Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Normalizácia relačných databáz

(Bakalárska práca)

Martin Vlčák

Vedúci: Dr. Tomáš Plachetka

Bratislava, 2009

Martin Vlčák

Normalizácia relačných databáz

Bakalárska práca

Katedra informatiky
Fakulta matematiky, fyziky a informatiky
Univerzita Komenského, Bratislava

2009

Čestne vyhlasujem, že som túto bakalársku prácu vypracoval samostatne,
s použitím uvedených zdrojov.

Bratislava, 2009

© Copyright 2009 Martin Vlčák
All Rights Reserved

Abstrakt

Kľúčové slová: Normalizácia, tretia normálna forma, Boyce-Coddova normálna forma, kľúče, relácia.

Zadanie bakalárskej práce

Navrhnúť a implementovať vybrané algoritmy pre normalizáciu relačných databáz.

Poďakovanie

Ďakujem Dr. Tomášovi Plachetkovi za cenné rady a pomoc pri písaní bakalárskej práce.

Obsah

1 Úvod **1**

1.1 Cieľ práce 1

2 Normalizácia relačných databáz **3**

3 Normalizačné algoritmy **5**

3.1 Uzáver množiny atribútov 6

3.2 Kľúče relácie 7

3.3 Minimálne pokrytie 10

3.4 Tretia normálna forma 15

3.5 Boyce-Coddova normálna forma 17

4 Implementácia normalizačných algoritmov **21**

4.1 Implementácia algoritmov 21

4.2 Applet pre prezentáciu algoritmov 26

A Príloha **29**

Literatúra **31**

Kapitola 1

Úvod

So vzostupom používania databázových systémov vzniká potreba správneho návrhu kvalitných databáz. Pri väčšom objeme dát môže reprezentácia relačnej databázy jedinou tabuľkou viesť k problémom. Vzniká riziko nekonzistencie, anomálie pri vynechávaní a modifikácii dát, potreba NULL hodnôt, plytvanie pamäťou.

Tento problém je možné riešiť použitím normalizačných algoritmov, ktoré slúžia na overenie kvality návrhu a predovšetkým na automatické generovanie vhodných tabuliek. Aj keď normalizačné algoritmy sú výpočtovo veľmi zložité, ich použitie nakoniec vedie k správne navrhnutej databáze.

1.1 Cieľ práce

V kapitole 2 vysvetlíme, prečo je vhodné používať normalizáciu pri navrhovaní databáz.

V ďalšej kapitole podrobne opíšeme vybrané algoritmy slúžiace na normalizáciu relačných databáz. Opíšeme hľadanie kľúčov, minimálneho pokrytia a tretej normálnej formy. Taktiež opíšeme priebeh algoritmu na testovanie, či zadaná množina funkčných závislostí je minimálnym pokrytím inej množiny funkčných závislostí a tiež algoritmu, ktorý otestuje či zadané tabuľky tvoria spolu rozklad do tretej normálnej formy, resp. Boyce-Coddovej normálnej formy. V poslednej kapitole opíšeme, ako sme implementovali tieto algoritmy.

Kapitola 2

Normalizácia relačných databáz

V tejto kapitole sme čerpali informácie z kníh Kroenkeho [Kro00], Elmasriho, Navatha [RE03], Connollyho a Begga [TC05]

Normalizácia je proces navrhovania databáz, ktorý začína skúmaním vzťahov medzi jednotlivými atribútmi. Výsledkom normalizácie je rozumné rozloženie atribútov do tabuliek. Žiadanou vlastnosťou databázy, ktorú možno dostať pomocou normalizácie je, že všetky atribúty medzi ktorými je funkčná závislosť, sú v jednej relačnej tabuľke.

Postup pri navrhovaní databázy môže byť taký, že sa najskôr pomocou vybraného modelovacieho UML nástroja vytvorí diagram, z ktorého sa dajú vyčítať funkčné závislosti medzi atribútmi, resp. počiatočné rozdelenie atribútov do jednotlivých relačných tabuliek. Následne sa na každú takúto tabuľku aplikuje normalizačný proces. Ak konkrétna tabuľka nebola v požadovanej normálnej forme, tak sa z nej požadovaná normálna forma vytvorí. Normalizáciu možno teda charakterizovať ako proces, ktorý je možné použiť v ľubovoľnom okamihu pri navrhovaní databázy.

Výhody databázy ktorá obsahuje normalizované relačné tabuľky, sú hlavne jednoduchší prístup užívateľa k dátam v databáze, lepšie udržiavanie dát a nižšie pamäťové nároky. Pri návrhu dobrej databázy nemožno zabúdať ani na správnu voľbu externých kľúčov. Sú to kľúče, ktoré slúžia na prepájanie jednotlivých tabuliek databázy navzájom. Ako príklad možno uviesť jednu tabuľku zamestnancov a ich osobných údajov, pričom táto tabuľka obsahuje ID zamestnanca a ID oddelenia, v ktorom pracuje. Druhá tabuľka nech obsahuje informácie o odde-

leniach a tiež obsahuje stĺpec ID oddelenia. V takomto prípade vidno, že pomocou ID zamestnanca možno určiť to, čo podľa ID oddelenia. Takáto redundancia je ale žiadaná, lebo umožňuje vhodne spájať jednotlivé tabuľky z databázy.

Na rozdiel od žiadanej redundancie pri externých kľúčoch, nechcená redundancia vnútri jednotlivých relačných tabuliek databázy môže spôsobiť viaceré problémy pri vkladaní, modifikácií a mazaní dát. Môžeme uvažovať predošlý prípad, ale s tým rozdielom, že dáta nie sú v dvoch tabuľkách, ale len v jednej. Potom dostávame v jednej tabuľke ID zamestnanca, stĺpce obsahujúce údaje o ňom a navyše pri každom zamestnancovi je aj ID oddelenia, v ktorom pracuje a aj všetky stĺpce obsahujúce informácie o danom oddelení. V takomto prípade je primárny kľúč tabuľky ID zamestnanca, ale všetky informácie o oddelení sú tiež určované aj ID oddelenia. Ak sa vkladá údaj do takejto tabuľky, treba vždy poznať o každom zamestnancovi aj podrobné informácie o oddelení, v ktorom pracuje. V takejto tabuľke sa nachádzajú teda inkonzistentné dáta, čo sú dáta, ktoré spolu navzájom nijak nesúvisia a sú zbytočné.

Pri správnom návrhu databáz je dôležité tiež poznať funkčné závislosti medzi jednotlivými atribútmi. Funkčné závislosti, ktoré platia medzi jednotlivými atribútmi, sa dajú určiť vo väčšine prípadov podľa prirodzenej logiky, čiže vzťahy platiace v reálnom svete možno aplikovať aj na databázu. V niektorých prípadoch je tiež ale potrebné pre lepšie fungovanie databázy, pridať ďalšie umelo vytvorené závislosti platiace v jednotlivých relačných tabuľkách.

Identifikácia funkčných závislostí je potrebná pre ďalšie aplikovanie normalizačných algoritmov. Pomocou funkčných závislostí je možné nájsť primárny kľúč danej tabuľky. Stačí s danými funkčnými závislosťami aplikovať algoritmus pre nájdenie kľúčov relácie. Ako primárny kľúč možno zvoliť hociktorý z množiny nájdených kľúčov, pričom ale platí, že primárny kľúč by nikdy nemal mať zbytočne veľa atribútov. Tak je teda najlepšie za primárny kľúč zvoliť ten nájdený kľúč, ktorý obsahuje najmenší počet atribútov.

Kapitola 3

Normalizačné algoritmy

V tejto kapitole postupne prezentujeme algoritmy súvisiace s normalizáciou. Postupne ukážeme pseudokódy jednotlivých algoritmov, opíšeme ich priebeh a analyzujeme ich zložitosť. V tejto kapitole sme pri popise jednotlivých algoritmov čerpali s materiálov Maiera [Dav80], Miltona, Diedericha [DM88] a Bernsteina [Ber76]. V tejto práci sme vybrali algoritmy, na ktorých sa dá jednoducho ukázať ich priebeh, ale v spomenutých materiáloch možno nájsť aj efektívnejšie verzie týchto algoritmov.

Definícia 3.0.1 *Množina atribútov, ktorá určuje všetky ostatné atribúty relácie, sa nazýva nadkľúčom relácie. Kľúč je nadkľúč, z ktorého keď sa odstráni jeden atribút, prestane byť nadkľúčom. [RE03]*

Definícia 3.0.2 *Uzáverom množiny atribútov $\mathcal{X}$ sa nazýva množina $\mathcal{X}^+$, ktorá obsahuje všetky atribúty, ktoré sa dajú odvodiť z množiny $\mathcal{X}$, použitím funkčných závislostí z množiny $\mathcal{F}$. [RE03]*

Definícia 3.0.3 *Dve množiny funkčných závislostí $\mathcal{F}$ a $\mathcal{G}$ sú ekvivalentné, ak každá funkčná závislosť z množiny $\mathcal{F}$ je odvoditeľná aj na množine funkčných závislostí $\mathcal{G}$ a zároveň každá funkčná závislosť z množiny $\mathcal{G}$ je odvoditeľná aj na množine funkčných závislostí $\mathcal{F}$ [RE03]*

Definícia 3.0.4 *Minimálne pokrytie množiny funkčných závislostí $\mathcal{F}$ je minimálna množina funkčných závislostí, ktorá je ekvivalentná s $\mathcal{F}$. [RE03]*

Definícia 3.0.5 *Atribút $A \in \mathcal{X}$ funkčnej závislosti $\mathcal{X} \rightarrow \mathcal{Y}$ sa nazýva redundantný, ak existuje $\mathcal{Z} \subseteq \mathcal{X}$ také, že $A \notin \mathcal{Z}$ a zároveň platí, že $\mathcal{Y} \subseteq \mathcal{Z}^{+}$. [Dav80]*

Definícia 3.0.6 *Relácia $\mathcal{R}$ je v tretej normálnej forme, ak pre každú jej závislosť $\mathcal{X} \rightarrow \mathcal{A}$ platí, že buď $\mathcal{X}$ je nadkľúč, alebo $\mathcal{A}$ je súčasťou kľúča relácie $\mathcal{R}$. [RE03]*

Definícia 3.0.7 *Funkčná závislosť $f \in \mathcal{F}$ tvaru $\mathcal{X} \rightarrow \mathcal{Y}$ sa nazýva redundantná, ak množina $\mathcal{X}^{+}$, ktorá bola vytvorená ako uzáver množiny $\mathcal{X}$ z funkčných závislostí $\mathcal{X} - f$, obsahuje všetky atribúty množiny $\mathcal{Y}$. [Dav80]*

Definícia 3.0.8 *Relácia $\mathcal{R}$ je v Boyce-Coddovej normálnej forme, ak pre každú jej závislosť $\mathcal{X} \rightarrow \mathcal{A}$ platí, že $\mathcal{X}$ je nadkľúč relácie $\mathcal{R}$. [RE03]*

3.1 Uzáver množiny atribútov

Hľadanie uzáveru nejakej množiny atribútov je dôležitou súčasťou skoro všetkých normalizačných algoritmov. V tejto časti ukážeme, ako sa pomocou metódy doClosure dá nájsť uzáver ľubovoľnej množiny atribútov. Nami prezentovaný algoritmus a aj iné algoritmy nájdenia uzáveru sú popísané v práci Miltona a Diedericha [DM88].

Metóda doClosure($\mathcal{S}$, $\mathcal{F}$) spraví uzáver množiny $\mathcal{S}$, čiže použitím všetkých závislostí z $\mathcal{F}$ získa všetky atribúty, ktoré je možné dosiahnuť z množiny $\mathcal{S}$. V tejto metóde sa postupne prechádza cez funkčné závislosti. Ak sa zistí, že množina atribútov $\mathcal{S}$ je nadmnožinou ľavej strany nejakej závislosti, zjednotí sa táto množina s atribútmi pravej strany a začína sa testovať od začiatku, pričom sa odstránila práve testovaná závislosť z množiny funkčných závislostí. Ak ľavá strana testovanej závislosti nie je podmnožinou testovanej podmnožiny relácie $\mathcal{R}$, tak algoritmus pokračuje testovaním ďalšej závislosti. Na výstupe metóda vráti upravenú množinu $\mathcal{S}$ tak, že $\mathcal{S}$ obsahuje okrem pôvodných atribútov aj ďalšie atribúty, ktoré sa z nej dajú odvodiť pomocou platných funkčných závislostí.

Pseudokód metódy doClosure($\mathcal{S}$, $\mathcal{F}$) je na obrázku 3.1.

VSTUP: Množina atribútov $\mathcal{S}$ ktorej uzáver sa hľadá, množina
funkčných závislostí $\mathcal{F}$
VÝSTUP: Množina atribútov, ktoré sa dajú odvodiť z $\mathcal{S}$
za použitia závislostí z množiny $\mathcal{F}$.

```
function doClosure(𝒮, ℱ)
BEGIN
i := 1
do
g := f_i ∈ ℱ
    if 𝒮 ⊇ lhs(g) then
        𝒮 := 𝒮 ∪ rhs(g)
        ℱ := ℱ - g
        i := 1
    else
        i := i + 1
    endif
while i <= |F|
return 𝒮
END
```

Obr. 3.1: Výpočet uzáveru množiny atribútov vzhľadom na množinu funkčných závislostí

3.2 Kľúče relácie

Kľúče relácie sú atribúty, z ktorých pomocou platných funkčných závislostí možno odvodiť všetky ostatné atribúty. Je dôležité pre ľubovoľnú reláciu vedieť povedať, či je nejaká jej podmnožina kľúčom, resp. nadkľúčom a takisto vedieť nájsť všetky kľúče danej relácie. V tejto časti sme sa preto venovali algoritmom na hľadanie kľúčov relácie a otestovaní jej ľubovoľnej podmnožiny, či je kľúčom.

Algoritmus nájdenia kľúčov dokáže nájsť všetky kľúče vstupnej relácie za použitia vstupnej množiny závislostí. Na nájdenie kľúčov treba v najhoršom prípade otestovať 2^n podmnožín vstupnej relácie $\mathcal{R}$, kde n je počet prvkov relácie $\mathcal{R}$.

Možno teda povedať, že algoritmus pre nájdenie kľúčov pracuje v exponenciálnej časovej zložitosti.

V tejto časti prezentujeme algoritmus nájdenia všetkých kľúčov metódou zdola nahor, pričom aj uvedieme pseudokód tohto algoritmu. Tiež popíšeme algoritmus, ktorý pracuje metódou zhora nadol. Pri postupe zdola nahor sa na začiatku vytvoria všetky podmnožiny a tie sa testujú, či je možné z nich pomocou množiny funkčných závislostí odvodiť celú reláciu. Na takto implementovanom algoritme sa dá dobre ukázať priebeh hľadania uzáveru jednotlivých podmnožín, ale tento algoritmus v porovnaní s inými algoritmami na nájdenie kľúčov nie je najlepší. Aj keď všetky algoritmy majú exponenciálnu zložitosť, existujú heuristiky pomocou ktorých sa dajú o niečo tieto algoritmy vylepšiť. Vylepšený algoritmus na hľadanie kľúčov možno nájsť v kuchárke normalizačných algoritmov od Jána Šturca [Stu], alebo iné v prácach Bernsteina, Beeriho [BB79] a Maiera [Dav80].

Algoritmus, ktorý hľadá všetky kľúče postupom zdola nahor, dostáva na vstup reláciu $\mathcal{R}$ a množinu všetkých závislostí $\mathcal{F}$ platiacich v relácií $\mathcal{R}$.

Na začiatku algoritmus vytvorí množinu $\mathcal{S}$, ktorá obsahuje všetky podmnožiny relácie $\mathcal{R}$. Algoritmus postupne prechádza cez všetky podmnožiny a zisťuje pomocou metódy *doClosure*, aké atribúty možno z danej podmnožiny dosiahnuť. Ak metóda *doClosure* vrátila množinu, ktorá obsahuje všetky atribúty vstupnej relácie $\mathcal{R}$, tak treba ešte otestovať, či nejaká iná podmnožina tejto množiny nie je zahrnutá medzi kľúčmi. Ak nie je, algoritmus pridá túto množinu medzi kľúče. Na testovanie, či nejaká podmnožina inej množiny je už zahrnutá medzi kľúčmi sme v uvedenom pseudokóde predpokladali, že existuje metóda subsetNotIn, ktorá to rozhodne.

Pseudokód algoritmu je na obrázku 3.2.

Algoritmus ktorý hľadá všetky kľúče relácie zhora nadol, dostáva na vstupe celú reláciu $\mathcal{R}$ a množinu funkčných závislostí $\mathcal{F}$. Tento algoritmus pracuje tak, že postupne prechádza cez všetky atribúty relácie a hľadá uzáver množiny, ktorá obsahuje reláciu, bez práve testovaného atribútu. Ak uzáver takejto množiny je celá relácia, algoritmus sa znova rekurzívne zavolá, pričom vstupnú reláciu predstavuje teraz množina vytvorená z relácie bez jedného atribútu. Takto sa algoritmus postupne môže vnárať, až nakoniec v najhoršom prípade otestuje všetky

```
VSTUP: Množina závislostí 𝓕, vstupná relácia 𝓡.
VÝSTUP: Množina všetkých kľúčov 𝓚.
function findKeys(𝓡, 𝓕)
BEGIN
𝓚 := ∅
𝓢 := Všetky podmnožiny relácie 𝓡
i = 1
for 𝓧 ∈ 𝓢 do
    𝓧 := doClosure(𝓧, 𝓕)
    if 𝓢 = 𝓡 then begin
        if subsetNotIn(𝓢,𝓚) then 𝓚 := 𝓚 ∪ 𝓢
    endif
endfor
return 𝓚
END
```

Obr. 3.2: Algoritmus hľadania všetkých kľúčov

podmnožiny vstupnej relácie $\mathcal{R}$.

Je vidieť, že hľadanie kľúčov zhora nadol je efektívnejšie ako zdola nahor, pretože pri takomto hľadaní kľúčov netreba nutne otestovať úplne všetky podmnožiny vstupnej relácie. Je to zapríčinené tým, že ak algoritmus prehlási o nejakej väčšej množine, že nie je kľúčom, tak už netreba testovať žiadnu z jej podmnožín.

Algoritmus na testovanie, či je nejaká podmnožina vstupnej relácie kľúčom, využíva algoritmus spravenia uzáveru množiny atribútov. Ak je výstupná množina tohto algoritmu totožná s celou reláciou, potom možno o testovanej množine atribútov prehlásiť, že je buď sama kľúčom, alebo obsahuje nejaký kľúč.

3.3 Minimálne pokrytie

Poznať minimálne pokrytie je dobré z dôvodu, že v minimálnom pokrytí sú odstránené všetky redundantné atribúty a aj všetky redundantné závislosti. Tiež je dobré vedieť testovať, či je nejaká množina funkčných závislostí minimálnym pokrytím inej množiny funkčných závislostí. V tejto časti sa preto venujeme hľadaniu a testovaniu minimálneho pokrytia.

Algoritmus minimálneho pokrytia sa skladá z troch častí. Sú to: Rozbitie pravých strán, minimalizácia ľavých strán, odstránenie redundantných závislostí. Algoritmus, ktorý hľadá minimálne pokrytie, má na vstupe množinu funkčných závislostí $\mathcal{F}$.

V priebehu algoritmu sa postupne mení vstupná množina funkčných závislostí. Po prvom kroku bude každá funkčná závislosť z $\mathcal{F}$ obsahovať na pravej strane len jednoduchý atribút, po druhom kroku sa na ľavých stranách vstupnej množiny $\mathcal{F}$ vynechajú redundantné atribúty a v poslednom kroku sa vynechajú aj celé redundantné závislosti. Po aplikovaní týchto troch krokov algoritmus vráti na výstupe množinu závislostí, ktorá je minimálnym pokrytím.

Algoritmus, ktorého priebeh sme tu uviedli, nájde jedno minimálne pokrytie. Ku každej množine závislostí existuje aspoň jedno minimálne pokrytie, ale môže existovať aj viacero minimálnych pokrytí. To, ktoré minimálne pokrytie algoritmus nájde, závisí od toho, v akom poradí sa testujú jednotlivé závislosti množiny $\mathcal{F}$ a takisto aj od toho v akom poradí sa testujú jednotlivé atribúty ľavej strany každej závislosti z $\mathcal{F}$.

Algoritmus nájde jedno minimálne pokrytie v čase $O(np)$,kde n je počet funkčných závislostí a p je polynóm.

Pseudokód algoritmu minimálneho pokrytia je na obrázku 3.3.

Prvý krok algoritmu minimálneho pokrytia je rozbitie pravých strán všetkých závislostí, tak že pravá strana bude obsahovať len jednoduchý atribút. Po aplikovaní tejto časti budú všetky závislosti v tvare $\mathcal{X} \rightarrow \mathcal{A}$, kde $\mathcal{A}$ je jednoduchý atribút. Algoritmus postupne prechádza cez všetky závislosti $\mathcal{F}$ a ak nájde závislosť, kde pravá strana obsahuje viacej ako jeden atribút, tak postupne prejde cez všetky atribúty pravej strany a k závislostiam pridá závislosť, kde ľavá strana je nezmenená a pravá strana je jednoduchý atribút z pôvodnej závislosti. Po prej-

```
VSTUP: Množina závislostí 𝓕
VÝSTUP: Upravená množina 𝓕 tak, že spĺňa
podmienku byť minimálnym pokrytím.
function minCover(𝓕)
𝓕 := splitPS(𝓕)
𝓕 := minLS(𝓕)
𝓕 := removeRedundantDependencies(𝓕)
return 𝓕
END
```

Obr. 3.3: Algoritmus hľadania minimálneho pokrytia množiny funkčných závislostí

dení všetkých atribútov na pravej strane sa odstráni celá pôvodná závislosť s viacerými atribútmi na pravej strane. Následne algoritmus pokračuje testovaním ďalšej závislosti. Po prejdení celej množiny závislostí $\mathcal{F}$ algoritmus vráti na výstupe množinu závislostí $\mathcal{F}$, kde pravá strana každej závislosti obsahuje už len jednoduchý atribút.

Pseudokód rozbitia pravých strán je na obrázku 3.4

Ďalším krokom algoritmu hľadanie minimálneho pokrytia je minimalizácia ľavých strán a vynechanie redundantných závislostí. Pri minimalizácii ľavých strán a pri vynechávaní redundantných závislostí sa používa metóda doClosure($\mathcal{S}$, $\mathcal{F}$), kde $\mathcal{S}$ je množina ktorej uzáver sa hľadá a $\mathcal{F}$ množina použitých funkčných závislostí.

Pri minimalizácii ľavých strán algoritmus dostáva na vstupe množinu funkčných závislostí $\mathcal{F}$. Počas svojho priebehu algoritmus postupne testuje každú závislosť $f \in \mathcal{F}$, tak že prejde cez všetky atribúty ľavej strany tejto závislosti. Pre každý atribút x ľavej strany sa urobí uzáver ľavej strany, ale bez atribútu x. Ak množina $\mathcal{S}$, ktorú vráti metóda doClosure(lhs(f) $- x$, $\mathcal{F}$), obsahuje celú pravú stranu f, tak testovaný atribút x je redundantným a môžeme ho vynechať. Po prejdení celej množiny $\mathcal{F}$ je ľavá strana každej závislosti bez redundantných atribútov. Na výstupe algoritmus vráti upravenú množinu funkčných závislostí $\mathcal{F}$.

```
VSTUP: Množina funkčných závislostí 𝓕
VÝSTUP: Upravená množina 𝓕, tak že každá závislosť z 𝓕 má
na pravej strane len jednoduchý atribút.
function splitPS(𝓕)
BEGIN
for f ∈ 𝓕 do
    if rhs(f) > 1 then
        for x ∈ rhs(f) do
            add(𝓕, x, lhs(f))
        endfor
        𝓕 := 𝓕 − f
    endif
endfor
return 𝓕
END
```

Obr. 3.4: Algoritmus rozbitia pravých strán množiny funkčných závislostí

Pseudokód minimalizácie ľavých strán je na obrázku 3.5.

Časťou, ktorá sa v algoritme minimálneho pokrytia volá ako posledná, je vynechanie redundantných závislostí. Na vstupe algoritmus dostáva množinu funkčných závislostí $\mathcal{F}$. V tejto časti algoritmus postupne prechádza všetky závislosti $f \in \mathcal{F}$. Nech metóda lhs(f) vráti ľavú stranu f. Pre každú závislosť f sa zavolá metóda doClosure(lhs(f), $\mathcal{F}$ - f). Zistia sa tak všetky atribúty, ktoré sú odvoditeľné z ľavej strany závislosti f, pričom sa ale použije množina závislostí $\mathcal{F}$ bez práve testovanej závislosti f. Ak množina, ktorú vráti metóda doClosure obsahuje všetky atribúty pravej strany f, možno povedať, že práve testovaná závislosť f je redundantná a možno ju teda vynechať z množiny závislostí. Po dokončení prechodu cez všetky závislosti už množina závislostí $\mathcal{F}$ neobsahuje žiadne redundantné závislosti, a keďže vynechanie redundantných závislostí je posledný krok algoritmu minimálneho pokrytia, tak po jeho aplikovaní sa množina závislostí $\mathcal{F}$ nazýva minimálnym pokrytím. Na výstupe algoritmus vráti takto upravenú

VSTUP: Množina závislostí $\mathcal{F}$

VÝSTUP: Upravená množina $\mathcal{F}$, tak že žiadna závislosť z $\mathcal{F}$ na ľavej strane neobsahuje žiadny redundantný atribút.

```
function minLS(𝓕)
BEGIN
for f ∈ 𝓕 do
    if lhs(f) > 1 then
        for x ∈ lhs(f) do
            𝓢 := doClosure(lhs(f) − x, 𝓕)
            if rhs(f) ⊆ 𝓢 then remove(lhs(f), x)
        endfor
    endif
endfor
return 𝓕
END
```

Obr. 3.5: Algoritmus minimalizácie ľavých strán funkčných závislostí

množinu funkčných závislostí $\mathcal{F}$.

Pseudokód časti, kde sa vynechávajú celé redundantné závislosti, je na obrázku 3.6.

Po zadefinovaní algoritmu, ktorý nájde minimálne pokrytie, je možné zaviesť aj algoritmus, ktorý otestuje, či nejaká množina funkčných závislostí $\mathcal{G}$ je minimálnym pokrytím inej množiny funkčných závislostí $\mathcal{F}$.

Algoritmus možno rozdeliť na 2 časti. V prvej časti algoritmus zistí, či je z ľavej strany každej funkčnej závislosti patriacej do $\mathcal{F}$ možné odvodiť jej pravú stranu, ale s použitím funkčných závislostí z množiny $\mathcal{G}$. Inými slovami, či možno povedať, že množina závislostí $\mathcal{G}$ pokrýva množinu závislostí $\mathcal{F}$. Toto sa dá realizovať pomocou volania metódy doClosure(lhs(f), $\mathcal{G}$), pre každú závislosť $f \in \mathcal{F}$. Ak množina, ktorú vráti metóda doClosure obsahuje celú pravú stranu závislosti f, tak algoritmus úspešne skončí svoju prvú časť. V opačnom prípade algoritmus skončí s negatívnou odpoveďou.

```
VSTUP: Množina závislostí 𝓕
VÝSTUP: Upravená množina 𝓕 tak, že 𝓕 neobsahuje
žiadnu redundantnú závislosť
function removeRedundantDependencies(𝓕)
BEGIN
for f ∈ 𝓕 do
    𝓢 := doClosure(lhs(f), 𝓕 - f)
    if rhs(f) ⊆ 𝓢 then 𝓕 := 𝓕 - f endif
endfor
return 𝓕
END
```

Obr. 3.6: Algoritmus vynechávania redundantných závislostí v množine funkčných závislostí

Vo svojej druhej časti musí algoritmus overiť, či je pokrytie $\mathcal{G}$ naozaj aj minimálnym pokrytím množiny funkčných závislostí $\mathcal{F}$. Toto sa dá realizovať pomocou volania metódy minCover($\mathcal{G}$). Ak množina, ktorú táto metóda vráti, je totožná s množinou funkčných závislostí $\mathcal{G}$, algoritmus prehlási $\mathcal{G}$ za minimálne pokrytie množiny $\mathcal{F}$.

Výpočtová zložitosť algoritmu, ktorý overí či je nejaká množina funkčných závislostí minimálnym pokrytím inej množiny funkčných závislostí, je rovnako ťažká úloha ako nájdenie minimálneho pokrytia. Druhá časť tohto algoritmu je totiž samotné nájdenie minimálneho pokrytia a prvá časť, kde sa hľadá uzáver nejakej množiny nie je výpočtovo zložitejšia ako nájdenie minimálneho pokrytia samotné. Zložitosť testovania minimálneho pokrytia je teda $O(np)$, kde n je počet jednotlivých funkčných závislostí a p je polynóm.

Pseudokód testovania minimálneho pokrytia je na obrázku 3.7.

```
VSTUP: Množina závislostí F, množina závislostí G o ktorej sa predpokladá,
že je minimálnym pokrytím F
VÝSTUP: true ak G je minimálnym pokrytím F, inak false
function testMinCover(F, G)
BEGIN
for f ∈ F do
    S := doClosure(lhs(f), G)
    if rhs(f) ⊈ S then return false endif
endfor
G_min := minCover(G)
if G_min ≠ G then return false
return true
END
```

Obr. 3.7: Algoritmus testovanie minimálneho pokrytia funkčných závislostí.

3.4 Tretia normálna forma

Algoritmus nájdenia tretej normálnej formy dostáva na vstup reláciu atribútov $\mathcal{R}$ a množinu závislostí, ktorá spĺňa podmienku byť minimálnym pokrytím $\mathcal{F}$. Na výstupe algoritmus vracia množinu tabuliek, ktoré tvoria tretiu normálnu formu.

Algoritmus v prvom kroku vytvorí množinu tabuliek tak, že prejde cez všetky závislosti a každá tabuľka obsahuje všetky prvky ľavej aj pravej strany jednej závislosti. Zároveň pre každú tabuľku platí, že nie je podmnožinou žiadnej inej tabuľky. V ďalšom kroku algoritmus overí, čí nejaká tabuľka obsahuje niektorý z kľúčov. Ak nie, algoritmus pridá do množiny tabuliek ďalšiu tabuľku, ktorá obsahuje všetky atribúty ľubovoľného kľúča relácie $\mathcal{R}$. Na výstupe algoritmus vráti množinu tabuliek, ktoré tvoria tretiu normálnu formu.

Pseudokód výpočtu tretej normálnej formy je na obrázku 3.8.

Algoritmus, ktorý overí, či nejaká relácia je v tretej normálnej forme je výpočtovo zložitý. Overenie podmienky, či je ľavá strana závislosti nadkľúč, alebo pravá strana súčasťou kľúča v relácií $\mathcal{R}$ je NP-ťažký problém.

Algoritmus, ktorý testuje, či vstupná množina tabuliek $\mathcal{S}$ tvorí rozklad do

```
VSTUP: Množina závislostí 𝓕, relácia 𝓡 ktorá obsahuje všetky
atribúty vystupujúce v množine závislostí 𝓕.
VÝSTUP: Množina tabuliek ktoré tvoria rozklad do 3NF vstupnej relácie 𝓡
function find3NF(𝓕, 𝓡)
BEGIN
𝓚 := findKeys(𝓕, 𝓡)
𝓕 := minCover(𝓕)
𝓢 := ∅
for f ∈ 𝓕 do
    𝓢 := 𝓢 ∪ { rhs(f) ∪ lhs(f) }
endfor
for 𝓧 ∈ 𝓢 do
    for 𝓨 ∈ 𝓢 do
        if 𝓧 ⊆ 𝓨 AND 𝓧 ≠ 𝓨 then remove(𝓢, 𝓧) endif
    endfor
endfor
if notContainsKey(𝓢, 𝓚) then add(k ∈ K, 𝓢)
return 𝓢
END
```

Obr. 3.8: Výpočet tretej normálnej formy

tretej normálnej formy s použitím funkčných závislostí $\mathcal{F}$, pre každú tabuľku nájde množinu $\mathcal{G}$ všetkých závislostí platných v danej tabuľke. To znamená, že množina $\mathcal{G}$ bude obsahovať také závislosti, ktorých ľavá aj pravá strana obsahuje len atribúty obsiahnuté v práve testovanej tabuľke. Následne algoritmus nájde všetky kľúče testovanej tabuľky s použitím všetkých funkčných závislostí $\mathcal{F}$. Čiže sa vytvorí množina $\mathcal{K}$, ktorá bude obsahovať výstup metódy findKeys($\mathcal{G}$, $\mathcal{F}$).

V ďalšom kroku algoritmus prejde všetky závislosti v množine $\mathcal{G}$ a otestuje či je ľavá strana každej závislosti nadkľúč, alebo pravá strana každej závislosti súčasťou nejakého kľúča. Ak algoritmus takýmto spôsobom s pozitívnym výsledkom otestuje každú tabuľku vstupnej množiny $\mathcal{S}$, tak vyhlási daný rozklad za

tretiu normálnu formu. V prípade, že niektorá tabuľka zo vstupu nespĺňa uvedenú podmienku, algoritmus zastaví svoju činnosť a povie, že vstupné tabuľky netvoria rozklad do tretej normálnej formy.

Pseudokód testovania tretej normálnej formy je na obrázku 3.9.

VSTUP: Množina závislostí $\mathcal{F}$, množina tabuliek $\mathcal{S}$ ktoré algoritmus otestuje či tvoria rozklad do 3NF.
VÝSTUP: true, ak tabuľky tvoria rozklad do 3NF, inak false
function test3NF($\mathcal{F}$, $\mathcal{S}$)
BEGIN
for $\mathcal{X} \in \mathcal{S}$ do
    $\mathcal{G}$:= všetky závislosti z $\mathcal{F}$ platné v $\mathcal{X}$
    $\mathcal{K}$:= findKeys($\mathcal{G}$, $\mathcal{F}$)
    for $f \in \mathcal{G}$ do
        for $k \in \mathcal{K}$ do
            if rhs(f) $\not\subseteq k$ OR lhs(f) $\not\supseteq k$ return false endif
        endfor
    endfor
endfor
return true
END

Obr. 3.9: Testovanie tretej normálnej formy

3.5 Boyce-Coddova normálna forma

Boyce-Coddovu normálnu formu (BCNF) možno dostať použitím algoritmu naivnej dekompozície. Možno tento algoritmus aplikovať hneď na vstupnú reláciu, alebo hľadať BCNF z už vytvorenej tretej normálnej formy. Pri naivnej dekompozícii sa na začiatku nájde závislosť, ktorá porušuje podmienku BCNF a minimalizuje sa jej ľavá strana. Následne sa relácia rozdelí na dve nové relácie. Jedna relácia obsahuje všetky atribúty ako pôvodná relácia, ale bez atribútov

pravej strany závislosti, ktorá porušovala podmienku BCNF. Druhá relácia obsahuje len atribúty tej závislosti, ktorá porušuje podmienku BCNF. Tento proces sa následne opakuje na všetky novo vzniknuté relácie, až dovtedy pokým všetky funkčné závislosti platné v jednotlivých reláciách nespĺňajú podmienku BCNF.

Algoritmus ktorý testuje, či vstupná množina tabuliek $\mathcal{S}$ tvorí rozklad do Boyce-Coddovej normálnej formy s použitím funkčných závislostí $\mathcal{F}$, má veľmi podobný priebeh ako algoritmus testovania tretej normálnej formy. Jediný rozdiel týchto dvoch algoritmov je v časti, kde sa overuje, či je ľavá strana nadkľúč a pravá strana súčasťou kľúča. Na rozdiel od algoritmu, kde sa testuje tretia normálna forma, treba v algoritme testovania Boyce-Coddovej normálnej formy aplikovať podmienku, že ľavá strana každej funkčnej závislosti musí byť nadkľúč. Ak je táto podmienka pre každú vstupnú tabuľku splnená, algoritmus prehlási vstupný rozklad za Boyce-Coddovu normálnu formu.

Problém rozhodnutia, či nejaká vstupná relácia s funkčnými závislosťami $\mathcal{F}$ ktoré v nej platia porušuje podmienku byť Boyce-Coddovou normálnou formou je NP-ťažké.

Pseudokód testovania Boyce-Coddovej normálnej formy je na obrázku 3.10.

VSTUP: Množina závislostí $\mathcal{F}$, množina tabuliek $\mathcal{S}$ ktoré algoritmus otestuje či tvoria rozklad do BCNF.
VÝSTUP: true, ak tabuľky tvoria rozklad do BCNF, inak false

```
function testBCNF(𝓕, 𝓢)
BEGIN
for 𝓧 ∈ 𝓢 do
    𝓖 := všetky závislosti z 𝓕 platné v 𝓧
    𝓚 := findKeys(𝓖, 𝓕)
    for f ∈ 𝓖 do
        for k ∈ 𝓚 do
            if rhs(f) ⊈ k return false endif
        endfor
    endfor
endfor
return true
END
```

Obr. 3.10: Testovanie Boyce-Coddovej normálnej formy

Kapitola 4

Implementácia normalizačných algoritmov

V tejto kapitole opíšeme ako sme implementovali jednotlivé normalizačné algoritmy v programovacom jazyku Java. Postupne opíšeme triedy *KeyFinder*, *NormalForm* a *MinimalCover*, ktoré implementujú algoritmy z kapitoly 3.

4.1 Implementácia algoritmov

Reláciu, ktorá obsahuje všetky atribúty, ktoré vystupujú vo funkčných závislostiach sme reprezentovali pomocou premennej *relation* typu hešovacia mapa. V tejto premennej sa mapuje meno každého atribútu na prislúchajúcu číselnú hodnotu, pre jednoduchšiu prácu s jednotlivými atribútmi.

V každej triede sme reprezentovali aj množinu funkčných závislostí použitím dvoch premenných typu spájaný zoznam hešovacích máp. Nazvali sme ich *leftSide* a *rightSide*. Premenná *leftSide* reprezentuje ľavú stranu množiny funkčných závislostí a premenná *rightSide* jej pravú stranu. Ľavú stranu každej funkčnej závislosti reprezentuje i-ta hešovacia mapa zo spájaného zoznamu *leftSide* a pravú stranu i-ta hešovacia mapa zo spájaného zoznamu *rightSide*.

Trieda *KeyFinder* slúži na implementáciu algoritmu nájdenia všetkých kľúčov zadanej relácie. Okrem premenných, ktoré obsahujú všetky definované triedy, obsahuje táto trieda ako privátne premenné spájané zoznamy hešovacích máp *subsets* a *result* ktoré sú inicializované ako prázdne spájané zoznamy. Spájaný

zoznam *result* slúži na uloženia nájdených kľúčov.

V konštruktore tejto triedy sa inicializujú premenné *leftSide*, *rightSide* a *relation*. Trieda *KeyFinder* obsahuje verejné metódy *findKeys* a *findKeysTopDown*. Metóda *findKeys* implementuje hľadanie všetkých kľúčov zdola nahor a vracia spájaný zoznam hešovacích máp, ktorý reprezentuje všetky kľúče vstupnej relácie. V tejto metóde sa postupne volajú ďalšie tri metódy. Ako prvá sa volá metóda *findSubsets*. Tá naplní premennú *subsets* všetkými podmnožinami vstupnej premennej *relation*. Následne metóda *findKeys* postupne prechádza for cyklom cez všetky hešovacie mapy obsiahnuté v premennej *subsets* a pre hešovaciu mapu na každej jej pozícii zavolá metódu *isKey*, ktorá dostane na vstup ako parameter hešovaciu mapu na aktuálnej pozícii spájaného zoznamu *subsets*.

Metóda *isKey* postupne prechádza pomocou while cyklu cez všetky prvky premennej *leftSide*. Ak premenná *leftSide* na aktuálnej pozícii obsahuje všetky hodnoty, ktoré obsahuje vstupná premenná, pridajú sa k vstupnej premennej aj všetky hodnoty obsiahnuté v premennej *rightSide* na rovnakej pozícii. Následne sa z premenných *leftSide* a *rightSide* odstránia hešovacie mapy na aktuálnej pozícii a while cyklus začne zase prechádzať od začiatku premennú *leftSide*. Po prejdení všetkých prvkov premennej *leftSide* sa overí, či po aplikovaní while cyklu obsahuje vstupná premenná všetky hodnoty, ktoré obsahuje aj premenná *relation*. Ak áno, metóda vráti hodnotu true, inak vráti hodnotu false.

Ak metóda *isKey* vráti hodnotu true, metóda *findKeys* pridá do spájaného zoznamu *result* nadkľúčov práve testovanú hešovaciu mapu. Po prejdení všetkých podmnožín dostaneme všetky nadkľúče vstupnej relácie. Posledná metóda, ktorá sa zavolá v metóde *findKeys* je metóda, ktorá odstráni hešovacie mapy ktorých prvky tvoria nadmnožinu niektorej hešovacej mapy. Táto metóda postupne prejde for cyklom cez všetky nadkľúče a pre každý prvok v zozname nadkľúčov nájde všetky jeho nadmnožiny a odstráni ich. Takto získaný výsledný spájaný zoznam *result* obsahuje už iba všetky kľúče vstupnej relácie.

Metóda *findKeysTopDown* implementuje algoritmus hľadania všetkých kľúčov zhora nadol. Táto metóda dostane ako parameter hešovaciu mapu, ktorá predstavuje reláciu, ktorej kľúče chceme nájsť. V tejto metóde sa rekurzivne testujú všetky podmnožiny vstupnej hešovacej mapy. A pomocou metódy *doClosure* z triedy minimálneho pokrytia sa hľadá ich uzáver. Ak nejaká hešovacia mapa

predstavuje kľúč, tak táto metóda pokračuje testovaním, jej o jeden atribút menších podmnožín. Výsledné hešovacie mapy ktoré predstavujú kľúče sa pridávajú do výsledného spájaného zoznamu *result*. Po vynorení zo všetkých rekurzívnych volaní, bude spájaný zoznam *result* obsahovať všetky kľúče danej relácie.

Znázornenie triedy KeyFinder je na obrázku 3.1.

KeyFinder
class

Fields

Properties

Public LinkedList leftSide{ get; set; }
Public LinkedList rifhtSide{ get; set; }
Public HashMap<Integer,String> relation{ get; set; }
Public LinkedList subsets{ get; set; }
Public LinkedList result{ get; set; }

Methods

+ void findSubsets( )
+ bool isKey( )
+ void findKeys( )
+ void findKeysTopDown()

Obr. 4.1: Znázornenie triedy KeyFinder

Algoritmus minimálneho pokrytia je implementovaný pomocou triedy *MinimalCover*. Táto trieda má metódy *splitPS*, *minLS*, *removeDependencies*, *doClosure*, *testMinCover* a *doMinCover*.

Metóda *doClosure* má na vstupe ako parameter hešovaciu mapu, ktorá predstavuje množinu atribútov, ktorých uzáver treba nájsť. Metóda prechádza while cyklom cez všetky hešovacie mapy obsiahnuté v spájanom zozname *leftSide*. Ak na nejakej pozícií hešovacia mapa v spájanom zozname *leftSide* obsahuje všetky prvky, ktoré obsahuje vstupná hešovacia mapa, pridajú sa do vstupnej mapy aj všetky atribúty obsiahnuté v mape v zozname *rightSide* na príslušnej pozícii a while cyklus sa nastaví na začiatok zoznamu *leftSide*. Tiež sa z premenných *leftSide* a *rightSide* odstránia mapy na práve testovanej pozícii. Po dokončení prechodu while cyklu metóda *doClosure* vráti vstupnú hešovaciu mapu rozšírenú

o ďalšie atribúty, ktoré sa dali z nej odvodiť.

Metóda *splitPS* implementuje rozbitie pravých strán množiny funkčných závislostí. Táto metóda for cyklom prechádza cez všetky hešovacie mapy zoznamu *rightSide* a ak je veľkosť konkrétnej hešovacej mapy na niektorej pozícii spájaného zoznamu *rightSide* väčšia ako 1, tak sa zo zoznamov *leftSide* a *rightSide* odstránia mapy reprezentujúce túto závislosť a zároveň vzniknú v zoznamoch *leftSide* a *rightSide* na konci nové prvky, tak aby spolu predstavovali novú závislosť, ktorá vznikla rozbitím jednej konkrétnej závislosti.

V metóde *minLS* sme implementovali minimalizáciu ľavých strán množiny funkčných závislostí pomocou dvoch for cyklov a volania metódy *doClosure*.

V metóde *removeDependencies* sme implementovali odstránenie redundantných funkčných závislostí, pomocou jedného for cyklu a volania metódy *doClosure*.

Metóda *doMinCover* obsahuje postupne volania metód *splitPS*, *minLS*, *removeDependencies*. Túto metódu sme následne využili pri implementácii algoritmu nájdenia minimálneho pokrytia.

Pomocou týchto metód sa dá implementovať aj algoritmus testovania, či nejaká množina vstupných závislostí je minimálnym pokrytím inej množiny funkčných závislostí. Dá sa to realizovať pomocou volania metód *doClosure* a *doMinCover* z triedy *MinimalCover*. To sme implementovali v metóde *testMinCover*.

Znázornenie triedy MinimalCover je na obrázku 4.2.

```
MinimalCover
class
Fields
Properties
  Public LinkedList leftSide{ get; set; }
  Public LinkedList rightSide{ get; set; }
  Public HashMap <Integer, String> relation{ get; set; }
Methods
  + void minLS( )
  + void removeDependencies( )
  + void splitRS()
  + HashMap<Integer, String> doClosure(HashMap<Integer, String> attributesSet)
  + bool testMinCover(LinkedList left,LinkedList right)
  + void doMinCover()
```

Obr. 4.2: Znázornenie triedy MinimalCover

Algoritmus nájdenia a testovania tretej, resp. Boyce-Coddovej normálnej formy sme implementovali pomocou triedy *NormalForm*. Okrem premenných, ktoré obsahujú všetky triedy, obsahuje táto trieda aj premennú *tables* typu spájaný zoznmam hešovacích máp. Táto premenná predstavuje tabuľky, ktoré tvoria rozklad do tretej normálnej formy. V konštruktore tejto triedy sa na začiatku inicializovali premenné *leftSide*, *rightSide* a *relation*. V konštruktore sa tiež vytvorila aj inštancia triedy *MinimalCover* a zavolala sa na nej metóda *doMinCover*. Čiže premenné *leftSide* a *rightSide* tejto inštancie, už v tomto okamihu predstavujú minimálne pokrytie. V konštruktore sa tiež inicializovala premenná *tables* tak, že sa naplnila mapami, ktoré sa zobrali zo zoznamov *leftSide* a *rightSide* minimálneho pokrytia.

Odstránenie jednotlivých podmnožín medzi tabuľkami sme implementovali pomocou metódy *removeSubsets*. V nej sme pomocou 2 for cyklov otestovali pre každú hešovaciu mapu v premennej *tables*, či nejaká iná hešovacia mapa neobsahuje všetky jej atribúty. Ak áno, tak sa táto hešovacia mapa zo zoznamu odstránila.

Posledná metóda v triede *NormalForm*, ktorá slúži na nájdenie tretej normálnej formy, je metóda *addKey*. V tejto metóde sa pomocou metódy *findKeys* z triedy *KeyFinder* vytvoril zoznam všetkých kľúčov relácie. Pomocou 2 for cyklov sa následne otestuje, či nejaká mapa v zozname *tables* obsahuje všetky atribúty nejakej mapy zo zoznamu všetkých kľúčov. Ak nie, pridá sa prvý kľúč, teda prvá mapa zo zoznamu kľúčov na koniec zoznamu *tables*. Takto vytvorený zoznam hešovacích máp *tables* už predstavuje tretiu normálnu formu.

Algoritmus, ktorý testuje, či je nejaká tabuľka v tretej normálnej forme sme implementovali metódou *test3NF* v triede *NormalForm*. Táto metóda dostala ako parameter hešovaciu mapu, ktorá predstavuje tabuľku, ktorú treba otestovať, či je v tretej normálnej forme. Pomocou metódy *findKeys* z triedy *KeyFinder* sa najskôr nájde zoznam hešovacích máp, ktorý predstavuje kľúče v danej tabuľke, pričom parametre v konštruktore triedy *KeyFinder* sú premenné *leftSide*, *rightSide* z tejto triedy *NormalForm* a tretím parametrom, ktorý predstavuje reláciu, ktorej kľúče treba nájsť, je hešovacia mapa z parametra tejto metódy.

Následne sa vyberú zo zoznamov *leftSide* a *rightSide* len tie hešovacie mapy, ktorých spájané zoznamy predstavujú závislosti platné vo vstupnej relácií. Následne sa for cyklom prejde cez tieto zoznamy a podľa zoznamu nájdených kľúčov

sa overí, či mapa na každej pozícii zoznamu *leftSide* a aj k nemu prislúchajúca mapa na tej istej pozícii zoznamu *rightSide* spĺňajú podmienku tretej normálnej formy. Overenie, či nejaký rozklad do tabuliek, ktoré sú reprezentované hešovacími mapami tvorí tretiu normálnu formu sa teda urobí tak, že pre každú hešovaciu mapu sa zavolá metóda *test3NF* triedy *NormalForm*.

Takým istým spôsobom ako testovanie tretej normálnej formy, sme implementovali aj algoritmus na testovanie Boyce-Coddovej normálnej formy v metóde *testBCNF* s rozdielom, že sa aplikovala podmienka, že každá závislosť platná v tabuľke musí spĺňať podmienku Boyce-Coddovej normálnej formy.

Znázornenie triedy NormalForm je na obrázku 4.3.

NormalForm
class
Fields
Properties
Public LinkedList leftSide{ get; set; }
Public LinkedList rightSide{ get; set; }
Public HashMap <Integer, String> relation{ get; set; }
Public LinkedList tables{ get; set; }
Methods
+ bool testBCNF(HashMap<Integer, String> table)
+ void removeSubsets()
+ bool test3NF(HashMap<Integer, String> table)
+ void addKey()

Obr. 4.3: Znázornenie triedy NormalForm

4.2 Applet pre prezentáciu algoritmov

Ako grafické rozhranie pre zobrazenie výsledkov jednotlivých algoritmov sme zvolili Java applet [JMS]. Applet je malý program, ktorý nie je určený na to aby bol sám spúšťaný, ale skôr na to aby bežal vo vnútri nejakej inej aplikácie. Takto sa dá applet vložiť do kódu nejakej html stránky pomocou tagu <applet>. Ten obsahuje aj atribút class, ktorého hodnota hovorí o ceste, kde je umiestnený skom-

pilovaný Java kód. Do webového prehliadača sa takto okrem html kódu dostane aj program, ktorý prezentuje daný applet. Samotný applet sa spúšťa v prehliadači na strane klienta a preto je potrebné pre správne fungovanie appletu mať nainštalovanú Javu a to aspoň JRE.

Môj applet pre zobrazovanie výsledkov pozostával z viacerých ďalších komponentov. Boli to komponenty JMenu, JButton, JPanel, JLabel, JCheckBox, JEditorPane a JScrollPane. Po načítaní stránky s appletom do prehliadača sa v applete zobrazí len horné menu typu JMenu, jedno prázdne plátno typu JEditorPane na ktorom sa zobrazujú výstupy jednotlivých algoritmov a tiež jedno plátno typu JEditorPane, ktoré slúži na zadávanie vstupov. V hornom menu appletu je možné si vybrať ktorú úlohu chceme riešiť.

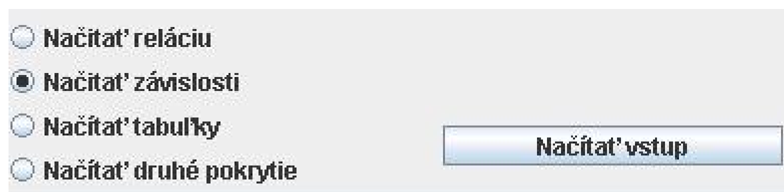
Ukážka výberu úlohy na riešenie z menu appletu je na obrázku 4.4.

Obr. 4.4: Ukážka výberu úlohy na riešenie z menu appletu

Pri načítavaní vstupov pre applet sme predpokladali, že užívateľ appletu má súbory, ktoré obsahujú v požadovanom tvare zapísanú reláciu, funkčné závislosti a rozklad do tabuliek. Obsah týchto súborov musí užívateľ skopírovať do plátna určeného na načítavanie vstupov v ľavej časti appletu a stlačiť tlačidlo nad ním, pričom ale musí byť pomocou JCheckBoxov zaznačené, ktorý typ vstupu sa práve načítava.

Výber typu načítaného obsahu je na obrázku 4.5.

Súbor užívateľa, ktorý obsahuje celú reláciu musí mať všetky atribúty relácie oddelené navzájom čiarkami. Napr. A,B,C,D . Vo vstupnom súbore ktorý obsahuje množinu závislostí musia byť závislosti navzájom oddelené znakom '|', ľavú stranu od pravej oddeľuje pomlčka, a atribúty každej strany sú navzájom oddelené čiarkou. Časť tohto súboru môže teda vyzerať napríklad takto: A,B-C,D|a,b-



Obr. 4.5: Výber typu načítaného obsahu

e. V súbore, ktorý obsahuje rozklad do tabuliek, musia byť tabuľky navzájom oddelené znakom '|' a jednotlivé atribúty každej tabuľky oddelené čiarkami.

Po výbere problému, ktorý sa ide riešiť, je pod horným menu appletu, umiestnené tlačidlo ktoré spustí riešenie jednotlivej úlohy. Po kliknutí sa na plátne typu JEditorPane v pravej časti appletu zobrazí buď celý výsledok, alebo len čiastkový a po opätovnom kliknutí sa zobrazí ďalšia časť výsledku.

Dodatok A

Príloha

Prikladáme CD, na ktorom je napálený applet na nájdenie kľúčov, minimálneho pokrytia, tretej normálnej formy a overenia či sú zadané vstupné tabuľky v tretej normálnej forme, alebo v Boyce-Coddovej normálnej forme spolu so zdrojovými kódmi appletu.

Literatúra

[BB79] Catriel Beeri and Philip A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Trans. Database Syst.*, 4(1):30–59, 1979.

[Ber76] Philip A. Bernstein. Synthesizing third normal form relations from functional dependencies. *ACM Trans. Database Syst.*, 1(4):277–298, 1976.

[Dav80] Maier David. Minimum covers in relational database model. *J. ACM*, 27(4):664–674, 1980.

[DM88] Jim Diederich and Jack Milton. New methods and fast algorithms for database normalization. *ACM Trans. Database Syst.*, 13(3):339–365, 1988.

[JMS] *The Source for Java Developers*. http://java.sun.com/.

[Kro00] David Kroenke. *Database Processing: Fundamentals, Design and Implementation*. Prentice Hall, 2000.

[RE03] Shamkant B. Navathe Ramez Elmasri. *Fundamentas of Database Systems*. Addison-Wesley, 2003.

[Stu] Jan Sturc. Kuchárka normalizačných algoritmov. http://www.dcs.fmph.uniba.sk/sturc/databazy/uvod/NFalgoritmy.pdf.

[TC05] Carolyn Begg Thomas Connolly. *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005.