeší separátně. Dá se použít libovolná strategie pro řešení podproblémů, vyvstává ale nutnost integrovat dílčí schémata.

Poznámky

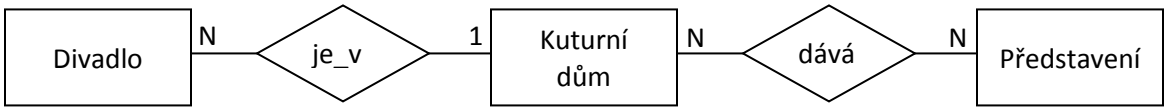
Strategie návrhu obecně nevedou k témuž výsledku, protože dokonce ani aplikace té samé strategie různými týmy nemusí vést ke stejnému výsledku. Výsledek záleží na abstrakci, na schopnostech, na úhlu pohledu.

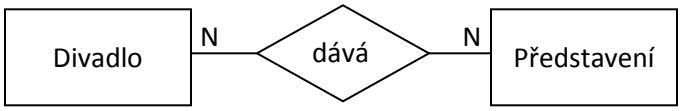
Integrace pohledů

Cíl této činnosti je vytvořit z více dílčích schemat jedno. Problém je, že schémata se mohou překrývat, některé části se ve schematech opakují, a je nutné unifikovat reprezentaci těchto částí. Vznikají různé konflikty, které dělíme na strukturální a konflikty ve jménech.

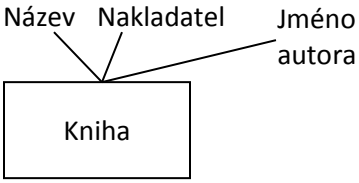
Strukturální konflikt

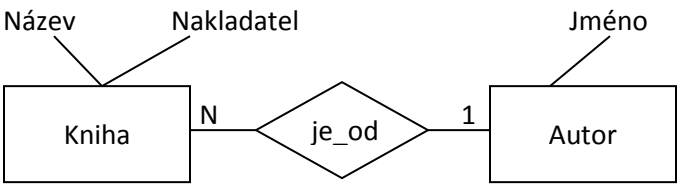
Tento konflikt může mít několik příčin. První z nich je odlišná úroveň abstrakce:



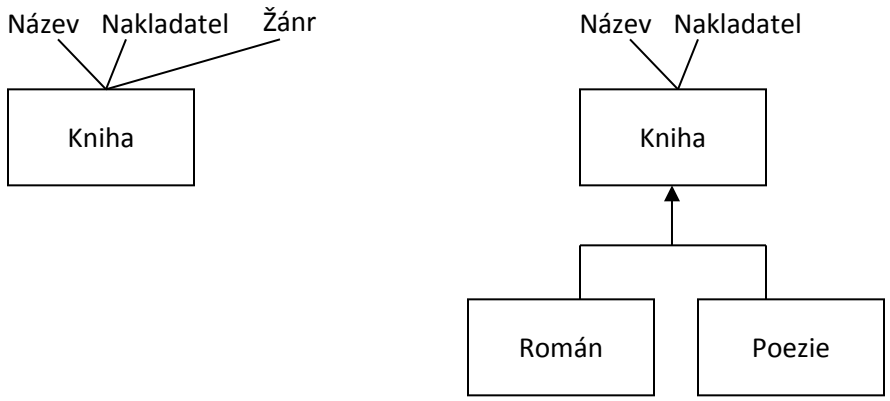


Další příčinou je sémantický relativismus, záměna atributu za vztah:



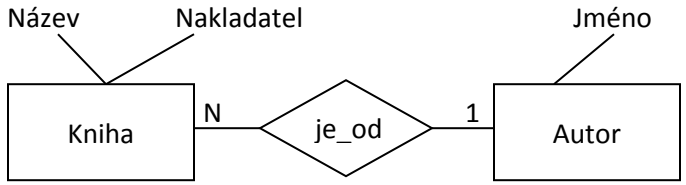


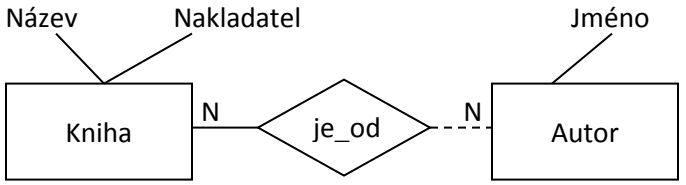
Jiný strukturální konflikt je různý pohled na hierarchie:



Další strukturální konflikt vyplývá z nekompatibilní specifikace vztahů:

- rozdílné kardinality
- rozdílné parciality





Řešení strukturálních konfliktů: obvykle se dává přednost obecnější variantě. Pokud se dva analytici koukají na tentýž problém různě, značí to určitou nejednoznačnost zadání, vybereme proto to řešení, které je bližší skutečné realitě.

Konflikt jmen

Synonyma

- dva analytici pojmenují stejnou věc různě
- dá se celkem snadno odhalit

Homonyma

- dva analytici pojmenují různé věci stejně
- to je horší, na to se přichází obtížněji

Metoda integrace pohledů

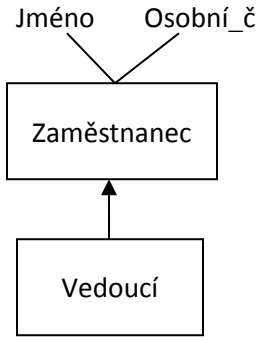
Pohledy integrujeme obvykle po dvojicích a tvoříme větší celek, nebo alternativně k celku přidáváme další a další dílčí schémata. Při integraci záleží na použité strategii návrhu, u smíšené strategie se začíná skeletálním schématem.

Důležité: při integraci nestačí ER diagram, je nutný verbální popis všech použitých pojmů, jinak není možné integraci provádět!

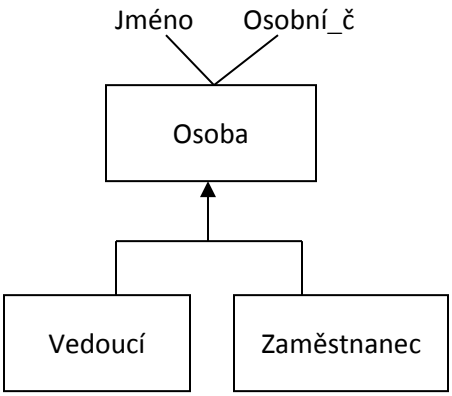
Příklad – konflikt jmen entit

- nejjednodušší řešení
    - konflikt je sice vyřešen, avšak je ignorován zřejmý rozdíl v úrovni abstrakce: vedoucí je patrně podtypem zaměstnance

- lepší řešení



- ještě obecnější řešení



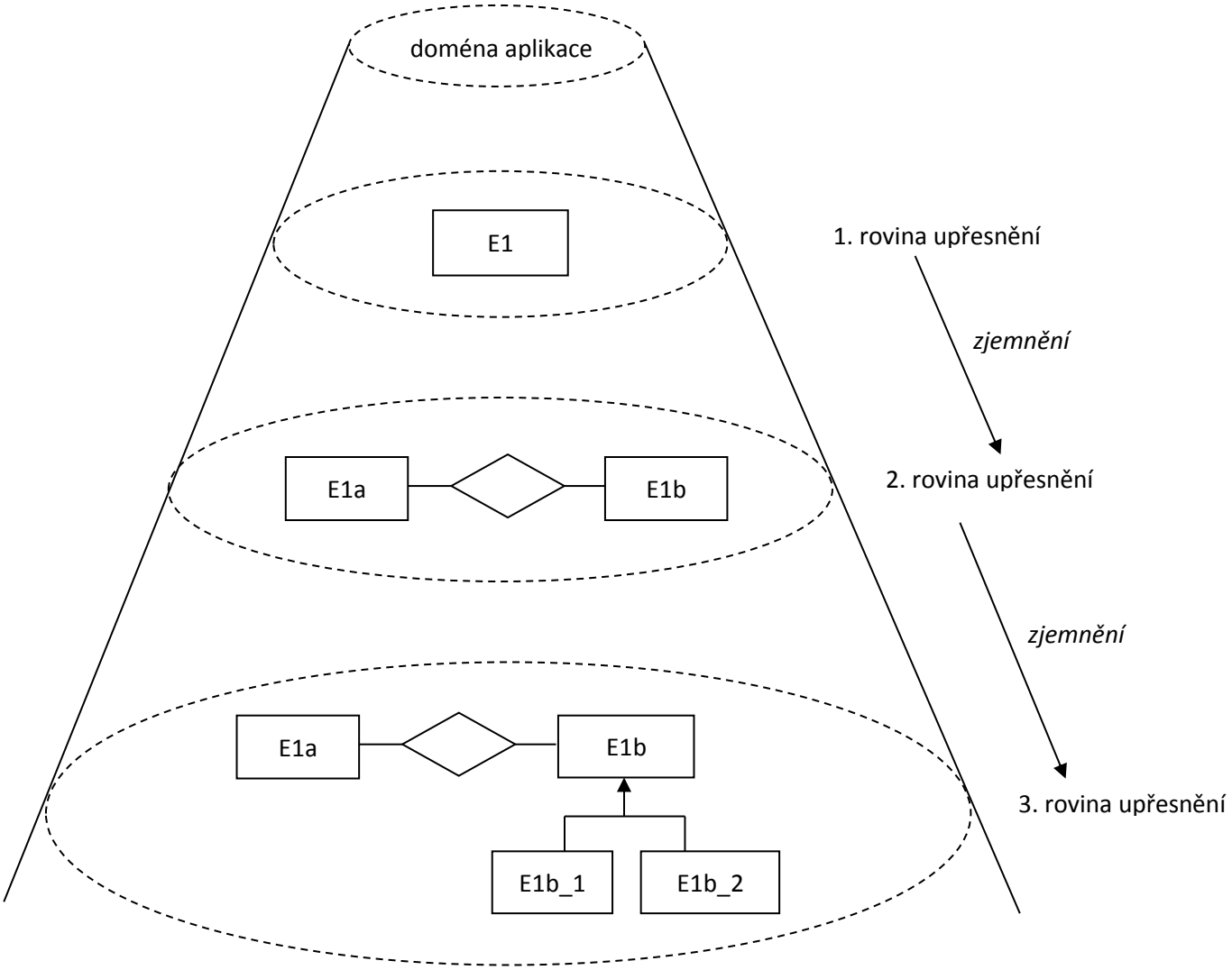
Kontrolní otázky

- Jaké jsou kroky logického návrhu databáze?
- K čemu slouží atributy?
- Jaké existují strategie návrhu databáze?
- K čemu slouží integrace pohledů?
- Co to je strukturální konflikt a jak se řeší?

Řešené příklady

Mějme databázi obchodního rejstříku.

<u>Ukázka strategie návrhu shora dolů:</u>



….. a dále zjemňujeme …..

1) Nejprve máme jedinou entitu: *obchodní rejstřík*.
2) Pak se zjistí, že jde o sledování vztahů mezi údaji o *firmách* a *osobách* a *místech*.
3) Pak se ujasňují typy vztahů (firma_sídlí, osoba_narozena, osoba_bydlí),
4) Pak může proběhnout rozdělení osob na *ředitele* a *jednatele*, rozdělení firem na *SRO* a *AS*.
5) Nakonec proběhne identifikace atributů a konečné upřesnění entit.

Strategie shora dolů představuje rozdělení problému na podproblémy. Chyba na začátku se propaguje do dalších etap vývoje. Můžeme říci, že „vidíme les a nevidíme stromy“. Při zjemňování nerozšiřujeme namodelovanou oblast (celá problémová doména je pokryta již v první úrovni přiblížení pomocí univerzální entity), pouze zvyšujeme úroveň detailu.

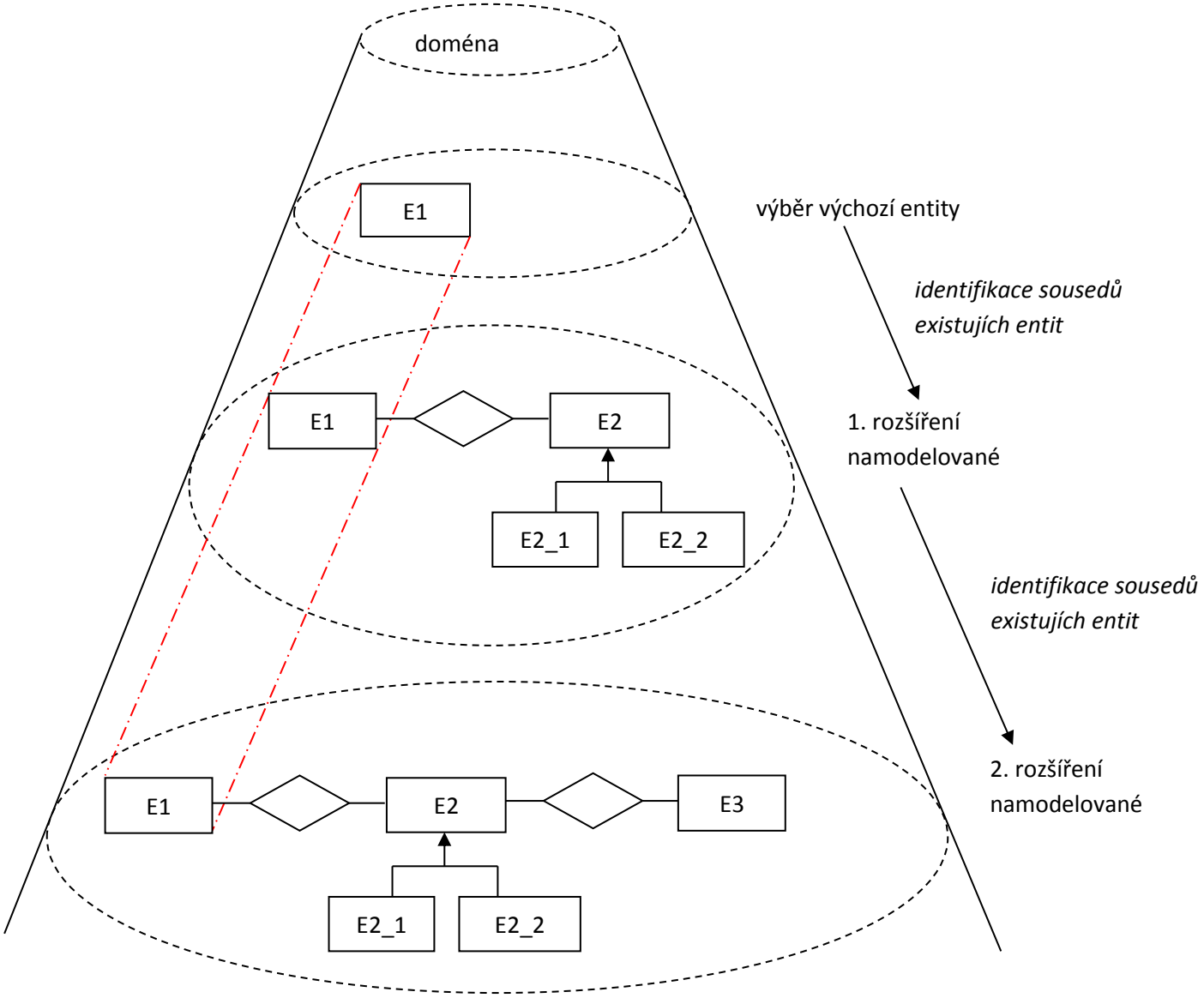
<u>Ukázka strategie návrhu zdola nahoru</u>

1) Vyjdeme z objektů na nejnižší úrovni, tj. z atributů:
   - jméno_ředitele, jméno_jednatele, název_sídla, ulice, číslo_popisné, název_sro, název_as, …

2) Dále zkonstruujeme z atributů entity.
3) Dále zkonstruujeme hierarchie entit.
4) Nakonec proběhne speicifikace vztahů, kardinality,…

Na počátku této strategie je nepřehledné množství dat, obvykle musíme provádět časté restrukturalizace. Můžeme říci, že „vidíme stromy a nevidíme les“. Stejně jako u strategie shora dolů zde nerozšiřujeme namodelovanou oblast. Celá problémová doména je pokryta již v první fázi modelování, kde již máme identifikovány všechny elementární atributy. Během modelování pouze konstruujeme entity, seskupujeme je do hierarchií a svazujeme je vztahy.

Ukázka strategie návrhu zevnitř ven



…… a dále rozšiřujeme …..

1) Najdeme jasné a konečné pojmy: FIRMA
2) Najdeme vztahy pojmů, které už máme: FIRMA je ve vztahu s OSOBAMI
3) … a tak pořád dokola, dokud to není hotové

Pohybujeme se stále na téže úrovni abstrakce, tj. entita OSOBA se v průběhu modelování již nemění, úroveň detailu modelu se nemění. Jinými slovy, to, co už máme, se ponechá, pouze odkrýváme další a další části modelu. Tedy na rozdíl od strategií shora dolů a zdola nahoru zde postupně rozšiřujeme namodelovanou oblast.

Úlohy k procvičení

1. Použijte zadání databáze z předchozí kapitoly (databáze pilotních průkazů). Vytvořte E-R model různými strategiemi.

---

2. Vytvořte evidenci fotbalových zápasů. Uvažujte ligu, hřiště, hráče apod., vše, co je podle vás potřeba modelovat v daném problému. Domluvte se se spolužákem, aby vytvořil svou variantu E-R modelu. Oba dva modely posléze integrujte, tj. vyřešte všechny konflikty a vytvořte jediný E-R model.

3 Transformace E-R schematu do RMD

Cíl kapitoly

Kapitola je zčásti opakováním látky z předmětu Databázové systémy 1. Poskytuje propojení mezi konceptuálním E-R modelem a relačním modelem dat. Cílem je naučit se převést libovolný E-R model do relačního modelu a to pokud možno optimálně vzhledem k plánovanému využití a se zobrazením všech integritních omezení.

Klíčové pojmy

Silný a slabý entitní typ, determinant vztahu, identifikační závislost, cizí klíč, ISA hierarchie.

Motivace

Výsledkem konceptuálního modelování je E-R model, který může být dost vzdálený logickému modelu, je nutné převést E-R model do relačního modelu. Transformace zajišťuje reprezentaci základních konstruktů bez ohledu na další vlastnosti (např. NF). RMD reprezentuje entitní a vztahové typy stejně – schematem relace, rekonstrukce E-R modelu z RMD tudíž není intuitivní.

Silný entitní typ

- přímočaré – silnému ent. typu odpovídá schema relace se stejnou množinou atributů
- primární klíč odpovídá identifikačnímu klíči entit. typu
- popisným typům atributů se přiřadí domény
- cíl – normalizovaná tabulka
    - závisí na citu analytika a množství funkčních závislostí (FZ), které uvažujeme
    - mělo by platit, že jediné netriviální funkční závislosti odvoditelné z ent. typu jsou závislosti atributů na identifikačním klíči, odhalení dalších FZ může indikovat další ent. typ

Možnosti transformace vícehodnotových atributů

- některé SŘBD je přímo umožňují
- jestliže víme max. počet výskytů atributu, pak pro ně „rezervujeme“ místo v relaci, nevyužité budou mít například hodnotu NULL
    - zabírá místo
- zavedeme další relaci, odpovídající vícehodnotovému atributu

Možnosti transformace skupinových atributů

- některé SŘBD je přímo umožňují
- jinak je nutno oželet hierarchickou strukturu atributu, uložíme pouze atomické složky a opět využijeme nové relace

Vztahový typ

Vztah 1:1

Mějme služební vozidla v podniku

- žádný vůz není využíván více zaměstnanci
- žádný zaměstnanec nevyužívá víc než jeden vůz

Reprezentace zde závisí na tom, zda je členství ve vztahu povinné či nikoli (parcialita).

Povinné členství pro oba ent. typy

- každý zaměstnec má právě jedno auto
- každé auto je přiděleno právě jednomu zaměstnanci

- **atributy obou entitních typů zařadíme do jediné relace** – slévání, přilepení atributů, vztah reprezentován implicitně
- klíčem bude buď č_osoby nebo SPZ

OSOBA(č_osoby, ... , SPZ, ...)

Povinné členství pouze pro jeden ent. typ

- každé auto je přiděleno právě jednomu zaměstnanci
- každý zaměstnec má žádné nebo jedno auto

- **dvě relace (VŮZ a OSOBA)**, do relace VŮZ přidáme atribut č_osoby
- klíčem by mohlo být i č_osoby

OSOBA(č_osoby, ...)
VŮZ(SPZ, ..., č_osoby)

Nepovinné členství pro oba ent. typy

- každé auto přiděleno žádné nebo jedné osobě
- každý zaměstnec má žádné nebo jedno auto

- nelze přilepit atributy ani k vozu, ani k osobě, anebo bychom museli jedině připustit prázdné hodnoty (v SQL toto lze)

- **vytvoříme třetí (vztahovou) relaci UŽÍVÁ**
    - atributy odpovídající identifikačním klíčům obou e. typů
    - klíčem nové relace může být č_osoby nebo SPZ

  OSOBA(č_osoby, ...)
  VŮZ(SPZ, ...)
  UŽÍVÁ(č_osoby, SPZ)

Vztah 1:N

- ent. typ PACIENT je determinantem ent. typu POKOJ, opačně to neplatí
- hraje roli pouze parcialita determinantu (PACIENT)

Povinné členství determinantu vztahu

- evidujeme pouze hospitalizované pacienty

- **přilepíme atribut č_pokoje k relaci PACIENT** (tj. k determinantu)

  PACIENT(rč, ..., č_pokoje)
  POKOJ(č_pokoje, ...)

Nepovinné členství determinantu vztahu

- evidujeme i ambulantní pacienty

- přilepíme-li atribut č_pokoje k relaci PACIENT jako v předchozím případě, musíme připustit prázdnou hodnotu (v SQL možné)

- **zavedení třetí (vztahové) relace UMÍSTĚN**
    - atributy odpovídající identifikačním klíčům obou e. typů
    - klíčem nové relace je klíč determinantu

  PACIENT(rč, ...)
  POKOJ(č_pokoje, ...)
  UMÍSTĚN(rč, č_pokoje)

Vztah M:N

- např. vůz může náležet více zaměstnancům, jeden zaměstnanec může mít přiděleno víc vozů

- **v každém případě budou tři relace** (dvě pro entity, jedna pro vztah)
- primárním klíčem vztahové relace bude dvojice příslušných cizích klíčů

OSOBA(č_osoby, ...)
VŮZ(č_vozu, ...)
POUŽÍVÁ(č_osoby, č_vozu)

Atributy vztahů

Vztahové atributy je třeba přidávat vždy do té relace, v níž jsou obsaženy identifikátory obou relací, jež jsou spolu ve vztahu. Jak to bude řešeno v příkladu na vztah 1:1:

- v prvních dvou případech transformace se přilepí tam, kde jsou klíče obou relací (č_osoby i SPZ)
- ve třetím případě se přidají do nové vztahové relace

Optimální reprezentace by měla také respektovat funkcionalitu systému. Uvažme opět příklad vztahu 1:1 – když se agenda autoparku příliš nevyužívá, můžeme si dovolit více menších relací (pružnější odezva např. při dotazech na zaměstnance), dotaz na autopark bude pomalejší (náročná operace spojení).

Řešení vztahů v SQL

Relační model (RMD) má oproti SQL několik odlišností, je tedy na místě zmínit se o tom, jak se transformace vztahů dá řešit v praxi. Zásadní odlišností je neexistence prázdných hodnot v RMD, která například při transformaci vztahu 1:N s *nepovinným členstvím* determinantu vede k nutnosti zavádět samostatnou vztahovou entitu. Naproti tomu v SQL bychom tento vztah implementovali podobně jako vztah 1:N s povinným členstvím determinantu – dvěma tabulkami s cizím klíčem v tabulce determinantu, u tohoto cizího klíče bychom ovšem povolili prázdnou hodnotu. Podobně je tomu u vztahu 1:1 – zde bychom asi velmi zřídka používali řešení se samostatnou vztahovou tabulkou. Buď bychom použili tabulku jedinou (v případě povinného členství obou entit), nejspíše ale použijeme opět dvě tabulky propojené cizím klíčem (s omezením nebo bez omezení prázdných hodnot).

Lze tedy říci, že asi nejčastější je reprezentace vztahů pomocí dvou tabulek a cizího klíče mezi nimi. Tímto způsobem lze řešit vztahy 1:1 i vztahy 1:N. Podle povinného členství může mít cizí klíč buď povolené nebo zakázané prázdné hodnoty. Z uvedeného ovšem vyplývá jistá nejednoznačnost transformace: máme-li v databázi dvě tabulky propojené cizím klíčem, nepoznáme z této struktury, o jaký typ vztahu se ve skutečnosti jedná. Další problém představuje fakt, že dvě tabulky s cizím klíčem *vždy* umožní ukládat záznamy dle principu 1:N, samotná struktura tabulek není schopna zajistit zajištění kardinality 1:1. Chceme-li tedy vynutit, aby takto implementovaný vztah byl skutečně 1:1, je to nutno zajistit programově, např. pomocí triggeru či na úrovni aplikace.

Slabý entitní typ

- identifikační závislost na id. vlastníkovi, slabý ent. typ má pouze parciální identifikační klíč
- identifikační vztah je speciálním případem vztahu 1:N, kde slabý entitní typ má povinné členství, tedy máme vyřešeno: k relaci slabého ent. typu přilepíme atributy odpovídající identifikačním klíčům id. vlastníků jako cizí klíče

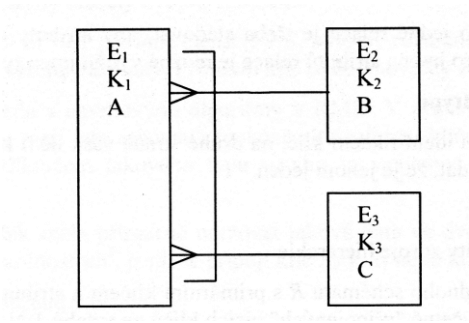
Entitní podtyp (ISA hierarchie)

Entitní podtyp narozdíl od slabého typu nemá žádný parciální klíč, je identifikován zdrojem ISA hierarchie. Jsou možné tři způsoby transformace:

- jediná relace se všemi atributy nadtypu i všech podtypů + další rozlišující atribut, jakého typu entita je
    - tzv. TPH (table-per-hierarchy) model
    - atributy, které nepatří do daného podtypu, musí zůstat prázdné (v čistém relačním modelu toto ovšem není možné, lze pouze v SQL)
    - výhody
        - přímočaré, rychlé
        - výhodné pro základní operace (čtení, zápis, update, delete)
        - minimalizuje počet tabulek v databázi
    - nevýhody
        - mnoho prázdných hodnot (u nevyužitých atributů ostatních podtypů) – riziko porušení integrity, redundance
        - porušuje třetí normální formu
        - obtížná manipulace s podtypy – nelze je snadno přidávat či odebírat
        - u atributů podtypů nelze efektivně vynutit omezení NOT NULL (nutno kontrolovat procedurálně např. triggerem v databázi, viz dále)
    - ošetření integritních omezení
        - nevyužité atributy musí být prázdné
        - kontrola NOT NULL omezení
- relace pro každý typ (nadtyp + všechny podtypy)
    - tzv. TPT (table-per-type) model
    - primární klíče podtypů slouží zároveň jako cizí klíče do nadtypu
    - výhody
        - zachování integrity jednotlivých tabulek, možnost dodržet třetí normální formu
        - nízká datová redundance
        - snadná manipulace s podtypy
    - nevýhody
        - pomalejší odezva v dotazech – nutnost častého spojení; počet spojení může být někdy roven počtu podtypů
        - zvýšení počtu tabulek v databázi

    - ošetření integritních omezení
        - nutno dbát na disjunktnost podtypů (nesmí se stát, že dvě relace podtypů obsahují tentýž primární klíč)
        - nutno kontrolovat, že pro každou n-tici v relaci nadtypu existuje odpovídající n-tice v *právě jedné* relaci podtypu
    - pro snadnější čtení dat z příliš složitých struktur je možno převést tuto hierarchii na model TPH pomocí pohledu (VIEW)
- relace pouze pro entitní podtypy
    - tzv. TPC (table-per-concrete type) model
    - v každé relaci se opakují atributy nadtypu; dílčí relace vlastně „nevědí“ o hierarchii – není mezi nimi žádný vztah
    - modifikace struktury hierarchie jsou obtížné – mnoho relací obsahuje tytéž atributy, zásahy do nadtypu a jeho vazeb je tedy nutno opakovat ve všech dílčích relacích
    - ošetření integritních omezení
        - některá integritní omezení (např. primární či cizí klíče) musejí být kontrolována nad celou množinou relací jaksi distribuovaně, je například nutno kontrolovat, aby více dílčích relací neobsahovalo tentýž primární klíč apod.

Výlučné vztahy



Reprezentace je opět možná více způsoby.
První způsob lze použít, mají-li klíče K2 a K3 stejnou doménu. Zavedeme tři relace

R1(<u>K1</u>, id, K, A)
R2(<u>K2</u>, B)
R3(<u>K3</u>, C)

kde *K* je cizí klíč odkazující buď na R2 nebo na R3 a *id* je atribut, rozlišující, který vztah to je.

Druhý způsob se používá, nemají-li klíče K2 a K3 stejnou doménu. Zavedeme tři relace

R1(K1, K2, K3, A) - zde nesmí být K2 i K3 současně neprázdné!
R2(<u>K2</u>, B)
R3(<u>K3</u>, C)

Při použití druhého způsobu musí být všechny cizí klíče – kromě jednoho – prázdné. To je požadavek, který v klasickém relačním modelu nelze splnit, lze jej ovšem bez potíží realizovat v SQL. Pokud však v SQL povolíme prázdné hodnoty cizích klíčů, bude se obtížněji vynucovat povinné členství ve vztahu

V obou případech je zapotřebí kontrolovat integritu (obvykle nějakými procedurálními prostředky, např. triggerem v databázi)

- v prvním způsobu nelze použít automatickou kontrolu referenční integrity v databázi (FOREIGN KEY), neboť jeden atribut může odkazovat na více relací; tuto kontrolu je tedy třeba naprogramovat ručně či zajistit jinak
- v druhém způsobu je třeba kontrolovat, aby byl neprázdný vždy nejvýše jeden cizí klíč; složitěji se též vynucuje povinné členství ve vztahu – musí být neprázdný *právě jeden* z cizích klíčů

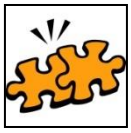
Normální forma výsledných relací

Cílem je, aby vznikající relace byly normalizované, aspoň ve třetí normální formě (3NF). Při transformaci to závisí na citu a zkušenosti analytika. Mělo by platit, že jediné závislosti, odvoditelné z ER diagramu, jsou závislosti atributů na klíči, odhalení dalších závislostí může indikovat další entitní typ.

Např. mějme relaci

PRACOVNÍK(<u>rč</u>, ..., č_místnosti, č_telefonu)

Platí-li funkční závislost č_telefonu → č_místnosti, tj. že podle telefonního čísla vždy jednoznačně poznáme místnost, pak relace není ve 3NF. Bylo by možné zvážit dekompozici na více relací (viz další přednášky), z hlediska uživatele nemá ale dekompozice smysl. V SQL lze problém částečně řešit pomocí pohledů.



Kontrolní otázky

- Co to je cizí klíč? Jak souvisí s referenční integritou?
- Jaká je souvislost dekompozice M:N vztahu a převodu M:N vztahu do RMD?
- Jsou nějaká integritní omezení E-R modelu, která se v RMD nedají explicitně vyjádřit?
- Jak se liší slabá entita od entitního podtypu?
- Kdy lze použít výlučný vztah?

Úlohy k procvičení

1. Transformujte do relačního modelu E-R diagram vytvořený v kapitole 1 (databáze pilotních průkazů).

---

2. Transformujte do relačního modelu E-R diagram vytvořený v kapitole 2 (databáze fotbalové ligy).

4 Modelování funkcí

Cíl kapitoly

Kapitola představuje úvod do funkční analýzy. Pozor, funkční analýza není analýza funkčních závislostí mezi daty (viz látka o normalizaci), nýbrž analýza procesů, které má vykonávat navrhovaný softwarový systém. Cílem kapitoly je naučit se

- co to je funkce, co to je proces
- jak lze modelovat funkce softwarového systému
- různé strategie funkčního modelování
- základy tvorby DFD (data flow diagram)

Klíčové pojmy

Funkční analýza, funkce, proces, data flow diagram.

Úvod

Nedílnou součástí analýzy a návrhu IS je funkční analýza, která zahrnuje modelování činností týkajících se dané aplikace. Funkční analýza si klade zejména tyto otázky:

- jaké informace vstupují do funkcí
- jak informace putují mezi funkcemi
- jaké informace vystupují z funkcí

Funkční analýza je první krok ve specifikaci aplikačních programů, jež budou nad databází operovat. Je to tedy komplement k datové analýze, umožní ji verifikovat a ověřit úplnost datového modelu.

Existuje více způsobů modelování funkcí

- hierarchie funkcí
- události – spouštěče funkcí
- DFD – data flow diagramy, diagramy toků dat
- stavové diagramy
- Petriho sítě

Při funkční analýze opět rozlišíme různé úrovně abstrakce (v seznamu jdeme od nejabstraktnější úrovně):

- (pracovní) funkce v organizaci
- proces v konceptuálním modelu
- procedura, program v programové úrovni

Co to je funkce

Funkce je něco, co organizace dělá, např. založení nového abonenta v evidenci vstupenek nebo odstranění studenta ze seznamu aktivních studentů apod.

Funkce má mít výstižné jméno a popis: identifikuj, změň, odstraň, odešli, spočítej,..., např. „založ nový požadavek na reservaci“.

V popisu funkce vždy specifikujeme CO chceme udělat, nikoli JAK. Ve funkci se nezmiňují organisační struktury podniku (oddělení...) či role uživatelů (ředitel...), vylučujeme též informace typu KDE, KDO, KDY, JAK.

Hierarchie funkcí

Modelování funkcí pomocí hierarchie funkcí je typicky výsledkem funkční analýzy strategií shora dolů. Tato strategie je analogická strategii shora dolů, kterou již známe z datové analýzy: problém modelujeme postupně s větší a větší úrovní detailu. Nejprve tedy určíme základní funkci, kterou případně rozdělíme na podfunkce, dílčí funkce. Vzniká tak strom (hierarchie) funkcí, pořadí funkcí na stejné úrovni stromu ovšem neurčuje pořadí vykonávání. Při dekompozici mateřské funkce musí dílčí funkce úplně a beze zbytku pokrýt nadřazenou funkci, nesmí se překrývat a musí být nutné. Listy stromu jsou atomické, nevyžadují již dekompozici. Jako spouštěč, trigger nějaké funkce může často fungovat nějaká událost.

V hierarchii funkcí můžeme rozlišit typy funkcí:

- atomické – dále nedělitelné
- elementární – mohou se skládat z dílčích funkcí, ale vždy se provedou celé nebo nic (obdoba transakcí – viz další přednášky)
    - převod peněz mezi účty
        - odečíst peníze z jednoho účtu
        - přičíst peníze na druhý účet
- společná – funkce, která se v hierarchii vyskytuje vícekrát
    - lze ji navrhnout, popsat a implementovat pouze jednou

Funkční analýza

Během funkční analýzy dochází k postupnému objevování funkcí, stejně jako tomu je v datové analýze. Základem je interview se zákazníkem. Při rozhovoru je třeba být investigativní, neboť pracovníci mají tendenci leckteré funkce považovat za tak samozřejmé, že se o nich ani nezmíní. Je proto výhodné používat další zdroje informací, analogie z jiných společností, apod.

Při funkční analýze lze opět aplikovat postup shora dolů nebo zdola nahoru, obojí lze kombinovat a vzájemně doplňovat a testovat. Výsledkem analýzy je schema funkcí, které uvádí funkce do vzájemných souvislostí.

Schema funkcí má být

- přesné a srozumitelné
- úplné
- stručné ve výrazivu
- vyvážené

Funkce mají samozřejmě vztah k datovému konceptuálnímu schematu. Během návrhu je tedy nutné testovat zda

- existují funkce, které zmiňují dané entity
- existují funkce, které vytvářejí, mění či odstraňují entity, vztahy, hodnoty atributů v koncept. schematu

Nesmí se tedy stát, že datový model obsahuje entitu, se kterou nepracuje žádná funkce. Naopak, funkce musí pracovat pouze s entitami, které existují v datovém modelu.

Každá funkce musí být specifikována. Specifikace funkce je popis její logiky, tj. jak se provádí, je to vlastně slovní popis algoritmu. Detailní specifikace je časově náročná, provádí se především u klíčových a složitých funkcí. Popis funkce by v každém případě měl být natolik exaktní, aby umožnil jednoznačnou implementaci.

Data flow diagram

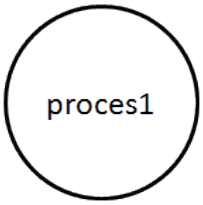
DFD je model toků dat. Místo funkcí se v něm obvykle hovoří o procesech. Základní konstrukty DFD jsou

- proces (process)
    - reprezentuje nějakou aktivitu v softwarovém systému
    - transformuje informace
    - předává informaci jiným procesům
    - může pokrývat několik pracovních funkcí
- tok dat (data flow)
    - výměna informace mezi procesy
- sklad dat (data store)
    - místo, kde jsou uložena data – šanon, databáze...
    - odpovídá nějaké podmnožině datového modelu
- rozhraní (interface), někdy se nazývá terminátor
    - např. externí uživatel
    - při dekomponovaném schematu může reprezentovat část IS popsanou jinde

DFD poskytuje funkční popis systému, nikoli popis implementace. Je to orientovaný graf, kde uzly: jsou procesy, rozhraní a sklady, hrany jsou toky dat. Při návrhu diagramu je dobré všechny objekty pojmenovat, a to i hlavní toky dat. Někdy lze procesy vnořovat do sebe, někdy se (vcelku rozumně) požaduje, aby mezi skladem a rozhraním byl vždy aspoň jeden proces.

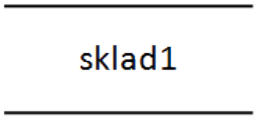
Použitá notace DFD

proces (process)

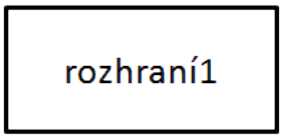


datový tok (data flow)

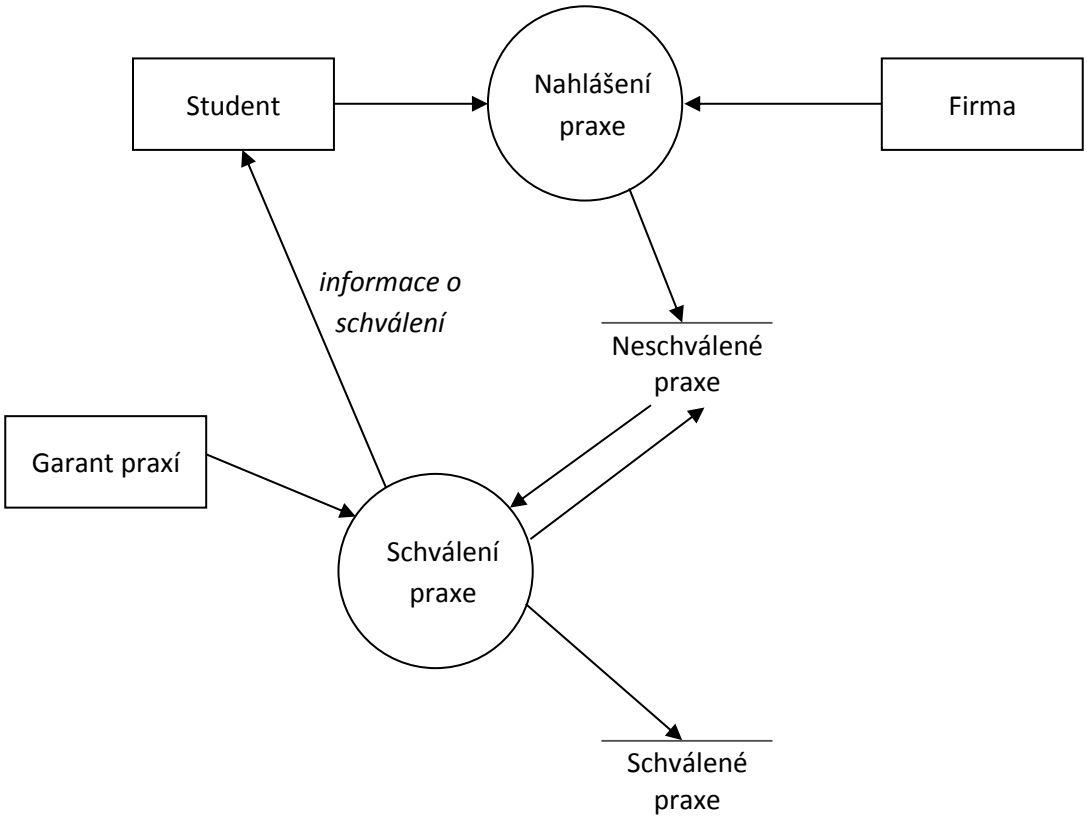
sklad dat (data store)



rozhraní (interface)



Příklad DFD



Použití DFD ve funkční analýze

Můžeme nejprve definovat hierarchii funkcí a poté z identifikovaných funkcí konstruovat DFD. To obnáší seskupování funkcí do procesů. Při tomto postupu lze využít různé metodologie.

Základní posloupnost kroků, kterou obvykle postupujeme při tvorbě DFD je

- identifikuj rozhraní
- identifikuj I/O toky dat mezi rozhraním a systémem
- navrhni skeletální DFD
- zjemňuj DFD libovolnou strategií tak dlouho, až jsou obsaženy všechny požadavky, ale schema ještě není procedurální

Můžeme použít opět různé strategie návrhu:

- shora dolů
    - zjemňování
- zdola nahoru
    - nejprve izolované procesy, které se postupně spojují
- zevnitř ven
    - lze postupovat od rozhraní nebo od skladu dat
- smíšená

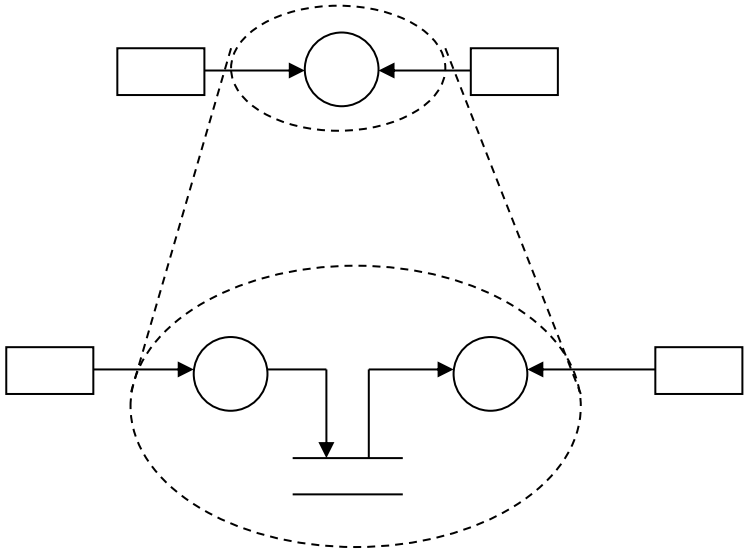
Jakákoli strategie je vlastně transformací diagramu od počátečního stavu ke koncovému. Obecná doporučení:

- schema má ukazovat CO, nikoli JAK
- nezjemňovat, přechází-li v procedurální popis, kde jednotlivé procesy jsou již redukovány na téměř jediný příkaz prograovacího jazyka
- nezjemňovat, ztrácí-li se globální struktura

Posouzení kvality DFD modelu

- funkční nezávislost
    - každý proces je autonomní, může být analyzován a měněn nezávisle na zbytku systému
- úplnost
    - DFD pokrývá všechny rysy aplikace, nic neschází
- korektnost
    - zachování ekvivalence hranic po zjemnění
- čitelnost
- minimalita
    - DFD neobahuje nadbytečné komponenty
- procesy a sklady se nepřekrývají

Ilustrace ekvivalence hranic:



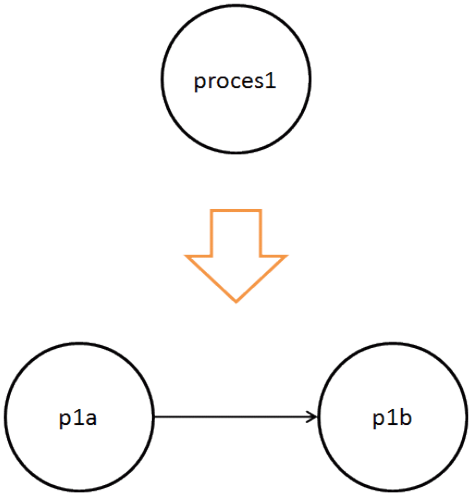
Během analýzy je nezbytná integrace datové a funkční analýzy. Probíhá vzájemné ovlivňování, například zjemnění schematu funkcí vyvolá nutnost zjemnit i ER model. Např. rozdělíme proces na dva, čímž je indukována nutnost zavést do ER modelu ISA hierarchii. Je nutná kontrola vzájemné úplnosti:

- každý pojem, zmiňovaný v tocích dat a zásobárnách, se vyskytuje v ER schematu
- každý datový prvek ER modelu musí mít k sobě proces, který jej vytváří, mění, odstraňuje

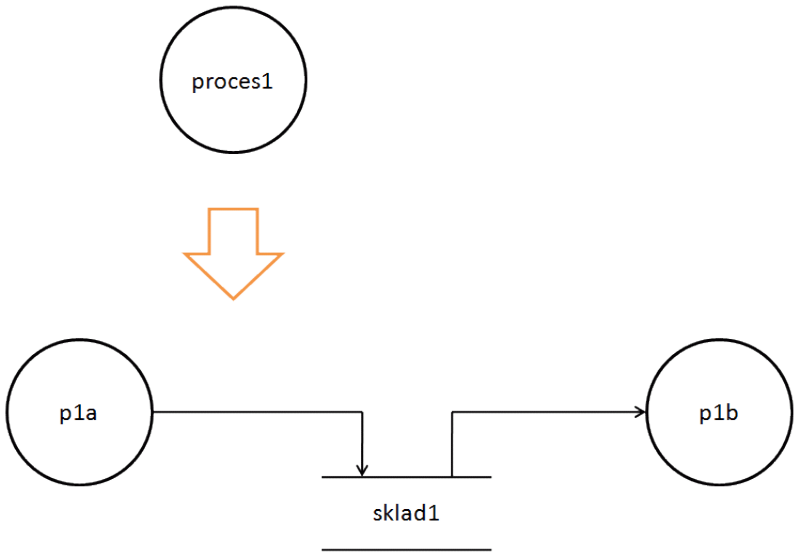
Transformace DFD při použití metody shora dolů

V této metodě postupně zjemňujeme funkční model a zvyšujeme jeho úroveň detailu. Zjemňování probíhá formou dělení jednoho objektu na více objektů při současném dodržování ekvivalence hranic. Se zjemňováním je nutno přestat, pokud již model přechází v procedurální popis. Při dekompozici se rovněž snažíme o nezávislost procesů. Následující příklady ilustrují možné způsoby dekompozice základních konstruktů.

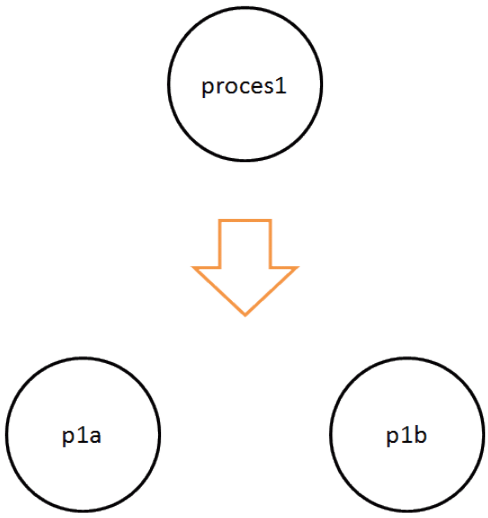
- dekompozice procesu s mezitokem



- dekompozice procesu s vloženým skladem

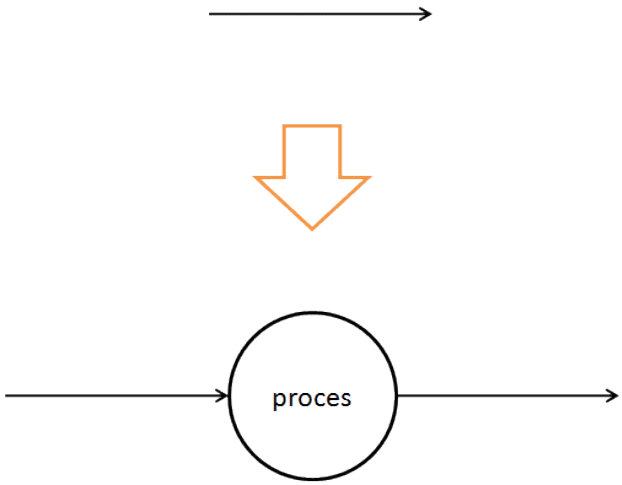


- dekompozice procesu bez spojení



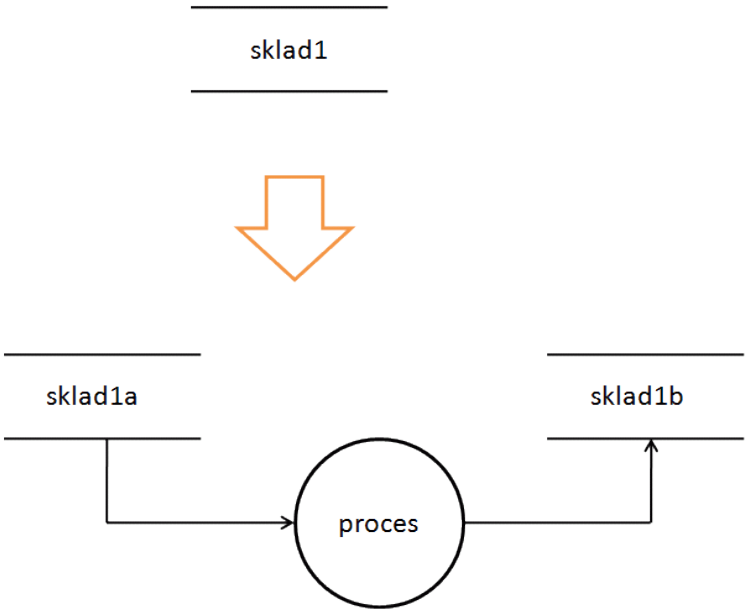
- dekompozice datového toku

- zjemnění datového toku



- dekompozice skladu

- dekompozice skladu s vloženým procesem



Kontrolní otázky

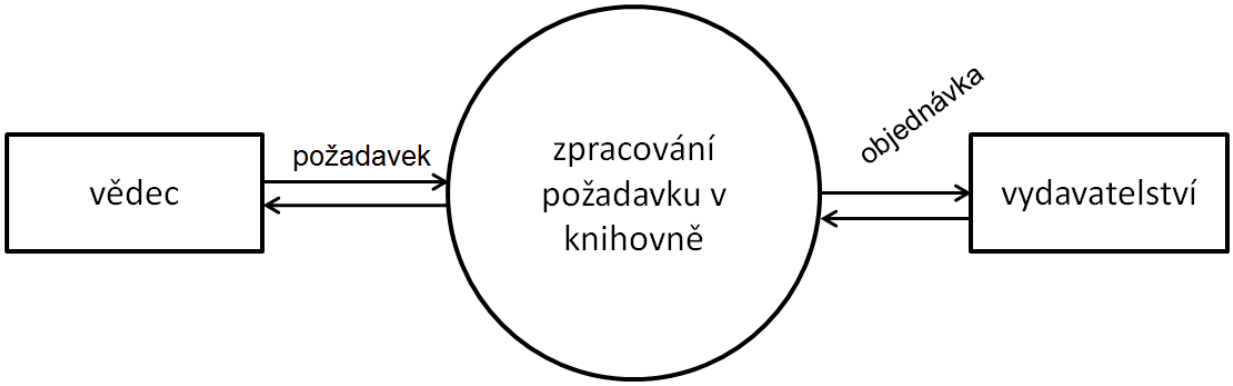
- Co to je funkce?
- Co to je proces a jaká je jeho úloha?
- Co to je sklad dat, rozhraní, datový tok, jaká je jejich úloha?
- Je možné, aby proces v DFD nebyl zapojen do žádného datového toku?
- Ovlivňuje funkční analýza datovou analýzu?

Řešené příklady

Příklad 1

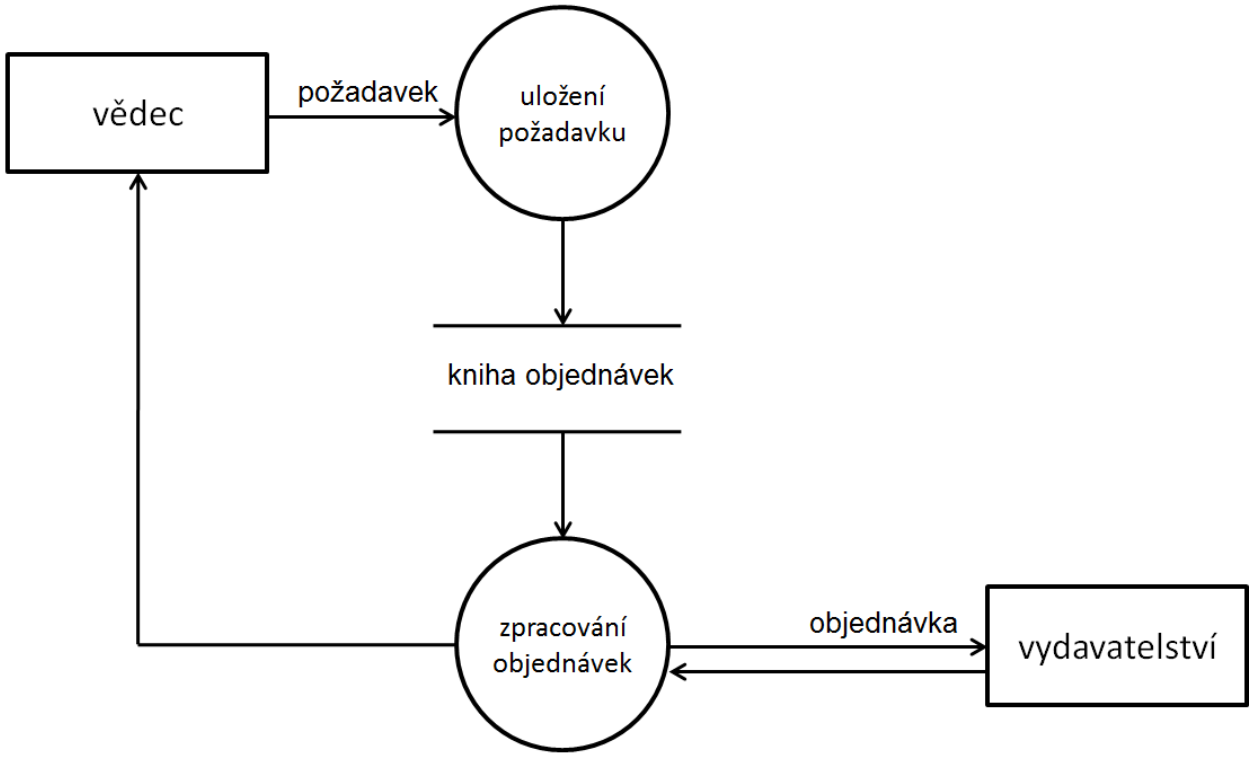
Máme systém vědecké knihovny, vědec si tu a tam přes knihovnu chce objednat knihu. Vědec tedy zašle požadavek do knihovny, tam se požadavek zaknihuje a pak se postupně zpracovává. Pokud je kniha v knihovně, rovnou se mu půjčí. Pokud se musí koupit, zkoumá se, zda ji lze financovat z grantu.

Vytvoříme DFD nejprve strategií shora dolů. Na počátku je kontextový model (neboli skeletální schema): jediný proces, zastupující celý systém, plus všechna známá rozhraní.



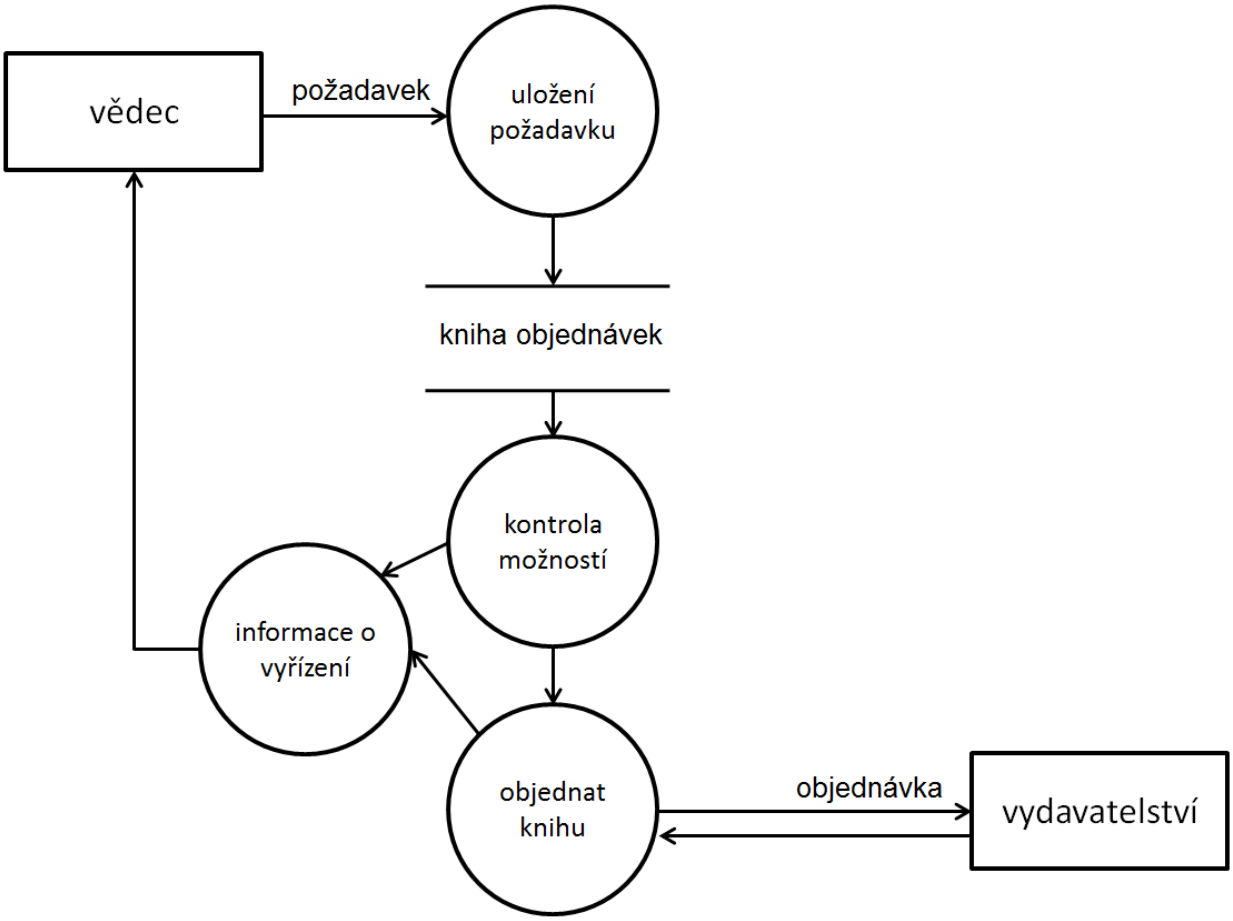
Následná dekompozice: *zpracování požadavku* je vlastně

- uložení požadavků
- zpracování objednávek



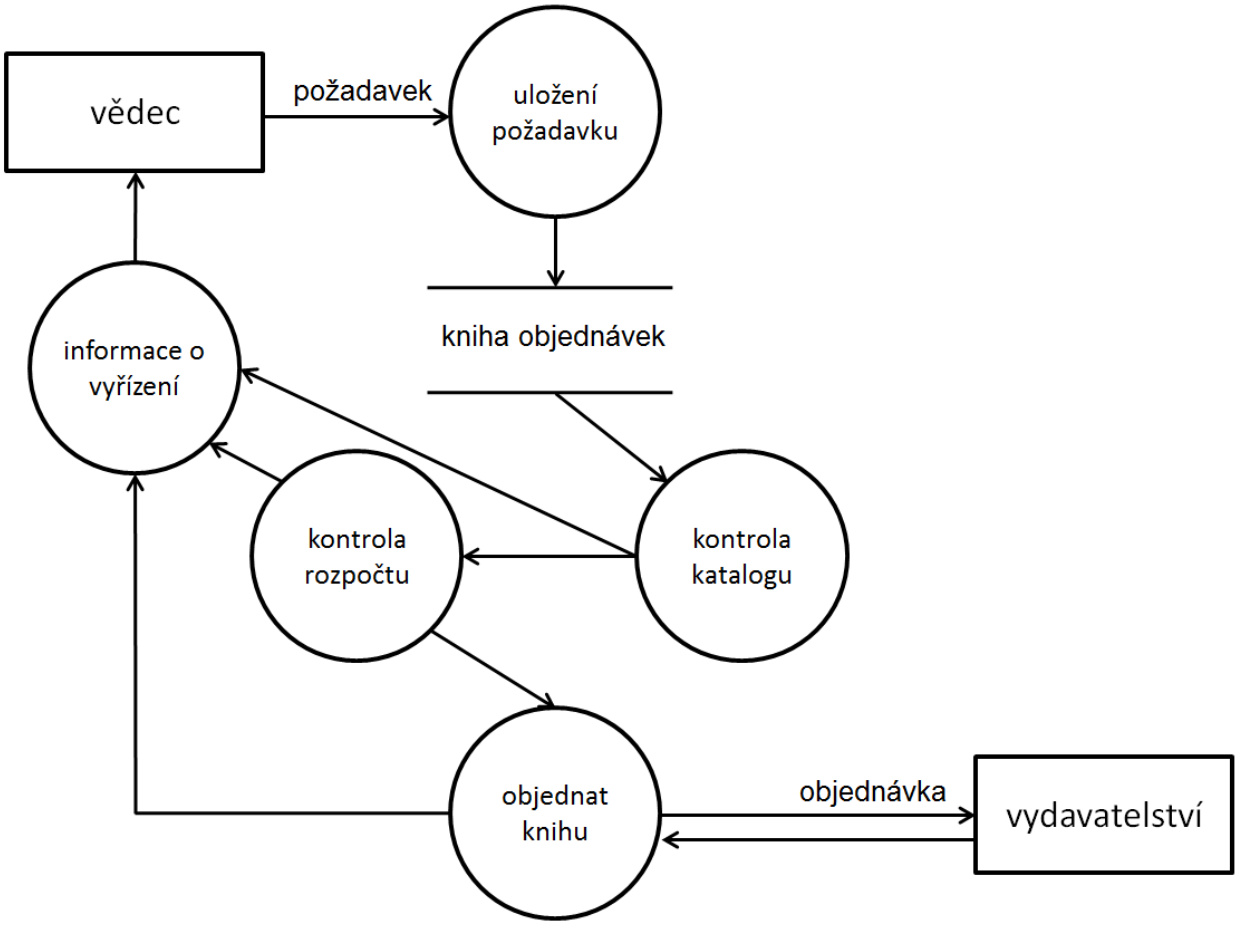
Následná dekompozice: *zpracování objednávek* je vlastně

- kontrola možností
- objednání knihy
- informace klientovi

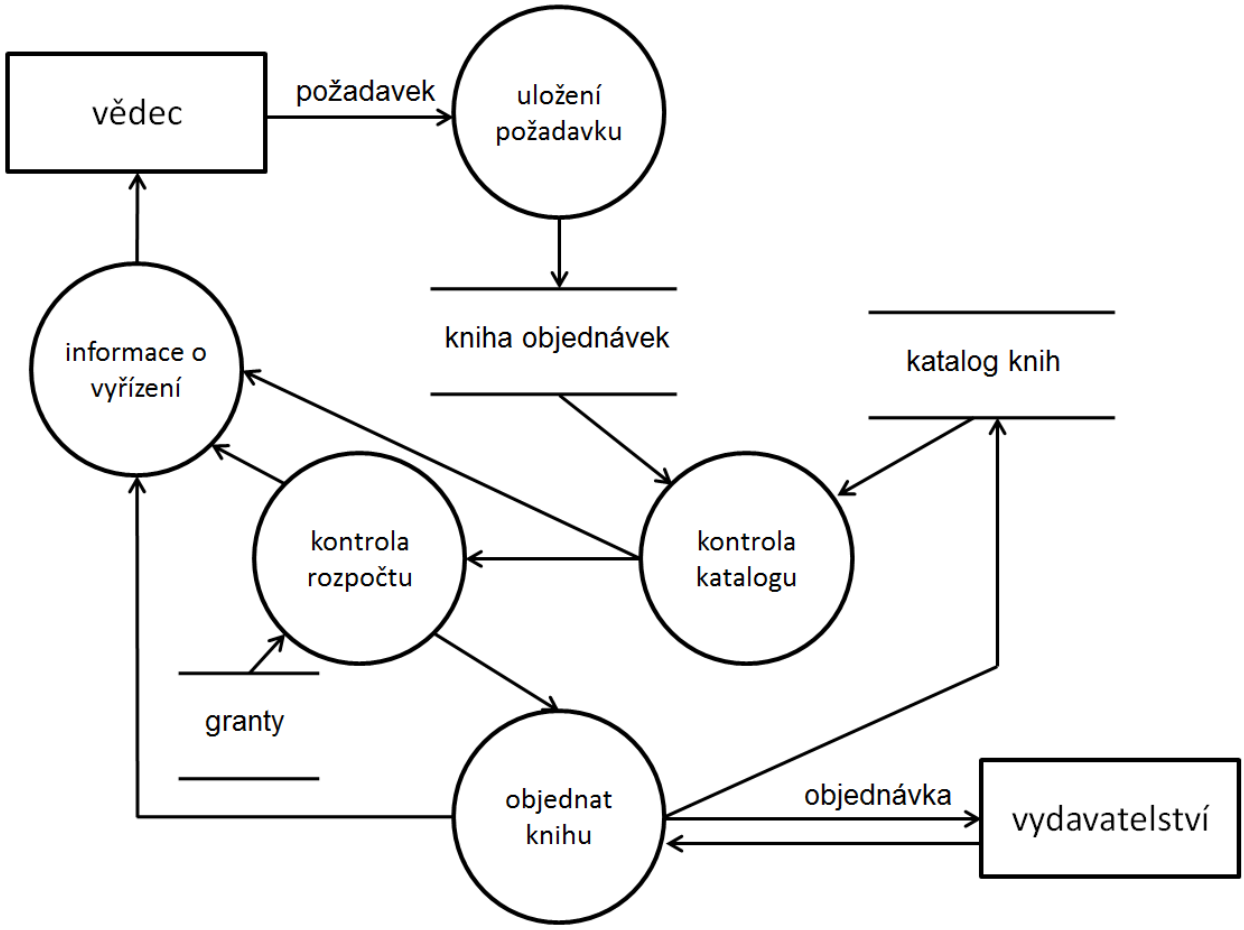


Následná dekompozice: *kontrola možností* se dělí na

- kontrolu katalogu
- kontrolu rozpočtu



Následná dekompozice: *kontrola rozpočtu* potřebuje ještě databázi grantů, *kontrola katalogu* potřebuje databázi knih.

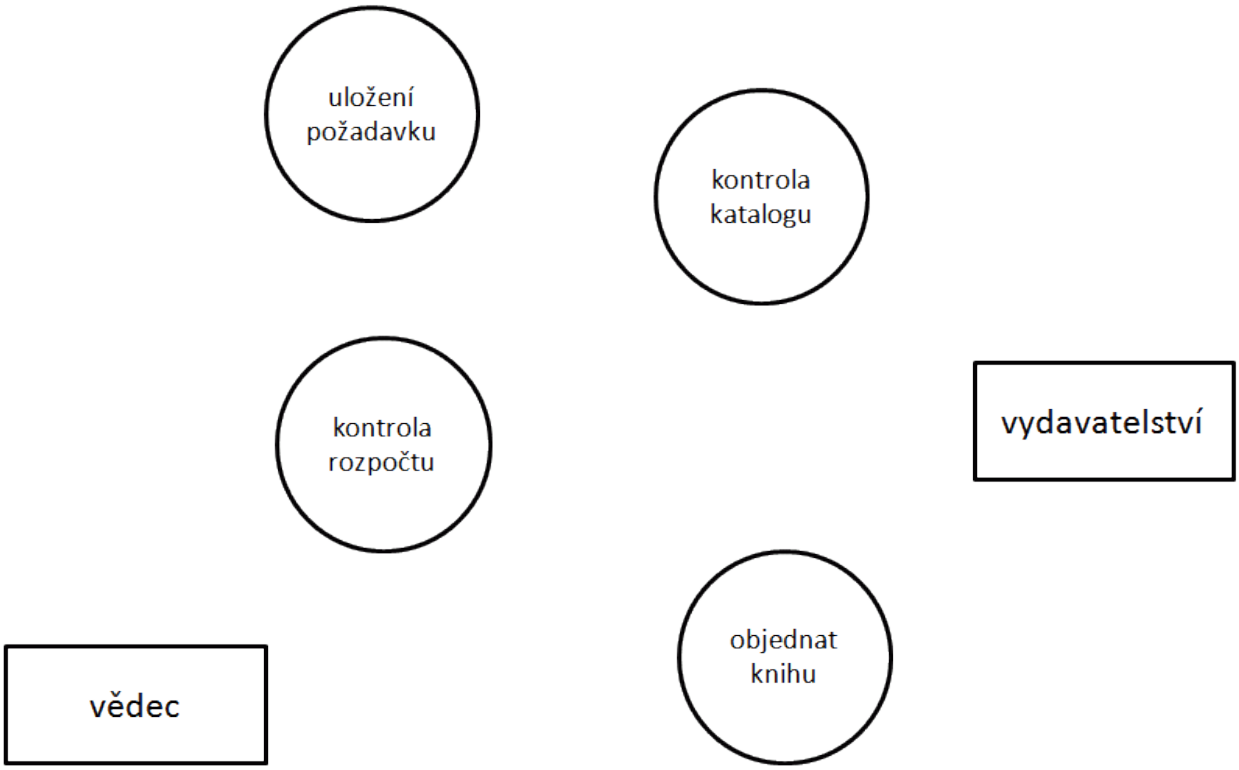


Pojďme nyní tentýž příklad řešit s použitím strategie zdola nahoru. Při této strategii následují kroky:

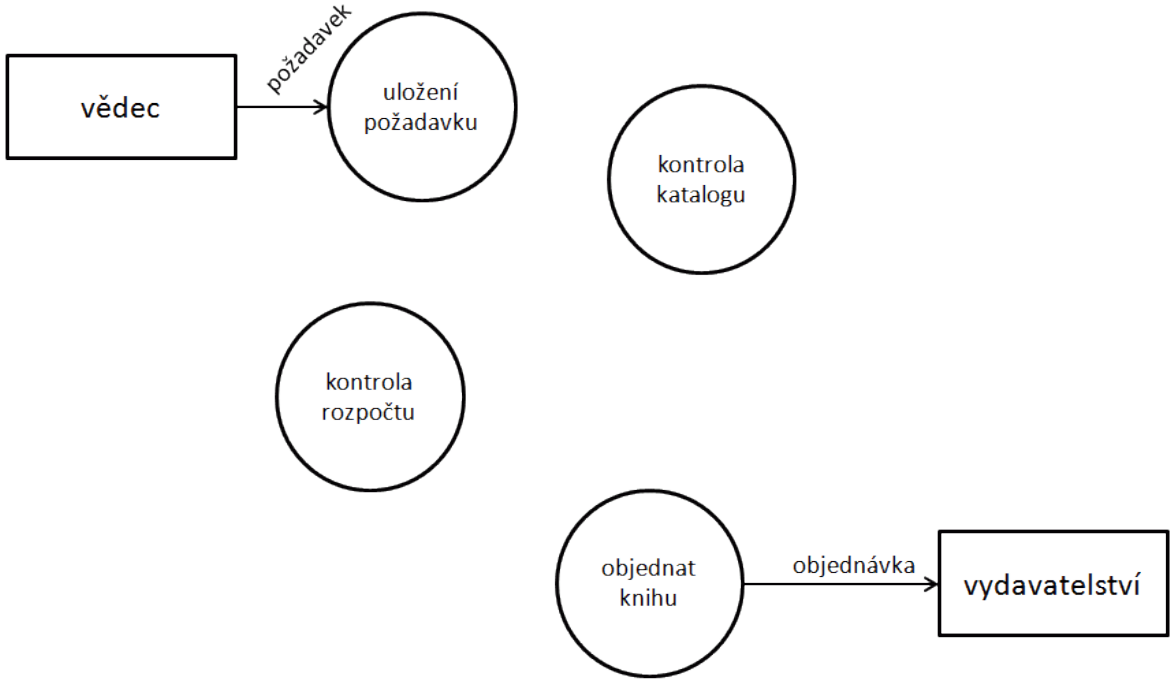
- identifikace všech elementárních procesů
- zavedení rozhraní
- zavedení toků a skladů

Na počátku je tedy množina elementárních procesů.

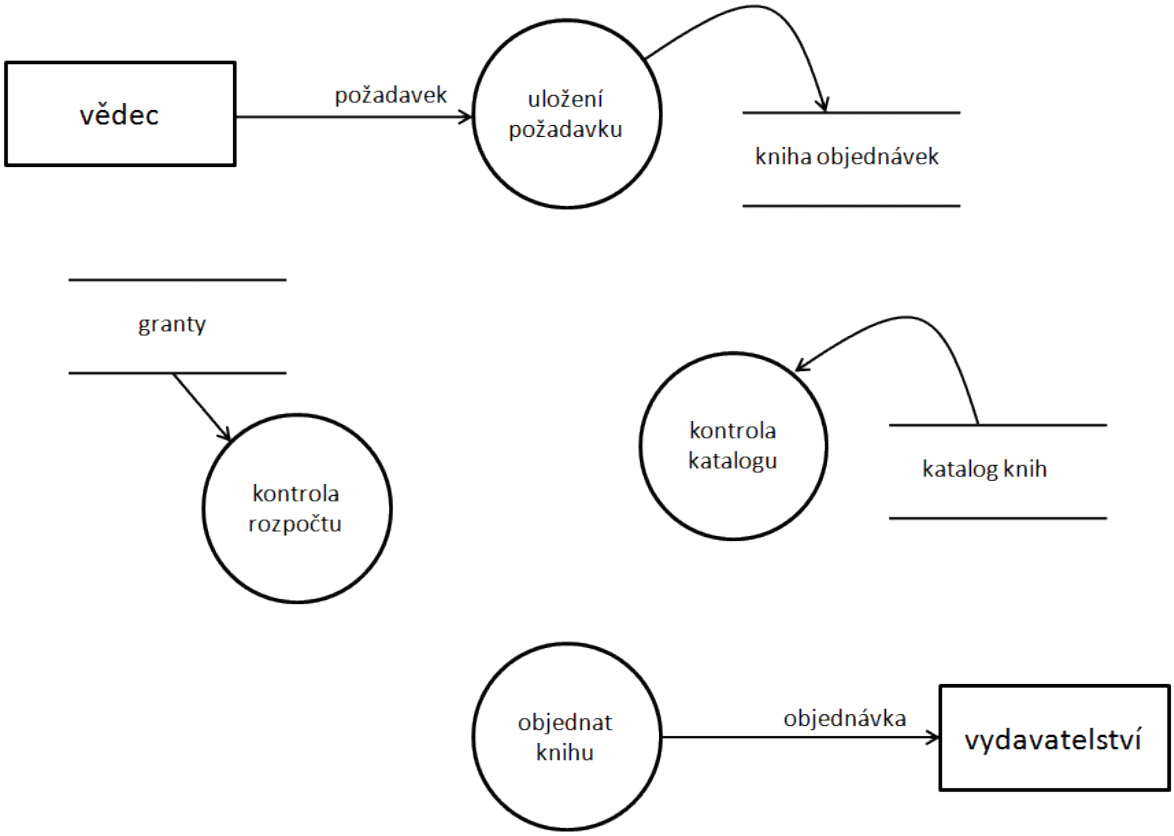
Dále přidáme známá rozhraní.



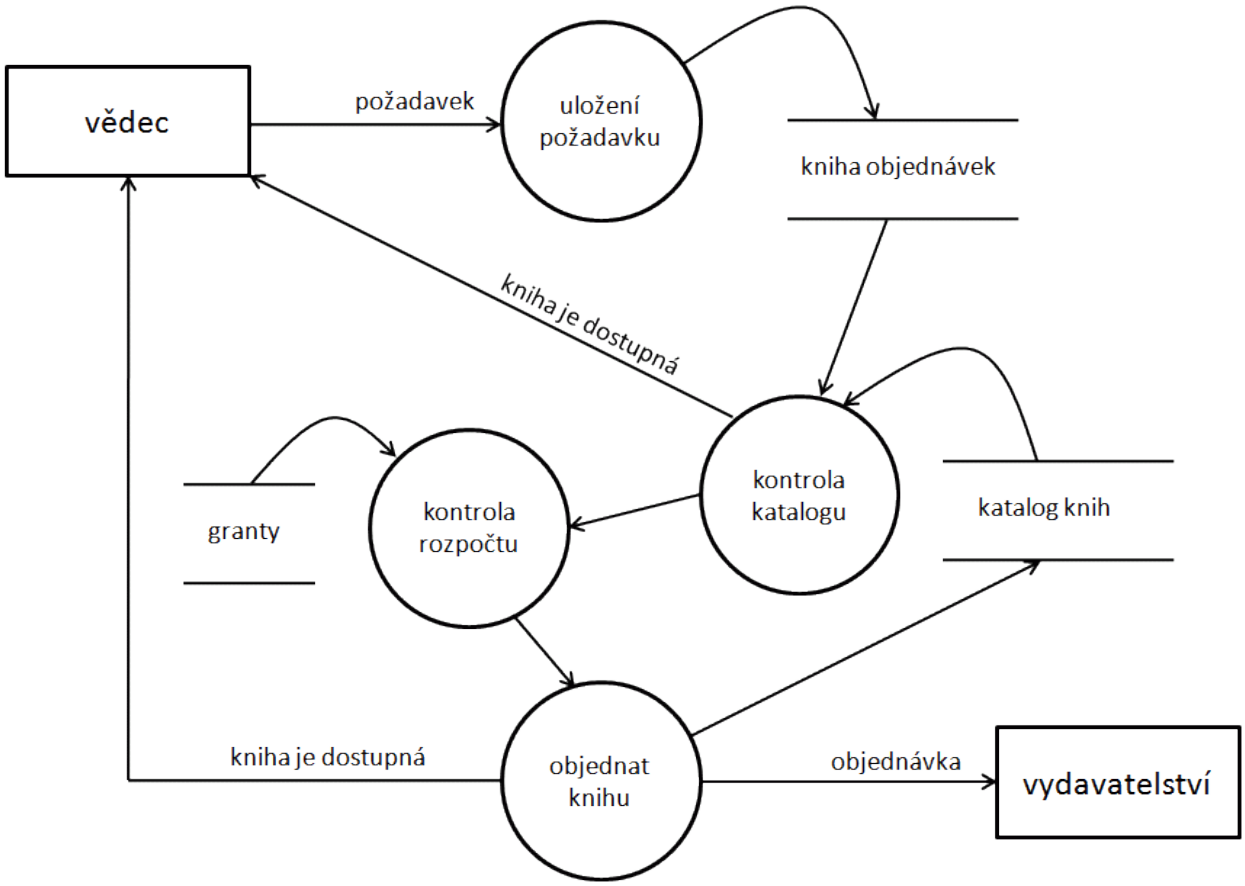
V dalším kroku propojujeme existující objekty datovými toky...



... a přidáváme sklady dat.

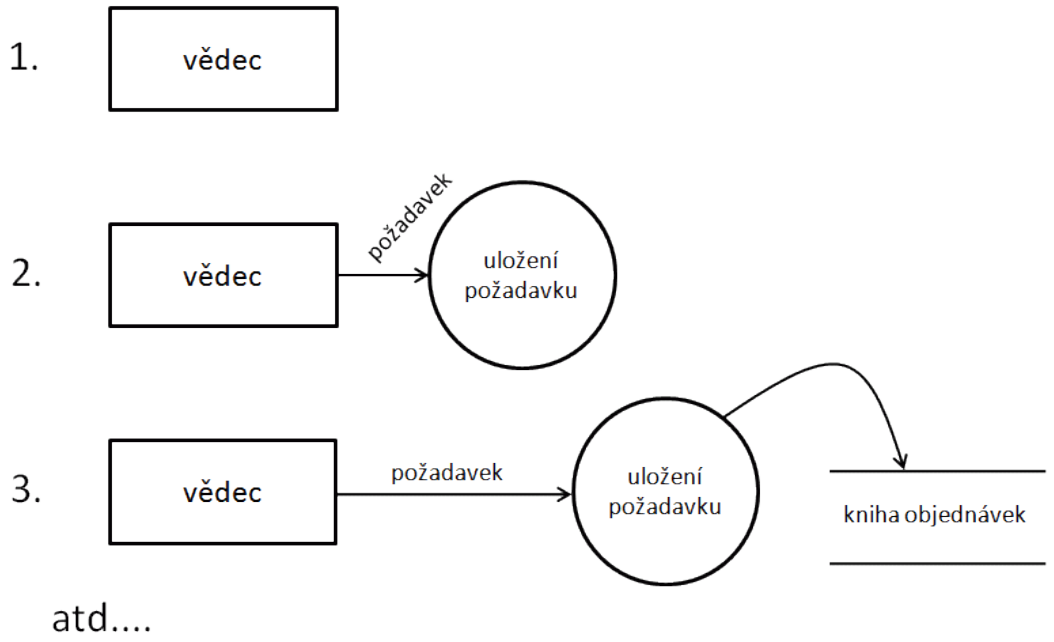


Výsledný model:



Při srovnání modelu vytvořeného strategií shora dolů a modelu vytvořeného strategií zdola nahoru je vidět, že se od sebe liší (v posledním modelu jsme zapomněli např. na zpětnou vazbu od vydavatelství či informaci klientovi o výsledku kontroly rozpočtu). To je ovšem poměrně přirozený jev; dokonce i tehdy, když dva týmy řeší tentýž problém stejnou strategií, se mohou výsledné modely lišit. Modelování reality je totiž do značné míry subjektivní záležitost a velmi záleží na zkušenosti a úhlu pohledu analytika.

Na závěr ukážeme princip tvorby DFD metodou zevnitř ven. Můžeme začít například od zvoleného nejdůležitějšího rozhraní a postupovat k sousedním objektům:



Úlohy k procvičení

1. IS pro studenty VŠ: Studenti se zapisují na předměty, z předmětů pak skládají zápočty a zkoušky, studijní výsledky zapisuje vyučující do evidence. Každou zkoušku lze absolvovat nejvýše třikrát, předmět lze jednou opakovat, je ale nutno se do něj znovu zapsat.

5 Funkční závislosti

Cíl kapitoly

Kapitola představuje úvod do problematiky funkčních závislostí. Pozor, nepleťte si tuto problematiku s funkční analýzou probíranou v předchozí kapitole. Funkční závislosti jsou pojmem, vztahujícím se k relačnímu modelu dat (RMD), pohybujeme se tedy jednak na zcela odlišné úrovni abstrakce než ve funkční analýze, zejména je ale analýza funkčních závislostí součástí **datové analýzy**. Rozbor funkčních závislostí je nezbytným podkladem pro normalizaci logického schematu, viz další kapitola. Cílem této kapitoly je naučit se

- co to je funkční závislost (FZ)
- proč funkční závislosti zkoumáme
- jak souvisí FZ s klíčem relace a co to je tranzitivní závislost na klíči
- co to jsou Armstrongova pravidla
- co to je uzávěr množiny funkčních závislostí a pokrytí množiny funkčních závislostí
- co to je uzávěr množiny atributů
- co to je minimální pokrytí a jak jej určujeme

Klíčové pojmy

Funkční závislost, uzávěr množiny FZ, pokrytí množiny FZ, uzávěr atributu, minimální pokrytí, redundantní atribut, redundantní FZ, klíč, tranzitivní závislost na klíči.

Motivace

Důležitým krokem při návrhu databáze je normalizace relací, resp. kontrola normálních forem relací. Normalizací se míní úprava struktury relace tak, aby relační schema dobře reprezentovalo data, aby byla omezena redundance a přitom se neztratily vazby mezi daty.

Příklad

Mějme relaci

PROGRAM(<u>název_k, jméno_f</u>, adresa, datum)

Problémy:

- některá informace bude zbytečně opakována víckrát
    - adresa kina bude uložena tolikrát, kolik filmů je v kině na programu
- změní-li se jméno ulice, musíme přepisovat mnoho prvků relace
- nehraje-li kino nic, ztrácíme jeho adresu

Řešení: dekompozice na dvě schemata relací:

KINO(název_k, adresa)
PROGRAM(název_k, jméno_f, datum)

Všimněte si, že všechny jmenované problémy zmizely.

Obecně vzato, důsledkem nevhodné (nenormalizované) struktury relací jsou zejména následující jevy:

- 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 do jiné entity
    - 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

Řešením je tudíž normalizace schematu.

Co je důvodem problémů v příkladu?

- hodnoty některých atributů závisejí na hodnotách jiných atributů
    - ke každému kinu přísluší právě jedna adresa
    - pro každé kino a film existuje nejvýše jedno datum, kdy se film dává

Závislost hodnoty určitého atributu na hodnotě jiného atributu (či atributů) je v podstatě funkce: přiřazení hodnoty jiné hodnotě. Jde tedy o speciální případ integritního omezení, neboť funkční závislost vymezuje možné kombinace dat v relaci, tj. množinu přípustných relací.

Funkční závislost

Funkční závislost (FZ) je něco jiného, než jsme řešili na předchozí přednášce (viz úvod kapitoly)! Jde o závislost mezi daty v **jedné** relaci, tedy nikoli o proces jako v předchozí kapitole, ale ani o závislost mezi dvěma či více relacemi nebo dokonce dvěma entitami.

Funkční závislosti z předchozího příkladu lze vyjádřit zápisem

název_k $\rightarrow$ adresa
{název_k, jméno_f} $\rightarrow$ datum

Množina funkčních závislostí nad schématem relace R(**A**) je množina všech zobrazení

$$f_i : X_i \rightarrow Y_i$$

takových, že $X_i, Y_i \subseteq$ **A**, kde i = 1..počet závislostí pro R(**A**). Říkáme, že n-tice z X_i **funkčně určuje** m-tici z Y_i, resp. že m-tice z Y_i **funkčně závisí** na n-tici z X_i. Jednoduše řečeno, hodnoty atributů X společně určují hodnoty atributů Y. Jde vlastně o zobecnění principu klíče (klíč funkčně určuje hodnoty všech atributů, viz dále).

Důležité: funkční závislost vyplývá z **obecné (konceptuální) závislosti dat**, nikoli z jejich konkrétní podoby! Např. máme tabulku studijních výsledků

student	předmět	známka
Novák	Teorie obvodů	3
Novák	Úvod do algebry	3
Haničinec	Teorie obvodů	1
Haničinec	Matematika 1	1

Podle těchto konkrétních dat to vypadá, že existuje FZ

student $\rightarrow$ známka

To ale ve skutečnosti určitě obecně neplatí, je to pouze náhoda. V příštím semestru si třeba student Novák zapíše Teorii elektromagnetického pole a dostane z ní jedničku.

Základní vlastnosti FZ

- pokud $X \rightarrow Y$ a zároveň $Y \rightarrow X$, pak jsou X a Y funkčně ekvivalentní, $X \leftrightarrow Y$
- pokud $X \rightarrow a$, kde $a \in$ **A**, tj. na pravé straně FZ je jeden atribut, závislost se nazývá **elementární**

Redefinice klíče

Máme-li schema relace R(**A**) a nějakou množinu atributů K, K $\subseteq$ **A**, pak K je klíčem schematu R, jestliže současně platí:

1) $K \rightarrow A$
2) neexistuje $K' \subset K$ taková, že $K' \rightarrow A$

Slovně lze uvedené vlastnosti vyjádřit tak, že klíč K musí funkčně určovat všechny atributy relace (tzv. identifikační schopnost) a zároveň musí být minimální v tom smyslu, že z množiny K nelze odebrat žádný atribut, aniž by byla narušena identifikační schopnost.

Uvědomte si, že uvedené dvě podmínky může splňovat více množin atributů, tj. jedna relace může mít více klíčů. Pak se jim říká kandidátní klíče, v praxi jeden z nich volíme jako **primární**.

Armstrongova pravidla

Mějme schema relace R(**A**) a množinu funkčních závislostí F. Nechť X, Y, Z $\subseteq$ A. Armstrongova pravidla jsou čtyři:

1) Jestliže Y $\subseteq$ X, potom X $\rightarrow$ Y
   - triviální funkční závislost, axiom (např. AB $\rightarrow$ A)
2) Jestliže X $\rightarrow$ Y a Y $\rightarrow$ Z, potom X $\rightarrow$ Z
   - tranzitivita
3) Jestliže X $\rightarrow$ Y a X $\rightarrow$ Z, pak X $\rightarrow$ YZ
   - kompozice
4) Jestliže X $\rightarrow$ YZ, pak X $\rightarrow$ Y a X $\rightarrow$ Z
   - dekompozice

Armstrongova pravidla

- jsou korektní: co z F odvodíme, platí pro libovolnou instanci R
- jsou úplná: lze jimi odvodit všechny FZ platné ve všech instancích R
- jsou nezávislá: odstraněním kteréhokoli z nich odstraníme podmínku úplnosti

Příklad aplikace Armstrongových pravidel

Mějme relaci R(P, U, H, M, S, Z) a množinu funkčních závislostí

F = {P $\rightarrow$ U, HM $\rightarrow$ P, HU $\rightarrow$ M, PS $\rightarrow$ Z, HS $\rightarrow$ M}

Co platí (co lze odvodit):

- HM $\rightarrow$ H (triviální)
- HU $\rightarrow$ H (triviální)
- HU $\rightarrow$ HM (kompozice HU $\rightarrow$ H a HU $\rightarrow$ M)
- HM $\rightarrow$ U (tranzitivita HM $\rightarrow$ P a P $\rightarrow$ U)
- HM $\rightarrow$ HU (kompozice HM $\rightarrow$ U a HM $\rightarrow$ H)
- HM a HU jsou funkčně ekvivalentní, HM $\leftrightarrow$ HU

Další příklad aplikace Armstrongových pravidel

Mějme relaci R(a,b,c,d,e,f) a množinu FZ

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

Co lze odvodit

- 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)

Tedy dvojice atributů {a,b} je určitě klíčem schematu relace.

Minimální pokrytí

Definujme nejprve uzávěr F+ množiny funkčních závislostí F. Je to množina **všech** funkčních závislostí odvoditelných z funkčních závislostí v F. Uzávěr F+ bychom získali např. opakovanou aplikací Armstrongových pravidel na všechny dosud získané funkční závislosti a jejich kombinace. Uzávěr je úplný tehdy, když již nelze odvodit žádnou další FZ, která by v něm ještě nebyla.

Řekneme, že závislost f je redundantní v množině funkčních závislostí F, jestliže (F – {f})+ = F+, tj. f lze odvodit z ostatních funkčních závislostí v F.

Pokrytí množiny funkčních závislostí F je množina G taková, že G+ = F+. Dalo by se tedy říci, že množina G má stejnou sémantiku či „stejný informační obsah“ jako množina F, třebaže může obsahovat zcela jiné funkční závislosti. Připomeňme, že pokrytím množiny F je i její uzávěr F+, neboť (F+)+ = F+.

Několik pojmů

- kanonické pokrytí – má na pravých stranách všech FZ pouze jednotlivé atributy.
- neredundantní pokrytí – neobsahuje žádné redundantní FZ, není ovšem dáno jednoznačně, záleží totiž na pořadí odebírání funkčních závislostí z původní množiny F.

Uzávěr množiny atributů X+ *vzhledem k množině funkčních závislostí F* je množina atributů funkčně závislých na X, tj. jsou to ty atributy, které lze z X odvodit pomocí pravidel v F.

Definujme dále redundantní atribut pro jistou FZ. Mějme v množině funkčních závislostí F závislost X$\rightarrow$ Y a mějme atribut A$\in$ X takový, že (X-A)+ = X+. Pak A je v X pro danou závislost redundantní. Jinými slovy, redundantní atribut je takový atribut z levé strany zkoumané FZ, který když z levé strany vypustíme, odvodíme nakonec stejné atributy jako před jeho vypuštěním. Zopakujme, že redundantní atributy je třeba vždy zkoumat ve vztahu k dané množině FZ.

Minimální pokrytí je takové pokrytí, které je kanonické, neredundantní a nemá v závislostech na levé straně žádné redundantní atributy. Motivací pro hledání minimálního pokrytí je fakt, že při normalizaci je vhodné, aby bylo funkčních závislostí co nejméně a byly co nejjednodušší. K minimálnímu pokrytí lze teoreticky nejlépe dospět odebíráním redundantních FZ z uzávěru množiny F+, nikoli z množiny F. To je ovšem náročné, obvyklý postup optimalizace množiny F je proto následující:

- vytvořit kanonické pokrytí ... F’
- odstranit z FZ v F’ redundantní atributy ... F’’
- odstranit z F’’ redundantní závislosti ... F’’’

Pak F’’’ je minimální pokrytí. Takovéto minimální pokrytí nemusí být dáno jednoznačně, záleží na pořadí odebírání redundantních atributů a redundantních FZ.

Příklad nalezení min. pokrytí

Mějme relaci R(A,B,C,D) a množinu funkčních závislostí F={A→ AC, B→AC, D→AB}

- kanonické pokrytí (vpravo vždy jediný atribut):

  F’={A→A, A→C, B→A, B→C, D→A, D→B}

- všechny závislosti jsou již redukované, netřeba odstraňovat redundantní atributy (vlevo vždy jediný atribut)
- odstranit redundantní závislosti

  A→A lze odstranit, je triviální
  A→C nelze odstranit (kdyby ano, pak A+=A)
  B→A nelze odstranit (kdyby ano, pak B+=BC)
  B→C lze odstranit (neb B→A a A→C)
  D→A lze odstranit (neb D→B a B→A)
  D→B nelze odstranit (kdyby ano, pak D+=D)

Minimální pokrytí:

F’’’={A→C, B→A, D→B}

Tranzitivní závislost na klíči

Nechť A není ani klíč ani nadklíč a nechť existuje závislost A → B. Pak B je tranzitivně závislé na klíči. Zdůvodnění: z definice klíče K dostaneme, že K → A a také K→ B (neboť klíč funkčně určuje všechny atributy). Jelikož A → B, pak K → A → B, tím pádem B je na klíči závislé nejen přímo, ale též tranzitivně přes A. Asi tušíme, že existence dvojí závislosti mezi klíčem a atributem B bude způsobovat problémy – povede k datové redundanci a nevhodné struktuře dat v databázi.

Kontrolní otázky

- Co to je funkční závislost?
- Co to je triviální funkční závislost?
- Jaká jsou Armstrongova pravidla a k čemu slouží?
- Jak souvisí funkční závislosti s klíčem relace?
- Co to je minimální pokrytí množiny FZ?
- Co to je tranzitivní závislost na klíči?

Úlohy k procvičení

1. Mějme schema R(**A**), definujte klíč K pomocí funkčních závislostí.

---

2. Mějme schema R(A,B,C,D,E) a množinu FZ F = {AD→E, B→CD, B→A}. Spočítejte uzávěry všech jednotlivých atributů. Je některý z nich klíčem? Přidejte nějakou funkční závislost tak, aby se stal klíčem také atribut A.

---

3. Mějme relaci R(A,B,C,D) a množinu F = {AB→C, A→D, D→B}. Určete klíč(e), spočítejte minimální pokrytí.

---

4. Mějme relaci R(A,B,C,D,E) a množinu F = {B→D, E→C, AC→D, CD→A, BE→A}. Určete klíč, spočítejte minimální pokrytí.

---

5. Mějme relaci R(A,B,C,D) a množinu F = {A→B, BC→D, D→BC, C→A}. Určete klíč, spočítejte minimální pokrytí.

6 Normální formy relací

Cíl kapitoly

Kapitola využívá poznatky předchozí kapitoly a využívá je k definici a práci s normálními formami relací. Cílem této kapitoly je naučit se

- co to je normální forma relace a k čemu je dobrá
- určit, zda je daná relace v první, druhé a zejména třetí normální formě
- dekompozicí převádět relace do normalizované podoby
- ověřovat pokrytí funkčních závislostí a bezeztrátovost dekompozice

Klíčové pojmy

První, druhá, třetí normální forma, normalizace, tranzitivní závislosti na klíči, pokrytí závislostí, bezeztrátová dekompozice.

Úvod

Aby schema relace splňovalo jisté požadavky (např. na malou redundanci, odstranění aktualizačních anomálnií atd.), zavádíme tzv. normální formy:

- první
- druhá
- třetí
- Boyce-Coddova
- (čtvrtá, pátá)...

Normální forma je nějaké omezení kladené na strukturu relace, zejména na závislosti mezi jednotlivými atributy, které má za cíl pokud možno optimální uložení dat v databázi. Motivaci pro normalizaci jsme uvedli na příkladu v předchozí kapitole.

První normální forma (1NF)

1NF nařizuje, že **každý atribut schématu musí být elementární a nestrukturovaný**, tj. databáze je plochá, tabulka je skutečně dvojrozměrný objekt. Tedy 1NF nedovoluje použití např. strukturovaných atributů.

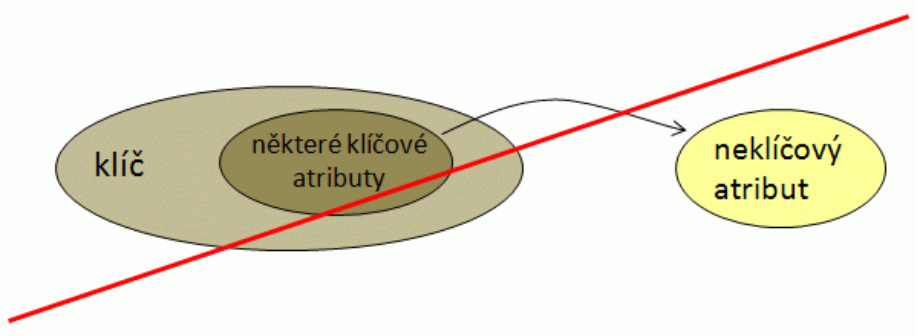
Například:

Relace OSOBA(jméno, rč, č_op) je v 1NF

Relace OSOBA(jméno, rč, adresa(město, ulice, čp)) není v 1NF, neboť obsahuje strukturovaný atribut adresa.

Druhá normální forma (2NF)

2NF říká, že **neexistují závislosti atributů na podmnožině žádného klíče**.



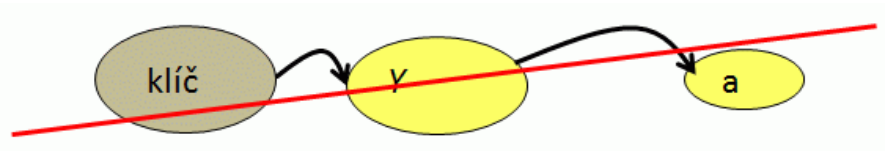
Tedy např. relace PROGRAM(<u>název_k, jméno_f</u>, adresa, datum) není v 2NF, neboť platí, že

název_k → adresa

a tedy neklíčový atribut adresa je funkčně závislý na části klíče (název kina). Motivační příklad z předchozí kapitoly tedy obsahoval relaci, která nebyla ve 2NF a tudíž vykazovala řadu nedostatků (viz předchozí kapitola).

Třetí normální forma (3NF)

Třetí normální forma je nejdůležitější. Říká, že **žádný neklíčový atribut není tranzitivně závislý na žádném klíči**.



Jelikož dle definice se 3NF testuje poměrně obtížně (vyžaduje výpočet uzávěru FZ), byla nalezena sada jednoduchých pravidel, jimiž je splnění 3NF ověřitelné velmi snadno. Třetí normální forma je splněna, platí-li pro každou funkční závislost $X \rightarrow a$ (kde $X \subseteq A$, $a \in A$) aspoň jedna z podmínek

- závislost je triviální
- X je nadklíč (tj. buď je to některý z kandidátních klíčů relace nebo ještě něco víc)
- a je částí nějakého klíče (tj. libovolného kandidátního klíče)

Postupujeme tedy tak, že zkoumáme postupně všechny funkční závislosti, platné pro danou relaci, a testujeme, zda je splněna aspoň jedna z uvedených podmínek. Pokud pro některou FZ neplatí ani jedno testovací pravidlo, pak relace není ve třetí normální formě.

Příklad

Mějme relaci zobrazenou jako tabulku:

Firma	**Kraj**	**Sídlo**
Agrostroj	Vysočina	Pelhřimov
KESAT	Vysočina	Jihlava
Chovanec Group	Pardubický	Pardubice

Na relaci víme, že platí

Firma → *všechno*

Sídlo → Kraj

Tato relace je ve 2NF, není ale ve 3NF (existuje tranzitivní závislost Kraje na klíči přes Sídlo). Důsledek je redundance hodnot atributu Kraj. Aplikujeme-li testování pomocí výše uvedených pravidel, pak pro první funkční závislost je splněna podmínka, že levá strana FZ je nadklíč (atribut Firma je klíčem, tedy je i nadklíčem), tato závislost tedy 3NF neporušuje. To není překvapivé, uvědomíme-li si, že jde o závislost neklíčových atributů na celém klíči. Pro druhou FZ ale neplatí ani jedno pravidlo: FZ není triviální, Sídlo není nadklíč, Kraj není součástí klíče. Tedy tato FZ porušuje 3NF.

Řešení: dekomponujeme relaci na dvě relace:

Firma	**Sídlo**
Agrostroj	Pelhřimov
KESAT	Jihlava
Chovanec Group	Pardubice

Sídlo	**Kraj**
Pelhřimov	Vysočina
Jihlava	Vysočina
Pardubice	Pardubický

V první relaci pak platí jediná FZ (Firma → Sídlo), na druhé relaci platí rovněž jediná FZ (Sídlo → Kraj). Obě schemata relací jsou tedy ve 3NF.

Boyce-Coddova norm. forma (BCNF)

BCNF je ještě silnější než 3NF. Říká, že **každý atribut** (včetně klíčových) **je netranzitivně závislý přímo na klíči**. Tedy pro každou netriviální závislost X → Y platí, že X obsahuje klíč schematu. Podobně jako v případě 3NF je k dispozici sada testovacích pravidel. Dané schema relace je v BCNF,

platí-li pro funkční závislost $X \rightarrow a$ (kde $X \subseteq A$, $a \in A$) aspoň jedna z podmínek

- závislost je triviální
- X je nadklíč

Srovnáním s pravidly pro testování 3NF vidíme, že jedna z podmínek zmizela, tedy BCNF je přísnější. Nesplnění BCNF opět vede k redundanci dat v databázi, nicméně v praxi se obvykle stačí omezit na 3NF.

Pokrytí závislostí a bezetrátovost

Zatím jsme normálními formami řešili problém aktualizačních anomálií a redundance. Také normalizace ale může přinášet problémy, neboť nevhodně provedená normalizace může vést k dalším potížím. Po normalizaci je totiž nutné, aby normalizovaná schemata

- měla stejnou sémantiku (pokrytí závislostí)
- obsahovala stejná data (bezeztrátové spojení)

Pokrytí závislostí

Vysvětleme požadavek pokrytí FZ. V původním schematu relace (před dekompozicí) existovala množina funkčních závislostí (pojmenujme ji F). Po dekompozici má každé nově vzniklé *i*-té schema svou vlastní množinu FZ, pojmenujme je F_i. Po dekompozici musí platit, že množina funkčních závislostí na nových schematech musí tvořit pokrytí původní množiny F, tj. všechny původní FZ musí být i nadále odvoditelné:

$$F+ = (\cup_{i=1..n} F_i)+$$

Slovně: sjednotíme-li všechny dílčí množiny F_i a ze sjednocení vytvoříme uzávěr, musí být tento uzávěr roven uzávěru původní množiny funkčních závislostí F.

Příklad

Mějme relaci

Firma	**Kraj**	**Sídlo**
Agrostroj	Vysočina	Pelhřimov
KESAT	Vysočina	Jihlava
Chovanec Group	Pardubický	Pardubice

s funkční závislostí *Sídlo* $\rightarrow$ *Kraj*.

Tato relace není ve 3NF, neboť Kraj je tranzitivně závislý na klíči přes atribut Sídlo. Provedeme nyní dekompozici na dvě relace:

Firma	**Sídlo**
Agrostroj	Pelhřimov
KESAT	Jihlava
Chovanec Group	Pardubice

Firma	**Kraj**
Agrostroj	Vysočina
KESAT	Vysočina
Chovanec Group	Pardubický

Každá z relací je ve třetí normální formě, neboť obsahuje pouze jedinou závislost neklíčového atributu na klíči. Přesto je dekompozice nevhodná: původní závislost *Sídlo* → *Kraj* již na nových relacích není odvoditelná: sjednocení množin FZ dekomponovaných relací je {*Firma*→*Sídlo*, *Firma*→*Kraj*}, a z těchto dvou FZ nelze závislost *Sídlo* → *Kraj* nijak odvodit. Správná dekompozice vypadá takto:

Firma	**Sídlo**
Agrostroj	Pelhřimov
KESAT	Jihlava
Chovanec Group	Pardubice

Sídlo	**Kraj**
Pelhřimov	Vysočina
Jihlava	Vysočina
Pardubice	Pardubický

Při této dekompozici jsou všechny FZ zachovány.

Bezeztrátová dekompozice

Definujme nyní bezetrátovou dekompozici. Požadavek bezeztrátovosti říká, že původní relační schema (před dekompozicí) by se mělo dát zrekonstruovat z dekomponovaných schemat pomocí přirozeného spojení. Tedy pro každou přípustnou relaci R‘ by mělo platit

$$R' = *_{i=1..n} R'[A_i],$$

kde A_i jsou množiny atributů dílčích relací po dekompozici. Tedy spojení projekcí původní relace na množiny atributů nových relací musí vést k původní relaci.

Příklad

Mějme relaci PROGRAM

Film	**Kino**	**Datum**	**Město**
Big Time	Aero	1.3.2000	Praha
Samotáři	Dukla	2.5.2005	Jihlava
Čokoláda	Světozor	18.4.2009	Praha

Tato relace porušuje 2NF (a tedy i 3NF) v důsledku závislosti města na kinu. Proveďme dekompozici na dvě relace

Film	**Datum**	**Město**
Big Time	1.3.2000	Praha
Samotáři	2.5.2005	Jihlava
Čokoláda	18.4.2009	Praha

Kino	Město
Aero	Praha
Dukla	Jihlava
Světozor	Praha

Obě vzniklé relace jsou v 3NF. Pokud nyní provedeme jejich přirozené spojení, dostaneme ovšem relaci s prvky, které v původní relaci nebyly:

Film	Kino	Datum	Adresa kina
Big Time	Aero	1.3.2000	Praha
Big Time	Světozor	1.3.2000	Praha
Samotáři	Dukla	2.5.2005	Jihlava
Čokoláda	Světozor	18.4.2009	Praha
Čokoláda	Aero	18.4.2009	Praha

Jak zajistit bezeztrátovou dekompozici? Mějme schema R(A,B,C), kde A, B, C jsou disjunktní množiny atributů, a funkční závislost B→C. Pak dekompozice na R1(B,C) a R2(A,B) je bezeztrátová. Naopak, je-li dekompozice R1(B,C) a R2(A,B) bezeztrátová, musí platit buď B→C nebo B→A. Jednoduše řečeno, při dekompozici vždy rozkládáme relaci podle kritické funkční závislosti, která porušuje 3NF. Tento princip vždy zaručí bezetrátovost, upozorněme ale, že nemusí zaručit pokrytí funkčních závislostí!

Návrh schematu databáze

Přihlédneme-li nyní k požadavku, aby databázové schema obsahovalo pouze normalizované relace (tabulky), lze využít dva způsoby návrhu:

1) na prvotní množině relačních schemat (vzniklé např. převodem z E-R modelu) provádíme normalizaci na jednotlivých schematech
2) vezmu všechny atributy všech relací a pomocí funkčních závislostí mezi nimi rozpoznávám entity

První způsob představuje riziko přílišného rozdrobení modelu, neboť každá relace se může teoreticky rozpadnout na mnoho dílčích relací, což snižuje efektivitu práce s databází. Druhý způsob má sice menší riziko rozdrobení, modelujeme ale na úrovni atributů, což není moc intuitivní, navíc v tomto případě víceméně ignorujeme konceptuální model. Přístupy lze nicméně různě kombinovat a dosahovat tak pokud možno lepších výsledků.

Důležité je poznání, že ne vždy je normalizace výhodná. Vznik většího množství menších tabulek vede k nutnosti častěji používat operaci spojení, která je výpočetně náročná. Může být tedy z hlediska výkonu databáze výhodnější, aby některé relace nebyly normalizované. Tím sice může narůst objem ukládaných dat a redundance, ale databáze bude mít rychlejší odezvu. Při návrhu databáze je tedy nutné volit optimální variantu z hlediska použití a výkonu. Dalším faktem je, že ne vždy se podaří provést normalizaci, která by splňovala jak podmínku pokrytí FZ, tak podmínku bezeztrátovosti. V takovém případě pak musíme ponechat relaci v původním tvaru nebo provést normalizaci pouze částečně.

Kontrolní otázky

- Co to je normální forma a k čemu je dobrá?
- Jaké znáte normální formy a jak se testují?
- Jak se provádí normalizace?
- Co znamená požadavek na pokrytí funkčních závislostí při dekompozici?
- Co znamená požadavek na bezeztrátovou dekompozici?

Úlohy k procvičení

1. Mějme schema R(**A**), v něm klíč K, nějakou množinu atributů Y a jednoduchý atribut C. Nalezli jsme závislost K→Y→C. Za jaké podmínky není C tranzitivně závislé na klíči?

---

2. Mějme jednoduchou evidenci návštěv u lékaře, evidujeme rodné číslo pacienta, číslo lékaře, datum návštěvy, adresu pacienta a odbornost lékaře. Všechno máme zpočátku v jediné relaci:

   NÁVŠTĚVA(<u>RČ_PAC, Č_LÉK, DATUM</u>, ADRESA_PAC, ODBORNOST).

- Identifikujte funkční závislosti.
- Ověřte, zda je schema ve 3NF. Pokud není, proveďte dekompozici.
- Nad původním i upraveným schematem formulujte dotazy: najdi pacienty, kteří bydlí v Praze; najdi lékaře, kteří ošetřili pacienty z Prahy; najdi všechny odbornosti lékařů, kteří ošetřili pacienty z Prahy

---

3. Mějme relaci R(A,B,C,D) a množinu F = {AB→C, A→D, D→B}. Určete klíč(e), spočítejte minimální pokrytí (viz úlohy k předchozí kapitole). Určete, zda je schema ve 3NF. Pokud není, proveďte dekompozici do 3NF a ověřte, zda dekompozice zachovává pokrytí závislostí a bezeztrátovost.

---

4. Mějme relaci R(A,B,C,D,E) a množinu F = {B→D, E→C, AC→D, CD→A, BE→A}. Určete klíč(e), spočítejte minimální pokrytí (viz úlohy k předchozí kapitole). Určete, zda je schema ve 3NF. Pokud není, proveďte dekompozici do 3NF a ověřte, zda dekompozice zachovává pokrytí závislostí a bezeztrátovost.

---

5. Mějme relaci R(A,B,C,D) a množinu F = {A→B, BC→D, D→BC, C→A}. Určete klíč(e), spočítejte minimální pokrytí (viz úlohy k předchozí kapitole). Určete, zda je schema ve 3NF. Pokud není, proveďte dekompozici do 3NF a ověřte, zda dekompozice zachovává pokrytí závislostí a bezeztrátovost.

7 Pokročilejší rysy jazyka SQL (1)

Cíl kapitoly

Kapitola staví nad poznatky o jazyce SQL, které student získal v předmětu Databázové systémy 1. Využívá se databázového stroje PosgreSQL. Cílem této kapitoly je

- naučit se pracovat s různými datovými typy
- seznámit se s vybranými funkcemi a operátory
- seznámit se s dalšími konstrukty jazyka SQL, jako je podmíněné větvení, klauzule LIMIT a OFFSET, COALESCE, a pod.
- naučit se pracovat s pohledy
- seznámit se s principem dědičnosti v PostgreSQL
- seznámit se s principem schemat v PostgreSQL

Klíčové pojmy

Datový typ, agregační funkce, operátor, pohled, schema, dědičnost.

Datové typy PostgreSQL

PostgreSQL nabízí širokou škálu nativních datových typů:

- číselné (numeric)
- znakové (character)
- binární (binary)
- datum a čas (date and time)
- boolean
- výčtové (enumerated)
- geometrické (geometric)
- pole (arrays)
- kompozitní datové typy (composite types)
- řetězce bitů (bit strings)
- a další... (síťové adresy, XML, měnové, OID,...)

Navíc lze přidávat a konstruovat další datové typy.

Číselné datové typy

K dispozici jsou celočíselné datové typy různé délky

- integer

- serial – automatické číslo
- čísla s plovoucí řádovou čárkou
    - real
    - speciální hodnoty Infinity, -Infinity a NaN
- desetinná čísla s pevnou řádovou čárkou
    - decimal (totéž co numeric) – velmi pomalé operace, přesnost až 1000 cifer

Znakové datové typy

- char(n) – řetězec délky přesně n
- varchar(n) – řetězec proměnné délky, max n
- text – řetězec neomezené délky

Datový typ char je v případě potřeby doplněn zprava na stanovenou délku mezerami, tyto mezery nejsou signifikantní, tj. neberou se v potaz. Naproti tomu varchar se ukládá přesně tak, jak je, tzn. že případné mezery JSOU signifikantní. Char neumožňuje uložit např. byte s hodnotou 0.

Binární datové typy

- bytea – řetězec bytů proměnné délky

Tento datový typ je analogický datovému typu BLOB z SQL standardu. Některé znaky se musí zadávat pomocí tzv. escape sekvencí ve tvaru

E'\\000'

Datové typy pro datum a čas

- timestamp – časové razítko
- time
- interval

Tyto typy umožňují též uchovávat informaci o časovém pásmu a zvládají různé formáty data a času:

- 1999-01-08, January 8, 1999, 1/8/1999, 990108,...

Jsou definovány speciální hodnoty:

- now
- today, tomorrow, yesterday
- infinity, -infinity

Datový typ boolean

- boolean

Kromě slov TRUE/FALSE lze použít rovněž

- t/f

- y/n
- yes/no
- on/off
- 1/0

PostgreSQL standardně vypisuje t/f.

Výčtové datové typy

- enum

Výčtový datový typ je nutno vytvořit, poté se použije dle jména typu:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Ve výčtovém typu je definováno uspořádání a to podle pořadí, v jakém byly prvky definovány. Pozor, i když více výčtových typů obsahuje tentýž identifikátor, nelze je porovnávat relačními operátory!

Geometrické datové typy

- point
- line
- box
- circle
- různé 2D objekty

Bitové řetězce

Bitové řetězce jsou posloupnosti cifer 0 a 1. Deklarují se podobně jako char a varchar:

- bit(n)
- bit varying(n)

Při vkládání do tabulky používáme escape sekvenci s B na začátku:

```
INSERT INTO test VALUES (B'101', B'00');
```

Pole

Pole se deklaruje pomocí hranatých závorek za použitým datovým typem: []. Pole lze použít skoro na jakýkoli datový typ: text[][]

Pole má velmi mnoho možností použití, jedna z nich je implementace vícehodnotového atributu.

```
CREATE TABLE sal_emp (
        name text,      pay_by_quarter integer[],       schedule text[][] );
```

Při vkládání hodnot do pole lze použít buď konstruktoru ARRAY nebo systému svorek a uvozovek. Doporučuji spíše konstruktor ARRAY, neb je přehlednější – pro každou dimenzi pole jej použijeme zvlášť.

```
INSERT INTO sal_emp
        VALUES ('Bill',
        ARRAY[10000, 10000, 10000, 10000],
        '{{"meeting", "lunch"}, {"training", "presentation"}}');
```

K prvkům pole se přistupuje pomocí indexů, lze používat interval indexů

```
SELECT pay_by_quarter[2:3] FROM sal_emp;
```

Kompozitní datové typy

Tyto datové typy jsou v podstatě obdoba tabulky. Mají velmi mnoho možností použití. Přístup k prvkům se děje tečkovou notací. Chceme-li ad hoc vytvořit jeden prvek kompozitního typu (tj. vlastně řádek tabulky), použijeme příkaz ROW:

```
ROW(1, 2.5, 'this is a test');
```

Kompozitní typ lze využít k implementaci strukturovaného skupinového atributu. Podobně jako výčtový typ, musí se kompozitní typ definovat:

```
CREATE TYPE complex AS (
        r    double precision,
        i    double precision
);
```

Přetypování

Datové typy lze do jisté míry převádět jeden na druhý. Existují implicitní konverze i explicitní konverze, které se nařizují příkazem

```
CAST (expression AS type)
```

Jako alternativu lze použít zápis

```
expression::type
```

Například převod čísla na textový řetězec můžeme provést takto:

```
CAST ( 12.0 AS text );
```

nebo

12.0::text;

Dostupné funkce a operátory

V PostgreSQL existují myriády vestavěných funkcí a operátorů pro všemožné operace nad datovými typy. V následujícím přehledu uvádíme některé z nich. Používáme přitom notaci, že termy uvedené v hranatých závorkách [] jsou volitelné, termy uvedené ve svorkách {} a oddělené svislým pruhem | jsou alternativa, tj. vybere se právě jeden z těchto termů.

Logické operátory

AND, OR, NOT, ...

Relační operátory

<, >, =, >=, <=, <> nebo !=

Testování příslušnosti do intervalu

x [NOT] BETWEEN a AND b (testuje se vč. okrajů intervalu)

Testování hodnoty NULL

Nedefinovaná hodnota NULL se netestuje relačním operátorem =, nýbrž speciálním operátorem IS NULL nebo IS NOT NULL. Je to proto, že hodnota NULL je nedefinovaná, a jako taková se nemůže ničemu rovnat, ani jiné nedefinované hodnotě.

x IS [NOT] NULL

Testování logické hodnoty

x IS [NOT] { TRUE | FALSE | UNKNOWN }

Matematické operátory

+, -, *, /, % (modulo), ^ (mocnina), ! (faktoriál), ...

Bitové operátory

& (and), | (or), # (xor), ~ (not), <<, >> (posuny)

Matematické funkce

abs, random, exp, div, log, pi, sqrt, sign, sin, cos, ...

Řetězcové funkce a operátory

s1 || s2 - zřetězení (je akceptován i jeden neřetězec, provede se konverze)
length, substring, overlay, chr, replace, strpos, ...

Podobnost

str [NOT] LIKE maska

V masce používáme zástupné znaky % a _, přitom % zastupuje libovolný i prázdný řetězec, znak _ zastupuje právě jeden libovolný znak. Kromě této možnosti existuje i mechanismus vyhodnocování regulárních výrazů.

Agregační funkce

Již víme, že na seskupenou tabulku (pomocí klauzule GROUP BY) lze aplikovat agregační funkce. Například:

```
SELECT MAX(délka) AS 'Nejdelší' FROM Silnice

SELECT COUNT(DISTINCT délka) FROM Silnice
```

PostgreSQL má řadu agregačních funkcí:

avg, max, min, count, sum, bool_and, bool_or

a další statistické funkce (stddev, variance, corr,...)

Při použití agregačních funkcí dávejte pozor na hodnoty NULL a nulový počet řádků. Například když je jako vstup agregační funkce použit nulový počet řádků, funkce count(*) vrací 0, ale funkce sum(x) vrací NULL!

Další funkce

- funkce pro formátování data a času
- funkce pro XML
- funkce pro síťové adresy
- funkce a operátory pro datum a čas
    - sčítání, odečítání, násobení, dělení
    - now, current_date, current_time, timeofday, ...
    - extract:

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
```

- funkce pro operace nad ENUM
- funkce a operátory pro geometrické typy
    - translace, rotace, změna měřítka, vzdálenost, překryv, vzájemná poloha, rovnoběžnost...
    - plocha, výška, šířka, typ cesty, ...

- funkce a operátory nad poli
    - porovnávání (lexikograficky), test podmnožiny, test průniku, zřetězení (||), array_fill, array_length, unnest, generate_subscripts, ...
- systémové funkce
    - current_database, user, version, ...
- zjišťování práv a spousta dalších věcí

Podmíněné výrazy

V PostgreSQL je možné neprocedurální vyhodnocení podmínek, např. pro zajištění integritních omezení. Konstrukt CASE zajišťuje běžné podmíněné větvení:

```
 SELECT a,
        CASE    WHEN a=1 THEN 'one'
                WHEN a=2 THEN 'two'
                ELSE 'other'
        END
FROM test;
```

nebo méně obecná verze, předpokládající test na rovnost

```
SELECT a,
        CASE a WHEN 1 THEN 'one'
               WHEN 2 THEN 'two'
               ELSE 'other'
        END
FROM test;
```

Podmínku CASE lze použít též při tvorbě tabulky pro definici integritních omezení.

Dalším podmíněných výrazem je konstrukt COALESCE, který vybere ze zadaného seznamu první hodnotu, která není NULL. Použití: řešení případů, kdy očekáváme NULL. Například:

```
COALESCE(Vypujcky.Cena, "Cena neni urcena");
```

Pokud je hodnota atributu Vypujcky.Cena jiná než NULL, vrátí se tato hodnota, jinak se vypíše hláška.

Klauzule LIMIT a OFFSET

PostgreSQL má možnost, jak vrátit jen podmnožinu řádků dotazu

```
SELECT select_list
        FROM table_expression
        [ ORDER BY ... ]
        [ LIMIT nlim ] [ OFFSET noffs ];
```

Klauzule LIMIT vrací nejvýše nlim řádků dotazu počínaje noffs+1 řádkem. Není-li OFFSET zadán, předpokládá se noffs=0. **Pozor:** v dotazu je důležité použít řazení, aby měl příkaz nějaký smysl, protože obecně je pořadí řádků vracených SELECTem nepředvídatelné.

Pohledy

Pohled je vlastně virtuální tabulka, jejíž obsah je dynamicky generován z tabulek, nad nimiž je dotaz vystavěn. Pohledy umožňují pohled na databázi ušít na míru uživateli, umožňují zejména

- odstínění nežádoucích dat
- odstínění změn DB
- předvypočtení souhrnů

Pohledy se využívají pro dotazy, v PostgreSQL je nelze aktualizovat (nelze do nich vkládat data a upravovat je). Pohledy lze používat pro konstrukci dalších pohledů, může tak vznikat vícevrstvá struktura.

```
CREATE VIEW Prazaci AS
        SELECT    * FROM Zakaznici
        WHERE adresa LIKE '%Praha%';

CREATE OR REPLACE VIEW PoctyVypujcek AS
        SELECT    rc, COUNT(c_kopie) FROM Vypujcky
        GROUP    BY rc;
```

Dědičnost

Tabulka může být vytvořena jako odvozenina (potomek) jiné tabulky (rodičovské, mateřské). Takový potomek dědí

- všechny sloupce rodiče
- některá integritní omezení: check, not null

Potomek naopak nedědí

- zbývající IO: unique, primary key, foreign key
- přístupová práva

Záludnosti

Dědičnost nezajistí žádnou pevnou vazbu mezi tabulkami, jako např. unikátní primární klíč přes sjednocení záznamů (tj. nelze tím automaticky zajistit realizaci entitního podtypu ISA hierarchie). Rovněž vkládání dat (INSERT) se děje jen na konkrétní tabulku, čili vkládání do rodiče nezajistí automaticky vložení do potomka. Na druhou stranu, dědičnost lze využít k výběru dat (SELECT) přes celý strom dědičnosti, viz příklady.

-- databáze měst, z nichž chceme ještě vydělit hlavní města

```
CREATE TABLE Cities (   name text,
                        population float,
                        altitude int );

CREATE TABLE Capitals ( state text ) INHERITS (cities);
```

-- vybereme záznamy z tabulky cities i capitals (tj. vč. potomků)

```
SELECT name, altitude FROM cities WHERE altitude > 500;
```

-- vybereme záznamy jen z tabulky cities

```
SELECT name, altitude FROM ONLY cities WHERE altitude > 500;
```

Schemata

Databáze v PostgreSQL je členěna do hierarchické struktury. Nejvyšší objekt je cluster, cluster obsahuje pojmenované databáze. Databáze obsahuje pojmenovaná schemata, schema obsahuje tabulky a další objekty. Dvě schemata mohou obsahovat tabulku stejného jména.

Schemata nejsou tak striktně oddělena jako databáze, uživatel (připojí-li se k DB) může používat všechna, má-li na to oprávnění. Schemata není možné vnořovat

Schemata

- pomáhají oddělovat různé skupiny uživatelů
- člení databázi na logické celky
- umožňují uchovávat objekty třetích stran (aplikace apod.), aby nekolidovaly identifikátory

Vytváření a rušení schemat:

```
CREATE SCHEMA myschema;
CREATE SCHEMA myschema AUTHORIZATION username;
  --majitelem nového schematu bude username
```

```
DROP SCHEMA myschema    [CASCADE];
--CASCADE nařídí odstranění všech vnořených objektů
```

Implicitně jsou všechny objekty umístěny ve schematu public. Přístup k vnořeným objektům se děje přes tečkovou notaci:

```
CREATE TABLE myschema.mytable (...);
```

Tečková notace bývá nepohodlná a obtěžující, proto se často nepíše – bez jejího použití prohledá stroj podle nějakého algoritmu seznam schemat (search path) a první vyhovující tabulka se vezme. Prohledávací seznam schemat se dá prohlížet příkazem

```
SHOW search_path;
```

a dá se nastavit pomocí

```
SET search_path TO myschema, public;
```

Search path implicitně začíná uživatelským schematem.

Uživatel může pracovat jen se schematem, které vlastní, nebo je k tomu nutné oprávnění USAGE (viz přednáška o oprávněních). Standardně mají všichni oprávnění CREATE a USAGE na schema PUBLIC, i to lze ale změnit:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

Způsoby použití

- ponechat pouze schema public
    - emuluje situaci, kdy schemata vůbec nejsou
- pro každého uživatele vytvořit jeho schema
    - jelikož se search path prohladává od začátku, uživatel bude pracovat (bez tečkové notace) s vlastním schematem
- pro instalované aplikace vytvořit vlastní schema
    - nutno zajistit patřičná oprávnění
    - uživatelé buď mohou používat kvalifikovaná jména (tečkovou notaci) nebo si zařadit schema do své search path

Kontrolní otázky

- K čemu se dá použít datový typ pole?
- K čemu se dá použít kompozitní datový typ?
- Jak se používají agregační funkce?
- Jak se používají podmíněné výrazy?
- K čemu je klauzule LIMIT a OFFSET?
- Co to je pohled a k čemu se používá?

Úlohy k procvičení

Při práci na úlohách používejte dokumentaci databázového stroje PostgreSQL, která je dostupná např. na internetu na adrese http://www.postgresql.org/docs/8.4/static/index.html.

---

1. Vytvořte tabulky pro jednoduchou evidenci knih v různobarevných regálech:

   regal(id_regalu, barva)
   kniha(id_knihy, nazev, autori, zanr, id_regalu)

   kde atribut autori bude řešen jako vícehodnotový atribut (použijte implementaci polem). Atribut id_regalu v tabulce kniha je cizí klíč odkazující na tabulku regal, tento cizí klíč bude mít nastaveno povinné členství knihy ve vztahu s regálem. Primární klíče implementujte datovým typem integer, pro ostatní atributy použijte varchar.

---

2. Do tabulek vložte nějaká data (nejprve do regálu, pak do knihy, jinak bude porušena referenční integrita). Určitě mějte aspoň tyto barvy regálu: modrý, zelený. V každé z těchto barev mějte přiřazenu aspoň jednu knihu.

---

3. K některé knize přidejte další dva autory, jednoho na začátek a druhého na konec seznamu (využijte funkce array_append, array_prepend, array_length). Následně tyto autory zase odeberte (jak ze začátku, tak z konce). Nakonec všechny autory u této knihy vymažte.

---

4. V databázi chceme také uchovávat rozměr regálu, definujte tedy nový kompozitní datový typ VySiHl se třemi komponentami (vyska, sirka, hloubka), všechny budou typu real. K tabulce regal přidejte další sloupec rozmery datového typu VySiHl. Vložte vhodná data v cm. Následně u modrého regálu zvětšte výšku o 20cm (nápověda: pokud se chcete v příkazu UPDATE odvolávat na původní hodnotu měněného atributu, musíte název atributu (zde rozmery), za rovnítkem ozávorkovat, jinak to překladač pochopí jako tabulku).

---

5. Vytvořte jednoduchý překladač pro anglické uživatele. Napište příkaz select, který bude při výběru knihy překládat žánr: román=novel, básně=poems, povídky=short stories, sci-fi=sci-fi, vojenské=military, pro ostatní žánry vrátí „unknown genre“.

---

6. Přidejte k tabulce regal sloupec kapacita (integer), který bude mít definováno integritní omezení, že musí být větší než nula. Toto IO je splněno, i když je ve sloupci NULL. Když nyní příkazem select vypíšete všechny záznamy, bude ve sloupci kapacita všude NULL. Upravte příkaz select tak, aby pro situaci, kdy je ve sloupci kapacita NULL vypsal -1.

8 Pokročilejší rysy jazyka SQL (2)

Cíl kapitoly

Kapitola pokračuje v představování pokročilejších rysů jazyka SQL, implementovaných v databázovém stroji PostgreSQL. Cílem této kapitoly je

- seznámit se s konceptem transakčního zpracování a naučit se transakce používat
- seznámit se s principy paralelního zpracování dat
- seznámit se s principy zotavení z chyb
- seznámit se s konceptem indexů
- seznámit se se způsobem udělování uživatelských oprávnění
- seznámit se s konceptem uživatelských rolí

Klíčové pojmy

Transakce, paralelní zpracování, zotavení z chyb, index, uživatelské oprávnění, uživatelská role.

Transakce

Transakce představují koncept, jak zajistit paralelní přístup k datům v databázi. Jsou rovněž prostředkem pro zotavení z chyb. Myšlenka transakce je následující: databázové operace se vykonávají v rámci nedělitelných programových jednotek – transakcí. Transakce je nedělitelná, buď se provede celá nebo vůbec, a jako atomická jednotka práce s databází převádí DB z jednoho konzistentního stavu do jiného konzistentního stavu. Dokud není transakce potvrzena, dílčí změny nejsou viditelné pro ostatní transakce, naproti tomu efekty úspěšné transakce jsou již trvalé. Řízení transakcí je převážně automatické, explicitně lze transakce řídit pomocí příkazů COMMIT a ROLLBACK. Pro zajištění chodu transakcí je nutné žurnálování (logging, viz dále).

Transakce je vlastně posloupnost akcí (čtení, zápis, výpočet...), s nimiž se zachází jako s jedním celkem. Jako příklad uveďme typický úkol – převod peněz z účtu na účet. V tomto úkolu se provádějí dvě dílčí operace:

- odečíst peníze ze zdrojového účtu
- přičíst peníze na cílový účet

Aby databáze zůstala v konzistentním stavu, musí proběhnout buď obě dílčí operace nebo žádná.

Transakce se může nacházet v různých stavech:

- aktivní – počátek provádění transakce
- částečně potvrzený – těsně před COMMIT
- chybný – vyskytla se chyba
- zrušený – po příkazu ROLLBACK
- potvrzený – po příkazu COMMIT

Je třeba říci, že transakce je možné explicitně definovat, nicméně pokud je nějaká operace nad databází zavolána bez „zabalení“ do transakce, databázový stroj stejně zavede transakci, obsahující tuto jedinou operaci. Tedy transakce jsou používány automaticky při jakékoli manipulaci s daty v databázi, nejen na vyžádání. Z toho též plyne, že každý příkaz je zapisován do žurnálu.

Paralelní zpracování

Není efektivní všechny transakce provádět za sebou (sériově), proto je snaha je vzájemně prokládat. Tím ovšem vzniká závislost jedné transakce na druhé. Uvědomme si přitom, že transakce nevidí změny, které právě provádí jiná transakce. Je zřejmé, že některé typy operací nelze libovolně prokládat, jmenujme např. čtení a zápis téhož objektu. Při prokládání transakcí je tedy nutné dodržení serializovatelnosti, tj. takové proložení operací, které vede ke stejnému výsledku, jako kdyby byly transakce prováděny za sebou. Serializovatelnost lze řešit algoritmicky, ale je to náročné.

Při prokládání transakcí jsou konfliktní především operace typu READ a WRITE. Řešením konfliktů je uzamykání objektů, k nimž transakce přistupuje. Existují různé protokoly zamykání:

Jednoduchý uzamykací protokol

- transakce zamkne objekt, chce-li jej používat
- transakce nezamyká již zamčený objekt
- transakce může odemknout jen to, co sama zamkla
- po ukončení transakce jsou všechny objekty odemknuty

Dvoufázový protokol

- v první fázi se postupně zamykají potřebné objekty, ale nic se neodemyká
- v druhé fázi se objekty postupně odemykají, ale žádný se již nesmí zamknout

Dvoufázový protokol zajišťuje uspořádatelnost, ovšem hrozí při něm velké nebezpečí uváznutí (deadlocku), kdy dvě transakce na sebe vzájemně čekají. Uzamykání objektů je možné na různé úrovni detailu (granularity). Nejhrubější je uzamknout celý systém, což je ekvivalentem sériového provádění transakcí. Jemnější je zamykání tabulek, nejjemnější pak zamykání řádků v tabulkách (record locking). Jemné uzamykání je náročnější a alokuje velké množství zdrojů, proto někdy databázový stroj přistupuje k tzv. eskalaci zámků, kdy snižuje granularitu zamykání tak, že např. namísto mnoha zamčených řádků jedné tabulky zamkne celou tuto tabulku.

Prokládání transakcí vede k několika typickým problémům:

- Lost update – dvě transakce po sobě zapíší různé hodnoty do téhož objektu. Potom je efekt první transakce ztracen.
- Dirty read – první transakce zapíše hodnotu do objektu, druhá transakce si tuto zapsanou hodnotu přečte, načež první transakce provede ROLLBACK, hodnota přečtená druhou transakcí tedy nebyla definitivní.
- Unrepeatable read – první transakce přečte hodnotu, druhá transakce tuto hodnotu změní, takže další čtení této hodnoty první transakcí vrátí jiný výsledek než poprvé
- Phantom problem – první transakce čte data z tabulky (provede např. SELECT), mezitím do této tabulky jiná transakce vloží nové řádky; první trasakce při dalším provedení téhož dotazu vrátí jiná data. Tento problém je speciálním případem neopakovatelného čtení (unrepeatable read).

Existují různé strategie na omezování výše uvedených nežádoucích jevů. Při řízení transakcí lze nastavit tzv. isolation level, tj. stupeň izolace, který určuje, které z problémů budou eliminovány. Čím vyšší stupeň izolace, tím lepší ochrana před anomáliemi, ovšem tím horší prostupnost databáze pro víceuživatelské operace.

Zotavení z chyb

Transakce je i jednotkou zotavení z chyb. Pokud dojde v průběhu transakce k chybě, vrátí se DB do stavu před transakcí. K tomu je nutné vést tzv. žurnál (log-file), do něhož se zapisují všechny akce, které se provádí, aby bylo možné je případně vrátit.

Žurnálování je možné zajistit dvěma způsoby:

- Odložená realizace změn – do žurnálu se zapisují operace, které ale nejsou prováděny ihned. Pokud všechny simulované transakce dopadnou dobře a jsou potvrzeny, změny se stabou viditelnými pro ostatní uživatele a mohou být zapsány do DB. Téměř nikdy tak není nutný ROLLBACK.
- Bezprostřední realizace změn – všechny změny se zapisují do DB rovnou, ale žurnál změny detailně eviduje a v případě kolize nebo problému se provede ROLLBACK.

PostgreSQL, podobně jako jiné databázové stroje, používá odloženou realizaci změn (označuje ji jako WAL – write-ahead logging). Důležité je, že změny vyvolané transakcí se mohou stát viditelnými už ve chvíli, kdy je na disk zapsán pouze soubor žurnálu (log-file), ale změna vlastních datových souborů může být ještě dále odložena. Dotaz na databázi v tomto stavu pak vyžaduje kombinaci informací v datových souborech a v žurnálu. Uvedený princip má svůj původ v pomalosti starých diskových systémů, kdy bylo podstatně rychlejší zapsat (a přečíst) log-file, než zapisovat celý datový soubor, který mohl být mnohdy velmi rozsáhlý a fragmentovaný na mnoho fyzických míst na disku. Pokud chceme vynutit fyzické zapsání změn z žurnálu do datových souborů, pak je nutno použít samostatný příkaz CHECKPOINT. V praxi není obvykle nutné příkaz CHECKPOINT volat, vhodné je to např. před zálohováním databáze.

Transakce v PostgreSQL

V PostgreSQL se transakce řídí příkazy BEGIN, COMMIT, ROLLBACK a SAVEPOINT. Existuje automatické i uživatelsky řízené zamykání.
Např.

```
BEGIN;
        UPDATE accounts SET balance = balance + 100.00
                WHERE name = 'Bob';
        UPDATE accounts SET balance = balance - 100.00
                WHERE name = 'Alice';
COMMIT;
```

Transakce v PostgreSQL řeší i problém současného vložení řádků do dvou tabulek při povinném členství ve vztahu. V tabulce lze u cizího klíče nastavit modifikátor DEFERRABLE, který zajistí, že kontrola referenční integrity může proběhnout až na konci transakce. V každé transakci nastává kontrola referenční integrity podle toho, jak je nastaveno výchozí chování, pro aktuální transakci lze chování vždy změnit příkazem

```
SET CONSTRAINTS name DEFERRED;
```

Ukázka definice cizího klíče:

```
...
fk      integer references tbl2(pk) deferrable initially deferred,
fk2     integer references tbl3(pk) deferrable,   ...
```

Kromě modifikátoru DEFERRABLE se u cizího klíče dá nastvit výchozí chování, tj. zda bude implicitně odložena kontrola referenční integrity (initially deferred) nebo zda se bude implicitně provádět kontrola okamžitě (initially immediate – výchozí nastavení).

Indexy

Odezvu DB lze zrychlit pomocí přídavných datových struktur, které usnadní vyhledávání. Ovšem pozor, každá přídavná struktura zvyšuje zároveň zátěž SŘBD, proto se musí užívat s rozmyslem. Takovouto pomocnou strukturou je tzv. index. Index přísluší tabulce (pozn.: pohled nelze indexovat) a představuje informaci o tom, kde v tabulce leží která data. Myšlenka indexu je podobná, jako rejstřík knihy: najdeme si v rejstříku, kde leží požadovaná informace, a rovnou nalistujeme příslušnou stránku. Indexy tedy slouží k rychlému nalezení záznamů, majících požadovanou hodnotu indexovaných atributů. Toho lze využít např. v operacích SELECT, UPDATE či DELETE, významné jsou indexy pro rychlou implemetnaci spojení (JOIN).

Fyzické uspořádání souborů na disku

Fyzická organizace databázových souborů na disku se může různit. Základní typy:

- hromada
  - neuspořádaný soubor záznamů proměnné délky
  - nalezení záznamu má složitost O(N)
- sekvenční soubor
  - neuspořádaný resp. uspořádaný soubor záznamů stejné délky
  - nalezení záznamu má složitost O(N) resp. O(log N)
- indexsekvenční soubor
  - sekvenční soubor setříděný dle primárního klíče plus přídavný víceúrovňový index
  - index neukazuje na jednotlivé záznamy, ale na krátké setříděné bloky více záznamů
- indexovaný soubor
  - sekvenční soubor (nemusí být setříděný ani souvislý) plus přídavný jednoúrovňový index
  - index ukazuje na jednotlivé záznamy

Lze zavádět i víceúrovňový index, což představuje vlastně indexaci indexu. Řekněme, že indexujeme podle sloupce, který obsahuje celá čísla. Pak první úroveň indexu ukazuje na jednotlivá čísla, druhá indexuje první číslo z každé pětice indexů první úrovně (třeba), atd...

Ve vztahu k fyzickému uspořádání záznamů na disku rozlišujeme v praxi tzv. clustered a non-clustered index. První z nich (clustered) zajišťuje, že záznamy na disku jsou fyzicky uloženy ve stejném pořadí jako v indexu. Příkladem může být uspořádání záznamů vzestupně dle primárního nebo cizího klíče.
Z uvedeného plyne, že tabulka může mít pouze jeden clustered index. Clustered index má tu výhodu, že operace vyhledávání jsou velmi rychlé, na druhou stranu jakákoli změna (UPDATE) indexovaného atributu vede k nutnosti fyzicky přeuspořádat celý datový soubor, což je naopak pomalé. Clustered index lze tedy použít např. u primárního klíče, kde se nepředpokládají časté změny. Druhý typ indexu, non-clustered, je sám o sobě setříděn, odpovídající datový soubor může však mít záznamy v libovolném pořadí. Při hledání záznamu je pak třeba nejprve vyhledat odpovídající položku v indexu a poté následovat ukazatel k fyzickému záznamu na disku. Je zjevné, že tento přístup je pomalejší než vyhledávání u clustered indexu, nicméně stále může být výrazně rychlejší než prohledávání tabulky bez indexu. Tabulka může nad sebou mít více non-clustered indexů. Pro úplnost dodejme, že použití indexu databázovým strojem není dogma: při vyhodnocování dotazu může databázový stroj usoudit, že bude výhodnější index ignorovat.

Vyhledávání v indexu

Pro vyhledávání v indexu lze zjevně použít binární vyhledávací strom: každý uzel X má dva potomky, levý potomek je vždy menší než uzel X, pravý potomek je vždy větší než X. Takový strom má složitost prohledávání $O(\log_2 N)$ – to může být ale pro velká množství záznamů stále ještě příliš mnoho (uvažme opakované přístupy na disk). Proto se zavádějí tzv. B-stromy, kde každý uzel má nikoli dva, ale m potomků. To vede ke snížení hloubky stromu a tím i k omezení počtu přístupů na disk. Obvyklý počet úrovní stromu je 2-3 pro řádově stovky záznamů.

Příklad:
Chceme vyhledat jeden záznam podle jeho primárního klíče. Pokud o uspořádání záznamů v tabulce nic nevíme, musíme projít celou tabulku. Pokud máme index, lze hledání výrazně urychlit: vytvoříme index nad sloupcem, podle kterého často vyhledáváme.

V PostgreSQL lze tvořit index pro více sloupců naráz:

```
CREATE TABLE test (
        major integer,
        minor integer,
        content varchar );

CREATE INDEX test_mm_index ON test (major, minor);

SELECT content FROM test WHERE major = 5 AND minor = 10;

DROP INDEX test_mm_index;
```

PostgreSQL umí tvořit indexy různými technikami, B-strom je implicitní. Umí též analyzovat, zda je index užíván či nikoli. Zajímavé je indexování skalární funkce dat, jímž lze vynutit dodržení některých IO, např. to, že se ve sloupci nesmí opakovat stejný řetězec ať už s malými či velkými písmeny:

```
CREATE UNIQUE INDEX test_lower_col1_idx ON test (lower(col1));
```

Lze též vytvářet index jen na podmnožinu záznamů

```
CREATE INDEX orders_unbilled_idx ON orders (order_nr)
        WHERE billed is not true;
```

Oprávnění

Databázové stroje umožňují dobře definovat přístupová práva k jednotlivým databázovým objektům. Spolu s pohledy to představuje efektivní nástroj k zajištění bezpečnosti DB. Někdy je správa oprávnění označována jako samostatná podmnožina jazyka SQL – tzv. DCL.

Základní principy:
- pokud uživatel vytvoří objekt, stane se jeho vlastníkem
- standardně smí s objektem manipulovat pouze jeho vlastník
- aby mohli objekt užívat i ostatní uživatelé, musejí jim být udělena patřičná oprávnění
- admin má přístup ke všemu

Existuje několik typů oprávnění pro různé akce nad objekty (liší se podle typu objektu):

- SELECT – právo číst tabulku
- INSERT – právo vložit záznam
- UPDATE, DELETE – právo upravovat data a mazat, obvykle potřebuje též práva pro SELECT
- TRUNCATE – právo naráz mazat obsah celé tabulky
- REFERENCES – právo vytvořit cizí klíč, nutno mít přiděleno na odkazující i odkazované tabulce
- TRIGGER – právo vytvořit trigger nad tabulkou
- CREATE – právo vytvářet objekty
- CONNECT – právo připojovat se k dané DB
- TEMPORARY – právo vytvářet dočasné tabulky
- EXECUTE – právo spouštět konkrétní funkci a s ní svázané operátory (jediné právo na funkce)
- USAGE
    - pro jazyky právo používat konkr. procedurální jazyk
    - pro schemata právo vidět objekty ve schematu

Tato oprávnění lze delegovat uživatelům, avšak odstranění a modifikaci objektu smí provádět vždy jen vlastník. Specifická práva vlastníka nelze delegovat, některá lze omezit, tj. vlastník si může například nastavit tabulku jako read-only. Pomocí příkazu ALTER TABLE lze vlastníka změnit. Je také možné udělit právo udělovat práva: pokud je toto právo uživateli odebráno, všechna práva jím udělená ostatním uživatelům jsou rovněž odebrána!

Udělení privilegia

Příkaz GRANT:

```
GRANT UPDATE ON accounts TO joe;
GRANT UPDATE ON accounts TO bob WITH GRANT OPTION;
```

Na místo konkrétního privilegia můžeme zapsat ALL a uživateli jsou předána všechna práva. Pro udělení práv všem uživatelům použijeme speciálního „uživatele“ PUBLIC. Privilegia se mohou kumulovat, tj. příkaz grant pouze přidá další právo k již nastaveným. Pouze práva udělená s uvedením WITH GRANT OPTION mohou být uživatelem delegována dalším uživatelům. V právech lze jít až na úroveň sloupců v tabulkách, tj. lze udělit právo číst třeba jen jeden sloupec:

```
GRANT SELECT jméno, příjmení ON osoba TO jirka;
```

Odebrání privilegia

Příkaz REVOKE

```
REVOKE ALL ON accounts FROM PUBLIC;
```

Uživatelské role

Pro abstrakci uživatelů PostgreSQL nabízí koncept uživatelských rolí. Role může být buď jediný uživatel (to je speciální případ) nebo obecně skupina uživatelů. Rolím lze udělovat práva, role mohou vlastnit objekty (tabulky apod.).

Vytvoření, modifikace, zrušení role:

```
CREATE ROLE name;
ALTER ROLE name RENAME TO newname;
DROP ROLE newname;
```

Role může být definována s různými parametry, které specifikují práva role. Práva role lze změnit pomocí příkazu ALTER ROLE.

Typy práv:

- běžný uživatel:

```
CREATE ROLE name LOGIN;
CREATE USER name;    -- totéž
```

- admin – musím být sám admin, abych mohl tuto roli vytvořit:

```
CREATE ROLE name SUPERUSER;
```

- možnost vytváření databáze:

```
CREATE ROLE name CREATEDB;
```

- možnost vytváření dalších rolí:

```
CREATE ROLE name CREATEROLE;
```

Uživatelé mohou být přiřazování do rolí, čímž jim mohou být hromadně udílena práva. Funguje to tak, že se uživateli přiřadí role příkazem GRANT, s rolí se tedy pracuje jako s oprávněním. Nejprve vytvoříme požadovanou „skupinovou“ roli:

```
CREATE ROLE skupina CREATEDB;
```

potom přidáme člena

```
GRANT    skupina    TO    jirka;
```

Členem skupiny může být pochopitelně opět skupina, tím vzniká hierarchie rolí. Dohromady s uživatelskými privilegii tak máme k dispozici mocný nástroj na správu uživatelských oprávnění v databázi.

Aby člen skupiny měl automaticky i práva skupiny, musí být definován s atributem INHERIT, potom dědí práva každé skupiny, jejímž je členem. Tento princip funguje tedy poněkud opačně, než bychom asi čekali: dědičnost je definována zespoda, ne shora.

Například:

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel;
GRANT admin TO joe;
GRANT wheel TO admin;
```

Vytvořili jsme hierarchii uživatelských rolí, kde nejvyšší je role wheel, do níž je zařazena role admin, do níž je zařazen uživatel joe. Uživatel joe díky atributu INHERIT dědí práva nadřazených skupin, tj. v našem případě vlastní práva skupiny admin, ale *nedědí* práva skupiny wheel, protože admin také nedědí práva skupiny wheel.

Člen skupiny může ovšem v určitých případech užívat práva určité skupiny, i když nemá nastaven atribut INHERIT. Po přihlášení do databáze může za tímto účelem použít příkaz SET ROLE. Člen skupiny se může tímto příkazem „přepnout“ na práva kterékoli skupiny, jíž je členem, ztrácí tím ovšem práva, která má sám a skupina je nemá – operace není aditivní.

Příklad:

- nastavení přesně práv role admin

```
SET ROLE admin;
```

- nastavení uživatelových původních práv

```
RESET ROLE;
```

Kontrolní otázky

- Co to je transakce a jak se používá?
- Jaké jsou dvě základní využití transakcí?
- K čemu jsou indexy?
- Jaké znáte typy fyzického uspořádání souborů na disku?
- Co to je oprávnění či privilegium v databázi?
- Jak se oprávnění používají?

Úlohy k procvičení

Při práci na úlohách používejte dokumentaci databázového stroje PostgreSQL, která je dostupná např. na internetu na adrese http://www.postgresql.org/docs/8.4/static/index.html. Použijte tabulky s daty z úloh k minulé kapitole (knihy v regálech).

---

1. Vytvořte pohled „modre“, který bude uchovávat knihy z modrého regálu. Použijte klauzuli CREATE OR REPLACE. Zkuste aktualizovat pohled, tj. vložit do něj nějaká data. Zkuste v tabulce knihy přeřadit nějakou knihu do modrého regalu a sledujte, co se děje s pohledem.

---

2. Pokuste se smazat nějaký regál, na který odkazuje aspoň jedna kniha. Pokud jste na začátku deklarovali foreign key bez dalších specifikací, nemělo by se to podařit. Změňte nyní specifikaci cizího klíče v tabulce kniha tak, aby se při vymazání příslušného regálu vymazaly i záznamy o všech knihách v něm. Nápověda: musíte nejprve odebrat IO cizího klíče (název najdete v popisu tabulky kniha, bude to něco jako "kniha_id_regal_fkey") – viz nápověda k příkazu ALTER TABLE. Následně musíte přidat nové tabulkové IO, které bude obsahovat patřičnou klauzuli.

---

3. Vyzkoušejte funkčnost transakcí. Vytvořte transakci, která bude obsahovat dva příkazy: prvním budete měnit barvu modrého regálu na pomněnkovou, druhým se pokusíte vložit do tabulky kniha záznam, který bude odkazovat na neexistující regál (id_regálu, které v tabulce regal neexistuje). Po zadání příkazu COMMIT zkontrolujte, jakou barvu má modrý regál.

---

4. Představte si, že se často stává, že vkládáte novou knihu ještě předtím, než vložíte regál, v němž bude kniha uložena. Přitom se nechcete vzdát omezení, které zajišťuje povinné členství knihy ve vztahu s regálem (cizí klíč je NOT NULL). Vyřešte situaci pomocí transakce, která bude obsahovat dva příkazy: prvním vkládáme knihu do dosud neexistujícího regálu, druhým vkládáme nový odpovídající regál. Aby vše fungovalo, musíte ještě nastavit patřičnou vlastnost cizího klíče, která zajistí kontrolu referenční integrity až na konci transakce (nikoli na konci insertu samotného).

---

5. Vytvořte novou tabulku police, která bude dědit vlastnosti tabulky regal a navíc bude mít ještě atribut nosnost. Do tabulky vložte nějaká data a to tak, aby některá police byla zelená. Nyní vypište všechny regály a police, které mají zelenou barvu. Dále vypište jen zelené regály bez polic.

9 Procedurální rozšíření jazyka SQL

Cíl kapitoly

SQL je sám o sobě neprocedurání jazyk. Aby byla odstraně a některá omezení z toho plynoucí, bývají k dispozici procedurální rozšíření jazyka SQL. Tato kapitola představuje procedurální rozšíření jazyka SQL, implementovaná v databázovém stroji PostgreSQL. Cílem této kapitoly je

- naučit se, co to je procedurální rozšíření jazyka SQL
- seznámit se s typy procedurálních rozšíření v PostgreSQL
- naučit se pracovat s SQL funkcemi a s funkcemi v jazyce PL/pgSQL

Klíčové pojmy

Procedurální rozšíření jazyka SQL, SQL funkce, procedurální funkce, vstupní parametry, návratová hodnota, jazyk PL/pgSQL.

Procedurální rozšíření - motivace

Jazyk SQL je neprocedurální jazyk, říká *CO* chceme udělat, ne *JAK*. Z tohoto důvodu postrádá možnosti sofistikovanějšího řešení některých problémů, například ošetření složitějších integritních omezení. Také dotazy, které přirozeně vedou na cykly, podmínky a podobné programátorské konstrukce, se neprocedurálním způsobem řeší obtížně. Mnohdy je obtížné přemýšlet „v množinách“, řešení nás často napadá snadněji s využitím klasických paradigmat sekvenčního funkcionálního programování. Nevýhoda čistě neprocedurálního jazyka SQL je také v tom, že pro čisté SQL (na serveru) a procedurální zpracování v klientovi se zvyšuje zátěž sítě.

Procedurální rozšíření SQL znamená doplnění jazyka SQL o možnost vestavěných funkcí, procedur, triggerů a podobně. Definované funkce lze volat buď explicitně, nebo automaticky při událostech jako je vkládání dat, manipulace s daty apod. SŘBD PostreSQL disponuje procedurálním jazykem PL/pgSQL – obdoba PL/SQL firmy ORACLE.

Funkce v PostgreSQL

Typy funkcí:

- SQL funkce
- procedurální funkce (psané v PL/pgSQL nebo jiném jazyce)
- vnitřní (interní funkce)
- funkce v jazyce C

Vytváření a odstranění

Funkce se obecně vytváří příkazem CREATE FUNCTION. Tělo funkce musí být deklarováno jako řetězec, tj. uzavřeme jej například do $$. Více k vytváření jednotlivých typů funkcí bude uvedeno dále v textu.

```
CREATE FUNCTION jméno_funkce(seznam_parametrů) RETURNS návratová_hodnota AS
$$
        tělo funkce;

$$ specifikace_jazyka;
```

Odstranění funkce:

```
DROP jméno_funkce (seznam parametrů);
```

Pro bezproblémové odstranění funkce je nutné zadat nejen název funkce, ale též její parametry. PostgreSQL totiž používá přetížení funkcí, kdy funkce je identifikována nejen svým jménem, ale též seznamem datových typů svých parametrů.

Parametry funkcí

Vstupní parametry funkcí mohou používat všechny základní i kompozitní datové typy, tj. i tabulky. Funkce nemusí mít žádné vstupní parametry. Pro deklaraci seznamu parametrů funkce stačí uvést jejich datové typy, např.

```
CREATE FUNCTION mojefunkce(int, real) ....
```

Přístup k parametrům se totiž děje standardně pomocí jejich pořadí v seznamu parametrů, používáme číslování od 1. První parametr je označen $1, podobně pro další parametry. Pokud jsou parametry strukturovaného datového typu, použijeme tečkovou notaci, například $3.name. Parametry lze ale v deklaraci seznamu parametrů pojmenovat, podobně jako v deklaraci sloupce tabulky se uvádí nejprve název parametru a následně jeho datový typ:

```
CREATE FUNCTION sales_tax(subtotal real) ....
```

Pozor: pokud používáme pojmenované parametry funkce, musíme dbát na to, aby jejich jméno nekolidovalo s žádným jiným identifikátorem použitým ve funkci (název tabulky, název proměnné, apod.). Překladač PostgreSQL totiž před vykonáním funkce provede primitivní textovou náhradu jména parametru za jeho číslo. Pokud by se ve funkci vyskytoval jiný řetězec, který by byl shodný s názvem některého parametru a přitom to nebyl parametr, bude rovněž textově nahrazen a funkce nebude správně fungovat!

Důležité také je, že parametry mohou uchovávat pouze datové hodnoty, nelze pomocí parametrů přímo předávat identifikátory. Jakým způsobem lze psát generické funkce a předávat jim pomocí parametrů identifikátory objektů bude popsáno v dalším textu (viz Dynamické SQL příkazy).

Návratová hodnota

Návratová hodnota může být jedna nebo více hodnot základního nebo kompozitního datového typu, tabulka, popřípadě void. Pro možnost vracení více hodnot se návratová hodnota deklaruje jako set neboli množina. Deklarace návratové hodnoty se děje pomocí klauzule RETURNS a následného datového typu.

Příklady deklarace návratové hodnoty

- funkce vrací jednu hodnotu typu integer:

  CREATE FUNCTION fce(int, real) RETURNS integer ...

- funkce vrací množinu hodnot typu integer:

  CREATE FUNCTION fce(int, real) RETURNS SETOF integer AS...

- funkce vrací tabulku se zadanými sloupci:

  CREATE FUNCTION fce(int, real) RETURNS TABLE(column1 int, column2 char(20)) AS...

- funkce nevrací nic:

  CREATE FUNCTION fce(int, real) RETURNS void AS ...

Další příklady práce s kompozitním datovým typem jako návratovou hodnotou jsou uvedeny k podkapitole o SQL funkcích.

Jako alternativu k návratové hodnotě lze používat výstupní parametry, deklarované za vstupními parametry s modifikátorem OUT:

```
CREATE FUNCTION soucet(integer, integer, OUT sum integer) AS .....
```

Volání funkce

Pro explicitní volání použijeme příkaz SELECT, za nímž následuje jméno funkce a předávané parametry v závorkách.

```
SELECT fname(...);
```

Zajímavé je, že volání funkce lze použít i v klauzuli FROM místo tabulky.

SQL funkce

SQL funkce není pravým procedurálním rozšířením, neboť pouze zapouzdřuje běžné SQL dotazy. SQL funkce vrací výsledek posledního SQL příkazu, může vracet buď pouze první řádek nebo celou výslednou tabulku. Nemusí také vracet nic, pokud je tak deklarováno. SQL funkce může obsahovat SELECT bloky, INSERT, UPDATE, DELETE, atd. Pokud není poslední příkaz SELECT a funkce má něco vracet, musí se použít klauzule RETURNING. SQL funkce nemůže obsahovat řízení transakcí.

Vytváření SQL funkce

Jako každá funkce, i SQL funkce se vytváří příkazem CREATE FUNCTION. Tělo funkce musí být deklarováno jako řetězec, tj. uzavřeme jej například do $$ nebo jiného nepravděpodobného řetězce znaků. Typ funkce (tj. zde SQL funkce) se udává za klauzulí LANGUAGE na samém konci deklarace funkce.

```
CREATE FUNCTION jméno(seznam parametrů) RETURNS datatype AS
$$
        příkazy...
$$ LANGUAGE SQL;
```

Příklady

```
CREATE FUNCTION clean_emp() RETURNS void AS $$
        DELETE FROM emp WHERE salary < 0;
$$ LANGUAGE SQL;
---
CREATE FUNCTION add(integer, integer) RETURNS integer AS
$$
        SELECT $1+$2;
$$ LANGUAGE SQL;

SELECT add(1,2) AS answer;
DROP add(int,int);
---
CREATE FUNCTION tf1(integer, numeric) RETURNS integer AS
$$
        UPDATE bank
                SET balance = balance-$2
                WHERE accountNo = $1
                RETURNING balance;
$$ LANGUAGE SQL;

SELECT tf1(17, 100.0);
```

Příklad použití kompozitních typů

Vytvoříme kompozitní datový typ (tabulku) emp. Tento datový typ budeme používat jako parametr ve funkci doubleSalary.

```
CREATE TABLE emp (name text, salary numeric, age integer);

CREATE FUNCTION doubleSalary(emp) RETURNS numeric AS $$
        SELECT $1.salary * 2      AS      salary;
$$ LANGUAGE SQL;

SELECT name, doubleSalary(emp.*)
FROM emp
WHERE name='Jenda';
```

Představíme-li si kompozitní datový typ jako tabulku, pak jednou hodnotou tohoto typu je jeden řádek tabulky. Z toho plyne, že pokud je použit kompozitní typ (resp. tabulka) jako návratová hodnota, funkce vrací jen jeden řádek!

```
CREATE FUNCTION newEmp() RETURNS emp AS $$
        SELECT text 'None' AS name, 1000.0 AS salary, 25 AS age;
$$ LANGUAGE SQL;

SELECT (new_emp()).name;
```

Aby funkce vracela víc řádků kompozitního datového typu, musíme návratovou hodnotu deklarovat jako množinu hodnotu, set.

```
CREATE FUNCTION getOldEmp(integer) RETURNS SETOF emp AS $$
        SELECT * FROM emp WHERE age>$1;
$$ LANGUAGE SQL;
```

Alternativně lze návratovou hodnotu deklarovat přímo jako tabulku.

```
CREATE FUNCTION sum_n_product (x int)
RETURNS TABLE(sum int, product int) AS $$
        SELECT $1 + tab.y,      $1 * tab.y
        FROM tab;
$$ LANGUAGE SQL;
```

Procedurální funkce

Procedurální funkce jsou funkce psané v nějakém procedurálním jazyce. My použijeme jazyk PL/pgSQL, nicméně PostgreSQL umožňuje i Python, Perl nebo Tcl. Jazyk PL/pgSQL je obdoba jazyka PL/SQL firmy ORACLE. Umožňuje:

- vytvářet funkce a obsluhu triggerů
- přidat řízení běhu programu k jazyku SQL
- provádět složité výpočty
- využívat všech datových typů v DB
- využívat operací a jazyka SQL
- být důvěryhodný pro server

Pro zprovoznění jazyka plpgsql je většinou nutné napsat příkaz:

```
CREATE LANGUAGE plpgsql;
```

Deklarace funkce v jazyce PL/pgSQL je obdobná jako deklarace SQL funkce, pouze jako specifikace jazyka se uvede LANGUAGE plpgsql.

```
CREATE FUNCTION func(parlist) RETURNS datatype AS
$$
        tělo funkce
$$ LANGUAGE plpgsql;
```

PL/pgSQL je blokově orientovaný jazyk, umožňuje též vnořování bloků. Blok má strukturu:

- návěští (label) – nepovinné, uzavřené do dvojitých lomených uvozovek: <<návěští>>
- deklarace proměnných – nepovinná, uvedená slovem DECLARE
- tělo bloku, uzavřené mezi klíčová slova BEGIN a END – nezaměňovat se začátkem transakce!

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

PL/pgSQL je case insensitive.

Komentář se zapisuje:

```
--
/*  ...  */
```

Deklarace proměnných

Všechny proměnné kromě iterátoru cyklů LOOP a FOR musejí být deklarovány. Mohou mít libovolný datový typ, i z jazyka SQL. Abychom mohli jako datový typ proměnné efektivně využívat například řádek tabulky nebo datový typ jiné proměnné, je zaveden mechanismus kopírování datového typu:

```
nová_proměnná1    jiná_proměnná%TYPE;
nová_proměnná2    tabulka%ROWTYPE;
```

Syntaxe:

```
name [ CONSTANT ] type [ NOT NULL ] [ { DEFAULT | := } expression ];
```

Příklady na deklaraci proměnných:

```
user_id       integer;
quantity      numeric(5);
url           varchar;
myrow         tablename%ROWTYPE;
myfield       tablename.columnname%TYPE;
arow          RECORD;    -- obecný kompozitní typ bez určení struktury
```

Vytváření procedurálních funkcí

Příklady na procedurální funkce:

```
CREATE FUNCTION SP(x int, y int, OUT sum int, OUT product int) AS $$
BEGIN
        sum := x + y;
        product := x * y;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION ext_sales(n int) RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
        RETURN QUERY SELECT quantity, quantity * price FROM sales     WHERE itemno = n;
END; $$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION merge_fields(t_row    table1) RETURNS text AS $$
DECLARE
        t2_row      table2%ROWTYPE;
BEGIN
        SELECT * INTO t2_row FROM table2 WHERE ... ;
        RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

Příkazy

- přiřazení

```
variable := expression;
```

- SQL příkazy nevracející data

```
BEGIN
        UPDATE mytab SET val = val + delta WHERE id = key;
END;
```

- dotazy vracející jeden řádek (do proměnné)

```
SELECT select_expressions INTO proměnná FROM ...;
INSERT ... RETURNING expressions INTO proměnná;
```

- nedělat nic

```
NULL;
```

- informace o zpracování příkazu

```
GET DIAGNOSTICS variable = item [ , ... ];
```

nebo proměnná FOUND

Dynamické SQL příkazy

Často chceme, aby SQL příkaz obsahoval pokaždé něco jiného (jiné identifikátory apod.). K tomu se využívá příkaz EXECUTE, za nímž následuje příkaz ve formě textového řetězce:

```
EXECUTE command_string
```

Uvnitř příkazového řetězce nutno dobře ošetřit uvozovky, lze použít např. funkce quote_ident() a quote_literal(). K tvorbě řetězce lze použít libovolnou řetězcovou funkci, např. oprátor zřetězení. Tím lze vytvořit příkaz dynamicky a předávat pomocí vstupních parametrů a proměnných i identifikátory objektů.

```
CREATE FUNCTION vypis(tabulka text, podmínka text) RETURNS void AS $$
BEGIN
        EXECUTE         ' SELECT * FROM ' || quote_ident(tabulka) ||
                        ' WHERE ' || quote_literal(podmínka);
END;
$$ LANGUAGE plpgsql;
```

Návrat z funkce

Návrat z funkce se standardně děje příkazem `RETURN;`

Při návratu z funkce lze vrátit též jednoduchou návratovou hodnotu: `RETURN hodnota;`

Pro vracení složitějších dat, například množiny řádků tabulky, je třeba použít jiný mechanismus. V každé funkci, vracející SETOF, je zavedena speciální proměnná pro výsledek, do níž se ukládají data. Tato proměnná je aditivní – lze do ní data postupně přidávat. K přidávání dat do návratové proměnné slouží příkazy RETURN QUERY a RETURN NEXT:

```
RETURN QUERY dotaz;     -- připojí do návratové proměnné výsledek dotazu
RETURN NEXT výraz;      -- připojí do návratové proměnné výsledek výrazu
```

Tyto příkazy *neukončí funkci*, to je nutno zařídit dalším samostatným příkazem RETURN!

Příklad

```
CREATE FUNCTION getAllTbl() RETURNS SETOF Tbl AS $$
DECLARE
        r       Tbl%rowtype;
BEGIN
        FOR r IN SELECT * FROM Tbl WHERE id > 0
        LOOP
                -- nějaké zpracování
                RETURN NEXT r; -- aktuální řádek SELECTu přidá do návratové proměnné
        END LOOP;
        RETURN;
END
$$ LANGUAGE plpgsql;
```

```
SELECT * FROM getAllTbl();
```

Podmínka IF a CASE

Podmínky IF-THEN-ELSE a CASE mají mnoho variant:

```
IF podmínka THEN příkaz1 END IF;
```

```
IF podmínka THEN
        příkaz1
ELSE
        příkaz2
END IF;
```

```
IF podmínka1 THEN
        příkaz1
ELSIF podmínka2 THEN
        příkaz2
ELSE
        příkaz3
END IF;
```

```
CASE proměnná
WHEN hodnota1 THEN          -- rovnost proměnné a hodnoty
        příkaz1
ELSE
        příkaz2
END CASE;
```

```
CASE WHEN podmínka THEN     -- analogie IF
        příkaz1
ELSE
        příkaz2
END CASE;
```

Příklady:

```
IF number = 0 THEN
        result := 'zero ';
ELSIF number > 0 THEN
        result := 'positive ';
ELSIF number < 0 THEN
        result := 'negative ';
ELSE
        result := 'NULL';
END IF;
```

```
CASE x
        WHEN 1, 2 THEN
                msg := 'jedna nebo dvě';
        ELSE
                msg := 'něco jiného než jedna či dvě';
END CASE;
```

```
CASE
        WHEN x BETWEEN 0 AND 10 THEN
                msg := 'x je mezi 1 a 10';
        WHEN x BETWEEN 11 AND 20 THEN
                msg := 'x je mezi 11 a 20';
END CASE;
```

Cykly

- nekonečná smyčka LOOP – provádí se, dokud nepřijde EXIT či RETURN

```
LOOP
        příkazy
END LOOP;
```

- podmíněná smyčka WHILE

```
WHILE podmínka LOOP
        příkazy
END LOOP;
```

- klíčová slova EXIT a CONTINUE:
    - EXIT ukončí celý cyklus,
    - CONTINUE skočí na další iteraci cyklu
    - nepodmíněně:

```
EXIT;
CONTINUE;
```

    - podmíněně:

```
EXIT WHEN podmínka ;
CONTINUE WHEN podmínka;
```

- iterační smyčka FOR, iterátor i není třeba deklarovat jako proměnnou

```
FOR i IN 1..10 LOOP
        -- i bude postupně 1,2,3,4,5,6,7,8,9,10
END LOOP;
----------------
FOR i IN REVERSE 10..1 LOOP
        -- i bude postupně 10,9,8,7,6,5,4,3,2,1
END LOOP;
----------------
FOR i IN REVERSE 10..1 BY 2 LOOP
        -- i bude postupně 10,8,6,4,2
END LOOP;
```

- procházení výsledků dotazů: „procházecí“ smyčka FOR

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
        mviews      cs_mviews%rowtype;
BEGIN
        FOR mviews IN SELECT * FROM cs_mviews ORDER BY key
        LOOP
                --    "mviews" obsahuje vždy jeden řádek z dotazu
        END LOOP;
        RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Ošetření chyb

Standardně každá chyba ukončí provádění funkce. Chování lze ošetřit výjimkou:

```
BEGIN
        statements
EXCEPTION
        WHEN condition [ OR condition ... ] THEN
                handler_statements
        [ WHEN condition [ OR condition ... ] THEN
                handler_statements
        ... ]
END;
```

V obsluze výjimky je stav proměnných zachován. Výjimka ukončí aktuální transakci a provede ROLLBACK.

Příklad:
vrátí hodnotu x+1, ale UPDATE neproběhne

```
BEGIN
        UPDATE mytab SET name = 'Joe' WHERE lastname = 'Jones';
        x := x + 1;
        y := x / 0;
        EXCEPTION
                WHEN division_by_zero THEN
                RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

K explicitnímu ošetření chyb se používá příkaz RAISE. Má různé úrovně priority, nejvyšší je EXCEPTION, která zastaví vykonávání aktuální transakce. Příkaz RAISE lze použít k ladicím výpisům (úroveň DEBUG) nebo vypisování hlášek:

```
RAISE NOTICE 'Calling cs_create_job';
```

Příkaz RAISE umožňuje též předat kód chyby nebo dosadit hodnotu do vypisované hlášky:

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE =    'unique_violation';
```

Kontrolní otázky

- Co to je procedurální rozšíření jazyka SQL?
- Jaké jsou v PostgreSQL typy funkcí , jak se funkce vytvářejí a volají?
- Jak se v PostgreSQL předávají funkcím parametry?
- Jak se v PostgreSQL předávají návratové hodnoty?
- Jak se v PostgreSQL používají dynamické SQL příkazy?

Úlohy k procvičení

Při práci na úlohách používejte dokumentaci databázového stroje PostgreSQL, která je dostupná např. na internetu na adrese http://www.postgresql.org/docs/8.4/static/index.html. Použijte tabulky s daty z úloh k minulé kapitole (knihy v regálech). Úlohy na sebe často navazují, využívejte výsledků předchozích úloh, např. naprogramované funkce.

---

1. Napište SQL funkce, které u zadané knihy (id) přidají zadaného autora na konec nebo začátek seznamu autorů. Pro práci s polem řetězců použijte funkci array_append, resp. array_prepend.

---

2. Napište SQL funkce, které u zadané knihy (id) odeberou autora z konce nebo začátku seznamu autorů. Dbejte na to, aby funkce správně fungovaly i pro okrajové situace (prázdný seznam autorů apod.). Pro práci s polem řetězců použijte indexaci pole intervalem hodnot.

---

3. Napište SQL funkci, která u zadané knihy (id) odebere autora ze zadaným indexem (tj. pořadím v seznamu). Dbejte na to, aby funkce správně fungovala i pro okrajové situace (prázdný seznam autorů apod.). Pro práci s polem řetězců použijte operátor zřetězení.

---

4. Prozkoumejte chování funkcí unnest(pole) a generate_subscripts(pole,dimenze), které jsouce zavolány na pole, rozvinou dané pole v tabulku, respektive vrátí tabulku platných indexů pro dané pole. Vyzkoušejte, jak funkce pracují, pokud jsou obě zapsány za klauzulí SELECT.

---

5. Napište SQL funkci autoriTbl, která pro zadanou knihu vrátí tabulku, v níž bude každému autorovi odpovídat řádek se třemi údaji: jméno autora, délka jména autora, index autora v poli. Vyzkoušejte v dotazu jednak za klauzulí SELECT, jednak za klauzulí FROM.

---

6. Napište SQL funkci, která zajistí, že se ze seznamu autorů u zadané knihy odstraní autor s druhým nejkratším jménem.

---

7. Napište procedurální funkci knihado1nf, která vytvoří novou tabulku kniha2, do níž převede tabulku kniha do 1NF: rozbalí pole autorů tak, že každý autor bude mít svůj sloupec, počet „autorských“ sloupců bude roven délce nejdelšího pole autoři v tabulce. Případné nevyužité hodnoty budou NULL. Nápověda: vytvoření kopie tabulky lze provést příkazem

   CREATE TABLE kopie AS TABLE original

---

8. **Další dotazy:** data nastavte tak, abyste měli aspoň jeden regál, na který neodkazuje žádná kniha. Dále mějte aspoň tři knihy, které mají různé autory, ale stejné názvy. V SQL (bez procedurálního rozšíření) napište následující dotazy:

   a. Vypište jednotlivé regály (id) s počtem knih v nich. Pokud je regál prázdný, vypíše se nula.
   b. Vypište jednotlivé regály (id), pokud je regál prázdný, vypíše se slovo „volný“, jinak se vypíše slovo „obsazený“. Nápověda: použijte podmínku CASE.
   c. Vypište id prázdných regálů. Dotaz realizujte nejméně třemi různými způsoby (způsobů je v PostgreSQL nejméně pět).
   d. Vytvořte pohled „poloprázdné“, který bude obsahovat regály (sloupce s názvy id_regalu, kapacita, pocet_knih), v nichž je uskladněno méně knih, než je polovina jejich kapacity. Pohled tedy bude obsahovat i prázdné regály. Nápověda: použití jakékoli agregační funkce předpokládá, že kromě ní bude za klauzulí SELECT pouze sloupec, podle nějž se tabulka seskupuje. Jestliže chceme mít takové sloupce dva (id_regalu a kapacita), je nutné tabulku seskupit podle dvou sloupců – GROUP BY id_regalu, kapacita.
   e. Vypište tři relativně nejprázdnější regály (tj. regály, které mají nejmenší procento obsazenosti), vč. procenta obsazenosti. Nápověda: vytvoříte nově vypočítávaný sloupec, který pojmenujete (např. „procento“) a budete podle něj řadit.
   f. Vytvořte pohled „dvojice“, který bude uchovávat seznam všech dvojic knih, které mají stejný název, ale různé autory. V seznamu bude název, autoři první knihy, autoři druhé knihy. Fakt, že za každou takovou dvojici knih budou v seznamu dva řádky s opačným pořadím autorů, není na škodu: např. pro knihy Nesmrtelnost od Borgese a od Kundery se objeví řádky (Nesmrtelnost, Borges, Kundera) a (Nesmrtelnost, Kundera, Borges). Nápověda: využijte spojení tabulky sama se sebou.

g. Složitější: vezměte pohled „dvojice“ a napište nad ním dotaz, který bude eliminovat dvojí zobrazení jednoho názvu knihy: např. pro knihy Nesmrtelnost od Borgese a od Kundery se objeví jediný řádek (Nesmrtelnost, Borges, Kundera). Nápověda: využijte jednoznačné uspořádání autorů a klíčové slovo DISTINCT.

10 Kursory

Cíl kapitoly

Kapitola pokračuje v procedurálním rozšíření jazyka SQL a představuje koncept kursoru. Cílem této kapitoly je

- seznámit se s konceptem kursorů
- naučit se deklarovat a otevřít kursor
- naučit se používat kursory

Klíčové pojmy

Kursor, otevření kursoru, kursor ve smyčce, kursor s dynamickým příkazem.

Motivace

Dotaz nad databází (např. SELECT) vrací obecně více řádků, tj. tabulku. Mnohdy je však třeba zpracovávat vracené řádky jeden po druhém. K tomu lze v jazyce PL/pgSQL použít buď procházecí FOR smyčku (viz předchozí kapitola) nebo tzv. kursor. Kursor je struktura „ukazující“ na řádky výsledku dotazu. Umožňuje získávat data dotazu postupně po řádcích, může předcházet potížím s pamětí, pokud je výsledek dotazu příliš rozsáhlý. Výhodou kursoru oproti procházecí FOR smyčce je to, že jej lze použít jako návratovou hodnotu z funkce, tedy jedna funkce může kursor zavést a otevřít a další funkce jej může používat.

Deklarace

Kursor se deklaruje jako kursorová proměnná datového typu refcursor.
Obecná deklarace kursoru se syntaxí:

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] IS query;
```

- přepínač SCROLL umožní kursoru rolovat nazpátek k předcházejícím řádkům
- argumenty slouží k dosazování hodnot do dotazu
- konkrétní hodnoty jsou dodány ve chvíli otevření kursoru
- slovo IS lze zaměnit s FOR

Kursory dělíme do dvou typů: unbound a bound. Unbound kursor lze použít pro jakýkoli dotaz, bound kursory už mají svůj dotaz přidělen při deklaraci.

Příklady deklarace

```
DECLARE
        curs1   refcursor;      -- nevázaný (unbound) kursor
        curs2   CURSOR   FOR   SELECT * FROM t1;
        curs3   CURSOR(key integer)   IS   SELECT * FROM t1 WHERE   unique1 = key;
```

Otevření kursoru

Unbound kursor

Kursor musí být deklarován jako unbound. Při otevření se kursoru přiřadí dotaz a otevře se, kursor nesmí být již otevřen. Dotaz musí být něco, co vrací řádky, typicky SELECT. V kursoru lze použít proměnnou PL/pgSQL, ale její hodnota se dosadí POUZE při otevření kursoru; její další změny nemají na funkci kursoru již vliv.

```
DECLARE
        curs1   refcursor;      -- nevázaný (unbound) kursor

OPEN   curs1   NO SCROLL   FOR   SELECT * FROM foo WHERE key = mykey;
```

Bound kursor

Bound kursor již má svůj dotaz dán deklarací, proto se jen předají případné parametry a otevře se, kursor pochopitelně nesmí být již otevřen. Stejně jako v případě unboudn kursoru, dotaz musí být něco, co vrací řádky, typicky SELECT, v kursoru lze použít proměnnou PL/pgSQL, ale její hodnota se dosadí pouze při otevření kursoru; její další změny nemají na funkci kursoru již vliv.

```
DECLARE
        curs2   CURSOR   FOR   SELECT * FROM t1;
        curs3   CURSOR(key integer)   IS   SELECT * FROM t1 WHERE   unique1 = key;

OPEN curs2;
OPEN curs3(42);
```

Použití kursoru

Pro použití musí být kursor otevřen, i když ne nutně v té samé funkci. Kursor je použitelný v rámci aktuální transakce, pak se zavře. Načtení dat se provádí příkazem FETCH:

```
FETCH [ direction FROM ] cursor INTO proměnná;
```

Příkaz FETCH vloží *další řádek* kursoru do zadané proměnné. Není-li v kursoru již další řádek, vloží tam hodnoty NULL. Tento mechanismus někdy nefunguje dobře; pro kontrolu, zda byl nějaký řádek skutečně vložen je lepší kontrolovat proměnnou FOUND. Pokud je proměnná FOUND na hodnotě TRUE, pak byl řádek vložen.

Klauzule direction:

- NEXT (nebo FORWARD) – vloží do proměnné další řádek
- PRIOR (nebo BACKWARD) – vloží do proměnné předchozí řádek
- FIRST – vloží do proměnné první řádek
- LAST – vloží do proměnné poslední řádek
- ABSOLUTE n – vloží do proměnné n-tý řádek dotazu absolutně
- RELATIVE n – vloží do proměnné n-tý řádek dotazu relativně vůči aktuální pozici
    - pokud n=0, vrací aktuální řádek bez posunu polohy

Bez zadání klauzule direction se předpokládá NEXT.

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

Příkaz MOVE

MOVE přesune kursor bez vracení dat, jinak má úplně stejné možnosti jako standardní FETCH, tj. bez specifikace direction se bere NEXT.

```
MOVE [ direction FROM ] cursor;

MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
```

Aktualizace dat pod kursorem

Je možné mazat či aktualizovat data v tabulce, kam ukazuje kursor, dotaz pod kursorem ale musí být pro tuto operaci vhodný, tj. žádné GROUP BY apod. Pokud chceme tuto možnost používat, je nejlepší definovat bound kursor jako FOR UPDATE

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;

UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

Uzavření kursoru

Kursor lze uzavřít a tím uvolnit zdroje dříve, než skončí aktuální transakce

```
CLOSE cursor;
```

Kursor jako návratová hodnota funkce

Kursor lze vracet z funkce:

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

-- funkce otevře kursor daný proměnnou a otevřený jej vrátí
CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS $$
BEGIN
        OPEN $1 FOR SELECT col FROM test;
        RETURN $1;
END; $$ LANGUAGE plpgsql;

--nejprve funkcí otevřeme kursor a poté jej použijeme
BEGIN;    -- začátek transakce
        SELECT reffunc('funccursor');
        FETCH ALL IN funccursor;
COMMIT;
```

Kursor ve smyčce

Kursor ve smyčce je de facto speciální varianta FOR cyklu:

- otevře
- procykluje
- uzavře při opuštění smyčky

Kursor nesmí být předtím otevřen.

```
FOR recordvar IN bound_cursorvar [ ( argument_values ) ] LOOP
        statements
END LOOP;
```

Kursor s dynamickým příkazem

Chceme-li například kursoru přiřadit pokaždé trochu jiný dotaz, použijeme analogii příkazu EXECUTE. Namísto

```
OPEN curs FOR SELECT * FROM mytable;
```

použijeme například

```
OPEN curs FOR EXECUTE 'SELECT * FROM ' || quote_ident($1)
```

Další poznámky

Použití kursoru je výkonově velmi náročné a znamená snížení výkonu databáze. Vyžaduje-li konkrétní databáze použití kursorů, jedná se pravděpodobně o důsledek špatného návrhu databáze. Kursory je tedy vhodné používat pouze ve výjimečných případech, např. při migraci dat mezi dvěma databázovými strukturami.

Kontrolní otázky

- Co to je kursor?
- Jaké jsou typy kursorů?
- Lze kursor používat ve smyčce?
- Lze kursor používat k aktualizaci dat?

Úlohy k procvičení

Při práci na úlohách používejte dokumentaci databázového stroje PostgreSQL, která je dostupná např. na internetu na adrese http://www.postgresql.org/docs/8.4/static/index.html. Použijte tabulky s daty z úloh k minulé kapitole (knihy v regálech).

---

1. Napište procedurální funkci knihado1nf, která vytvoří novou tabulku kniha2, do níž převede tabulku kniha do 1NF: rozbalí pole autorů tak, že každý autor bude mít svůj sloupec, počet „autorských“ sloupců bude roven délce nejdelšího pole autoři v tabulce. Případné nevyužité hodnoty budou NULL. Řešte pomocí kurzoru namísto procházecí FOR smyčky. Nepoužívejte kurzorovou variantu FOR cyklu, použijte excplicitní otevření kursoru, získání řádku, kontrolu, zda byl řádek nalezen, a cyklus.

11 Triggery

Cíl kapitoly

Kapitola pokračuje v procedurálním rozšíření jazyka SQL a představuje koncept triggeru. Trigger je funkce, která se automaticky spustí při nějaké nastalé události v databázi. Cílem této kapitoly je

- seznámit se s konceptem triggerů
- naučit se deklarovat triggery
- naučit se používat triggery vč. předávání hodnot

Klíčové pojmy

Trigger, row-level trigger, statement-level trigger,after-trigger, before-trigger.

Trigger

Trigger je jakákoli procedurální funkce (ne SQL funkce), která se volá při určité události v DB. Kursor lze volat:

- před příkazem INSERT, UPDATE, DELETE (before-trigger)
- po příkazu INSERT, UPDATE, DELETE (after-trigger)
- jednou pro každý modifikovaný řádek (row-level trigger)
- jednou pro každý SQL příkaz (statement-level trigger)
    - kursornemůže sahat na jednotlivé řádky

Triggerová funkce nesmí mít žádné parametry a musí vracet datový typ trigger. Při vytváření triggeru nejprve definujeme příslušnou funkci, pak ji teprve učiníme triggerem a navážeme ji na konkr=tní událost pomocí příkazu CREATE TRIGGER.

Before-trigger

Before-trigger se spouští PŘED asociovaným příkazem.

- statement-level
    - spustí se těsně před vyvoláním asociovaného SQL příkazu
- row-level
    - spustí se těsně před tím, než se začne operovat s daným řádkem

After-trigger

After-trigger se spouští PO asociovaném příkazu.

- statement-level
  - spustí se na konci vykonávání asociovaného SQL příkazu
- row-level
  - spustí se na konci vykonávání asociovaného SQL příkazu, ale předtím, než se spouští statement-level trigger

Návratové hodnoty

- statement-level trigger
  - příslušná funkce by měla vždy vracet NULL
- row-level trigger
  - mohou vracet NULL nebo řádek tabulky
  - before-trigger
    - pokud before-trigger vrátí NULL, operace s daným řádkem se vůbec neprovede
    - pro příkazy INSERT a UPDATE bude vložen ten řádek, který je vrácen triggerem, trigger tedy může ovlivnit, co se vloží nebo co se bude měnit
    - pokud nechceme ani jednu z výše uvedených dvou možností, je nutno pečlivě dbát na to, aby trigger vracel přesně ten samý řádek, na němž se bude operovat
  - after-trigger – je jedno co vrací

Použití

Row-before-trigger je určen typicky pro kontrolu IO, konzistence vkládaných dat, vložení aktuálního času apod. Není vhodný pro kontrolu referenční integrity, neboť before-trigger nemůže s jistotou vědět, jak příkaz dopadne.

Row-after-trigger provádí např. zápisy aktualizovaných hodnot do vázaných tabulek popř. kontrolu konzistence.

Funkce triggeru může spouštět SQL příkazy, na nichž visí další trigger, a tak donekonečna, může se volat i rekurzivně.

Ilustrativní příklad

```
CREATE FUNCTION emp_stamp() RETURNS trigger AS $$
BEGIN
        ...
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
        FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Deklarace

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
        ON table
        [   FOR   EACH   { ROW | STATEMENT }   ]
        EXECUTE PROCEDURE funcname (arguments);
```

Odstranění triggeru

```
DROP TRIGGER name ON table;
```

Triggerové funkce

Trigger je konstrukce, která přiřadí nějaké události v databázi nějakou obslužnou funkci. Obecně jedna obslužná funkce může být přiřazena více triggerům. Triggerové funkce jsou psané v procedurálním jazyce, např. PL/pgSQL, nesmějí mít vstupní parametry neboť parametry se předávají jinak (viz dále). Triggerové funkce musí vracet jednu hodnotu datového typu trigger.

Když je triggerová funkce zavolána, automaticky se vytvoří následující proměnné:

- NEW
    - datový typ record
    - pro INSERT/UPDATE v row-level triggerech uchovává nově vkládaný řádek
    - pro statement-level triggery a DELETE je NULL
- OLD
    - datový typ record
    - pro DELETE/UPDATE v row-level triggerech uchovává původní či odstraňovaný řádek
    - pro statement-level triggery a INSERT je NULL
- TG_NAME
    - datový typ name
    - jméno právě spuštěného triggeru
- TG_WHEN
    - datový typ text
    - obsahuje slovo AFTER nebo BEFORE, podle typu triggeru
- TG_LEVEL
    - datový typ text
    - obsahuje slovo STATEMENT nebo ROW, podle typu triggeru
- TG_NARGS
    - datový typ integer
    - obsahuje počet parametrů, definovaných v triggeru
- TG_ARGV[]
    - datový typ array of text
    - obsahuje jednotlivé parametry z deklarace triggeru, indexujeme od 0

- TG_OP
    - datový typ text
    - obsahuje operaci, na kterou je trigger navázán (INSERT, DELETE, UPDATE)
- další proměnné s informacemi, na jaké tabulce operujeme (TG_TABLE_NAME), v jakém schematu (TG_TABLE_SCHEMA), na jakém objektu (TG_RELID)

Triggerové funkce musí vracet buď NULL, nebo přesně takový řádek, jaký odpovídá tabulce, na níž byl trigger spuštěn. Lze tedy modifikovat proměnnou NEW a upravit data, která chceme změnit. Triggerová funkce může ukončit celou transakci tím, že vyvolá chybu.

Příklad:

Kdykoli se něco děje na tabulce zaměstnanců, je o tom učiněn záznam do auditové tabulky

```
CREATE TABLE emp (
        empname text NOT NULL,
        salary integer );

CREATE TABLE emp_audit (
        operation char(1)    NOT NULL,
        stamp    timestamp    NOT NULL,
        userid    text    NOT NULL,
        empname    text    NOT NULL,
        salary integer );

CREATE FUNCTION process_emp_audit() RETURNS trigger AS $$
BEGIN
        IF (TG_OP = 'DELETE') THEN
                INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
                RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
                INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
                RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
                INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
                RETURN NEW;
        END IF;
        RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_audit
        AFTER INSERT OR UPDATE OR DELETE ON emp
        FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

Další poznámky

Podobně jako kursory, i triggery jsou výkonově náročné a znamenají omezení výkonu databáze. Je tedy třeba zvážit použití triggerů v databázovém stroji.

Tato kapitola pojednávala o tzv. DML triggerech (data manipulation), které se vážou na operace jako INSERT, UPDATE, DELETE. Kromě nich některé databázové stroje obsahují tzv. DDL triggery (data definition), které operují nad příkazy CREATE, ALTER či DROP. Tyto triggery mohou sloužit například k auditování strukturálních změn databáze.

Kontrolní otázky

- Co to je trigger?
- Jaké jsou typy triggerů?
- Jak se triggerům předávají parametry?

Úlohy k procvičení

Implementujte ISA hierarchii. Mějme entitní nadtyp OSOBA (atributy rodné číslo, jméno, příjmení) a entitní podtypy STUDENT (navíc atribut ročník) a UČITEL (navíc atribut plat). Realizujte třemi tabulkami: OSOBA(rč, jméno, příjmení), STUDENT(rč, ročník) a UČITEL(rč, plat). Pamatujte, že ISA hierarchie znamená pokrytí množiny, tj. entitní podtypy beze zbytku pokrývají entitní nadtyp a navíc jsou vzájemně disjunktní. Tedy osoba je vždy buď student nebo učitel, musí být právě jedno z toho.

Pomocí triggerů ošetřete co nejvíc variant operací nad tabulkami:

1. Pokus o INSERT do tabulky OSOBA nebude povolen, protože by nebylo určeno, zda je osoba student nebo učitel.

---

2. Předchozí trigger a funkci odstraňte (DROP TRIGGER...)

3. Vytvořte systém triggerů následovně:
    a. Při INSERTu do OSOBY kontrolovat, zda je dané rč přítomno buď ve studentovi nebo v učiteli a pokud ano, vloží se údaje i do osoby, jinak se oznámí chyba (raise exception...) a INSERT se neprovede.
    b. Při INSERTu do STUDENTA nebo UČITELE se nejprve zkontroluje, zda dané rč není přítomno v osobě, pokud ano, ohlásí se chyba a zápis se neprovede. Pokud dané rč ještě není, provede se následně INSERT do tabulky OSOBA. Tento bod vede na kombinaci row-before a row-after triggeru.

---

4. DELETE nad tabulkou OSOBA provede i DELETE v odpovídajícím podtypu

---

5. DELETE nad tabulkou STUDENT/UČITEL provede i DELETE v OSOBĚ. Proveďte tak, že pokud se nepodaří DELETE v OSOBĚ, neprovede se ani v ent. podtypu

---

6. UPDATE rodného čísla v OSOBĚ se propaguje i do entitního podtypu

---

7. UPDATE rodného čísla ve STUDENTOVI/UČITELI se provede i v OSOBĚ, tím bude současně ošetřena unikátnost rodného čísla. Proveďte tak, že pokud se nepodaří UPDATE v OSOBĚ, neprovede se ani v ent. podtypu

---

8. Proveďte dotazy:
    a. vypište jména a příjmení všech studentů/učitelů
    b. vypište jména a příjmení všech osob, s příznakem, zda jsou učitel nebo student

12 Datové sklady a OLAP

Cíl kapitoly

Kapitola se stručně zmiňuje o pojmech jako datový sklad, OLAP či data mining. Tyto techniky mohou být považovány za jakousi abstraktnější vrstvu nad běžnými (obvykle relačními) databázemi. Cílem této kapitoly je především upozornit na pojmy datový sklad, OLAP, OLTP či business intelligence.

Klíčové pojmy

Datový sklad, data warehouse, OLAP, OLTP, data mining, business intelligence.

Business intelligence

Podniky se v rámci konkurenčního boje vždy snažily porozumět prostředí, v němž působí. Tato znalost je nezbytným podkladem pro dlouhodobé strategické plánování, ale i pro okamžité taktické manévry. Ještě relativně nedávno neexistovaly dostatečné počítačové prostředky pro sběr, ukládání a analýzu těchto dat. I v situaci, kdy již v podniku existoval jakýsi informační systém, nebyla podpora tohoto typu běžná. Postupně však vznikaly potřebné standardy v ICT, které zpřístupnily obrovské objemy elektronických dat, jež se poté staly živnou půdou pro business intelligence.

Business intelligence (BI) zahrnuje

- procházení velkých objemů dat
- extrakci důležitých informací
- změna získaných informací na znalosti použitelné při řízení podniku

Business intelligence je definována jako sada konceptů a metod pro zlepšení rozhodování podniků pomocí podpůrných systémů založených na faktech.

Datové sklady a OLAP

Pro podporu rozhodování nelze většinou použít standardní databáze. Byly proto vyvinuty speciální typy databází – datové sklady (data warehouses, OLAP) – které obsahují data, která jsou předzpracovaná např. pro manažerské analýzy a záměrně obsahují vysokou míru redundance. Datové sklady jsou tedy využívané především v business intelligence, kde nelze běžné databáze efektivně použít. Datové sklady z principu a záměrně nesplňují pravidla normalizace databází.

Důvody pro použití datových skladů pro rozhodování jsou především tyto:

- často je IS podniku postaven na více DB
    - komponenty jako je účetní systém, správa skladu, personální agenda, správa objednávek, mají často své vlastní databáze
    - jednotlivé komponenty spolu sice komunikují, každá jednotlivá DB však má jinou strukturu, může používat jiné datové typy a formáty apod.
    - získávání integrovaných informací je tudíž téměř nemožné
- provozní systémy a databáze nebyly konstruovány pro účel obchodního rozhodování
    - jsou dělené podle aplikačních oblastí (fakturace, řízení zásob, prodej produktů) a nikoli podle hlavních subjektů organizace (zákazníci, produkty, dodavatelé)
    - poskytují podrobná, aktuální a neustále se měnící data, ovšem pro rozhodování potřebujeme data sumarizovaná, historická, která se mění jen pomalu
    - jsou zaměřeny na aktuální provoz, nepotřebná starší data jsou archivována a nebývají snadno dostupná; pro rozhodování však často potřebujeme právě tuto historii
    - jsou optimalizovány pro zpracování obřího množství jednoduchých a předvídatelných operací, zatímco pro rozhodování potřebujeme naopak zpracovat malé množství nepředvídatelných a komplikovaných operací
    - podporují paralelní přístup mnoha uživatelů, zatímco rozhodování je obvykle v rukách jen hrstky manažerů

Řešení spočívá v zavedení datových skladů a technik data mining:

- všechna provozní a archivní data analyzujeme a přepočítáme a uložíme znovu jinam a v jiné formě
- na integrovaná data aplikujeme sofistikované analytické nástroje
- využíváme především agregací – vypočítávání souhrnných informací z tisícovek a milionů záznamů.

Datový sklad je tedy ze samotného principu naprosto nenormalizovaná struktura. Zdrojem dat pro datový sklad jsou provozní databáze, případně soubory a externí zdroje (web). Zdrojová data jsou extrahována, přetransformována a uložena do datového skladu, v datovém skladu jsou data uchovávána již ve více či méně sumarizované a předzpracované podobě. Toto sice zvyšuje nároky na úložný prostor, ale zrychluje to práci – data se nemusejí stále znovu přepočítávat, mohou být současně uloženy i různé úrovně sumarizace. Tam, kde pro běžnou agendu máme databázi a nad ní pracující aplikační programy, pro účely business intelligence máme datový sklad a nad ním speciální nástroje pro získávání komplexních informací. Datový sklad tedy uchovává de facto znovu totéž jako provozní DB, ale vše na jednom místě, vyčištěné, konzistentní a v takové struktuře, která vyhovuje obchodnímu rozhodování.

Prosté uložení dat v datovém skladu neposkytne samo o sobě žádoucí efekt, je třeba ukryté informace nějak extrahovat, „vydolovat“ (odtud název data mining). Základní analýzu je analytik schopen provést na datovém skladu i bez pokročilých nástrojů, vzhledem k růstu objemu a složitosti dat ale mohou data skrývat neočekávané trendy či závislosti, které je obtížné odhalit => potřeba sofistikovaných technik data miningu.

Někdy se jako synonymum datových skladů uvádí zkratka OLAP (on-line analytical processing). Tento termín ale spíše označuje přístup k rychlému získávání odpovědí na složité multidimenzionální dotazy. Protipólem je termín OLTP (on-line transactional processing), který označuje tradiční databáze, o nichž pojednává tento učební text. Problematika OLAP je poměrně rozsáhlá a překračuje rozsah předmětu.

Kontrolní otázky

- Co to je datový sklad?
- Jaký je rozdíl použití datového skladu a standardní databáze?

Literatura

[1] Pokorný, J. – Halaška, I.: Databázové systémy. Skripta FEL ČVUT, Vydavatelství ČVUT, Praha, 2004.

[2] Pokorný, J.: Konstrukce databázových systémů. Skripta FEL ČVUT, Vydavatelství ČVUT, Praha, 2001.

[3] Conolly, T. – Begg, C. – Holowczak, R.: Mistrovství – databáze: Profesionální průvodce tvorbou efektivních databází. 1. vydání, Computer press, Brno, 2009.

[4] *PostgreSQL: The world's most advanced open source database* [online]. 2011 [cit. 2011-10-07]. Dostupné z WWW: <http://www.postgresql.org/>.