

Úvod do databázových systémov

<http://www.dcs.fmph.uniba.sk/~plachetk>
/TEACHING/DB2013

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2013–2014

Niekoľko organizačných vecí: prednáška, cvičenia

Prednáška: Streda 11:30-13:00, B

Cvičenia: **Tento a budúci týždeň bežia teoretické cvičenia, v akváriách (t.j. NIE v M217 a M218).** Praktické cvičenia budú až niekedy v 3–4 týždni semestra. Info o praktickom týždni bude na prednáške a na WWW

- Rozvrh cvičení:

DB1 Streda 14:50, F1.328, 2iai1 2iai3

DB2 Streda 16:30, M.XII, 2iai2, *i*

DB3 Štvrtok 14:00, M.V, 2i2

DB4 Štvrtok 15:40, M-V, 2i1 2iai4

~~DBX Štvrtok 8:10, M-IV~~

V tomto týždni platí tento rozvrh. Za svoje priradenie do skupiny zodpovedáte sami, nezaťažujte s tým svojich učiteľov. **Na WWW prednášky nájdete link na systém organizácie cvičení.** Od budúceho týždňa platí rozvrh zapísaný v tomto systéme

Niekoľko organizačných vecí: cvičenia



Úvod do databázových systémov cvičenia

Login

Študent [Odhlásiť](#)

- [Zmeniť e-mail](#)
- [Zmeniť heslo](#)

Menu

- [Hlavná stránka](#)
- [Rozvrh podľa mena](#)
- [Rozvrh podľa skupín](#)
- [Zmeny rozvrhu, obsadenosť skupín](#)

Linky

- [Prednáška](#)

Obsadenie skupín podľa predmetov

DB teória

Názov	Čas	Miestnosť	Volných/Celkom/Fronta	Stav	Zmeniť stav?
DB1	Str 14:50	M-VI	5/28/0	Sem nechcem	Zmenit
DB2	Str 17:20	M-VI	12/28/0	Sem nechcem	Zmenit
DB3	Stv 08:10	M-VII	1/28/0	Sem nechcem	Zmenit
DB4	Stv 09:50	M-VII	12/28/0	Tu som	

DB praktikum

Názov	Čas	Miestnosť	Volných/Celkom/Fronta	Stav	Zmeniť stav?
DB1A	Str 14:50	M-217	2/14/0	Sem nechcem	Zmenit
DB1B	Str 14:50	M-218	3/14/0	Sem nechcem	Zmenit
DB2A	Str 17:20	M-217	6/14/0	Sem nechcem	Zmenit
DB2B	Str 17:20	M-218	6/14/0	Sem nechcem	Zmenit
DB3A	Stv 08:10	M-217	0/14/1	Sem chcem (1)	Zmenit
DB3B	Stv 08:10	M-218	1/14/0	Sem nechcem	Zmenit
DB4A	Stv 09:50	M-217	5/14/0	Tu som	
DB4B	Stv 09:50	M-218	7/14/0	Sem nechcem	Zmenit

Aktuality

Momentálne žiadne.

Tip: Zmenu robte len v nutnom prípade. (Inak môžete prísť o miesto v pôvodnej skupine!)

Systém pre organizáciu databázových cvičení | 2008-2010 | F. Hanes, F. Vojtko, T. Plachetka | FMFI, Univerzita Komenského Bratislava

Niekoľko organizačných vecí: konzultácie, hodnotenie

Konzultácie: okrem štandardného termínu (**utorok 14:00-16:00**) je možné termín dohodnúť osobne (napr. cez e-mail). **V skúškovom období konzultácie neposkytujem**

Hodnotenie: nutnou podmienkou kvalifikácie na skúšku je priebežné štúdium počas semestra („zápočet“, vyjadrený v percentách). Hodnotí sa najmä príprava na cvičenia a domáce úlohy
Hodnotenie práce počas semestra má rovnakú váhu ako skúška.
Na úspešné absolvovanie kurzu treba získať z každého čiastočného hodnotenia („zápočet“, „skúška“ aspoň 40% a zároveň v priemere aspoň 50%)

Tip: obzvlášť na cvičenia sa oplatí prísť včas, začnú krátkym (hodnoteným) písomným testom

Niekoľko organizačných vecí: niekoľko doporučení

Niekoľko doporučení (platia aj pre študentov, ktorí kurz opakujú):

- **Mám rád otvorenú (neanonymnú) kritiku.** Vecné komentáre a otázky k prednáškam a cvičeniam rozhodne neprispievajú k zhoršeniu hodnotenia—dokonca môžu prispiť k zlepšeniu
- Na vlaňajšej web stránke prednášky sú zverejnené úlohy z písomiek, aj s riešeniami. **Ak nájdete chybu v riešení niektornej vlaňajšej alebo tohoročnej úlohy, vysvetlite v čom tá chyba spočíva a pošlite mi korektné riešenie (e-mail).** Na chybu v prednáške môžete upozorniť okamžite. **Takúto prácu pokladám za zmysluplnú a honorujem ju bonusovými bodmi. (Bonusové body sa netýkajú kvalifikácie na skúšku, môžu len zlepšiť známku.)**
- **Naučte sa samostatne riešiť úlohy,** napríklad tie z predošlých písomných testov—nestačí prečítať si riešenie. (Riešenia staršie než 2–3 roky berte s rezervou. V tých môžu byť občas chyby, ktoré neplánujem opraviť.)
Začnite hned, nie až v decembri pred skúškou

Niekoľko organizačných vecí

- **Len pre študentov informatiky** (nie aplikovanej): do 2 týždňov, niektorý deň o 18:00, bude zhromaždenie venované ročníkovým projektom. Informácia o presnom termíne a mieste sa objaví ca. týždeň predtým na prednáške a WWW. Skúste dovtedy, každý sám sa seba, vymysliť a sformulovať na pár riadkoch projekt, na ktorom by ste radi pracovali

Čo je databáza (a čo je databázový systém)

Kolekcia nejakých dát (údajov)? Napríklad: databáza kníh, databáza fotografií, databáza používateľov v operačnom systéme, televízny program, databáza študentov na študijnom oddelení, ...

Áno, aj. Čo majú tieto príklady spoločné?

Napríklad, **dáta majú štruktúru**. Isteže, knihy sú popísané inak ako fotografie. Ale aj dve veľmi rôzne knihy majú spoločné **atribúty** (autor, názov, vydavateľ, rok vydania, ISBN atď.), líšia sa len hodnotami atribútov

Idea: Nevyrábať špecializované systémy zvlášť pre každú aplikáciu, ale vyrobiť **jeden univerzálny systém**, ktorý vie pracovať s **ľubovoľnou (pevnou) štruktúrou**. Ten istý systém je potom možné použiť na ukladanie údajov o knihách rovnako dobre ako o študentoch

Príklad (archaického) databázového systému: Masterfile

MASTERFILE III
DATA FILING AND RETRIEVAL For
HOME And BUSINESS

FOR AMSTRAD CPC 6128
(Also extended 128K CPC 464/664)

Dealer Details and Invoices

British United Freight 493 Western Avenue Gloucester GL9 5JN		Tel: 0452 677332 Contact: Mike Horne Ref: BUF	
Invoice	Tax point	Amount	Date paid
84294	20 Aug 84	£235.00	
84299	29 Aug 84		
84550	01 Oct 84		
84553			

Summary of Business Assets

Description	Maker	Model	Value
Microcomputer, 64K RAM + 32K ROM	Amstrad	CPC 464	£199.00
Microcomputer, 128K RAM + disc	Amstrad	CPC 6128	£299.00
Disc interface and 1st drive, 3"	Amstrad	DDI-1	£149.95
Dot-matrix printer 50cps 80col	Antler	DMP-2000	£159.95
Executive briefcase	British Steel	ATB109	£42.00
Wire paper clip	Canon	BWC	£8.01
Answering machine	British Telecom	BT12836	£185.00
Photocopier, single-feed	IBM	PC-18	£650.00
Executive jet aircraft	Lear	305-xxes	£3,200.00
Typewriter, electric	Olivetti	Cloud-Cuckoo	£130.00
Dictation machine	Philips	Leterra 36	£190.00
Coffee maker	Salter	510	£38.00
Parcel scale	Silicarn	HD5349	£119.95
Microcomputer, 48K	Maymaster	250P	£129.95
Letter scale		Trumspec	£10.00
Totals:		375KL	£5,202.294.26

* ALTER Record * I:insert E:erase A:alter R:raw data Other:exit
File: FILE2 Records:0016 Selected:0016 Parents:0000 RAM used:02K from 64K

Campbell Systems 

MASTERFILE III

MASTERFILE III is an information filing and retrieval program for use with the Amstrad CPC6128 or equivalent computer. The program runs under ASMDOS and requires just a single disc drive.

Designed, written and published by Campbell Systems, copyright 1986.
Author: John A Campbell

Reproduction or translation of this program or its documentation without the prior written permission of Campbell Systems is unlawful. Every effort has been made to ensure the correct performance of this software, but no liability can be accepted for direct or consequential loss as a result of its use.

Acknowledgement: We thank ARNOR Ltd whose MAXAM and PROTEXT software was used to develop, test, and document MASTERFILE III.

CPC6128, AMSTRAD and ASMDOS are trademarks of
Amstrad Consumer Electronics PLC.

CAMPBELL SYSTEMS
7 STATION ROAD, EPPING, ESSEX CM16 4HA, ENGLAND
TEL: EPPING (0378) 77762

Príklad (archaického) databázového systému: Masterfile

MASTERFILE 128 ver 3.1

Copyright 1986 CAMPBELL SYSTEMS

VERSION

FILE STATISTICS

← MAIN MENU

MESSAGE WINDOW

- Load/Save/Merge/CAT/Erase.....T
- Display/Print.....D
- Add new record.....A
- Select all records.....U
- Invert selection.....I
- Search the file.....S
- Purge all/some recs.....P
- Sort into Order.....O
- Data Names.....N
- Format report.....F
- Choose report from list.....C
- Load file direct.....L
- Invoke User BASIC.....U
- Export Data.....E
- List BASIC program.....B
- Send printer controls.....Q
- Adjust colours.....*

File: IN085 Records:0468 Selected:0036 Parents:0104 RAM used:21K from 64K

„Data Names“ umožňuje vytvoriť resp. zmeniť štruktúru databázy
(tej štruktúre sa v súčasnosti hovorí databázová schéma)

Príklad (archaického) databázového systému: Masterfile

Data Names

When creating a new file, the first step is to establish the data names and data references to be used in the file. Data names are listed when you press [N] at the main menu. Initially there are none to see, and you must follow the menus and prompts to insert them one by one.

```
          Data Names  
D:Description  
M:Maker  
T:Model  
S:Serial Number  
U:Replacement Value  
  
I:insert E:erase P:print X:exit  
File: FILE2    Records:0016  Selected:0016  Parents:0000  RAM used:02K from 64K
```

„Data names“ sa v súčasnosti nazývajú „attributes“ (atribúty)

Príklad (archaického) databázového systému: Masterfile

Summary of Business Assets			
Description	Maker	Model	Value
Microcomputer, 64K RAM + 32K ROM	Amstrad	CPC 464	£199.98
Microcomputer, 128K RAM + disc	Amstrad	CPC 6128	£299.98
Disc interface and 1st drive, 3"	Amstrad	DDI-1	£149.95
Dot-matrix printer 50cps 80col	Amstrad	DMP-2000	£159.95
Executive briefcase	Antler	AT0109	£42.00
Wire paper clip	British Steel	BMC	£8.01
Answering machine	British Telecom	BT2836	£185.00
Photocopier, single-feed	Canon	PC-10	£650.00
String, ball of	IBM	BOS-Exec	£8.48
Executive jet aircraft	Lear	Cloud-Cuckoo	£5,200,000.00
Typewriter, electric	Olivetti	Leterra 36	£130.00
Dictation machine	Philips	S18	£190.00
Coffee maker	Philips	HD5349	£30.00
Parcel scale	Salter	250P	£119.00
Microcomputer, 48K	Silicarn	Transpec	£129.95
Letter scale	Maymaster	375KL	£10.00
Totals:			£5,202,294.26

Top record = 0081 [H] for menu

File: FILE2 Records:0016 Selected:0016 Parents:0000 RAM used:02K from 64K

Pohľad na dátá uložené v databáze. Záznamy (records) sú organizované v riadkoch tabuľky. Záznamy sa dajú pridávať, odoberať, meniť, ... Záznamov môže byť **veľmi veľa**. Treba v nich vedieť **rýchlo vyhľadávať**

Problém s „hierarchickými“ dátami: Alby a pesničky

Ak album obsahuje viacero pesničiek (tracks), tak sa v tabuľke opakuje:

Album	Track
ABBA: Mamma Mia	Mamma Mia
ABBA: Mama Mia	Hey, Hey Helen
ABBA: MAMA MIA	Tropical Loveland
ABBA: MAMMA MIA	SOS
ABBA: Mamma Mia	Man In The Middle
ABBA: Mamma Mia	Bang-A-Boomerang
ABBA: Mamma Mia	I Do, I Do, I Do, I Do, I Do
So Nyeo Shi Dae: Girls Generation	Girls' Generation
Shi Dae: Girls' Generation	Oh La La!
So Nyeo Shi Dae: Girls' Generation	Baby Baby
So Nyeo Shi Dae: Girls' Generation	Complete

Redundancia, t.j. opakovanie rovnakých dát, znamená nielen plynvanie priestorom. Väčším problémom je prevádzka databázy, napr. vkladanie dát (veľa písania), aktualizácia dát (oprava preklepov) atď. Čo s tým?

Príklad: Albumy a pesničky

Track	Album
Mamma Mia	ABBA: Mamma Mia
Hey, Hey Helen	So Nyeo Shi Dae: Girls' Generation
Tropical Loveland	
SOS	
Man In The Middle	
Bang-A-Boomerang	
I Do, I Do, I Do, I Do, I Do	
Girls' Generation	
Oh La La!	
Baby Baby	
Complete	

Masterfile III ponúkal „parent-child“ abstrakciu: každá pesnička (child) patrí do nejakého albumu (parent). Rôzne pesničky môžu patriť do **rovnakého** albumu. Dáta jedného albumu sú **na jednom mieste!**

Príklad: Alby a pesničky

Album	Album_ID	Album_ID	Track
ABBA: Mamma Mia	0	0	Mamma Mia
So Nyeo Shi Dae: Girls' Generation	1	0	Hey, Hey Helen
		0	Tropical Loveland
		0	SOS
		0	Man In The Middle
		0	Bang-A-Boomerang
		0	I Do, I Do, I Do, I Do, I Do
		1	Girls' Generation
		1	Oh La La!
		1	Baby Baby
		1	Complete

Masterfile **navonok** prezentoval jednu tabuľku (a smerníky). Používateľ pracoval (len) s albumami. Mohol sa „vnoriť“ do niektorého albumu, pracovať potom (len) s pesničkami v tom albume a potom sa z albumu „vynoriť“ Ašak **interne** Masterfile používal dve tabuľky. Smerníky sa reprezentujú nejakými internými identifikátormi, ktoré príslušné záznamy spájajú

?–1960

- Jednoúčelové, navzájom nekompatibilné aplikácie, **práca s dátami „zadrôtovaná“ do štruktúr zvoleného programovacieho jazyka**, všelijaké formáty súborov vo všelijakých súborových systémoch, všelijaké počítače a pamäťové médiá, ...
- Priemyselné využitie aplikácií: bankové systémy, rezervačné systémy pre letecké spoločnosti, byrokracia veľkých firiem, knižnice

1960–1970

- Návrhy **jednotných DBMS** (Database Management Systems):
 - hierarchický (stromový) dátový model, à la albumy-pesničky
 - sietový (grafový) dátový model → štandard **CODASYL**, Committee on Data Systems and Languages
- Problém s CODASYL: ako automaticky prechádzať (prehľadávať) veľký graf? → potreba „programovacieho jazyka vyššej úrovne“

1970–dnes

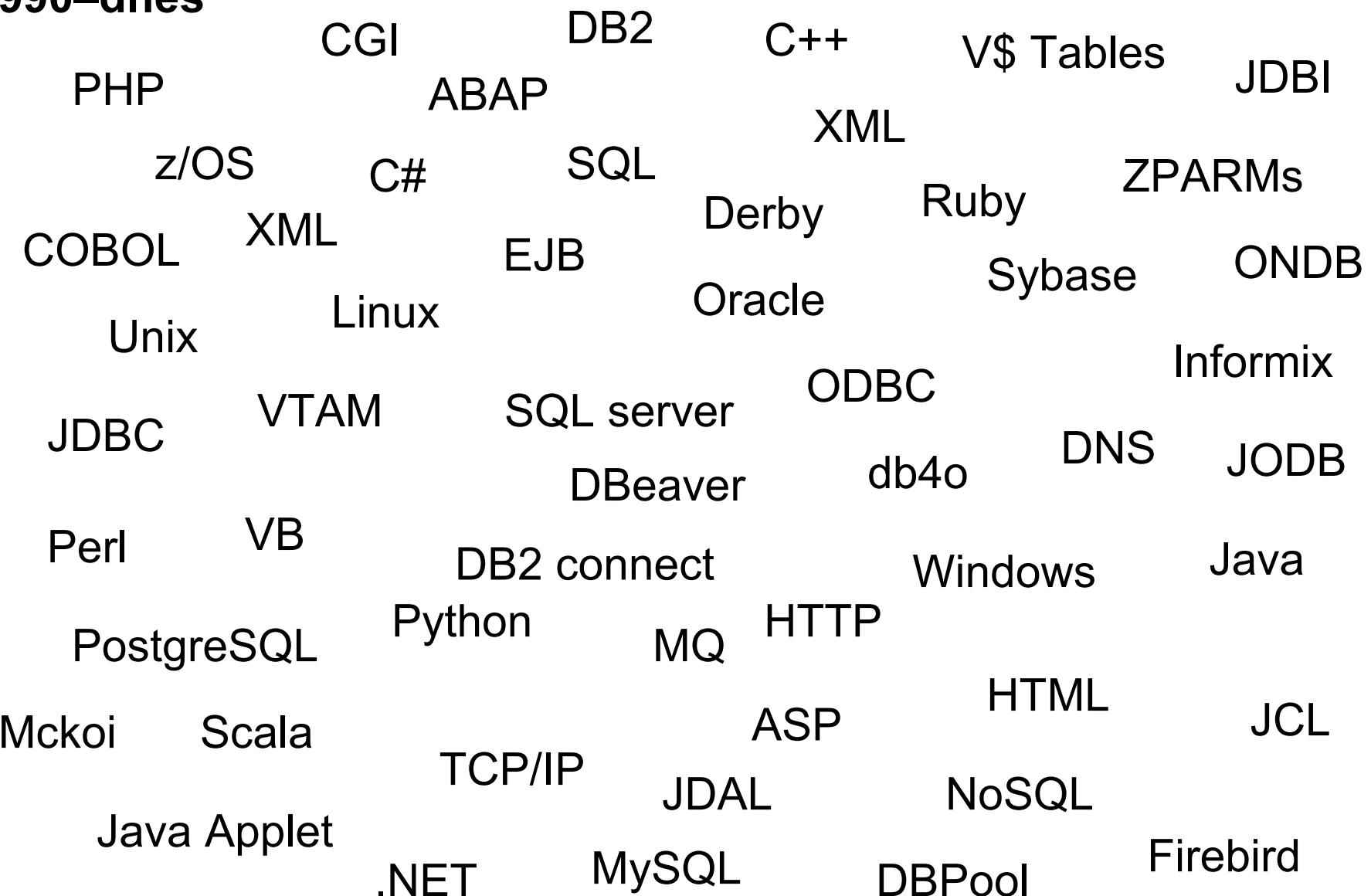
- Edgar F. Codd: A Relational Model of Data for Large Shared Data Banks (IBM, 1970). Idea 1: každá dátová štruktúra sa dá reprezentovať **reláciou** (tabuľkou). Idea 2: s malou, vhodne zvolenou množinou **tabuľkových operácií** sa dá „urobiť **všetko**“. Toto sa dá presne formalizovať a matematicky dokázať → **SQL**
- Codd bol v skutočnosti jedným z mnohých, ale tento článok sa väčšinou označuje ako prelomový

1986–dnes

- SQL, Standard Query Language, ISO a ANSI štandard od 1986, niekoľkokrát aktualizovaný (rozšírený). Posledná významná aktualizácia v 1999
 - **relačný dátový model**
 - príkazy na vytvorenie typov atribútov, vytvorenie relácií atď. (DDL, Data Definition Language)
 - kľúčový príkaz DML (Data Manipulation Language) je **SELECT**, ktorým sa formulujú dotazy
 - transakčné príkazy (BEGIN / COMMIT / ROLLBACK)
 - ...

História databázových systémov v svetovom merítku

1990–dnes



Dotazy (queries)

assets			
Description	Producer	Model	Value
Microcomputer, 64kB RAM	Amstrad	CPC464	199
Microcomputer, 128kB RAM	Amstrad	CPC6128	299
Disk interface	Amstrad	DDI	149
Printer	Amstrad	DMP2000	159
Executive briefcase	Antler	AT0109	42
Wire paper clip	British Steel	BWC	1
Answering machine	British Telecom	BT2836	185
Photocopier	Canon	PC10	650
Dictation machine	Philips	DM510	190
Coffee maker	Philips	HD5349	30

assets(Description, Producer, Model, Value)

Ktoré výrobky od Amstrad sú lacnejšie než £200?

Toto ešte Masterfile vedel (na každý riadok sa aplikuje rovnaká podmienka)

Dotazy (queries)

assets(*Description*, *Producer*, *Model*, *Value*)

Ktoré výrobky od Amstrad sú lacnejšie než £200?

V Pascale či v C hovoríme ako vypočítat' výsledok:

```
for i = 1 to N      ... kde 1 až N sú čísla riadkov tabuľky (poľa) assets  
    if assets[i].Producer = Amstrad and assets[i].Value < 200 then  
        print(assets[i].Model, assets[i].Description)
```

Všimnite si, že ešte pred napísaním tohto programu nám muselo byť jasné, **čo chceme vypočítat'**. Napríklad, museli sme si vyjasniť, že „výrobok“ znamená dvojicu [*Model*, *Description*]. V hovorovom jazyku je často problém vyjadriť sa presne. Okrem toho, chápanie hovorových viet je subjektívne. Presnejšie snáď bolo: „Nájdi (usporiadane) dvojice [*Model*, *Description*] z takých riadkov tabuľky *assets*, v ktorých *Producer* = ‘Amstrad’ a zároveň *Value* < 200.“

Dotazy (queries)

`assets(Description, Producer, Model, Value)`

Ktoré výrobky od Amstrad sú lacnejšie než £200?

Matematika (matematická logika) ponúka jazyk, ktorý umožňuje presne vyjadriť **čo treba vypočítať**: `{[Model, Description]}`:

`$\exists \text{Value } (\text{assets}(\text{Description}, \text{'Amstrad'}, \text{Model}, \text{Value}) \wedge \text{Value} < 200)$`

Tento jazyk sa nazýva relačný kalkul. Na reláciu `assets` sa dívame nie ako na tabuľku, ale ako na predikát, t.j. funkciu 4 argumentov, ktorá vracia TRUE len pre tie štvorce, ktoré sú v `assets` („spolu v jednom riadku tabuľky `assets`“)

Datalog je programovací jazyk priamo odvodený z matematickej logiky. Aj v Datalogu vyjadrujeme **čo treba vypočítať** (teda nie ako to vypočítať):

```
answer(Model, Description) ←  
    assets(Description, 'Amstrad', Model, Value), Value < 200.
```

Syntax Datalogu je veľmi jednoduchá, vyjadrovacia sila je veľká

Dotazy (queries)

assets(Description, Producer, Model, Value)

Ktoré výrobky od Amstrad sú lacnejšie než £200?

Aj v **SQL** vyjadrujeme **čo treba vypočítať** (teda nie ako to vypočítať):

```
select Model, Description
```

```
from assets
```

```
where Producer = 'Amstrad' and Value < 200;
```

Výsledkom dotazu (akéhokoľvek) je opäť relácia (tabuľka)

Syntax SQL je zbytočne „barokná“. SQL ponúka všetjaké (zbytočné) výrazové prostriedky, ktoré nijako nezvyšujú výpočtovú silu—a skôr komplikujú porozumenie dotazom a tvorbu kompilátorov. Navyše, niektoré dôležité veci sa stratia

Dotazy (queries)

assets(Description, Producer, Model, Value)

Ktoré výrobky od Amstrad sú lacnejšie než £200?

K „baroknej“ syntaxi SQL. Nasledujúce dotazy vyjadrujú to isté (malichernosti zanedbáme):

select Model, Description from assets

where Producer = 'Amstrad' and Value < 200;

select Model, Description from assets where Producer = 'Amstrad'

intersect

select Model, Description from assets where Value < 200;

select Model, Description from assets

where Producer = 'Amstrad' and [Model, Description] in

(select Model, Description from assets where Value < 200);

Ktorý z horeuvedených spôsobov je výpočtovo najviac efektívny?

Dotazy (queries)

assets(Description, Producer, Model, Value)

Ktoré výrobky od Amstrad sú lacnejšie než £200?

V **relačnej algebre** vyjadrujeme **ako vypočítat'** výslednú reláciu:

answer = $\Pi_{\text{Model}, \text{Description}} (\sigma_{\text{Producer} = \text{'Amstrad'} \wedge \text{Value} < 200} (\text{assets}))$

Relačná algebra má niekoľko (málo) operátorov, ktoré dostanú tabuľku resp. tabuľky a vypočítajú výslednú tabuľku.

π je **projekcia**. $\Pi_{\text{Description}, \text{Model}} (\text{assets})$ z relácie assets vyberie stĺpce Description a Model

σ je **selekcia**. $\sigma_{\text{Producer} = \text{'Amstrad'} \wedge \text{Value} < 200} (\text{assets})$ z relácie assets vyberie riadky, ktoré spĺňajú uvedenú podmienku

\times je **kartézskej súčin** dvoch relácií. O tom neskôr

\cap , \cup , $-$ sú **množinové operácie** (relácie musia byť „kompatibilné“)

S týmto sa dá vyjadriť “všetko”. (OK, takmer. Treba pridať ešte 1–2 operátory.)

Dotazy (queries)

`assets`(`Description`, `Producer`, `Model`, `Value`)

Ktoré výrobky od Amstrad sú lacnejšie než £200?

Pozor, **rôzne „ekvivalentné zápisy dotazu“ v relačnej algebre** znamenajú **rôzne výpočty!**

Napríklad,

$\Pi_{\text{Description}, \text{Model}} (\sigma_{\text{Producer} = \text{'Amstrad'}} \wedge \text{Value} < 200 \text{ (assets)})$

môže byť **výpočtovo oveľa efektívnejšie** než

$\Pi_{\text{Description}, \text{Model}} (\sigma_{\text{Producer} = \text{'Amstrad'}} \text{ (assets)}) \cap \Pi_{\text{Description}, \text{Model}} (\sigma_{\text{Value} < 200} \text{ (assets)})$

Pritom oba tieto dotazy počítajú rovnaký výsledok (pre akékoľvek naplnenie relácie assets). Akurát počítajú iným spôsobom. Keď operátory π , σ , \cap naprogramujete trebárs ako funkcie v C či v Pascale, tak si uvedomíte rozdiel medzi tými dvomi výpočtami

Dotazy (queries)

assets			
Description	Producer	Model	Value
Microcomputer, 64kB RAM	Amstrad	CPC464	199
Microcomputer, 128kB RAM	Amstrad	CPC6128	299
Disk interface	Amstrad	DDI	149
Printer	Amstrad	DMP2000	159
Executive briefcase	Antler	AT0109	42
Wire paper clip	British Steel	BWC	1
Answering machine	British Telecom	BT2836	185
Photocopier	Canon	PC10	650
Dictation machine	Philips	DM510	190
Coffee maker	Philips	HD5349	30

assets(Description, Producer, Model, Value)

Ktorí výrobcovia dodávajú každý svoj výrobok za menej než £200?

Toto už Masterfile **nevedel** (bolo to treba urobiť ručne, resp. s iným nástrojom)

Dotazy (queries)

assets(Description, Producer, Model, Value)

Ktorí výrobcovia dodávajú každý svoj výrobok za menej než £200?

Relačný kalkul:

$$\{P: (\exists D \exists M \exists V \text{ assets}(D, P, M, V)) \wedge \\ (\forall D \forall M \forall V (\text{assets}(D, P, M, V) \Rightarrow V < 200))\}$$

Alebo, ekvivalentne:

$$\{P: (\exists D \exists M \exists V \text{ assets}(D, P, M, V)) \wedge \\ \neg (\exists D \exists M \exists V \text{ assets}(D, P, M, V) \wedge V \geq 200)\}$$

Prečo nestačí napísat' $\{P: \forall D \forall M \forall V (\text{assets}(D, P, M, V) \Rightarrow V < 200)\}$?

(Ekvivalentne, $\{P: \neg (\exists D \exists M \exists V \text{ assets}(D, P, M, V) \wedge V \geq 200)\}$)

Dotazy (queries)

assets(Description, Producer, Model, Value)

Ktorí výrobcovia dodávajú každý svoj výrobok za menej než £200?

{P: ($\exists D \exists M \exists V \text{ assets}(D, P, M, V) \wedge$
 $\neg (\exists D \exists M \exists V \text{ assets}(D, P, M, V) \wedge V \geq 200)$)}

Datalog:

```
delivers_some_expensive_product(P) ← assets(_, P, _, V), V >= 200.  
answer(P) ← assets(_, P, _, _),  $\neg$  delivers_some_expensive_product(P).
```

SQL:

```
create temporary table delivers_some_expensive_product as  
select Producer from assets where Value >= 200;
```

```
select Producer from assets where Producer not in  
(select Producer from delivers_some_expensive_product);
```

Dotazy (queries)

assets(Description, Producer, Model, Value)

Ktorí výrobcovia dodávajú každý svoj výrobok za menej než £200?

$$\{P: (\exists D \exists M \exists V \text{ assets}(D, P, M, V)) \wedge \\ \neg (\exists D \exists M \exists V \text{ assets}(D, P, M, V) \wedge V \geq 200)\}$$

Relačná algebra:

delivers_some_expensive_product = $\pi_{\text{Producer}}(\sigma_{\text{Value} \geq 200}(\text{assets}))$

answer = $\pi_{\text{Producer}}(\text{assets}) - \text{delivers_some_expensive_product}$

Transakcie

Od databázových systémov sa vyžaduje nielen riešenie manipulácie s dátami (dotazy, vkladanie, mazanie a aktualizácie dát). **Musia počítať so súčasným prístupom viacerých používateľov k databáze**

Príklad: Mary a John majú spoločný účet v banke, Acc, na ktorom je teraz £1000. Mary je práve v banke a v pokladni ukladá na Acc £200. John v tej chvíli vyberá z Acc £10

```
deposit(Acc, 200)
```

```
{
```

```
    int mbalance;  
    mbalance = read(Acc);  
  
    mbalance = mbalance + 200;  
    write(Acc, mbalance);
```

```
}
```

1000
1200
1200

```
withdraw(Acc, 10)
```

```
{
```

```
    int jbalance;  
  
    jbalance = read(Acc);  
  
    jbalance = jbalance - 10;  
    if (jbalance < 0)  
        ERROR();  
    write(Acc, jbalance);
```

1000

990

990

Na účte Acc má byť £1190, lenže je len £990. Mary ani John za to nemôžu. Kto teda?

Transakcie

Problém strateného vkladu spôsobilo „zlé časovanie“. Presnejšie, **problém spôsobila banka**, ktorej databázový systém nerátal so „zlým časovaním“

Od databázových systémov sa vyžaduje, aby „každému klientovi vytvorili ilúziu, ako keby s databázou pracoval sám“. Okrem toho sa vyžaduje **odolnosť voči výpadkom v ľubovoľnom momente**. Napríklad, kedykoľvek môže vypadnúť elektrický prúd v banke či v bankomate. Kedykoľvek sa môže prerušiť spojenie medzi klientom a bankou. Kedykoľvek sa môže sa pokaziť hard-disk či počítač. Atď. Systém sa musí z takýchto havárií zotaviť, **databáza musí zostať konzistentná**

Centrálné postavenie databáz v informatike

Programovanie
a metodológia
programovania

Matematická logika

Dátové štruktúry
a zložitosť

Databázy

Operačné
systémy

Distribuované
systémy

Počítačové siete

Teória databázových systémov súvisí s mnohými oblastami informatiky

„Some of today's most beautiful theoretical results in mathematical logic are in ***finite model theory***, an area derived directly from database theory“ (T. Griffin,
University of Cambridge, UK)

Príklady databázových aplikácií

- Banky, poist'ovne
- Akciové a iné burzy (eBay)
- Knižnice
- Rezervačné systémy (letecké spoločnosti, železnice, hotely, cestovné kancelárie, požičovne áut, ...)
- Supermarkety
- Firmy (účtovníctvo, sklady, adresáre, ...)
- Fyzika (meteorológia, geológia, geografia, ...)
- Štatistika
- Operačné systémy (filesystémy, databázy užívateľov, databázy procesov, ...)
- ...

Náplň kurzu (nie nutne v tomto poradí, nie nutne všetko)

- Matematické základy relačného modelu (relácie, domény, vzťah relačnej algebry a relačného kalkulu)
- Dotazovacie jazyky (relačný kalkul, Datalog, SQL, relačná algebra)
- Modelovanie reality (návrh databáz)
- Teória návrhu relačných databáz (funkčné závislosti, klúče, normálne formy)
- Transakcie (operácie, rozvrhy, obnoviteľnosť, sériovateľnosť)
- Fyzický dátový model (fyzické plány, indexy)

Len okrajovo, možno vôbec: bezpečnosť, kompresia dát, XML, ODBC, JDBC, ...

Na cvičeniach sa budú cvičiť modro označené témy
(vybrané časti z nich)

Literatúra

- **H. Garcia-Molina, J.D. Ullman, J. Widom:** Database Systems, The Complete Book, Prentice Hall 2003
 - **R. Elmasri, S.B. Navathe:** Fundamentals of Database Systems, Addison-Wesley 2006
 - **J.D. Ullman, J. Widom:** A First Course in Database Systems, Prentice Hall 1997;
- <http://www-db.stanford.edu/~ullman/fcdb.html>**
- **M. Kifer, P.A. Bernstein, P.M. Lewis:** Database Systems: An Application-Oriented Approach, Addison-Wesley 2006
 - **S. Abiteboul, R. Hull, V. Vianu:** Foundations of Databases
 - **C.J. Date:** An Introduction to Database Systems, Addison-Wesley 2003

Literatúra

Stále aktuálna (hoci stará) kniha o transakciách:

P.A. Bernstein, V. Hadzilacos, N. Goodman: Concurrency Control and Recovery in Database Systems, Addison-Wesley 1987; <http://research.microsoft.com/pubs/ccontrol>

Online materiály zo Stanford University (obzvlášť doporučujem vypočuť si videoznamy krátkych prednášok J. Widom):

<http://www.db-class.org>

Československá literatúra:

- **J. Pokorný:** Databázové systémy a jejich použití v informačních systémech, Akadémia 1992
- **A. Scheber:** Databázové systémy, Alfa 1988

- **Časopisy:**
 - ACM TODS (Transactions on Database Systems)
 - ACM SIGMOD (Management of Data)
- **Konferencie:**
 - PODS: Principles of Database Systems
 - VLDB: International Conference on Very Large Data Bases
 - EDBT: International Conference on Extended Database Technology
 - FODO: International Conference on Foundations of Data Organization
 - SIAM Data Mining Conference
 - Datakon, Datasem

Spoločné charakteristiky databázových aplikácií

- Dáta majú štruktúru, ktorá sa zriedka mení, objem dát je veľký
- Dáta nie sú uložené na používateľovom počítači, pre prístup k dátam sa využíva počítačová siet (klient-server)
- Dotazy sú zložité („Koľko výrobkov, ktoré našej firme dodávajú iba čínski dodávatelia, sme predali vlni v Terchovej?”)
- Množstvo užívateľov pristupuje k dátam súčasne
- Vyžaduje sa vysoká priepustnosť
- Vyžaduje sa vysoká odolnosť voči neočakávaným výpadkom
- Vyžaduje sa vysoký stupeň bezpečnosti, t.j. ochrany proti neoprávnenému prístupu k dátam a zlým úmyslom používateľov
- Prístup koncových používateľov k dátam musí byť jednoduchý a silný (API, GUI)

Databázové systémy (DBMS) implementujú to, čo je v rôznych databázových aplikáciách spoločné

Prečo nastačí napríklad filesystem?

- Žiadna ochrana proti úplnej či čiastočnej strate dát (ak vypadne prúd, tak operácia, ktorá mala modifikovať dva súbory, mohla stihnuť modifikovať len jeden z nich)
- Žiadna alebo nedostatočná bezpečnosť (práva na prístup k súborom sú príliš „hrubozrnné“)
- Žiaden API (application programming interface) pre prístup k dátam
- Žiadna alebo nedostatočná ochrana dát v prípade súčasného prístupu viacerých užívateľov k dátam (jeden číta, druhý modifikuje)
- Žiadna kompatibilita s inými databázovými systémami

„Definícia“ databázových systémov (DBMS)

„Definícia“ DBMS (Database Management System): **Systém, ktorý poskytuje pohodlné, bezpečné, mnohoužívateľské prostredie pre efektívnu manipuláciu s veľkým objemom perzistentných dát**

Slovo „pohodlné“ treba chápať v profesionálnom význame, t.j. rozhranie DBMS by malo umožniť **vyjadriť** “**všetko**” **s minimom vyjadrovacích prostriedkov** (pohodlie aplikačného programátora je druhoradé)

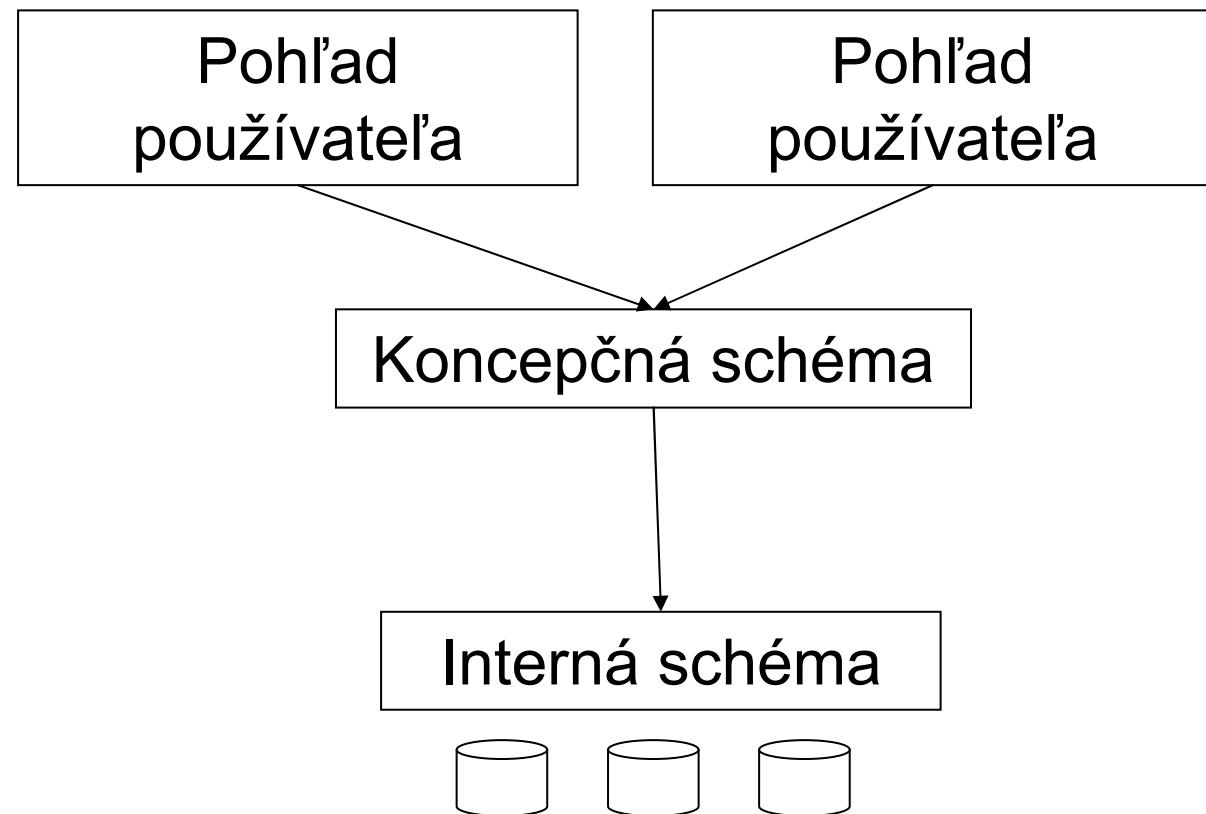
Trojstupňová architektúra DBMS (ANSI/SPARC)

Cieľ: nezávislosť programov od dát (pohľad používateľa nesmie byť závislý na internej reprezentácii dát)

Rôzne aplikáčné pohľady (napr. vyššie dotazovacie jazyky)

Koncepčný dátový model (napr. relačný)

Fyzický dátový model a perzistentné uloženie dát na médiách



Koncepčný dátový model je matematická notácia dát a manipulácie s dátami (algebra)

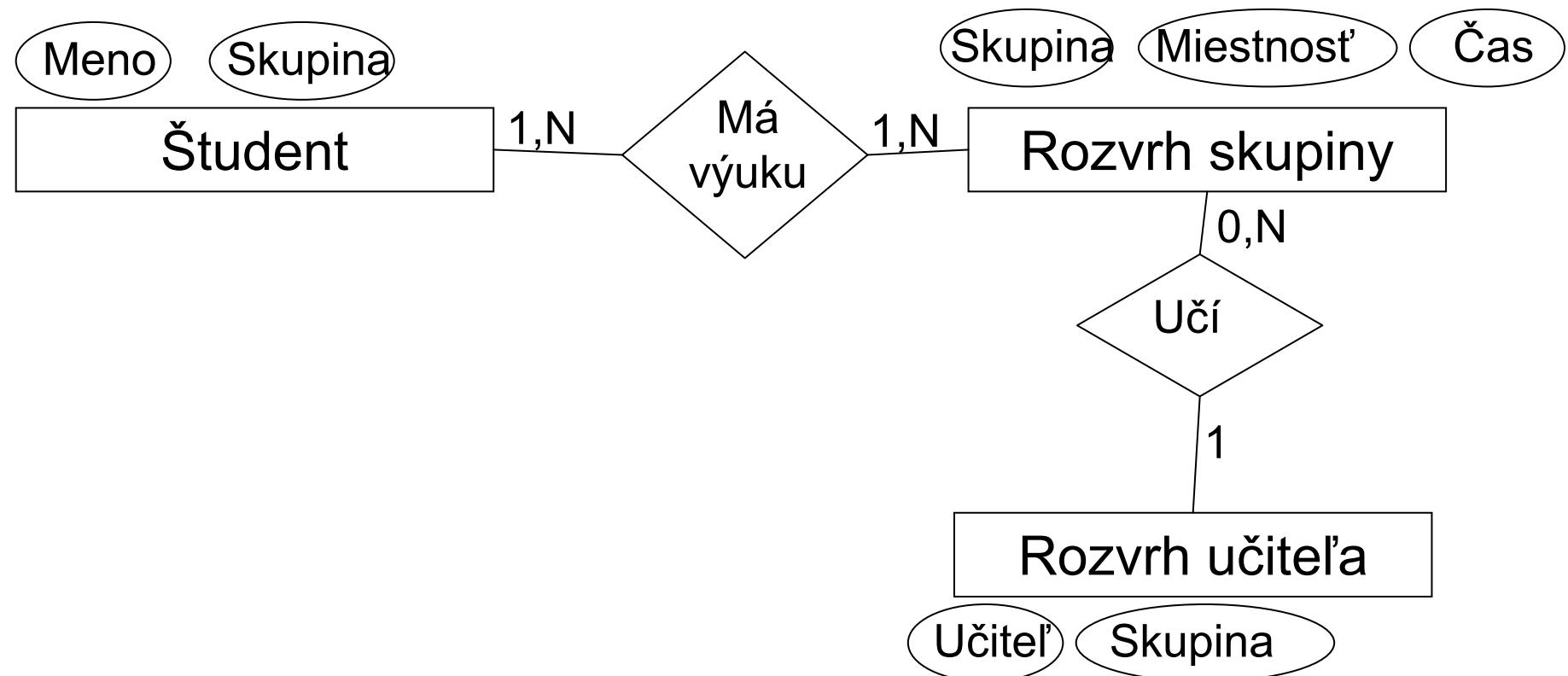
Používané dátové modely:

- entitno-relačný
- relačný
- navigačný (XML)
- objektový

Entitno-relačný dátový model

Entitno relačný model popisuje entity, atribúty entít a vzťahy medzi entitami. **Nemá však žiadne operácie**

Príklad ER diagramu:



Relačný dátový model

Jediným konceptom je relácia (tabuľka). Toto vymyslel Codd zhruba v 1970

DBMS ako napr. Oracle, Sybase, MySQL, Postgres atď. používajú **relačný dátový model**, preto sa im hovorí relačné DBMS

Relačné operácie:

- zjednotenie, prienik, rozdiel
- kartézsky súčin
- selekcia
- projekcia
- ...

XML (navigačný dátový model)

XML sa používa na popis štruktúrovaných textov. **Koncepty sú strom (linearizovaný do textu správnym uzátvorkovaním tagov) a odkaz (link).** DTD (Document Type Definition) je jazyk, v ktorom sa popisuje gramatika tagov (t.j. syntakticky správne vnorenia tagov) pre daný typ dokumentu

Operácie pre prehľadávanie stromu (XPath):

- home
- up, down
- first_sibling, next_sibling, prev_sibling
- first_down, next_down, prev_down

XPath je jednoduchý jazyk na “manuálne” traverzovanie stromu, t.j. zápis operácií (konceptná úroveň). XQuery je vyšší dotazovací jazyk (aplikáčná úroveň)

Úvod do databázových systémov

[http://www.dcs.fmph.uniba.sk/~plachetk
/TEACHING/DB2013](http://www.dcs.fmph.uniba.sk/~plachetk/TEACHING/DB2013)

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2013–2014

Definícia: **relácia** je podmnožina kartézskeho súčinu množín.

Formálne, n-árna relácia r je definovaná ako

$$\langle r \subseteq D_1 \times D_2 \times \dots \times D_n, D_1, D_2, \dots, D_n \rangle$$

Prvkami relácie sú **usporiadané n-tice** (hovorí sa im tiež n-tuples, tuples, n-tice, záznamy). Jednotlivým zložkám tých n-tíc sa hovorí **atribúty** (argumenty) relácie. Množinám D_1, D_2, \dots, D_n sa hovorí **domény (typy) atribútov** relácie r

Ako vieme popísat **konkrétnu** reláciu? Najjednoduchšie je **vymenovať** všetky n-tice, ktoré do nej patria. Napríklad boolovskú reláciu f vieme popísat takto:

$$f = \{[0, 0, 0], [0, 1, 0], [1, 0, 0], [1, 1, 1]\}.$$

Treba ešte dodať, že $f \subseteq \{0, 1\} \times \{0, 1\} \times \{0, 1\}$, t.j., že atribúty X, Y a Z môžu mať v relácii $f(X, Y, Z)$ len hodnoty 0 a 1. Domény atribútov obvykle nebudeme uvádzat'

Relácie

Sú aj nekonečné relácie. Napríklad:

plus = {[X, Y, Z] ∈ N × N × N: Z = X + Y}

mensi = {[X, Y] ∈ N × N: X < Y}

V týchto prípadoch sú domény dôležité, lebo toto sú **iné** relácie:

realne_plus = {[X, Y, Z] ∈ R × R × R: Z = X + Y}

realne_mensi = {[X, Y] ∈ R × R: X < Y}

Reláciu tvoria **usporiadane n-tice**. Kartézsky súčin nie je komutatívny, takže **na poradí atribútov záleží**. Ale ak sú atribúty relácie pomenované, tak ich môžeme ľubovoľne premiešať.

Napríklad relácie r(Mesto, Psc) a r(Psc, Mesto) budeme v prípade pomenovaných atribútov považovať za ekvivalentné. **V SQL sa atribúty relácie adresujú menom**: Mesto, Psc

Alebo: r.Mesto, r.PSC.

Predikáty

Definícia: Nech r je n -árna relácia

$\langle r \subseteq D_1 \times D_2 \times \dots \times D_n, D_1, D_2, \dots, D_n \rangle$. Potom **n -árny predikát prislúchajúci relácii** r je funkcia $r_p: D_1 \times D_2 \times \dots \times D_n \rightarrow \text{boolean}$, pričom $r_p(A_1, \dots, A_n) = \text{TRUE}$ práve vtedy keď $[A_1, \dots, A_n] \in r$

Je praktické nerozlišovať medzi menom relácie a menom predikátu, ktorý tej relácii prislúcha. Konvenciou bude, že mená predikátov a relácií budú písané malými písmenami. Konštanty v argumentoch predikátov budú písané malými písmenami (vždy), premenné veľkými písmenami (vždy)

Napríklad, reláciu $r(\text{Mesto}, \text{Psc})$ „stotožníme“ s predikátom $r(\text{Mesto}, \text{Psc})$ tak, že relácia $r(\text{Mesto}, \text{Psc})$ bude obsahovať (všetky a len tie) n -tice $[M, P]$, pre ktoré platí predikát $r(M, P)$

Atribúty (argumenty) predikátov sa adresujú pozíciou. Keďže nemajú meno, na ich poradí záleží

Relačný kalkul (logika 1. rádu)

	Relačný kalkul
Konštanta	$a, b, jozo, \dots, 1, 2, 3, \dots$
Premenná	X, Y, Z, \dots
Zložený term (funkčný symbol)	$f(t_1, t_2, \dots, t_n)$
Atomická formula (predikát)	$p(t_1, t_2, \dots, t_n)$
Konjunkcia	$p \wedge q$
Disjunkcia	$p \vee q$
Implikácia	$p \Rightarrow q$
Negácia	$\neg p$
Existenčný kvantifikátor	$\exists X p(X)$
Všeobecný kvantifikátor	$\forall X p(X)$
Zátvorky	(\dots)

Relačný kalkul (logika 1. rádu)

Pre úpravu formúl budeme používať nasledujúce pravidlá (pozor, **rôzne premenné musia dostat' rôzne mená**):

$$(\forall X p(X)) \wedge q \equiv \forall X (p(X) \wedge q)$$

$$(\forall X p(X)) \vee q \equiv \forall X (p(X) \vee q)$$

$$(\exists X p(X)) \wedge q \equiv \exists X (p(X) \wedge q)$$

$$(\exists X p(X)) \vee q \equiv \exists X (p(X) \vee q)$$

$$\forall X p(X) \equiv \neg (\exists X \neg p(X))$$

Pravidlá pre implikáciu v princípe netreba, lebo $p \Rightarrow q \equiv \neg p \vee q$, ale je dobré získať cit pre implikáciu:

$$(\forall X p(X)) \Rightarrow q \equiv \exists X (p(X) \Rightarrow q)$$

$$(\exists X p(X)) \Rightarrow q \equiv \forall X (p(X) \Rightarrow q)$$

Samozrejme, okrem pravidiel pre kvantifikátory treba poznáť pravidlá výrokovej logiky (napr. De Morganove)

Relačný kalkul: dotazy

Dotazy v relačnom kalkule sú logické formuly. Výsledkom dotazu je usporiadaná množina hodnôt voľných premenných, po ktorých dosadení tá formula platí. Ak dotaz nemá voľné premenné, tak je výsledkom hodnota tej formuly, t.j. TRUE, resp. FALSE

(Formálne je výsledkom formuly vždy TRUE, resp. FALSE. Ale nás **zaujímajú práve tie n-tice** hodnôt voľných premenných, ktoré danú formulu spĺňajú.)

Príklad (malá násobilka): `krat(Cinitel1, Cinitel2, Sucin)`
nad doménou $\{1, 2, \dots, 10\} \times \{1, 2, \dots, 10\} \times \{1, 2, \dots, 100\}$

`krat(1,1,3)` je FALSE

`krat(1,1,1)` je TRUE

Relačný kalkul: dotazy

$\text{krat}(\text{bimbo}, 1, 1)$ je FALSE (bimbo je konštanta, ktorá nepatrí do domény $\{1, 2, \dots, 10\}$; ale aj takýto dotaz má zmysel).

$\{[X, Y] : \text{krat}(X, Y, 4)\}$ je množina usporiadaných dvojíc $[X, Y]$, pre ktoré $\text{krat}(X, Y, 4) = \text{TRUE}$,

t.j. $\{[X, Y] : \text{krat}(X, Y, 4)\} = \{[1, 4], [2, 2], [4, 1]\}$.

$\{X : \text{krat}(X, X, 4)\} = \{[2]\}$, teda $\{2\}$.

$\{X : \text{krat}(X, 3, 4)\} = \emptyset$.

$\{X : \text{krat}(X, 3, \text{bimbo})\} = \emptyset$.

$\{Y : \exists X \text{ krat}(X, Y, 4)\} = \{1, 2, 4\}$.

Relačný kalkul: dotazy

$\{[X, Y] : \text{krat}(X, 3, Y) \wedge \text{krat}(Y, 3, 9)\} = \{[1, 3]\}$.

$\{X : \exists Y (\text{krat}(X, 3, Y) \wedge \text{krat}(Y, 3, 9))\} = \{1\}$.

Pozor, **nie je korektné** písat' napr. $\{X : \text{krat}(X, Y, 4)\}$, lebo vo formuli $\text{krat}(X, Y, 4)$ sú **dve** voľné premenné, X a Y. Ak nás vo výsledku skutočne zaujíma len X (hovoríme, že Y chceme odprojektovať), tak správne je: $\{X : \exists Y \text{krat}(X, Y, 4)\}$.

$\{[X, Y, Z] : \text{krat}(X, Y, Z)\}$ je celá malá násobilka.

Čo je $\{[X, Y, Z] : \neg \text{krat}(X, Y, Z)\}$? Toto nie je prázdna množina, lebo obsahuje napríklad trojicu [7, 7, 7]. Ak ignorujeme domény, tak obsahuje tiež trojicu [bimbo, 1, 1], lebo $\text{krat}(\text{bimbo}, 1, 1) = \text{FALSE}$. Tá množina obsahuje **všetky trojice okrem malej násobilky**. Aj $[\pi, 7.8, \mathfrak{R}]$. To vyzerá nebezpečne. A aj je.

Relačný kalkul: bezpečnosť dotazov (safety)

Obmedzenia, ktoré zjednodušujú/umožňujú výpočet dotazov:

- Každý atomický predikát, ktorý sa vyskytuje v dotaze, je pravdivý len na **konečnej množine konštantných n-tíc**. Táto množina sa volá **množina faktov**, alebo **extenzionálna databáza (EDB)**
- Každá premenná použitá v dotaze musí byť použitá aj v nejakom pozitívnom EDB kontexte (toto upresníme neskôr)

Tieto obmedzenia garantujú „bezpečnosť“ formúl. Za pravdivé považujeme len to, čo sa dá priamo dokázať z faktov (t.j. vypočítat')

Dôvod: čo je výsledkom napr. $\neg \text{krat}(X, Y, 42)$? Do výsledku patrí takmer celý svet. Napr. dvojica [bimbo, jumbo] patrí do výsledku, ale skúste vypočítať všetky dvojice, ktoré patria do výsledku! (To nejde, lebo výsledok obsahuje napríklad všetky dvojice reálnych čísel; a tiež dvojice nám dosiaľ neznámych vecí v dosiaľ neznámych galaxiách atď.).

Relačný kalkul: definície predikátov

V relačnom kalkule sa dá „programovať“. To znamená, že vieme vyrábať z existujúcich predikátov nové (zložitejšie) predikáty.

Príklad: $\text{prvocisla} = \{Z : \forall X \forall Y (\text{krat}(X, Y, Z) \Rightarrow (X=1 \vee Y=1))\}$
je množina prvočísel, ale iba na prvý pohľad. V skutočnosti je to obľudná množina, lebo do nej patrí napríklad jumbo.

Skúsme to „naopak“:

$\text{zlozene_cisla} = \{Z : \exists X \exists Y \text{krat}(X, Y, Z) \wedge X \neq 1 \wedge Y \neq 1\}$
je množina zložených čísel (jumbo v nej nie je).

Teraz je už jednoduché nájsť množinu prvočísel:

$$\begin{aligned}\text{prvocisla} &= \{Z : \exists X \exists Y \text{krat}(X, Y, Z) \wedge Z \notin \text{zlozene_cisla}\} = \\ &= \{Z : \exists X \exists Y \text{krat}(X, Y, Z) \wedge \neg(Z \in \text{zlozene_cisla})\} = \\ &= \{Z : \exists X \exists Y \text{krat}(X, Y, Z) \wedge \neg(\exists X \exists Y \text{krat}(X, Y, Z) \wedge X \neq 1 \wedge Y \neq 1)\} = \\ &= \{Z : (\exists X \exists Y \text{krat}(X, Y, Z)) \wedge (\forall X \forall Y (\text{krat}(X, Y, Z) \Rightarrow (X=1 \vee Y=1)))\}.\end{aligned}$$

Všimnite si účel toho „ $\exists X \exists Y \text{krat}(X, Y, Z) \wedge$ “. To zabezpečuje, aby sa medzi prvočísla nedostal jumbo atď>.

Relačný kalkul: definície predikátov

Použili sme pomenované množiny (definovali sme nové relácie):

$$\text{zlozene_cisla} = \{Z: \exists X \exists Y \text{krat}(X, Y, Z) \wedge X \neq 1 \wedge Y \neq 1\}$$

prvocisla =

$$\{Z: \exists X \exists Y \text{krat}(X, Y, Z) \wedge \neg (\exists X \exists Y \text{krat}(X, Y, Z) \wedge X \neq 1 \wedge Y \neq 1)\}.$$

Ale vieme z predikátu $\text{krat}(X, Y, Z)$ vyrobiť **nové predikáty** $\text{zlozene_cislo}(Z)$ a $\text{prvocislo}(Z)$? Áno, vieme:

$$\forall Z ((\exists X \exists Y \text{krat}(X, Y, Z) \wedge X \neq 1 \wedge Y \neq 1) \Rightarrow \text{zlozene_cislo}(Z))$$

$$\forall Z ((\exists X \exists Y \text{krat}(X, Y, Z)) \wedge \neg \text{zlozene_cislo}(Z)) \Rightarrow \text{prvocislo}(Z))$$

Všimnite si čo tie definície hovoria:

„Pre všetky Z , $\text{zlozene_cislo}(Z)$ platí ked...“.

Dotaz na prvočísla vieme napísat jednoducho: $\{Z: \text{prvocislo}(Z)\}$.

Toto totiž v skutočnosti znamená $\{Z: \text{prvocislo}(Z) \wedge$

$$\forall Z ((\exists X \exists Y \text{krat}(X, Y, Z)) \wedge \neg \text{zlozene_cislo}(Z)) \Rightarrow \text{prvocislo}(Z)) \wedge$$

$$\forall Z ((\exists X \exists Y \text{krat}(X, Y, Z) \wedge X \neq 1 \wedge Y \neq 1) \Rightarrow \text{zlozene_cislo}(Z))\}.$$

Relačný kalkul: definície predikátov

Rekurzia je v relačnom kalkule samozrejmostou. Napríklad, daná je relácia, resp. predikát $r(\text{Rodic}, \text{Dieta})$. Chceme definovať predikát $a(\text{Predok}, \text{Potomok})$, ktorý je pravdivý pre také dvojice, že osoba Predok je predkom osoby Potomok. Sú dva prípady:

1. Osoba Predok je rodičom osoby Potomok (priamy potomok).
2. Osoba Predok je predkom *nejakej* osoby Rodic, pričom Potomok je dieťaťom osoby Rodič.

$$\begin{aligned} & \forall \text{Predok} \ \forall \text{Potomok} \ (\\ & (\\ & r(\text{Predok}, \text{Potomok}) \ /* \text{ prípad 1 } */ \\ & \vee (\exists \text{ Rodic} \ a(\text{Predok}, \text{Rodic}) \wedge r(\text{Rodic}, \text{Potomok})) \ /* \text{ prípad 2 } */ \\ &) \\ & \Rightarrow a(\text{Predok}, \text{Potomok})) \end{aligned}$$

Datalog je programovací jazyk veľmi podobný jazyku Prolog (deklaratívne programovanie, logické programovanie).

Syntax programov:

<atóm>.

<hlava> ← <atóm>.

<hlava> ← <telo>.

<telo> ← <atóm> | not <atóm> | <telo>, <atóm>.

- Riadky začínajúce znakom ‘%’ označujú komentáre
- Všetko, čo je na pravej strane pravidla oddelené čiarkami, sa nazýva podiel’ (podiel’ je niečo, čo nadobúda boolovskú hodnotu). Príklady podiel’ov: „mult(2, 3, X)“, „not plus(4, Y, Y)“, „greater(3, 4)“, „3 > 4“, „not X = 0“
- Žiadne domény (typy)
- Na ľavej strane pravidla sa konštruujú len pozitívne atómy

Datalog: fakty (extenzionálna databáza, EDB)

Príklad (malá násobilka):

krat(1, 1, 1). krat(1, 2, 2). krat(1, 3, 3). krat(1, 4, 4). krat(1, 5, 5).
krat(1, 6, 6). krat(1, 7, 7). krat(1, 8, 8). krat(1, 9, 9). krat(1, 10, 10).
krat(2, 1, 2). krat(2, 2, 4). krat(2, 3, 6). krat(2, 4, 8). krat(2, 5, 10).
krat(2, 6, 12). krat(2, 7, 14). krat(2, 8, 16). krat(2, 9, 18). krat(2, 10, 20).
krat(3, 1, 3). krat(3, 2, 6). krat(3, 3, 9). krat(3, 4, 12). krat(3, 5, 15).
krat(3, 6, 18). krat(3, 7, 21). krat(3, 8, 24). krat(3, 9, 27). krat(3, 10, 30).
krat(4, 1, 4). krat(4, 2, 8). krat(4, 3, 12). krat(4, 4, 16). krat(4, 5, 20).
krat(4, 6, 24). krat(4, 7, 28). krat(4, 8, 32). krat(4, 9, 36). krat(4, 10, 40).
krat(5, 1, 5). krat(5, 2, 10). krat(5, 3, 15). krat(5, 4, 20). krat(5, 5, 25).
krat(5, 6, 30). krat(5, 7, 35). krat(5, 8, 40). krat(5, 9, 45). krat(5, 10, 50).
krat(6, 1, 6). krat(6, 2, 12). krat(6, 3, 18). krat(6, 4, 24). krat(6, 5, 30).
krat(6, 6, 36). krat(6, 7, 42). krat(6, 8, 48). krat(6, 9, 54). krat(6, 10, 60).
krat(7, 1, 7). krat(7, 2, 14). krat(7, 3, 21). krat(7, 4, 28). krat(7, 5, 35).
krat(7, 6, 42). krat(7, 7, 49). krat(7, 8, 56). krat(7, 9, 63). krat(7, 10, 70).
krat(8, 1, 8). krat(8, 2, 16). krat(8, 3, 24). krat(8, 4, 32). krat(8, 5, 40).
krat(8, 6, 48). krat(8, 7, 56). krat(8, 8, 64). krat(8, 9, 72). krat(8, 10, 80).
krat(9, 1, 9). krat(9, 2, 18). krat(9, 3, 27). krat(9, 4, 36). krat(9, 5, 45).
krat(9, 6, 54). krat(9, 7, 63). krat(9, 8, 72). krat(9, 9, 81). krat(9, 10, 90).
krat(10, 1, 10). krat(10, 2, 20). krat(10, 3, 30). krat(10, 4, 40). krat(10, 5, 50).
krat(10, 6, 60). krat(10, 7, 70). krat(10, 8, 80). krat(10, 9, 90). krat(10, 10, 100).

Datalog: pravidlá (intenzionálna databáza)

Príklad (malá násobilka):

```
zlozene_cislo(Z) ←
    krat(X, Y, Z),
    not X = 1,
    not Y = 1.
```

```
prvocislo(Z) ←
    krat(X, Y, Z),
    not zlozene_cislo(Z).
```

Prepis do ASCII (‘←’ je ‘:-’, ‘not’ je ‘\+’, singleton premenné sú ‘_’):

```
zlozene_cislo(Z) :-
    krat(X, Y, Z),
    \+ X = 1,
    \+ Y = 1.
```

```
prvocislo(Z) :-
    krat(_, _, Z),
    \+ zlozene_cislo(Z).
```

Datalog: pravidlá (intenzionálna databáza)

Program v Datalogu sa skladá z **pozitívnych** faktov a z pravidiel s **pozitívnou hlavou** (negatívne fakty sa nedajú definovať).

Program sa „spúšťa“ tým, že mu kladieme dotazy.

?- zlozene_cislo(Z).

[4], [6], [8], [9], [10], [12], [14], [15], [16], [18], [20], [21], [24], [25], [27], [28], [30],
[32], [35], [36], [40], [42], [45], [48], [49], [50], [54], [56], [60], [63], [64], [70], [72],
[80], [81], [90], [100]

37 tuples.

?- prvocislo(C).

[1], [2], [3], [5], [7]

5 tuples.

?- zlozene_cislo(13).

No

?- prvocislo(13).

No

```
zlozene_cislo(Z) ←
    krat(X, Y, Z),
    not X = 1,
    not Y = 1.
```

```
prvocislo(Z) ←
    krat(X, Y, Z),
    not zlozene_cislo(Z).
```

Zvláštne... Prečo 13 nie je ani zložené číslo, ani prvočíslo?

Datalog: sémantika (o čom to je)

Datalogový program je konjunkciou implikácií (pravidlo v Datalogu je implikácia), pričom všetky premenné sú kvantifikované všeobecným kvantifikátorom

Nasledujúce dva riadky (extenzionálna databáza)

$\text{krat}(1, 1, 1).$

$\text{krat}(1, 2, 2).$

sú tiež len skratkami za implikácie:

$\text{krat}(1, 1, 1) \leftarrow \text{TRUE}.$

$\text{krat}(1, 2, 2) \leftarrow \text{TRUE}.$

Prečítať sa dajú napríklad takto:

1. Platí $\text{krat}(1, 1, 1)$ a **zároveň** platí $\text{krat}(1, 2, 2)$.

2. **Z TRUE vyplýva**, že predikát krat platí pre trojicu $[1, 1, 1]$ **aj** pre trojicu $[1, 2, 2]$.

3. Predikát $\text{krat}(X, Y, Z)$ platí, ak bud' $[X, Y, Z] = [1, 1, 1]$ **alebo** $[X, Y, Z] = [1, 2, 2]$.

Všetky tieto spôsoby sú správne, lebo znamenajú to isté (**domáca úloha**)

Datalog: sémantika (o čom to je)

Datalogový program je konjunkciou implikácií (pravidlo v Datalogu je implikácia), pričom všetky premenné sú kvantifikované všeobecným kvantifikátorom

Uvažujme pre jednoduchosť program s jediným pravidlom:

$\text{krat}(1, 1, 1) \leftarrow \text{TRUE}.$

To zodpovedá formuli (ktorá definuje predikát krat):

$\text{TRUE} \Rightarrow \text{krat}(1, 1, 1).$

Ekvivalentne, „ $\text{krat}(X, Y, Z)$ platí, ak $[X, Y, Z] = [1, 1, 1]$ “:

$\text{krat}(X, Y, Z) \leftarrow X = 1, Y = 1, Z = 1.$

$\forall X \forall Y \forall Z ((X = 1 \wedge Y = 1 \wedge Z = 1) \Rightarrow \text{krat}(X, Y, Z)).$

Na niektorých prednáškach sa definície píšu s **obojsmernou** implikáciou, a zrazu je **jednosmerná**!

Formula $\text{TRUE} \Rightarrow \text{krat}(1, 1, 1)$ nevylučuje, že platí $\text{krat}(\text{jumbo}, \text{bimbo}, \pi)$. Na druhej strane, tá definícia nevylučuje ani to, že $\text{krat}(\text{jumbo}, \text{bimbo}, \pi)$ neplatí. Čo vlastne platí? Čo vypočíta dotaz „?- $\text{krat}(X, Y, Z)$ “ pre tento program?

Datalog: sémantika (o čom to je)

Našťastie, pre program $\text{krat}(1, 1, 1)$.
dotaz $?- \text{krat}(X, Y, Z)$ vypočíta $[1, 1, 1]$. Nič iné.

Datalog počíta **minimálny model programu** (t.j. minimálne relácie, pre ktoré je daný program splnený)

Horeuvedený program zodpovedá formuli

TRUE \Rightarrow $\text{krat}(1, 1, 1)$, ekvivalentne $\text{krat}(1, 1, 1)$.

Teda $\text{krat}(1, 1, 1)$ určite platí (v každom modeli, teda aj v minimálnom). V niektorom modeli môže platiť napríklad $\text{krat}(\text{jumbo}, \text{bimbo}, \pi)$, ale neplatí to v každom modeli. „Nezmysly“ sa do výsledku nedostanú, lebo **minimálna** relácia krat , ktorá spĺňa tú formulu, je $\{[1, 1, 1]\}$.

$$\{[X, Y, Z] : \text{min}_{\text{krat}} (\text{krat}(1, 1, 1) \wedge \text{krat}(X, Y, Z))\} = \{[1, 1, 1]\}$$

V skutočnosti je táto sémantika veľmi intuitívna.

Väčšinou sa definície $s \Rightarrow$ správajú podobne ako definície $s \Leftrightarrow$.

Ešte k sémantike Datalogu

Uvažujme program (ktorý pozostáva len z EDB faktov):

colour(black).

colour(white).

Čo vypočíta dotaz ?- colour(C).

Samotný program zodpovedá formuli colour(black) \wedge colour(white)

Existencia iných farieb okrem black a white programu neodporuje.

Napriek tomu intuitívne očakáme, že vo výsledku sa objavia **len** tie 2 farby

Tento program spolu s dotazom zodpovedá formuli
colour(black) \wedge colour(white) \wedge colour(C)

Sémantika dotazov v Datalogu je vskutku intuitívna: dotaz skutočne vráti **len** {black, white}. Dôvod je ten, že colour = {black, white} je **minimálna** relácia colour, pre ktorú je splnená formula zodpovedajúca tomu programu

Bezpečnosť pravidiel a programov v Datalogu (safety)

Obmedzenia pre pravidlá v Datalogu (J. Šturm vo svojej prednáške používa trošku slabšie, ale komplikovanejšie pravidlá):

Definícia: **Pravidlo je bezpečné**, ak každá premenná použitá kdekoľvek tom pravidle je použitá aj v nejakom **nie negovanom EDB** podcieli

Definícia: **Program je bezpečný**, ak každé jeho pravidlo je bezpečné

Bezpečnosť Datalogového pravidla, resp. programu sa dá ľahko strojovo overiť.

Príklad (Ulmann/Widom):

p(X, Y) ← q(X, Z), $\neg r(W, X, Z)$, $W < Y$.

Toto pravidlo nie je bezpečné (safe) z niekoľkých dôvodov:

- premenná Y je použitá v hlave pravidla, ale v žiadnom EDB podcieli. Toto vo všeobecnosti znemožňuje vymenovať konečnú množinu hodnôt pre Y, ktoré sú „kandidátmi“ pre odvodenie p(X, Y)
- premenná Y je použitá v aritmetickom podcieli (<), ale v žiadnom EDB podcieli
- premenná W je síce použitá len v negovanom v EDB podcieli a v aritmetickom podcieli

Datalog: predsa len isté trampoty s negáciou

“Holič holí všetkých mužov v meste, ktorí neholia sami seba“
(Russel's paradox)

man(barber).

man(mayor).

shaves(barber, X) \leftarrow man(X), \neg shaves(X, X).

Tento program je bezpečný, lebo X sa vyskytuje v relačnom podcieli man(X). (OK, je rekurzívny, ale rekurziu chceme.)

Holí holič starostu?

? shaves(barber, mayor).

Po dosadení do pravidla:

shaves(barber, mayor) \leftarrow man(mayor), \neg shaves(mayor, mayor).

Výsledkom tohto dotazu je TRUE, lebo shaves(mayor, mayor) nie je v databáze pravdivých faktov a očividne sa nedá odvodiť

Kto holí starostu?

? shaves(X, mayor). % {X: shaves(X, mayor)}

Výsledkom je {barber}

Datalog: predsa len isté trampoty s negáciou

“Holič holí všetkých mužov v meste, ktorí neholia sami seba“
man(barber).

man(mayor).

shaves(barber, X) \leftarrow man(X), \neg shaves(X, X).

Holič holič seba samého?

? **shaves(barber, barber)**.

Po dosadení do pravidla:

shaves(barber, barber) \leftarrow man(barber), \neg **shaves(barber, barber)**.

Ak by sme povedali, že shaves(barber, barber) = FALSE, tak potom aj
 \neg shaves(barber, barber) by muselo byť FALSE, ak má platiť tá implikácia.

Lenže to je protirečenie. Takže v dvojhodnotovej logike je správne

shaves(barber, barber) = TRUE.

Všimnite si, že to pravidlo pre shaves(barber, X) je implikácia, nie ekvivalencia.
Vtedy Russelov paradox v skutočnosti nie je paradoxom. (Ani žiaden iný paradox sa v Datalogu nepodarí vyrobiť.)

Dá sa však naozaj **dokázať**, že holič holí sám seba?

Čo znamená **dokázať**? Čo je **dôkaz**?

Datalog versus relačný kalkul

Programy (dotazy) v Datalogu zodpovedajú logickým formulám

	Relačný kalkul	Datalog
Konštanta	a, b, jozo, ..., 1, 2, 3, ...	a, b, jozo, ..., 1, 2, 3, ...
Premenná	X, Y, Z, ...	X, Y, Z, ..., _, ...
Zložený term	$f(t_1, t_2, \dots, t_n)$	$f(t_1, t_2, \dots, t_n)$
Atomická formula	$p(t_1, t_2, \dots, t_n)$	$p(t_1, t_2, \dots, t_n)$
Konjunkcia	$p \wedge q$	p, q
Disjunkcia	$p \vee q$	$p; q$ (resp. 2 pravidlá)
Implikácia	$p \Rightarrow q$	$q \leftarrow p$ (resp. $q :- p$)
Negácia	$\neg p$	$\text{not } p$ (resp. $\text{\textbackslash+ } p$)
Existenčný kvantifikátor	$\exists X p(X)$	
Všeobecný kvantifikátor	$\forall X p(X)$	
Zátvorky	(...)	

Zdá sa, že v Datalogu niečo chýba. V skutočnosti **nič nechýba!**

Datalog: príklad inej databázy

EDB:

%dodavatel(Firma, Mesto)

dodavatel(hp, bratislava).

dodavatel(hp, kosice).

dodavatel(ibm, bratislava).

%dodava(Firma, Výrobok, Cena, Lehota)

dodava(hp, laserjet4, 200, 2).

dodava(hp, pc1, 400, 3).

dodava(ibm, pc1, 300, 7).

%objednavky(Klient, Výrobok)

objednavky(jozo, pc1).

objednavky(anicka, laserjet4).

Datalog a kalkul: konjunkcia, disjunkcia, \exists

%bratislavski dodavatelia

dodavatel_ba(F) \leftarrow dodavatel(F, bratislava).

$\forall F \text{ (dodavatel}(F, \text{bratislava}) \Rightarrow \text{dodavatel_ba}(F))$

%kto dodava **nieco** z toho, co objednal jozo

dodava_nieco_jozovi(F) \leftarrow objednavky(jozo, V),

dodava(F, V, _, _).

$\forall F ((\exists V \text{ (objednavky}(jゾzo, V) \wedge \exists C \exists L \text{ dodava}(F, V, C, L))) \Rightarrow \text{dodava_nieco_jozovi}(F))$

%kto dodava pc1 bud do 4 dni **alebo** lacnejsie ako za 15000

dodava_pc1_do4_alebo_do15000(F) \leftarrow dodava(F, pc1, C, _),
 $C \leq 15000.$

dodava_pc1_do4_alebo_do15000(F) \leftarrow dodava(F, pc1, _, L),
 $L \leq 4.$

$\forall F (\exists C \exists L_1 (\text{dodava}(F, pc1, C, L_1) \wedge C \leq 15000) \vee (\exists L \exists C_2 (\text{dodava}(F, pc1, C_2, L) \wedge L \leq 4)))$

$\Rightarrow \text{dodava_pc1_do4_alebo_do15000}(F))$

Datalog a kalkul: negácia

%kto **nedodava nieco** z toho, co objednal jozo
nedodava_nieco_jozovi(F) ← objednavky(jozo, V),
¬ dodava(F, V, _, _).
 $\forall F (\exists V (\text{objednavky}(joso, V) \wedge$
 $(\neg \exists C1 \exists L1 \text{ dodava}(F, V, C1, L1)))$
 $\Rightarrow \text{nedodava_nieco_jozovi}(F))$

Pozor, toto nie je bezpečný program. Treba povedať, **odkiaľ sa vyberá firma F**, ktorá je kandidátom na výsledok:

d(F, V) ← dodava(F, V, _, _).
nedodava_nieco_jozovi(F) ← dodavatel(F, _),
objednavky(joso, V), ¬ d(F, V).
 $\forall F (\exists V (\text{objednavky}(joso, V) \wedge$
 $\exists M \text{ dodavatel}(F, M) \wedge \neg (\exists C1 \exists L1 \text{ dodava}(F, V, C1, L1)))$
 $\Rightarrow \text{nedodava_nieco_jozovi}(F))$

Datalog a kalkul: všeobecný kvantifikátor

%který dodavatel dodava **VSETKO**, co objednal jozo
 $\forall F ((\exists M \text{ dodavatel}(F, M) \wedge \forall V (\text{objednavky}(jizo, V) \Rightarrow \exists C \exists L \text{ dodava}(F, V, C, L))) \Rightarrow \text{dodava_vsetko_jozovi}(F))$

Toto sa v Datalogu nedá vyjadriť priamo. Datalog „pozná“ len existenčný kvantifikátor. Lenže vieme, že

$$\forall X p(X) \equiv \neg (\exists X \neg p(X)).$$

Ten dotaz treba preformulovať:

Pre ktorého dodávateľa **neplatí**, že **nedodáva niečo** Jožovi

Teraz už vieme zapísat' predikát `dodava_vsetko_jozovi`
`dodava_vsetko_jozovi(F) ← dodavatel(F, _),`
`¬ nedodava_nieco_jozovi(F).`

Ako sme videli, `nedodava_nieco_jozovi` je už jednoduché napísat'

Preklad relačnej formuly do Datalogu

Program v Datalogu je konjunkcia implikácií (pravidlo v Datalogu je implikácia), pričom všetky premenné sú kvantifikované všeobecným kvantifikátorom.

Každý program v Datalogu sa dá podľa tohto automaticky preložiť do formuly relačného kalkulu

Definícia. **Formula relačného kalkulu je bezpečná**, ak je dokázateľne ekvivalentná formule, ktorá vzniká automatickým kánonickým prekladom nejakého bezpečného Datalogového programu (konjunkcia implikácií, všetky premenné viazané \forall)

Bezpečnosť ľubovoľného Datalogového programu sa dá rozhodnúť strojovo, stačí syntakticky overiť bezpečnosť každého pravidla. Bezpečnosť formuly relačného kalkulu sa nemusí vždy podať rozhodnúť (ekvivalencia formúl je vo všeobecnosti nerozhodnuteľný problém) — a kým sa to nepodarí, formulu nepovažujeme za bezpečnú

Metóda prekladu formúl do Datalogu:

- 1. Premenovanie premenných, lokalizácia kvantifikátorov**
- 2. Eliminácia všeobecných kvantifikátorov a implikácií**
- 3. Definícia pomocných predikátov**
- 4. Bezpečnosť pomocných predikátov**
- 5. Prepis do pravidiel Datalogu**

Preklad relačnej formuly do Datalogu

Príklad: „every man who loves all animals is loved by someone“.

$\forall X (\text{man}(X) \Rightarrow ((\forall Y (\text{animal}(Y) \Rightarrow \text{loves}(X, Y))) \Rightarrow \exists Y \text{ loves}(Y, X)))$

1. Premenovanie premenných, lokalizácia kvantifikátorov

$\forall X (\text{man}(X) \Rightarrow ((\forall A (\text{animal}(A) \Rightarrow \text{loves}(X, A))) \Rightarrow \exists Y \text{ loves}(Y, X)))$

2. Eliminácia všeobecných kvantifikátorov a implikácií

$\neg \exists X (\text{man}(X) \wedge \neg(\neg \exists A (\text{animal}(A) \wedge \neg \text{loves}(X, A))) \vee \exists Y \text{ loves}(Y, X)))$

$\neg \exists X (\text{man}(X) \wedge \neg(\exists A (\text{animal}(A) \wedge \neg \text{loves}(X, A)) \vee \exists Y \text{ loves}(Y, X)))$

$\neg \exists X (\text{man}(X) \wedge \neg \exists A (\text{animal}(A) \wedge \neg \text{loves}(X, A)) \wedge \neg \exists Y \text{ loves}(Y, X))$

3. Definícia pomocných predikátov (pre sekvencie $\neg \exists \dots$)

$\neg f$

$f \leftarrow \exists X (\text{man}(X) \wedge \neg g(X) \wedge \neg h(X))$

$g(X) \leftarrow \exists A (\text{animal}(A) \wedge \neg \text{loves}(X, A))$

$h(X) \leftarrow \exists Y \text{ loves}(Y, X)$

Preklad relačnej formuly do Datalogu

Príklad: „every man who loves all animals is loved by someone“.

$\neg f$

$f \leftarrow \exists X (\text{man}(X) \wedge \neg g(X) \wedge \neg h(X))$

$g(X) \leftarrow \exists A (\text{animal}(A) \wedge \neg \text{loves}(X, A))$

$h(X) \leftarrow \exists Y \text{loves}(Y, X)$

4. Bezpečnosť pomocných predikátov

$g(X) \leftarrow \text{man}(X) \wedge \exists A (\text{animal}(A) \wedge \neg \text{loves}(X, A))$

$h(X) \leftarrow \text{man}(X) \wedge \exists Y \text{loves}(Y, X)$

5. Prepis do Datalogu

answer \leftarrow not **f**.

f \leftarrow **man(X)**, not **g(X)**, not **h(X)**.

g(X) \leftarrow **man(X)**, **animal(A)**, not **loves(X, A)**.

h(X) \leftarrow **man(X)**, **loves(**_, **X****)**.

Preklad relačnej formuly do Datalogu

Ešte jeden príklad: „kto dodava vsetko, co objednal jozo”

$\exists M \text{ dodavatel}(F, M) \wedge \forall V ((\text{objednavky}(joso, V) \Rightarrow \exists C \exists L \text{ dodava}(F, V, C, L)))$

1. Premenovanie premenných, lokalizácia kvantifikátorov

2. Eliminácia všeobecných kvantifikátorov a implikácií

$\exists M \text{ dodavatel}(F, M) \wedge$

$\neg (\exists V (\text{objednavky}(joso, V) \wedge \neg (\exists C \exists L \text{ dodava}(F, V, C, L))))$

3. Definícia pomocných predikátov

$\text{dodava_vsetko_jozovi}(F) \leftarrow \exists M \text{ dodavatel}(F, M) \wedge \neg \text{nedodava_nieco_jo}(F)$

$\text{nedodava_nieco_jo}(F) \leftarrow \exists V (\text{objednavky}(joso, V) \wedge \neg (\exists C \exists L \text{ dodava}(F, V, C, L)))$

4. Bezpečnosť pomocných predikátov

$\text{nedodava_nieco_jo}(F) \leftarrow \exists M \text{ dodavatel}(F, M) \wedge$

$\exists V (\text{objednavky}(joso, V) \wedge \neg (\exists C \exists L \text{ dodava}(F, V, C, L)))$

5. Prepis do Datalogu

$d(F, V) \leftarrow \text{dodava}(F, V, _, _).$

$\text{nedodava_nieco_jo}(F) \leftarrow \text{dodavatel}(F, _), \text{objednavky}(joso, V),$
 $\neg d(F, V).$

$\text{dodava_vsetko_jozovi}(F) \leftarrow \text{dodavatel}(F, _), \neg \text{nedodava_nieco_jo}(F).$

- Ak sa obmedzíme na bezpečné programy, tak **relačný kalkul a Datalog sú jazyky s rovnakou vyjadrovacou silou**, t.j. ľubovoľný bezpečný program v Datalogu sa dá ekvivalentne vyjadriť ako bezpečná formula relačného kalkulu a naopak
- V súčasnosti v najrozšírenejším dotazovacím jazykom v IT priemysle je **SQL** (Structured Query Language)

Základná syntax príkazu **SELECT**:

SELECT <zoznam atribútov>

FROM <zoznam relácií>

WHERE <podmienka>

Preklad Datalog→SQL je jednoduchý a nevyžaduje **žiadnu** predošlú znalosť SQL

Datalog a SQL: bratislavski dodavatelia

%dodavatel(Firma, Mesto)

%dodava(Firma, Výrobok, Cena, Lehota)

%objednavky(Klient, Výrobok)

%bratislavski dodavatelia

dodavatel_ba(FIRMA) ← dodavatel(FIRMA, bratislava).
? dodavatel_ba(FIRMA).

SELECT d.Firma

FROM dodavatel d

WHERE d.Mesto = 'bratislava'

Algoritmus prekladu jednoduchých pravidiel:

- 1.Za FROM vymenovať všetky nenegované relácie (pomenované predikáty) na pravej strane, pričom každá relácia dostane jednoznačné meno (alias)
- 2.Vo WHERE vymenovať v konjunkcii všetky vzťahy medzi atribútmi
- 3.Do SELECT vymenovať všetky atribúty, ktoré idú do výsledku (t.j. tie, ktoré sú v hlove pravidla)

Datalog a SQL: kto dodava nieco jozovi

%dodavatel(Firma, Mesto)

%dodava(Firma, Vyroboek, Cena, Lehota)

%objednavky(Klient, Vyroboek)

%kto dodava nieco z toho, co objednal jozo

dodava_nieco_jozovi(F) ← objednavky(jozo, V),
dodava(F, V, _, _).

SELECT d.Firma

FROM objednavky o, dodava d

WHERE o.Klient = 'joso' AND o.Vyroboek = d.Vyroboek

Datalog a SQL: výrobky dodávane aspon 2 dodavatelia

%dodavatel(Firma, Mesto)

%dodava(Firma, Výrobok, Cena, Lehota)

%objednavky(Klient, Výrobok)

% výrobky dodávane aspon 2 dodavatelia

výrobky_aspon2d(V) \leftarrow dodava(F1, V, _, _), dodava(F2, V, _, _),
F1 \neq F2.

SELECT d1.Výrobok

FROM dodava d1, dodava d2

WHERE d1.Výrobok = d2.Výrobok and d1.Firma <>> d2.Firma

Datalog a SQL: kto dodava pc1 bud rychlo alebo lacno

%dodavatel(Firma, Mesto)

%dodava(Firma, Vyrolok, Cena, Lehota)

%objednavky(Klient, Vyrolok)

%kto dodava pc1 bud do 4 dni alebo lacnejsie ako za 15000

dodava_pc1_do4_alebo_do15000(F) ←
 dodava(F, pc1, C, _), C <= 15000.

dodava_pc1_do4_alebo_do15000(F) ←
 dodava(F, pc1, _, L), L <= 4.

SELECT d.Firma

FROM dodava d

WHERE d.Vyrolok = 'pc1' AND d.Cena <= 15000

UNION

SELECT d.Firma

FROM dodava d

WHERE d.Vyrolok = 'pc1' AND d.Lehota <= 4

Datalog a SQL: kto nedodava nieco jozovi

%dodavatel(Firma, Mesto)

%dodava(Firma, Vyrobo, Cena, Lehota)

%objednavky(Klient, Vyrobo)

%kto nedodava nieco z toho, co objednal jozo

dd(F, V) \leftarrow dodava(F, V, _, _).

nedodava_nieco_jozovi(F) \leftarrow objednavky(jozo, V),
dodavatel(F, _), \neg dd(F, V).

SELECT d.Firma

FROM objednavky o, dodavatel d

WHERE o.Klient=’joso’ AND NOT EXISTS (

SELECT *

FROM dodava dd

WHERE dd.Firma = d.Firma AND dd.Vyrobo = o.Vyrobo)

Datalog a SQL: kto dodava jozovi vsetko (1)

```
%dodavatel(Firma, Mesto)
%dodava(Firma, Vyrolok, Cena, Lehota)
%objednavky(Klient, Vyrolok)
```

%kto dodava VSETKO, co objednal jozo

dd(F, V) ← dodava(F, V, _, _).

**nedodava_nieco_jozovi(F) ← dodavatel(F, _),
objednavky(jozo, V), ¬ dd(F, V).**

dodava_vsetko_jozovi(F) ← dodavatel(F, _),
¬ nedodava_nieco_jozovi(F).

```
CREATE TEMPORARY TABLE nedodava_nieco_jozovi AS
SELECT d.Firma
FROM dodavatel d, objednavky o
WHERE o.Klient = 'joso' AND NOT EXISTS (
    SELECT * FROM dodava dd
    WHERE dd.Firma = d.Firma AND dd.Vyrolok = o.Vyrolok)
```

Datalog a SQL: kto dodava jozovi vsetko (2)

%dodavatel(Firma, Mesto)

%dodava(Firma, Vyroboek, Cena, Lehota)

%objednavky(Klient, Vyroboek)

%kto dodava VSETKO, co objednal jozo

dd(F, V) \leftarrow dodava(F, V, _, _).

nedodava_nieco_jozovi(F) \leftarrow dodavatel(F, _),
 objednavky(jozo, V), \neg dd(F, V).

dodava_vsetko_jozovi(F) \leftarrow dodavatel(F, _),
 \neg nedodava_nieco_jozovi(F).

SELECT d.Firma

FROM dodavatel d

WHERE NOT EXISTS (

SELECT * FROM nedodava_nieco_jozovi nnj

WHERE nnj.Firma=d.Firma)

Datalog a SQL: prvocisla

%krat(X,Y,Z)

prvocislo(Z) \leftarrow krat(_, _, Z), \neg neprvocislo(Z).

neprvocislo(Z) \leftarrow krat(X, Y, Z), \neg (X=1), \neg (Y=1).

WITH neprvocislo AS /* WITH sa viaže sa na nasledujúci SELECT */

SELECT k.Z /* a dá sa použiť miesto CREATE TEMP TABLE */

FROM krat k

WHERE NOT(k.X=1) AND NOT(k.Y=1)

SELECT k.Z

FROM krat k

WHERE NOT EXISTS (

SELECT *

FROM neprvocislo np

WHERE k.Z=np.Z)

Datalog a SQL: šťastní pijani

capuje(Krcma, Alkohol)

lubi(Pijan, Alkohol)

navstivil(Idn, Pijan, Krcma)

vypil(Idn, Alkohol, Mnozstvo)

Najdite píjanov, ktorí v každej krčme, ktorú navštívili, našli nejaký alkohol, ktorý ľúbia (nemuseli ho vypiť, stačí keď ho tam čapovali).

pijani(P) \leftarrow lubi(P, _). % Píjan je ten, kto lubi alkohol...

pijani(P) \leftarrow navstivil(_, P, _). % alebo navstevuje krcmy.

stastni_pijani(P) \leftarrow pijani(P), not niekde_necapoвали_nic(P).

%Stastnemu píjanovi sa nikdy nestalo, že v niektornej krčme, ktoru navstivil, necapovali nic, co nas píjan lubi

Datalog a SQL: šťastní pijani

capuje(Krcma, Alkohol)

lubi(Pijan, Alkohol)

navstivil(Idn, Pijan, Krcma)

vypil(Idn, Alkohol, Mnozstvo)

niekde_necapovali_nic(P) \leftarrow navstivil(_, P, K),
not lubi_nieco(P, K).

%nas pijan nelubi nic z krcmy K, t.j. neplati ze lubi nieco z K
lubi_nieco(P, K) \leftarrow lubi(P, A), capuje(K, A).

Finálny dotaz:

? stastni_pijani(P).

Datalog a SQL: šťastní pijani

capuje(Krcma, Alkohol),
lubi(Pijan, Alkohol),
navstivil(Idn, Pijan, Krcma),
vypil(Idn, Alkohol, Mnozstvo)

pijani(P) \leftarrow lubi(P, _).

pijani(P) \leftarrow navstivil(_, P, _, _, _).

CREATE TEMPORARY TABLE **pijani** AS

SELECT I.Pijan

FROM lubi I

UNION

SELECT N.pijan

FROM navstivil n

lubi_nieco(P, K) \leftarrow lubi(P, A), capuje(K, A).

CREATE TEMPORARY TABLE **lubi_nieco** AS

SELECT I.Pijan, c.Krcma

FROM lubi I, capuje c

WHERE I.Alkohol = c.Alkohol

Datalog a SQL: šťastní pijani

capuje(Krcma, Alkohol),
lubi(Pijan, Alkohol),
navstivil(Idn, Pijan, Krcma), vypil(Idn, Alkohol, Mnozstvo)

niekde_necapovali_nic(P) \leftarrow navstivil(_, P, K),
not lubi_nieco(P, K).

```
CREATE TEMPORARY TABLE niekde_necapovali_nic AS
SELECT n.Pijan
FROM navstivil n
WHERE NOT EXISTS (
    SELECT *
    FROM lubi_nieco ln
    WHERE n.Pijan = ln.Pijan AND n.Krcma = ln.Krcma)
```

Datalog a SQL: šťastní pijani

capuje(Krcma, Alkohol),
lubi(Pijan, Alkohol),
navstivil(Idn, Pijan, Krcma),
vypil(Idn, Alkohol, Mnozstvo)

stastni_pijani(P) \leftarrow pijani(P), not niekde_necapovali_nic(P).

```
CREATE TEMPORARY TABLE stastni_pijani AS
SELECT p.Pijan
FROM pijani p
WHERE NOT EXISTS (
    SELECT *
    FROM niekde_necapovali_nic nnn
    WHERE nnn.Pijan = p.Pijan)
```

Datalog a SQL: šťastní pijani

capuje(Krcma, Alkohol),
lubi(Pijan, Alkohol),
navstivil(Idn, Pijan, Krcma),
vypil(Idn, Alkohol, Mnozstvo)

Finálny dotaz:

? stastni_pijani(P).

```
SELECT sp.Pijan  
FROM stastni_pijani sp
```

Alebo (bez použitia TEMPORARY TABLEs):

```
WITH pijani AS (SELECT ...),  
WITH lubi_nieco AS (SELECT ...),  
WITH niekde_necapovali_nic AS (SELECT ...),  
WITH stastni_pijani AS (SELECT ...),  
SELECT sp.Pijan  
FROM stastni_pijani sp
```

Algoritmus prekladu Datalog → SQL

Univerzálny algoritmus prekladu Datalogových programov do SQL:

Pre každý predikát p (ktorý nie je v EDB) generuj

`create temporary table p as ...`

Pre každé pravidlo definujúce p generuj `select ...`

Ak je pravidiel definujúcich p viacej, spoj selecty generované v predošлом kroku pomocou `union`

Pre všetky nenegované podciele pravidla použi algoritmus prekladu jednoduchých pravidiel (vo where klauze príkazu `select` vzniká vždy konjunkcia)

Negované podciele pravidla prekladaj nakoniec. Pre každý negovaný podiel pridaj do where klauzy
... `and not exists (select * ...)`

SQL: WITH, TEMPORARY TABLEs, VIEWs

WITH, CREATE TEMPORARY TABLE a CREATE VIEW sú ekvivalentné z hľadiska výsledku hlavného SELECTu. Líšia sa v „side-effects“

WITH r AS (SELECT ...),
SELECT ...

Žiadne side-effects (WITH je len súčasťou nasledujúceho SELECT). Bohužiaľ mnohé systémy WITH neimplementujú. Napríklad PostgreSQL dlho nepoznal WITH (*toto sa zmenilo v 2011, vo verzii 8.4*)

CREATE TEMPORARY TABLE r AS (SELECT ...);
SELECT ...

Vytvorí reláciu r (možno nie materializovanú, ale určite aspoň jej meno) **len počas trvania transakcie**. Keď transakcia skončí, r automaticky zanikne. V systémoch, ktoré nepoznajú WITH, je toto vhodná alternatíva

CREATE VIEW r AS (SELECT ...);
SELECT ...

Vytvorí permanentnú reláciu r (možno nie materializovanú, ale určite aspoň jej meno). Relácia r sa stáne trvalou súčasťou databázy, t.j. nezanikne pokial' sa tak explicitne nepovie príkazom DROP VIEW

Úvod do databázových systémov

<http://www.dcs.fmph.uniba.sk/~plachetk>
/TEACHING/DB2013

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2013–2014

Základné časti SQL

SQL, Structured Query Language má zhruba 2 časti: **1.DDL, Data Definition Language** (vytváranie používateľov, vytváranie databázových schém atď); **2.DML, Data Manipulation Language** (práca so schémou)

Základné príkazy DML:

INSERT	vkladanie záznamov do tabuľky
DELETE	rušenie záznamov v tabuľke
UPDATE	modifikácia záznamov v tabuľke
SELECT	výber (vyhľadávanie) záznamov v tabuľke

Jazyk SQL je bohatý. Spätnú kompatibilitu zabezpečuje štandardizácia: SQL-87 (ANSI), SQL-92 (SQL2), SQL-99 (SQL3), SQL-2003, SQL-2006 (ISO/IEC 9075-14:2006). Oficiálny štandard nie je voľne k dispozícii (je dostupný za peniaze od ISO, resp. ANSI), ale 99% je k dispozícii na
<http://www.wiscorp.com/SQLStandards.html>

Príkaz SELECT

Všeobecný tvar príkazu SELECT (detaily zanedbáme):

SELECT	zoznam atribútov
FROM	zoznam relácií (kartézsky súčin, resp. join)
WHERE	selekčná podmienka (obsahuje spájacie podmienky)
GROUP BY	zoznam grupovacích atribútov
HAVING	selekčná podmienka v agregovanej relácii
ORDER BY	usporiadanie výslednej relácie
...	napr. LIMIT, EXPORT atď

Základný tvar príkazu SELECT (detaily zanedbáme) :

SELECT	zoznam atribútov
FROM	zoznam relácií
WHERE	selekčná podmienka (spájacie podmienky)

Sémantika SELECT... FROM... WHERE... je takáto:

1. Urobí sa kartézsky súčin relácií za FROM
2. Na ten kartézsky súčin sa aplikuje selekčná podmienka za WHERE (ktorá odfiltruje riadky, ktoré tú podmienku nespĺňajú)
3. Nakoniec sa urobí projekcia na atribúty, ktoré sú za SELECT (čo odfiltruje stĺpce, ktoré nie sú v projekcii)

Príkaz SELECT: selekcia

```
SELECT *
FROM produkt
WHERE Meno = 'žiarovka' AND Cena > 100;
```

V podmienke WHERE sa dá použiť:

mená atribútov

porovnávacie operátory: =, <>, <, >, <=, >=

aritmetické operátory: +, -, *, /

operácie s reťazcami, napr. zretazenie: ||, &

logické spojky: NOT, AND, OR

porovnanie regulárnych výrazov: s LIKE p

špeciálne funkcie pre dátum a čas a rôzne built-in funkcie

produkt (Meno, Cena, Kategória, Výrobca)

kontrakt (Predávajúci, Kupujúci, Sklad, Výrobok)

podnik (Meno, Adresa, Okres)

osoba (Meno, Rodné_číslo, Telefón, Mesto)

Príkaz SELECT: projekcia a premenovanie atribútov

Projekcia (výber podmnožiny atribútov):

```
SELECT Meno, Cena  
FROM produkt  
WHERE Meno = 'žiarovka' AND Cena > 100;
```

Premenovanie atribútov vo výslednej tabuľke:

```
SELECT Meno AS Názov, Cena AS Ponuka  
FROM produkt  
WHERE Meno = 'žiarovka' AND Cena > 100;
```

produkt (Meno, Cena, Kategória, Výrobca)
kontrakt (Predávajúci, Kupujúci, Sklad, Výrobok)
podnik (Meno, Adresa, Okres)
osoba (Meno, Rodné_číslo, Telefón, Mesto)

Príkaz SELECT: usporiadanie výsledkov

```
SELECT  Meno, Cena  
FROM    produkt  
WHERE   Meno = 'žiarovka' AND Cena > 100;  
ORDER BY Kategória DESC, Meno ASC;
```

Usporiadanie je štandardne rastúce, t.j. default je ASC (ascending), pokial' nepoužijeme kľúčové slovo DESC (descending)

V zozname za ORDER BY sa môže vyskytnúť aj viac atribútov, pre každý nezávisle môže byť vzostupné alebo klesajúce utriedenie

produkt (Meno, Cena, Kategória, Výrobca)
kontrakt (Predávajúci, Kupujúci, Sklad, Výrobok)
podnik (Meno, Adresa, Okres)
osoba (Meno, Rodné_číslo, Telefón, Mesto)

Príkaz SELECT: zjednotenie, prienik, rozdiel

```
(SELECT Meno  
      FROM osoba  
     WHERE Mesto = 'Trnava')  
UNION  
(SELECT Meno  
      FROM osoba, kontrakt  
     WHERE Kupujúci = 'Novák' AND Sklad = 'Zohor');
```

Podobne sa používa **INTERSECT** (prienik) a **EXCEPT** (rozdiel).
Pozor, tabuľky musia byť kompatibilné, t.j. musia mať tie isté atribúty (inak treba atribúty premenovať tak, aby mali rovnaké mená)

produkt (Meno, Cena, Kategória, Výrobca)
kontrakt (Predávajúci, Kupujúci, Sklad, Výrobok)
podnik (Meno, Adresa, Okres)
osoba (Meno, Rodné_číslo, Telefón, Mesto)

Príkaz SELECT: spojenie (join)

```
SELECT  Meno, Sklad
FROM    osoba, kontrakt
WHERE   osoba.Meno = kontrakt.Kupujúci AND Mesto = 'Trnava'
        AND Výrobok = 'vysávač'
```

produkt (Meno, Cena, Kategória, Výrobca)
kontrakt (Predávajúci, Kupujúci, Sklad, Výrobok)
podnik (Meno, Adresa, Okres)
osoba (Meno, Rodné_číslo, Telefón, Mesto)

Príkaz SELECT: jednoznačnosť atribútov

Najdite mená osôb, ktoré si kúpili elektroniku:

```
SELECT  osoba.Meno  
FROM    osoba, kontrakt, produkt  
WHERE   osoba.Meno = Kupujúci  
        AND Výrobok = produkt.Meno  
        AND Kategória = 'elektronika';
```

produkt (Meno, Cena, Kategória, Výrobca)
kontrakt (Predávajúci, Kupujúci, Sklad, Výrobok)
podnik (Meno, Adresa, Okres)
osoba (Meno, Rodné_číslo, Telefón, Mesto)

Príkaz SELECT: n-ticové premenné (aliasy relácií)

Dvojice podnikov vyrábajúcich tie isté kategórie tovarov (t.j. aspoň jedna kategória je pre nich spoločná):

```
SELECT  p1.Výrobca, p2.Výrobca  
FROM    produkt AS p1, produkt AS p2  
WHERE   p1.Kategória = p2.Kategória  
        AND p1.Výrobca <> p2.Výrobca;
```

Toto je „dobrý programátorský štýl“. Aliasy relácií doporučujem používať **vždy** (aj tam, kde to nie je nutné)

produkt (Meno, Cena, Kategória, Výrobca)
kontrakt (Predávajúci, Kupujúci, Sklad, Výrobok)
podnik (Meno, Adresa, Okres)
osoba (Meno, Rodné_číslo, Telefón, Mesto)

Príkaz SELECT: INNER join

Syntax spojení:

```
SELECT X, Y, Z  
FROM r JOIN s ON r.Y = s.Y;
```

znamená to isté čo

```
SELECT X, Y, Z  
FROM r INNER JOIN s ON r.Y = s.Y;
```

znamená to isté čo

```
SELECT X, Y, Z  
FROM r, s  
WHERE r.Y = s.Y
```

Príkaz SELECT: vonkajšie spojenia (OUTER JOIN)

```
SELECT X, Y, Z  
FROM r LEFT OUTER JOIN s ON r.Y = s.Y;
```

je to isté čo

```
SELECT X, Y, Z  
FROM r, s  
WHERE r.Y = s.Y  
UNION  
SELECT X, Y, null  
FROM r  
WHERE Y NOT IN (SELECT Y FROM s);
```

Podobne sa definuje **RIGHT OUTER JOIN** a **FULL OUTER JOIN**

Príkaz SELECT: null hodnoty

SQL rozširuje všetky domény o hodnotu **null**. null predstavuje neznámu, neexistujúcu aj neurčenú hodnotu.

Pozor, SQL používa **trojhodnotovú logiku**:

NOT		AND	TRUE	FALSE	NULL	OR	TRUE	FALSE	NULL
null	null	null	null	false	null	null	true	null	null

Je dobré predstaviť si true = 1, null = 1/2, false = 0.

Potom platí $\neg p = 1 - p$, $p \wedge q = \min(p, q)$, $p \vee q = \max(p, q)$.

Pre všetky ostatné operácie a funkcie platí: Ak jeden z operandov resp. argumentov je null, potom aj výsledok je null

Príkaz SELECT: kontraintuitívne dotazy (null)

Príklad, na ktorom SELECT funguje (na prvý pohľad) kontraintuitívne:

A	B	C
null	6	7

```
SELECT r.C  
FROM r  
WHERE A < 1 OR A >= 1
```

Zdalo by sa, že selekčná podmienka $A < 1 \text{ OR } A \geq 1$ platí vždy (ak A je typu INTEGER), takže tento SELECT by mal intuitívne vrátiť tabuľku s jedným stĺpcom a jedným riadkom, ktorý obsahuje hodnotu 7. Lenže v našej tabuľke A=null, takže výraz $A < 1 \text{ OR } A \geq 1$ sa vyhodnotí ako null (t.j. nie ako true). Tým pádom výsledkom dotazu je prázdna tabuľka. Vyskúšajte!

Príkaz SELECT: kontraintuitívne dotazy (join)

```
SELECT      r.A  
FROM        r, s, t  
WHERE       r.A = s.A OR r.A = t.A;
```

Vyberá hodnoty A z r, ktoré sú v s alebo t. Skutočne?

A čo robí tento dotaz?

```
SELECT      r.A  
FROM        r, s, t  
WHERE       r.A = s.A;
```

Príkaz SELECT: poddotazy vracajúce práve 1 záznam

```
SELECT kontrakt.Výrobok  
FROM kontrakt  
WHERE Kupujúci =  
    (SELECT Meno  
     FROM osoba  
     WHERE Rodné_číslo = '450626/7887');
```

Takýto poddotaz musí vrátiť práve jednu hodnotu.
Inak nastane chyba (**run-time error**).

produkt (Meno, Cena, Kategória, Výrobca)
kontrakt (Predávajúci, Kupujúci, Sklad, Výrobok)
podnik (Meno, Adresa, Okres)
osoba (Meno, Rodné_číslo, Telefón, Mesto)

Príkaz SELECT: poddotazy vracajúce tabuľku

Nájdite spoločnosti, ktorých produkty kupuje Jožo:

```
SELECT podnik.Meno  
FROM podnik, produkt  
WHERE podnik.Meno = Výrobca  
      AND produkt.Meno IN  
          (SELECT Výrobok  
           FROM objednávka  
           WHERE Kupujúci = 'Jožo');
```

V takomto kontexte sa dá použiť aj:

s > **ALL** stĺpec relácie (true ak je s väčšie ako všetky hodnoty)
s > **ANY** stĺpec relácie (true ak je s väčšie ako niektorá hodnota)

Pozor: ALL sa nedá všeobecne použiť na vyjadrenie všeobecného kvantifikátora! (Napríklad, „Nájdite osoby, ktoré nekúpili nič“ sa nedá vyjadriť pomocou ALL)

Príkaz SELECT: podmienky na vrátené n-tice

```
SELECT podnik.Meno  
FROM podnik, produkt  
WHERE podnik.Meno = Výrobca  
      AND (produkt.Meno, Cena) IN  
            (SELECT Výrobok, Cena  
             FROM kontrakt  
             WHERE Kupujúci = 'Jožo');
```

Príkaz SELECT: opakovaný výskyt relácie v dotaze

Najdite názvy filmov, ktoré sa vyskytli viac než raz (neskôr):

`SELECT Názov`

`FROM film AS f1`

`WHERE Rok < ANY`

`(SELECT Rok
 FROM film
 WHERE Názov = f1.Názov);`

film (Názov, Rok, Režisér, Dĺžka)

Názvy filmov neidentifikujú film (názov sa môže použiť znova v niektorom z neskorších rokov).

Všimnite si platnosť (scope) premenných!

Príkaz SELECT: agregačné funkcie

produkt (Meno, Cena, Kategória, Výrobca)

```
SELECT sum(Cena)
FROM produkt
WHERE Výrobca = 'Vinárske Závody'
```

SQL obsahuje viacero agregačných funkcií:

sum, min, max, avg, stdev, count, ...
(sem patria tiež exists, in, any, all)

Všetky agregačné funkcie sa aplikujú na jeden atribút, s výnimkou
count (kde na atribútoch vôbec nezáleží):

```
SELECT count(*)
FROM kontrakt
```

Výnimkou je tiež funkcia exists, ktorá sa aplikuje na celú reláciu
(exists r je true, ak r je neprázdna a false ak r je prázdna)

Príkaz SELECT: grupovanie a agregácia

Väčšinou chceme aplikovať agregačné funkcie na časti tabuľky.

Dotaz na objem predaja jednotlivých výrobcov:

```
SELECT      produkt.Meno, sum(Cena)
FROM        produkt, kontrakt
WHERE       produkt.Meno = kontrakt.Výrobok
GROUP BY    produkt.Meno
```

1. Vypočíta sa relácia (t.j., SELECT ... FROM ... WHERE ...).
2. Relácia sa zoskupí podľa atribútov za GROUP BY.
3. Aplikuje sa agregačná funkcia na každú skupinu.

Projekcia SELECT môže obsahovať len:

- (1) zoskupené atribúty (atribúty v GROUP BY)
- (2) agregáty, t.j. výsledky agregačných funkcií.

produkt (Meno, Cena, Kategória, Výrobca)

Príkaz SELECT: podmienky na agregáty

Väčšinou chceme aplikovať agregačné funkcie na časti tabuľky.

Dotaz na objem predaja jednotlivých výrobcov; vybrať sa majú len tie produkty, ktoré majú viac ako 100 kupujúcich:

```
SELECT      Výrobok, sum(Cena)
FROM        produkt, kontrakt
WHERE       produkt.Meno = Výrobok
GROUP BY    Výrobok
HAVING     count(Kupujúci) > 100
```

Za HAVING nasledujú podmienky na agregované skupiny.

produkt (Meno, Cena, Kategória, Výrobca)
kontrakt (Predávajúci, Kupujúci, Sklad, Výrobok)

Príkaz SELECT: odstránenie duplikátov

```
SELECT DISTINCT p1.výrobca, p2.výrobca  
FROM produkt AS p1, produkt AS p2  
WHERE p1.Kategória = p2.Kategória  
      AND p1. Výrobca <> p2. Výrobca;
```

produkt (Meno, Cena, Kategória, Výrobca)

Príkaz SELECT: zachovanie duplikátov

Pozor: Operátory UNION, INTERSECT a EXCEPT pracujú s množinami (bez duplikátov) a nie s multimnožinami (s duplikátmi).

Dá sa však vynútiť, aby pracovali s multimnožinami (s duplikátmi):

```
(SELECT Meno  
      FROM osoba  
     WHERE Mesto = 'Trnava')
```

UNION ALL

```
(SELECT Meno  
      FROM osoba, kontrakt  
     WHERE Kupujúci = Meno AND Sklad = 'Zohor');
```

kontrakt (Predávajúci, Kupujúci, Sklad, Výrobok)
osoba (Meno, Rodné_číslo, Telefón, Mesto)

Najskôr:

Definovať typy dát

Potom:

Definovať schému databázy (množinu tabuľiek s atribútmi)

- Vytvorenie tabuľiek
- Odstránenie tabuľiek
- Modifikácia schémy tabuľiek

Nakoniec:

Definovať indexy a rôzne obmedzenia v záujme udržania konzistencie databázy

SQL DDL: typy dát (domains)

- Reťazce (pevnej alebo premennej dĺžky) CHAR, VARCHAR
- Bitové reťazce INTEGER, SHORTINT
- Pohyblivá čiarka REAL, DOUBLE
- Dátum a čas DATE/TIME

Domény sa používajú pri definícii tabuľiek.

Definícia domény (alias pre typ):

```
CREATE DOMAIN adresa AS VARCHAR(55);
```

SQL DDL: vytvorenie tabuľky

```
CREATE TABLE osoba(  
    Meno          VARCHAR(30),  
    Rodné_číslo  INTEGER,  
    Vek           SHORTINT,  
    Mesto         VARCHAR(30),  
    Pohlavie     BIT(1),  
    Dátum_narodenia DATE  
);
```

SQL DDL: odstránenie tabuľky

`DROP osoba;`

SQL DDL: modifikácia tabuľky

```
ALTER TABLE osoba  
    ADD Telefón CHAR(16);
```

```
ALTER TABLE osoba  
    DROP Vek;
```

SQL DDL: preddefinované hodnoty (defaults)

```
CREATE TABLE osoba (
    Meno          VARCHAR(30),
    Rodné_číslo  INTEGER,
    Vek           SHORTINT DEFAULT 100,
    Mesto         VARCHAR(30) DEFAULT 'Trnava',
    Pohlavie     CHAR(1)    DEFAULT '?',
    Dátum_narodenia DATE
);
```

SQL DDL: indexy

Indexy sú veľmi dôležité pre **urýchlenie výpočtu dotazov**. Na druhej strane, spomaľujú aktualizácie a zväčšujú rozsah databázy

Majme tabuľku:

osoba (Meno, Rodné_číslo, Vek, Mesto)

Index na **Rodné_číslo** umožňuje prístup k riadku s daným rodným číslom **ovel'a rýchlejšie** ako pri sekvenčnom prehľadávaní tabuľky

Problém výberu indexov súvisí s návrhom a prevádzkováním bázy dát. (Jeho exaktné riešenie je veľmi ťažké.)

SQL DDL: vytvorenie indexu

```
CREATE INDEX index_rc ON osoba(Rodné_číslo)
```

Môžeme vytvárať indexy aj pre viacero atribútov:

```
CREATE INDEX index_dvojity ON  
osoba (Meno, Rodné_číslo)
```

Prečo sa automaticky neindexuje všetko?

SQL DDL: vytvorenie VIEW

Views (pohľady) sú dotazy zapamätané v databáze. Používajú sa ako virtuálne tabuľky (za FROM). Možno na ne špecifikovať prístupové práva, aj indexy. Umožňujú, aby rôzni užívatelia videli rôzne časti databázy

VIEWS sú užitočné pri štruktúrovaní zložitých dotazov. Taktiež sú užitočné pri aktualizácii databázy, ktorá prebieha cez viacero tabuľiek naraz

Kontrakty na elektronické výrobky:

```
CREATE VIEW nákupy_elektroniky AS  
    SELECT Výrobok, Kupujúci, Predávajúci, Sklad  
    FROM kontrakt, produkt  
    WHERE kontrakt.Výrobok = produkt.Meno  
        AND produkt.Kategória = 'elektronika'
```

SQL DDL: použitie VIEW

```
CREATE VIEW trnavskí_zákazníci AS  
    SELECT Kupujúci, Predávajúci, Výrobok, Sklad  
    FROM osoba, kontrakt  
    WHERE osoba.Mesto = 'Trnava' AND  
          osoba.Meno = kontrakt.Kupujúci
```

Neskoršie použitie VIEW:

```
SELECT Meno, Sklad  
FROM trnavskí_zákazníci , produkt  
WHERE trnavskí_zákazníci.Výrobok = produkt.Meno  
      AND produkt.Kategória = 'textil'
```

SQL DDL: odstránenie VIEW

```
DROP VIEW trnavskí_zákazníci;
```

SQL DDL: aktualizácia cez VIEW

Aktualizácia databázy cez views je OK, ale treba s ňou narábať opatrne. Môže sa napríklad stať, že aktualizácia prebieha cez neexistujúce atribúty:

```
CREATE VIEW nákupy_tesco AS  
    SELECT Sklad, Predávajúci, Kupujúci  
    FROM kontrakt  
    WHERE Sklad = 'Tesco'
```

```
INSERT INTO nákupy_tesco  
    VALUES ('Tesco', 'Jožo', 'Fero');
```

vloží riadok ('Jožo', 'Fero', 'Tesco', **null**) do tabuľky kontrakt.

kontrakt (**Predávajúci, Kupujúci, Sklad, Výrobok**)

SQL DDL: kontrainuitívna aktualizácia cez VIEW

```
CREATE VIEW trnava AS  
    SELECT Predávajúci, Výrobok, Sklad  
    FROM osoba, kontrakt  
    WHERE osoba.Mesto = 'Trnava' AND  
          osoba.Meno = kontrakt.Kupujúci
```

Čo spôsobí nasledujúci INSERT? (domáca úloha)

```
INSERT INTO trnava  
VALUES ('Jožo', 'sandále', 'Kmart')
```

Niečo to urobí (čo presne?), ale výsledok je konstraintitívny

produkt (Meno, Cena, Kategória, Výrobca)
kontrakt (Predávajúci, Kupujúci, Sklad, Výrobok)
podnik (Meno, Adresa, Okres)
osoba (Meno, Rodné_číslo, Telefón, Mesto)

Úvod do databázových systémov

[**http://www.dcs.fmph.uniba.sk/~plachetk**](http://www.dcs.fmph.uniba.sk/~plachetk)
/TEACHING/DB2013

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2013–2014

Formalizácia dotazov

- **Databáza je štruktúra** pre relačný (predikátový) kalkul
- **Dotaz je formula** $\varphi(X_1, \dots, X_n)$, kde X_1, \dots, X_n sú voľné premenné
- **Výsledok dotazu** $\varphi(X_1, \dots, X_n)$ je množina usporiadaných n-tíc $[X_1, \dots, X_n]$, pre ktoré platí $\varphi(X_1, \dots, X_n)$

Ako **počítať** výsledok dotazu? O tom je **relačná algebra**.

Operandami relačnej algebry sú **relácie**. Relácia sa dá reprezentovať tabuľkou, kde mená atribútov označujú stĺpce tabuľky (hlavičkový riadok). Zatiaľ predpokladajme, že tabuľka je **množina riadkov** (bez duplikátov)

V ďalšom texte bude **X** označovať vektor atribútov $[X_1, \dots, X_n]$, **Y** bude označovať vektor atribútov $[Y_1, \dots, Y_n]$ a podobne

- **Formálna špecifikácia SQL.** Hoci syntax SELECT je veľmi „barokná“, sémantika sa dá presne popísať relačnou algebrou
- **Optimalizácia dotazov** v DBMS. SELECT sa dá priamočiaro transformovať na ekvivalentný výraz relačnej algebry (operátorový strom)
 - Pri optimalizácii sa tento výraz transformuje na ekvivalentný, výpočtovo efektívnejší algebraický výraz (resp. na postupnosť priradení), ktorému sa hovorí **logický plán**
 - Logický plán zapísaný v relačnej algebре sa namapuje do **fyzických operátorov** konkrétneho DBMS. Takto vznikne **fyzický plán** výpočtu dotazu

Pohľad programátora:

- Dotaz, resp. program v **relačnom kalkule, Datalogu a SQL** vyjadruje **ČO** treba vypočítať (**nie ako**). Nie je príliš dôležité, ako “efektívne” je program napísaný. O optimalizáciu sa stará stroj, nie programátor
- Dotaz, resp. program v **relačnej algebre** vyjadruje **AKO** sa vypočíta výsledná relácia z EDB relácií

(V konečnom dôsledku z návodu AKO sa niečo počíta vyplýva aj ČO sa počíta. Lenže ľudský spôsob myslenia je „najskôr ČO, až potom AKO“, t.j. „najskôr špecifikácia, až potom implementácia“)

Základné operátory relačnej algebry

- **Zjednotenie, prienik, rozdiel** (vyžaduje sa, aby relácie mali rovnakú schému, t.j. aby boli rovnakého typu)
- **Selekcia**: výber riadkov
- **Projekcia**: výber stĺpcov
- **Kartézsky súčin a join**: skladanie relácií
- **Premenovanie** relácií a atribútov relácií
- ...

Zjednotenie, prienik, rozdiel

Relačná algebra

$$r_1 \cup r_2 =$$

$$r_1 \cap r_2 =$$

$$r_1 - r_2 =$$

Relačný kalkul:

$$\{\mathbf{X}: r_1(\mathbf{X}) \vee r_2(\mathbf{X})\}$$

$$\{\mathbf{X}: r_1(\mathbf{X}) \wedge r_2(\mathbf{X})\}$$

$$\{\mathbf{X}: r_1(\mathbf{X}) \wedge \neg r_2(\mathbf{X})\}$$

Všimnite si, že **negácia sa vyjadruje ako rozdiel relácií** (inak sa ani vyjadriť nedá). Relácia r_1 vystupuje ako „pozitívny kontext“, t.j. obsahuje množinu kandidátov na výsledok. Negácia spôsobí len vyniechanie niektorých n-tíc z r_1 (tých, ktoré sú v r_2)

Kalkul:

Nech r je typu $r(X, Y)$. Potom

$$\Pi_X(r) = \{X : \exists Y \ r(X, Y)\}$$

Tabuľková definícia: $r_2 := \Pi_X(r_1)$

- r_2 vzniká kopírovaním riadkov r_1 , pričom pre každý riadok r_1 sa do r_2 skopírujú len tie atribúty, ktoré sú v X
- Nakoniec treba eliminovať duplikované riadky v r_2 (ak počítame s množinami)

Príklad (Ullman)

Relation sells

Bar	Beer	Price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

prices := $\Pi_{\text{Beer}, \text{Price}}(\text{sells})$

Beer	Price
Bud	2.50
Miller	2.75
Miller	3.00

Kalkul:

- Nech r je typu $r(\mathbf{X})$. Potom

$$\sigma_{c(\mathbf{X})}(r) = \{\mathbf{X} : r(\mathbf{X}) \wedge c(\mathbf{X})\}$$

Tabuľková definícia: $r_2 := \sigma_c(r_1)$

- r_2 vzniká kopírovaním riadkov r_1 , pričom do r_2 sa skopírujú len tie riadky, pre ktoré platí podmienka c

Príklad (Ullman)

Relation sells

Bar	Beer	Price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

joe_menu := $\sigma_{\text{Bar}=\text{"Joe's"}}(\text{ sells})$

Bar	Beer	Price
Joe's	Bud	2.50
Joe's	Miller	2.75

Kartézsky súčin: \times

Kalkul:

- Nech r_1 je typu $r_1(\mathbf{X})$ a r_2 je typu $r_2(\mathbf{Y})$, pričom $\mathbf{X} \cap \mathbf{Y} = \emptyset$. Potom
 $r_1 \times r_2 = \{[\mathbf{X}, \mathbf{Y}] : r_1(\mathbf{X}) \wedge r_2(\mathbf{Y})\}$

Tabuľková definícia: $r_3 := r_1 \times r_2$

- r_3 vzniká kopírovaním všetkých dvojíc riadkov r_1 a r_2
- Spoločné atribúty v r_1 a r_2 jednoducho nedovolíme, aby sme sa vyhli problémom v pomenovaní atribútov r_3 . (Vlastne môžeme dovoliť aj spoločné atribúty, ale vo výsledku potom musíme dôsledne používať prefixy atribútov—aby bolo jasné, z ktorej relácie pochádzajú.)

Kartézsky súčin: ×

Príklad (Ullman)

$$r3 := r1 \times r2$$

$r1($

A,	B
1	2
3	4

 $)$

$r2($

B,	C
5	6
7	8
9	10

 $)$

$r3($

A,	r1.B,	r2.B,	C
1	2	5	6
1	2	7	8
1	2	9	10
3	4	5	6
3	4	7	8
3	4	9	10

 $)$

Join (theta-join): \bowtie_c

Kalkul:

- Nech r_1 je typu $r_1(\mathbf{X})$ a r_2 je typu $r_2(\mathbf{Y})$, pričom $\mathbf{X} \cap \mathbf{Y} = \emptyset$. Potom
 $r_1 \bowtie_{c(\mathbf{X}, \mathbf{Y})} r_2 = \{[\mathbf{X}, \mathbf{Y}] : r_1(\mathbf{X}) \wedge r_2(\mathbf{Y}) \wedge c(\mathbf{X}, \mathbf{Y})\}$

Tabuľková definícia: $r_3 := r_1 \bowtie_{c(\mathbf{X}, \mathbf{Y})} r_2$

- r_3 vzniká kopírovaním všetkých dvojíc riadkov r_1 a r_2 , pričom do r_3 sa skopírujú len tie dvojice, pre ktoré platí podmienka c
- Spoločné atribúty v r_1 a r_2 jednoducho nedovolíme, aby sme sa vyhli problémom v pomenovaní atribútov r_3 . (Vlastne môžeme dovoliť aj spoločné atribúty, ale vo výsledku potom musíme dôsledne používať mená relácií ako prefixy atribútov.)
- Theta-join je selekcia aplikovaná na kartézsky súčin

Join (theta-join): \bowtie_c

Príklad (Ullman)

sells(Bar, Beer, Price)		
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Coors	3.00

bars(Name, Addr)	
Joe's	Maple St.
Sue's	River Rd.

bar_info := sells $\bowtie_{\text{sells.bar} = \text{bars.name}}$ bars

bar_info(Bar, Beer, Price, Name, Addr)				
Joe's	Bud	2.50	Joe's	Maple St.
Joe's	Miller	2.75	Joe's	Maple St.
Sue's	Bud	2.50	Sue's	River Rd.
Sue's	Coors	3.00	Sue's	River Rd.

Kalkul:

- Bežná substitúcia používaná matematikmi. Substituovať možno nielen mená atribútov, ale aj meno relácie

Tabuľková definícia: $r_2 := P_{r_2(Y)}(r_1)$

- r_2 je kópiou r_1 , len sa (možno) volá inak a jej (niektoré) atribúty sa volajú inak

Konkrétna syntax operátora P nie je príliš dôležitá. Avšak musí z nej byť jasné, ktoré meno je pôvodné a ktoré meno je nové.

Napríklad, pre reláciu $\text{lubi}(\text{Pijan}, \text{Alkohol})$ sa $P_{\text{Ochmelka}} := \text{Pijan}$ (lubi) chápe rovnako ako $P_{\text{Ochmelka} \leftarrow \text{Pijan}} (\text{lubi})$ alebo $P_{\text{Ochmelka}, \text{Alkohol}} (\text{lubi})$

Premenovanie: P

Príklad (Ullman)

$$\text{bars}(\text{Name}, \text{Addr})$$

Name	Addr
Joe's	Maple St.
Sue's	River Rd.

$$P_{r(\text{Bar}, \text{Addr})} \text{bars}$$
$$r(\text{Bar}, \text{Addr})$$

Bar	Addr
Joe's	Maple St.
Sue's	River Rd.

Natural join: \bowtie

Kalkul:

- Nech r_1 je typu $r_1(X, Z)$ a r_2 je typu $r_2(Y, Z)$, pričom Z sú spoločné atribúty r_1 a r_2 . Potom

$$r_1 \bowtie r_2 = \{[X, Y, Z] : r_1(X, Z) \wedge r_2(Y, Z)\}$$

Tabuľková definícia: $r_3 := r_1 \bowtie r_2$

- r_3 vzniká kopírovaním všetkých dvojíc riadkov r_1 a r_2 , pričom spoločné atribúty (atribúty s rovnakým menom) sú testované na rovnosť a sú kopírované iba raz
- Natural join sa dá vyjadriť pomocou premenovania, theta-joinu a projekcie. Často je však prirodzené zlúčiť dve relácie do jednej na základe **rovnosti spoločných atribútov**

Natural join: \bowtie

Príklad (Ullman)

sells(Bar, Beer, Price)			bars(Name, Addr)	
Joe's	Bud	2.50		
Joe's	Miller	2.75		
Sue's	Bud	2.50		
Sue's	Coors	3.00		

bar_info := sells \bowtie P_{bars(Bar, Addr)}(bars)

bars.Name bolo treba premenovať na **bars.Bar**, aby natural join fungoval.

bar_info(Bar, Beer, Price, Addr)			
Joe's	Bud	2.50	Maple St.
Joe's	Miller	2.75	Maple St.
Sue's	Bud	2.50	River Rd.
Sue's	Coors	3.00	River Rd.

Ekvivalentné notácie zložených výrazov relačnej algebry

Tri formy notácie výrazu: bary, ktoré budú sídlia na Maple St., alebo predávajú Bud za menej ako 3\$ (Ullman)

Výraz s operátormi

$$\begin{aligned}\Pi_{\text{Name}}(\sigma_{\text{Addr} = \text{"Maple St."}}(\text{bars}) \cup \\ P_{\text{Name} := \text{Bar}}(\Pi_{\text{Bar}}(\sigma_{\text{Price} < 3 \text{ AND Beer} = \text{"Bud"}}(\text{sells}))))\end{aligned}$$

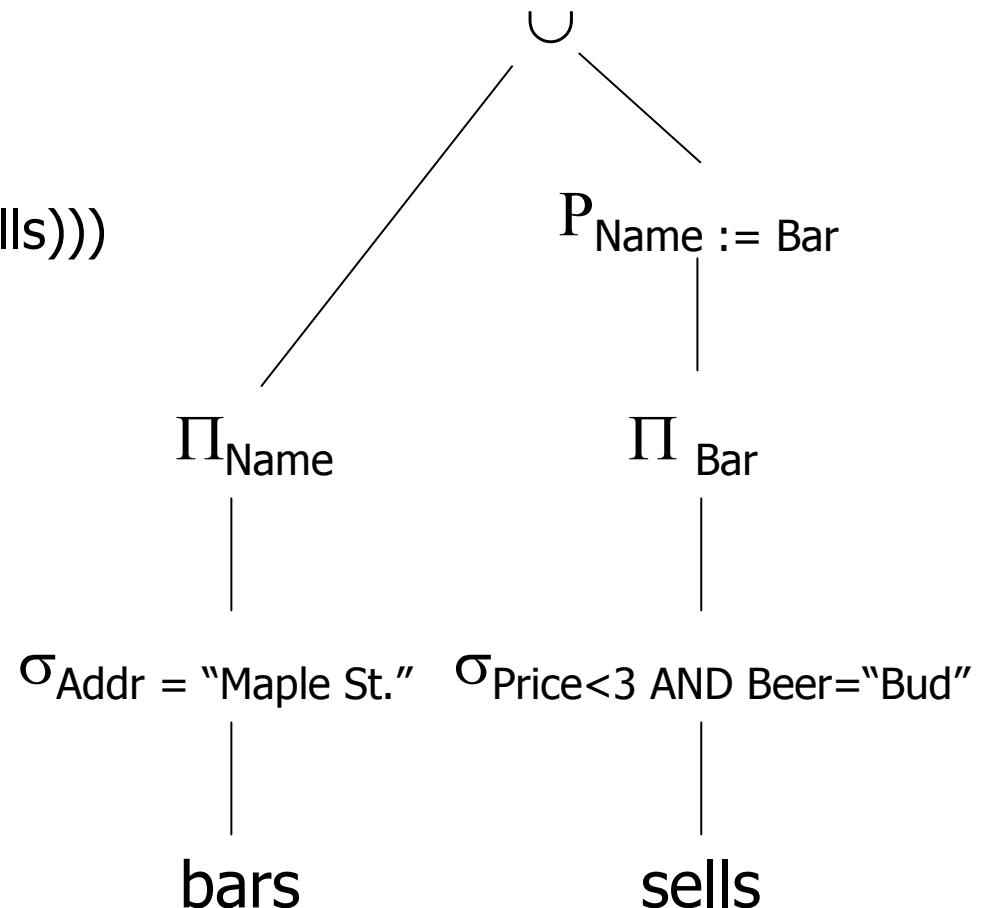
Postupnosť priradení

$$r := \Pi_{\text{Name}}(\sigma_{\text{Addr} = \text{"Maple St."}}(\text{bars});$$

$$s := P_{\text{Name} := \text{Bar}}(\\ \Pi_{\text{bar}}(\sigma_{\text{Price} < 3 \text{ AND Beer} = \text{"Bud"}}(\text{sells})));$$

$$t := r \cup s$$

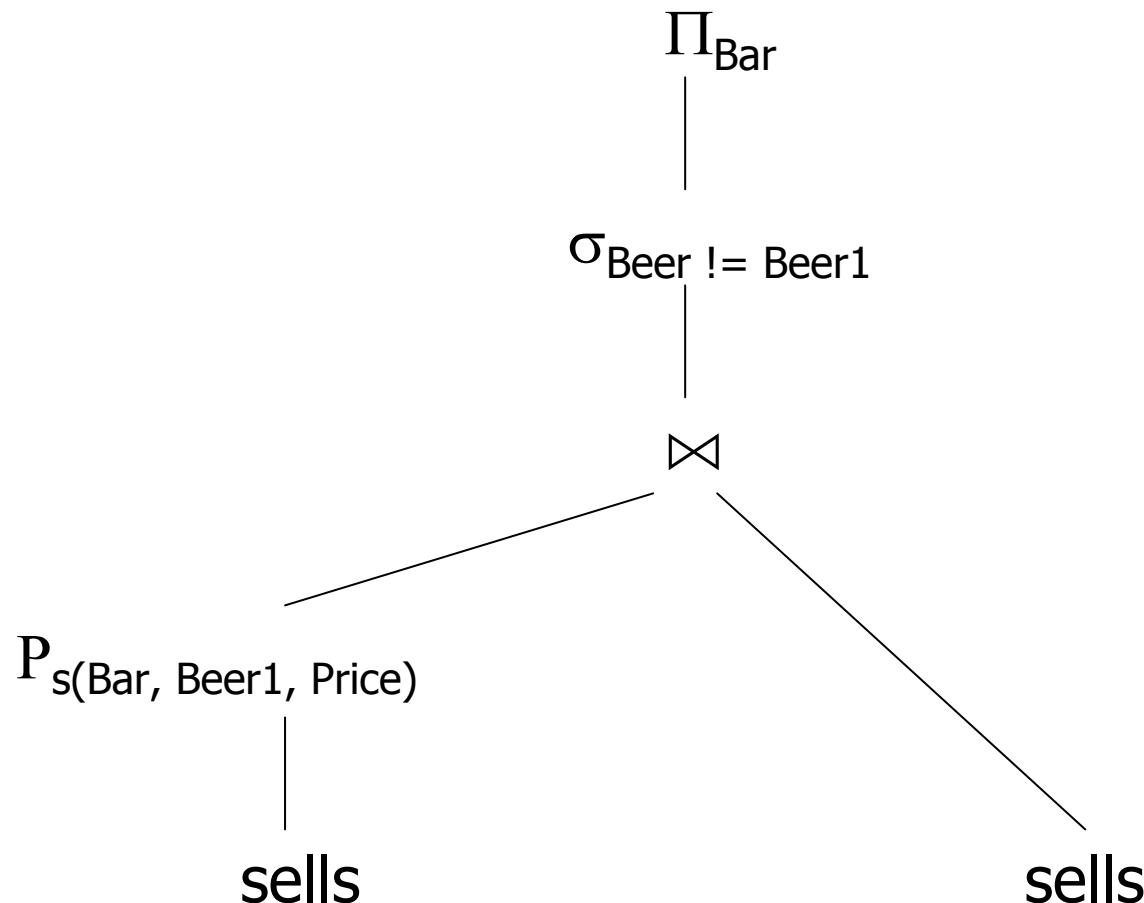
Strom výrazu



Ekvivalentné notácie relačnej algebry

Ešte príklad (Ullman): bary, ktoré predávajú (aspoň) dva rôzne druhy piva za rovnakú cenu

$$\Pi_{\text{Bar}}(\sigma_{\text{Beer} \neq \text{Beer1}}(P_{s(\text{Bar}, \text{Beer1}, \text{Price})} (\text{sells}) \bowtie \text{sells}))$$



Niekteré zákony (množinovej) relačnej algebry

- Prirodzené spojenie (natural join) a zjednotenie sú komutatívne, asociatívne a idempotentné
- Platia distributívne zákony
 - $r \bowtie (s \cup t) = (r \bowtie s) \cup (r \bowtie t)$
 - $r \bowtie (s - t) = (r \bowtie s) - (r \bowtie t)$
- Ak $Y \subseteq X$, tak potom $\Pi_Y \Pi_X (r) = \Pi_Y (r)$
- Ak podmienka c neobsahuje atribúty s , tak potom $\sigma_c(r \times s) = \sigma_c(r) \times s$
- Ak podmienka c_{rs} obsahuje atribúty r aj s , podmienka c_r obsahuje len atribúty r , a podmienka c_s obsahuje len atribúty s , tak potom

$$\sigma_{c_{rs} \wedge c_r \wedge c_s}(r \times s) = \sigma_{c_r}(r) \bowtie_{c_{rs}} \sigma_{c_s}(s)$$

Optimalizácia na úrovni relačnej algebry: príklad

dodava

Firma	Vyrobok	Cena	Lehota
-------	---------	------	--------

firmy

Firma	Mesto
-------	-------

objednavky

Klient	Vyrobok
--------	---------

Ktorý klient objednal výrobok, ktorý vie dodat' niektorá firma z Nuernberg?

SELECT o.Klient

FROM objednavky o, dodava d, firmy f

WHERE f.Mesto = 'nuernberg' **and** f.Firma = d.Firma

and d.Vyrobok = o.Vyrobok

Optimalizácia na úrovni relačnej algebry: príklad

Ktorý klient objednal výrobok, ktorý vie dodáť niektorá firma z Nuernberg?

dodava

Firma	Vyrobok	Cena	Lehota
vobis	pc386	2000	4
quelle	pc386	1900	9
vobis	pc486	2900	4
escom	pc486	3000	5
vobis	pc586	5000	7
escom	pc586	5900	9
vobis	hp4l	1400	6
vobis	hddisk	13	0
escom	hddisk	12	0
quelle	cdrom	400	4

dodavatelia

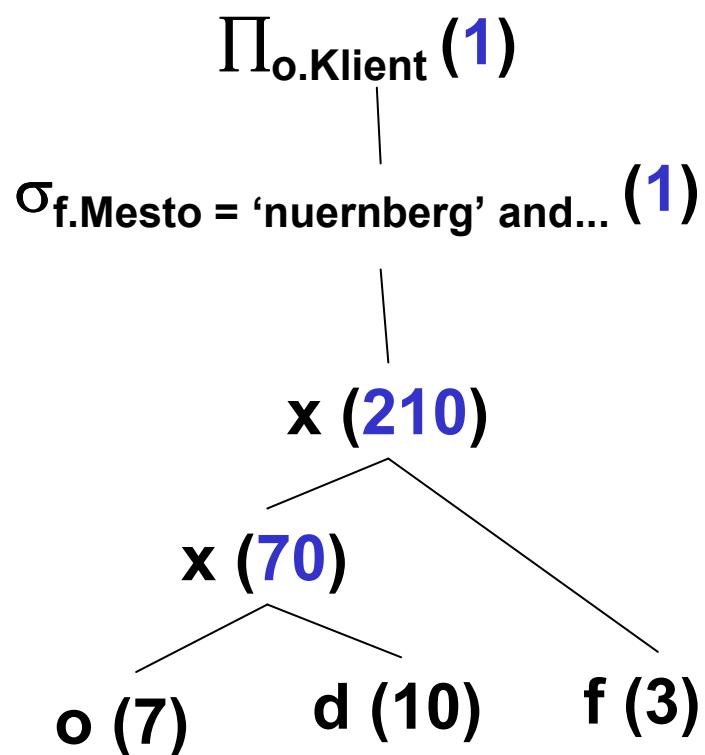
Firma	Mesto
vobis	ulm
escom	ulm
quelle	nuernberg

objednavky

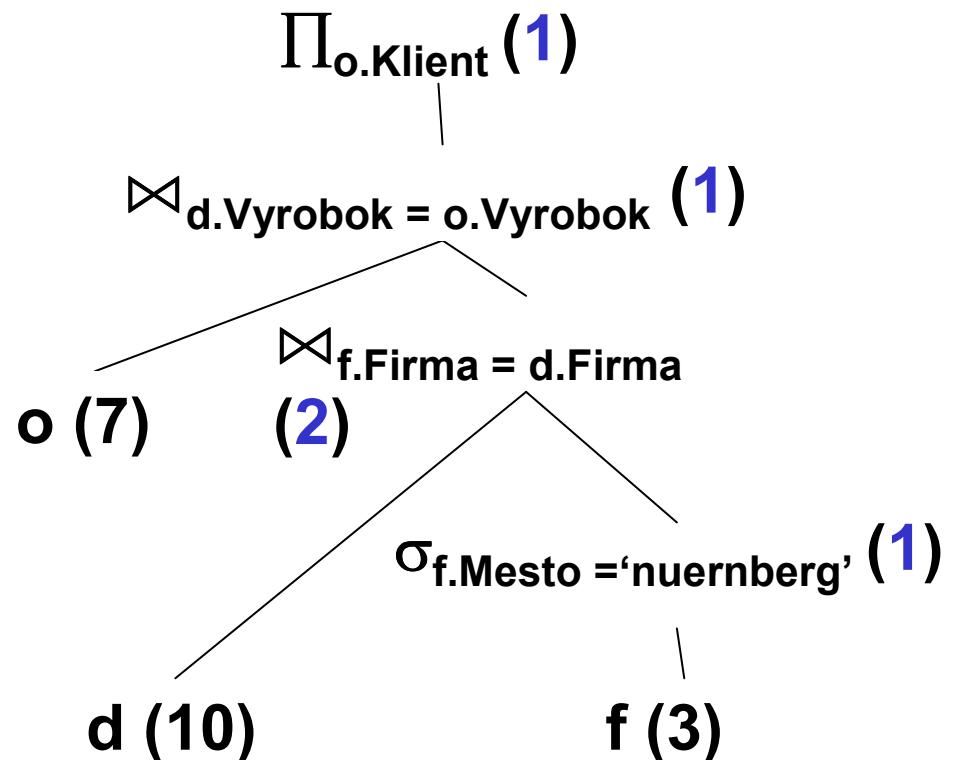
Klient	Vyrobok
meier	pc486
meier	hddisk
reich	pc586
reich	hp4l
reich	hddisk
arm	pc386
arm	hddisk

Optimalizácia na úrovni relačnej algebry: príklad

```
SELECT o.Klient  
FROM objednavky o, dodava d, firmy f  
WHERE f.Mesto = 'nuernberg' and f.Firma = d.Firma  
and d.Vyrobok = o.Vyrobok
```



Medzivýsledok: **282 riadkov**



Medzivýsledok: **5 riadkov**

Výsledok optimalizovaného dotazu musí byť rovnaký ako výsledok pôvodného dotazu!

- **Selekciu urob čo najskôr**
- **Vyhýbaj sa kartézskym súčinom, nahrad' ich joinom.** Ak nasleduje selekcia, pridaj ju do joinovacej podmienky
- Postupnosť unárnych operácií (selekcie a projekcie) spoj do jednej operácie a zviaž s nasledujúcou operáciou
- Výsledok opakovaného podvýrazu ulož, ak je veľkosť tabuľky malá (materialisation). Opakované čítanie malej tabuľky môže byť oveľa rýchlejšie ako opakovaný výpočet
- **Nájdi optimálne poradie joinov** technikou dynamického programovania. Poradí pre N tabuliek je zhruba 2^N , ale aj tak sa to oplatí pre $N < 8$. Pre väčšie N použi greedy techniku na vytvorenie ľavolineárneho binárneho stromu

Výpočet jednoduchého SELECT

SELECT X_1, \dots, X_n

FROM r_1, \dots, r_m

WHERE c

Projekcia selekcie kartézskeho súčinu

$$\Pi_{X_1, \dots, X_n} \sigma_c(r_1 \times \dots \times r_m)$$

Projekcia joinu

$$\Pi_{X_1, \dots, X_n} (r_1 \bowtie_c \dots \bowtie_c r_m)$$

Toto ešte nie je celkom presné, aj keď to zhruba vyjadruje ako sa „kanonický“ SELECT počíta. Nevenovali sme pozornosť premenovaniam relácií a atribútov. A hlavne sme nevenovali pozornosť **duplicátom**

Multimnožiny (bags)

- **Multimnožina (bag)** je množina s duplikátmi. Napríklad $\{1, 2, 3, 1, 2\}$ je multimnožina. Aj $\{1, 2, 3\}$ je multimnožina. Každá množina je multimnožinou, ale nie nutne naopak
- **SQL počíta nad multimnožinami.** Dôvodom je snaha ušetriť, eliminácia duplikátov je rovnako zložitá ako triedenie. (Na druhej strane, skutočne sa šetri, keď je duplikátov veľa?)
- **Relačná algebra počíta nad multimnožinami**
- Operátory relačnej algebry sa dajú definovať aj pre multimnožiny
 - *Zjednotenie, prienik a rozdiel* treba poopravíť (pre UNION ALL, ...)
 - *Projekcia* pre multimnožiny neeliminuje duplikáty
 - *Selekcia, kartézsky súčin a joiny* sú definované ako predtým (podľa tabuľkovej sémantiky)

Multimnožiny (bags)

Príklad (Ullman)

$$r(\quad \begin{array}{|c|c|} \hline A, & B \\ \hline 1 & 2 \\ 5 & 6 \\ 1 & 2 \\ \hline \end{array} \quad)$$

$$\sigma_{A < 3 \wedge B < 4}(r) =$$

$$\begin{array}{|c|c|} \hline A & B \\ \hline 1 & 2 \\ 1 & 2 \\ \hline \end{array}$$

Multimnožiny (bags)

Príklad (Ullman)

$$r(\quad \begin{array}{|c|c|} \hline A, & B \\ \hline 1 & 2 \\ 5 & 6 \\ 1 & 3 \\ \hline \end{array} \quad)$$
$$\Pi_A(r) = \begin{array}{|c|} \hline A \\ \hline 1 \\ 5 \\ 1 \\ \hline \end{array}$$

Multimnožiny (bags)

Príklad (Ullman)

$$r(\begin{array}{|c|c|} \hline A, & B \\ \hline 1 & 2 \\ 5 & 6 \\ 1 & 2 \\ \hline \end{array})$$
$$s(\begin{array}{|c|c|} \hline B, & C \\ \hline 3 & 4 \\ 7 & 8 \\ \hline \end{array})$$

$r \times s =$

A	r.B	s.B	C
1	2	3	4
1	2	7	8
5	6	3	4
5	6	7	8
1	2	3	4
1	2	7	8

Multimnožiny (bags)

Príklad (Ullman)

$r($

A,	B
----	---

 $)$

A,	B
1	2
5	6
1	2

$s($

B,	C
----	---

 $)$

B,	C
3	4
7	8

$r \bowtie_{r.B < s.B} s =$

A	r.B	s.B	C
1	2	3	4
1	2	7	8
5	6	7	8
1	2	3	4
1	2	7	8

Multimnožiny (bags)

Zjednotenie: multimnožiny sa „zreťazia“.

Príklad: $\{1, 2, 1\} \cup \{1, 1, 2, 3, 1\} = \{1, 1, 1, 1, 1, 2, 2, 3\}$

Prienik: vo výsledku sa prvok objaví toľkokrát, kol'kokrát je minimum jeho výskytu v operandoch

Príklad: $\{1, 2, 1, 1\} \cap \{1, 2, 1, 3\} = \{1, 1, 2\}$

Rozdiel: vo výsledku sa prvok objaví toľkokrát, kol'kokrát sa vyskytuje v prvom operande mínus kol'kokrát sa vyskytuje v druhom operande (samozrejme, aspoň nulakrát)

Príklad: $\{1, 2, 1, 1\} - \{1, 2, 3\} = \{1, 1\}$

Multimnožiny (bags)

Pozor, **nie všetky vlastnosti operácií s množinami sú zachované pre multimnožiny!**

Príklad (Ullman): zjednotenie množín je idempotentné (t.j. $s \cup s = s$), ale zjednotenie multimnožín nie je

Ďalšie operátory relačnej algebry

- Eliminácia duplikátov: Δ
- (Triedenie: T)
- Full join: OUTERJOIN
- Grupovanie a agregácia: Γ

Eliminácia duplikátov: Δ

- $r_2 := \Delta(r_1)$

r_2 je kópiou r_1 , ale bez duplikovaných riadkov

Príklad (Ullman)

$$r = (\begin{array}{|c|c|} \hline \text{A} & \text{B} \\ \hline \end{array})$$

A	B
1	2
3	4
1	2

$$\Delta(r) =$$

A	B
1	2
3	4

Triedenie: T

- $L := T_{X_1, \dots, X_n}(r)$

L je **zoznam**, ktorý vznikol utriedením r najprv podľa X_1 , potom podľa X_2, \dots , nakoniec podľa X_N (rovnosti sa riešia náhodne). Ak je pred niektorým atribútom šípka nadol, tak sa triedi zostupne, inak sa triedi vzostupne

Príklad (Ullman)

$$r = (\begin{array}{|c|c|} \hline A & B \\ \hline 1 & 2 \\ 3 & 4 \\ 5 & 2 \\ \hline \end{array})$$

$$T_B(r) = [(5, 2), (1, 2), (3, 4)]$$

Full join: OUTERJOIN

- $r_3 := r_1 \text{ OUTERJOIN } r_2$

Full join je podobný theta-joinu, ale do výsledku navyše pribudnú riadky z r_1 a r_2 , ktoré sa s ničím nespájajú. Chýbajúce hodnoty v týchto riadkoch sa doplnia špeciálnymi hodnotami **null**

Príklad (Ullman)

$$r_1 = (A \mid B)$$

A	B
1	2
4	5

$$r_2 = (B \mid C)$$

B	C
2	3
6	7

(1,2) sa spája s (2,3),
ale (4,5) z r1 a (6,7) z r2 sa nespájajú s ničím

$r_1 \text{ OUTERJOIN } r_2 =$

A	B	C
1	2	3
4	5	null
null	6	7

Grupovanie a agregácia

$$r_2 := \Gamma_X(r_1)$$

O každom atribúte $X \in X$ platí

- buď X je atribútom r_1 (v tom prípade atribútu X hovoríme **grupovací atribút**), alebo
- $X = AGG(Y)$, kde Y je atribútom r_1 a AGG je nejaká agregačná funkcia, ktorá zo stĺpca Y vyrába jednu hodnotu (v tom prípade atribútu X hovoríme **agregovaný atribút**)

\forall SQL, $AGG \in \{\text{SUM}, \text{COUNT}, \text{AVG}, \text{STDEV}, \text{MAX}, \text{MIN}\}$

Operátor $\Gamma_X(r_1)$

1. najprv vytvorí z relácie r_1 skupiny, pričom riadky v každej zo skupín majú rovnaké hodnoty grupovacích atribútov
2. potom vypočíta agregované atribúty pre každú skupinu

Výsledkom je tabuľka, v ktorej každej skupine prináleží jeden riadok

Grupovanie a agregácia

Príklad (Ullman)

$$r = (A \quad B \quad C)$$

A	B	C
1	2	3
4	5	6
1	2	5

$$\Gamma_{A, B, \text{AVG}(C)}(r) = ??$$

1.krok: zgrupuj r podľa A a B :

A	B	C
1	2	3
1	2	5
4	5	6

2.krok: vypočítaj average C pre každú grupu:

A	B	AVG(C)
1	2	4
4	5	6

Grupovanie a agregácia presnejšie

Definícia: $\Gamma_{A_1, A_2, \dots, A_n}(r)$ je agregačný operátor aplikovaný na r:

- A_i je bud' **grupovací atribút alebo agregácia**

$\langle \text{SUM}(A_i) \mid \text{COUNT}(A_i) \mid \text{AVG}(A_i) \mid \text{STDEV}(A_i) \mid \text{MIN}(A_i), \text{MAX}(A_i) \rangle$

(kde A_i je **agregovaný atribút**). Nech **G** je množina grupovacích atribútov a **A** je množina agregovaných atribútov. (r môže obsahovať aj iné atribúty, t.j. také, ktoré nepatria do **G** ani do **A**.)

- Medzivýsledkom je relácia r' , ktorá vzniká projekciou r na množinu atribútov **G** s následným odstránením duplikátov:
 $r' := \Delta(\Pi_{\mathbf{G}}(r))$. Každý riadok r' zodpovedá skupine riadkov v r
- Výsledkom je relácia r' rozšírená o agregované atribúty, pričom hodnota agregovaného atribútu v riadku výsledku sa vypočíta aplikáciou príslušnej agregačnej funkcie na zodpovedajúcu skupinu riadkov v r

Syntax a sémantika SELECT s GROUP BY a HAVING:

SELECT <S_attr>	5
FROM r_1, r_2, \dots, r_n	1
WHERE <w_cond>	2
GROUP BY <G_attr>	3
HAVING <h_cond>	4

$$\Pi_{S_attr}(\sigma_{h_cond}(\Gamma_{G_attr}, \text{agg}(Attr)(\sigma_{w_cond}(r_1 \times r_2 \times \dots \times r_n))))$$

5 4 3 2 1

Pozor! Keď sa použije GROUP BY, tak atribúty v klauzách
SELECT a HAVING sú **len** buď niektoré z grupovacích
atribútov G_attr alebo sú výsledkom agregácie agg(Attr)

Rozšírenie jazykov o agregáciu: SQL

Daná je relácia $r(A, B, C)$. Čo je výsledkom nasledujúceho SQL príkazu?

```
select A, B, C  
from r  
group by A, B, C
```

Úvod do databázových systémov

[**http://www.dcs.fmph.uniba.sk/~plachetk**](http://www.dcs.fmph.uniba.sk/~plachetk)
/TEACHING/DB2013

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2013–2014

Niekoľko organizačných vecí

Budúci týždeň, 6.11. a 7.11., budú **praktické cvičenia v terminálkach M217 a M218** (v rovnakých časoch ako obvykle)

Neskôr budú cvičenia stále v akváriách

Grupovanie a agregácia: SQL

Príklad: *dodava(Firma, Výrobok, Cena, Lehota),
objednavky(Klient, Výrobok)*

Najdite klientov spolu s počtom objednaných výrobkov, ktoré dodáva HP. Pozor, HP môže dodávať ten istý výrobok za rôzne ceny pre rôzne dodacie lehoty!

```
SELECT o.Klient, COUNT(DISTINCT o.Výrobok) AS Od_hp  
FROM objednavky o, dodava d  
WHERE o.Výrobok = d.Výrobok AND d.Firma = 'hp'  
GROUP BY o.Klient
```

To DISTINCT je naozaj potrebné (kvôli tej výstrahe)

Grupovanie a agregácia: SQL

```
SELECT o.Klient, COUNT(DISTINCT o.Vyrobok) AS Od_hp  
FROM objednavky o, dodava d  
WHERE o.Vyrobok = d.Vyrobok AND d.Firma = 'hp'  
GROUP BY o.Klient
```

dodava			
Firma	Vyrobok	Cena	Lehota
hp	pc	500	1
hp	pc	400	2

objednavky	
Klient	Vyrobok
heinz	pc

objednavky ▷ dodava					
o.Klient	o.Vyrobok	d.Firma	d.Vyrobok	d.Cena	d.Lehota
heinz	pc	hp	pc	500	1
heinz	pc	hp	pc	400	2

Heinz objednal jeden výrobok od hp, nie dva!

Grupovanie a agregácia: relačná algebra

dodava(Firma, Vyrobok, Cena, Lehota),

objednavky(Klient, Vyrobok)

Najdite klientov spolu s počtom objednaných výrobkov, ktoré dodáva HP. Pozor, HP môže dodávať ten istý výrobok za rôzne ceny pre rôzne dodacie lehoty!

$P_{Klient, Od_hp}(\Gamma_{o.Klient, COUNT(o.Vyrobok)})$

$\Delta \Pi_{o.Klient, o.Vyrobok} \sigma_{d.Firma='hp'}(P_o(objednavky) \bowtie P_d(dodava)))$

V tomto prípade treba použiť operátor Δ , lebo relačná algebra počíta s multimnožinami

Grupovanie a agregácia: relačná algebra

$P_{\text{Klient}, \text{Od_hp}}(\Gamma_{o.\text{Klient}}, \text{COUNT}(o.\text{Vyrobok}))$

$\Delta \Pi_{o.\text{Klient}, o.\text{Vyrobok}} \sigma_{d.\text{Firma} = 'hp'}(P_o(\text{objednavky}) \bowtie P_d(\text{dodava})))$

dodava			
Firma	Vyrobok	Cena	Lehota
hp	pc	500	1
hp	pc	400	2

objednavky	
Klient	Vyrobok
heinz	pc

$\Pi_{o.\text{Klient}, o.\text{Vyrobok}} (\text{objednavky} \bowtie \text{dodava})$	
o.Klient	o.Vyrobok
heinz	pc
heinz	pc

Heinz objednal jeden výrobok od hp, nie dva. Ak z projektovanej relácie odstránime duplikáty, ostane len jeden riadok.

Grupovanie a agregácia: relačný kalkul

- Grupovanie a agregácia sa chápe ako rozšírenie všeobecného kvantifikátora. Agregačný operátor \heartsuit (subtotal) viaže všetky voľné premenné, ktoré NIE SÚ za \heartsuit vymenované; a viaže tiež premenné, ktoré sú agregované niektorou agregačnou funkciou (J. Šturm preferuje symbol \oplus , ale aj \heartsuit sa podobá na všeobecný kvantifikátor). Výsledkom je predikát s novými premennými, ktoré vznikli agregáciou, alebo ostali voľné—tie sa použijú na grupovanie

Pozor, relačný kalkul počíta s množinami (bez duplikátov)!

Grupovanie a agregácia: relačný kalkul

Definícia: Nech p je predikátová formula s voľnými premennými

$\mathbf{G} = \{G_1, \dots, G_p\}$, $\mathbf{A} = \{A_1, \dots, A_q\}$, $\mathbf{B} = \{B_1, \dots, B_r\}$,

ktorá zodpovedá relácii P s atribútmi $\mathbf{G} \cup \mathbf{A} \cup \mathbf{B}$. Množiny \mathbf{G} , \mathbf{A} , \mathbf{B} sú po dvojiciach disjunktné. Nech

$\text{AGG}_i \in \{\text{SUM}, \text{COUNT}, \text{AVG}, \text{MIN}, \text{MAX}\}$, $i = 1, \dots, q$.

Potom

♥ $B_1, \dots, B_r, A'_1 = \text{AGG}_1(A_1), \dots, A'_q = \text{AGG}_q(A_q)$ (p)

je predikátová formula s voľnými premennými

$G_1, \dots, G_p, A'_1, \dots, A'_q$, ktorá zodpovedá relácii

$\Gamma_{G_1, \dots, G_p, A'_1 = \text{AGG}_1(A_1), \dots, A'_q = \text{AGG}_q(A_q)} (\Delta (p))$

Grupovacie premenné sa v relačnom kalkule píšu „naopak“!

Grupovanie a agregácia: relačný kalkul

Príklad: $dodava(Firma, Výrobok, Cena, Lehota)$,
 $objednavky(Klient, Výrobok)$

Najdite klientov spolu s počtom objednaných výrobkov, ktoré dodáva HP. Pozor, HP môže dodávať ten istý výrobok za rôzne ceny pre rôzne dodacie lehoty!

Toto je nesprávne (kvôli tej výstrahe):

{[K, P]: \heartsuit C, L, P = count(V)
(objednavky(K, V) \wedge dodava(hp, V, C, L))}

Správne je:

{[K, P]: \heartsuit P = count(V)
(\exists C \exists L objednavky(K, V) \wedge dodava(hp, V, C, L))}

Grupovanie a agregácia: Datalog

Špeciálny predikát **subtotal**, ktorého **prvým argumentom je n-árny predikát p** s navzájom disjunktnými množinami argumentov $G_1, \dots, G_p, A_1, \dots, A_q, B_1, \dots, B_r$. Premenné B_1, \dots, B_r mimo subtotalu neexistujú

Druhým argumentom subtotal je zoznam $[G_1, \dots, G_p]$.

Tretím argumentom subtotal je zoznam

$[A'_1 = \text{agg}_1(A_1), \dots, A'_q = \text{agg}_q(A_q)]$, kde agg_i je niektorá z agregačných funkcií.

Výsledkom je predikát, ktorý zodpovedá výrazu

♥ $B_1, \dots, B_r, A'_1 = \text{AGG}_1(A_1), A'_q = \text{AGG}_1(A_q) \ p(\dots)$

Pozor, Datalog počíta s **množinami** (bez duplikátov)!

Grupovanie a agregácia: Datalog

Príklad: *dodava(Firma, Výrobok, Cena, Lehota),
objednavky(Klient, Výrobok)*

Najdite klientov spolu s počtom objednaných výrobkov, ktoré dodáva HP. Pozor, HP môže dodávať ten istý výrobok za rôzne ceny pre rôzne dodacie lehoty!

```
objednava_od_hp(K, V) ←  
    objednavky(K, V), dodava(hp, V, _, _).  
  
answer(K, P) ←  
    subtotal(objednava_od_hp(K, V), [K], [P = count(V)]).
```

Pozor, Datalog počíta s **množinami** (bez duplikátov)!
Po projekcii duplikáty zaniknú.

Ešte príklad na grupovanie a agregáciu: Datalog

Príklad:

*capuje(Krcma, Alkohol), lubi(Pijan, Alkohol),
navstivil(Idn, Pijan, Krcma), vypil(Idn, Alkohol, Mnozstvo)*

Nájdite trojice [Krcma, Alkohol, Mnozstvo], ktoré hovoria koľko ktorého alkoholu sa vypilo v jednotlivých krčmách (nezaujímajú nás alkoholy, ktoré sa v jednotlivých krčmách nikdy nepili)

vypilo_sa_pri_navsteve(K, A, M, I) :-
 navstivil(I, _, K), vypil(I, A, M).

answer(K, A, M) :-
 subtotal(vypilo_sa_pri_navsteve(K, A, X, I), [K, A],
 [M = sum(X)]).

Všimnite si atribút Idn, ten **je** dôležitý (lebo sa počíta s množinami)

Ešte príklad na grupovanie a agregáciu: relačný kalkul

Príklad:

capuje(Krcma, Alkohol), lubi(Pijan, Alkohol),

navstivil(Idn, Pijan, Krcma), vypil(Idn, Alkohol, Mnozstvo)

Najdite trojice [Krcma, Alkohol, Mnozstvo], ktoré hovoria koľko ktorého alkoholu sa vypilo v jednotlivých krčmách (nezaujímajú nás alkoholy, ktoré sa v jednotlivých krčmách nikdy nepili)

Relačný kalkul: {[K, A, M]: $\exists A \ capuje(K, A) \wedge$

♥ I, M = sum(X) ($\exists P (navstivil(I, P, K) \wedge vypil(I, A, X)))}$

Všimnite si atribút Idn, ten je dôležitý (lebo sa počíta s množinami)

Ešte príklad na grupovanie a agregáciu: SQL

Príklad:

capuje(Krcma, Alkohol), lubi(Pijan, Alkohol),

navstivil(Idn, Pijan, Krcma), vypil(Idn, Alkohol, Mnozstvo)

Nájdite trojice [Krcma, Alkohol, Mnozstvo], ktoré hovoria koľko ktorého alkoholu sa vypilo v jednotlivých krčmách (nezaujímajú nás alkoholy, ktoré sa v jednotlivých krčmách nikdy nepili)

WITH vypilo_sa_pri_navsteve AS

SELECT n.Krcma, v.Alkohol, v.Mnozstvo, ~~n.Idn~~

FROM navstivil n, vypil v

WHERE n.Idn = v.Idn,

Idn je možné vynechať,
lebo SQL počíta s
multimnožinami

SELECT vspn.Krcma, vspn.Alkohol, SUM(vspn.Mnozstvo)

FROM vypilo_sa_pri_navsteve vspn

GROUP BY vspn.Krcma, vspn.Alkohol

Ešte príklad na grupovanie a agregáciu: relačná algebra

Príklad:

capuje(Krcma, Alkohol), lubi(Pijan, Alkohol),

navstivil(Idn, Pijan, Krcma), vypil(Idn, Alkohol, Mnozstvo)

Nájdite trojice [Krcma, Alkohol, Mnozstvo], ktoré hovoria koľko ktorého alkoholu sa vypilo v jednotlivých krčmách (nezaujímajú nás alkoholy, ktoré sa v jednotlivých krčmách nikdy nepili)

$P_{Krcma, Alkohol, Mnozstvo} \Gamma_{vspan.Krcma, vspan.Alkohol, SUM(vspan.Mnozstvo)}$

$P_{vspan} (\prod_{n.Krcma, v.Alkohol, v.Mnozstvo, n.Idn} (P_n \text{ navstivil} \bowtie P_v \text{ vypil}))$

Idn je možné vyniechať,
lebo relačná algebra
počíta s multimnožinami—
ale môžeme ho tam dať

Rekapitulácia dotazovacích jazykov

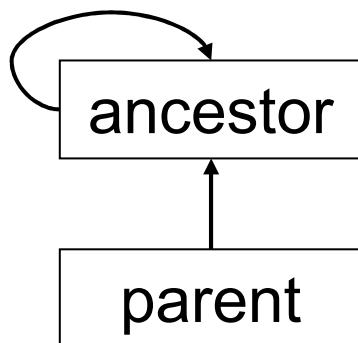
- Dotazy sa dajú vyjadriť v 4 jazykoch, ktoré sú (pre bezpečné dotazy) rovnako silné, t.j. je možné napísať kompilátor z ľubovoľného jazyka do ľubovoľného
 - **Relačný kalkul:** (bezpečné) predikátové formuly s reláciami
 - **Datalog:** bezpečné predikátové formuly v “normálnej” forme
 - „Kanonický“ **SQL:**
SELECT... FROM... WHERE (+NOT EXISTS)...
GROUP BY... HAVING (+NOT EXISTS)...
UNION...
- **Relačná algebra:** výrazy nad reláciami, s operátormi zjednotenia, prieniku, rozdielu, kartézskeho súčinu, selekcie, projekcie, premenovania a agregácie
- **Chýba ešte rekurzia (resp. iterácia)**

Rekurzia: Datalog

Príklad (predkovia):

$\text{ancestor}(X, A) :- \text{parent}(X, A).$

$\text{ancestor}(X, A) :- \text{parent}(X, P), \text{ancestor}(P, A).$



Rekurzívne programy obsahujú cyklus v grafe závislostí predikátov (hrana z uzla p do uzla q, ak existuje pravidlo s hlavou q, ktorého telo obsahuje p)

Komplikovanejšia rekurzia (počíta to isté čo predošlý program):

$\text{ancestor}(X, A) :- \text{parent}(X, A).$

$\text{ancestor}(X, A) :- \text{ancestor}(X, Y), \text{ancestor}(Y, A).$

Rekurzia: relačná algebra

Operátor minimal fixpoint (tranzitívny uzáver): Φ

- $\Phi(<\text{výraz}>)$ iteruje výraz, kým sa relácie použité vo výraze menia (t.j. kým sa mení obsah aspoň 1 relácie výrazu)

Príklad:

`ancestor := parent(X, A)`

`ancestor := ancestor ∪ ΠX, A(ancestor ⋙ parent)`

`ancestor := ancestor ∪ ΠX, A(ancestor ⋙ parent)`

...

sa zapíše ako

ancestor := parent

$\Phi(\text{ancestor} := \text{ancestor} \cup \Pi_{X, A}(\text{ancestor} \bowtie \text{parent}))$

alebo

ancestor := {}

$\Phi(\text{ancestor} := \text{parent} \cup \Pi_{X, A}(\text{ancestor} \bowtie \text{parent}))$

Rekurzia: relačný kalkul

V relačnom kalkule netreba zavádzať špeciálny operátor na vyjadrenie rekurzie. Rekurzia sa vyjadrí prirodzeným spôsobom

Príklad (predkovia Johna):

`ancestor(X, A) :- parent(X, A).`

`ancestor(X, A) :- parent(X, P), ancestor(P, A).`

`answer(A) :- ancestor(john, A).`

{ $A : \text{ancestor}(\text{john}, A) \wedge$
 $(\forall X \forall A (\text{parent}(X, A) \Rightarrow \text{ancestor}(X, A))) \wedge$
 $(\forall X \forall A \forall P ((\text{parent}(X, A) \wedge \text{ancestor}(P, A)) \Rightarrow \text{ancestor}(X, A)))$ }

V SQL-99 bola pridaná rekurzia: rekurzívne VIEW, resp. rekurzívne WITH.

Bohužiaľ, existujúce systémy väčšinou neimplementujú WITH RECURSIVE. Spoliehajú na cykly resp. rekurziu v externých jazykoch mimo SQL (napr. PHP). Existujúce systémy, ktoré WITH RECURSIVE implementujú (Oracle, Firebird, DB2), neimplementujú plnú sémantiku (implementujú len rekurziu cez 1 reláciu). Do PostgreSQL bol WITH RECURSIVE dodaný v 2011.

Príklad:

```
WITH RECURSIVE ancestor AS (
    SELECT p.X FROM parent p
    UNION
    SELECT p.X FROM parent p, ancestor a WHERE p.Y=a.X
)
SELECT a.X FROM ancestor a
```

Výpočet pevného bodu: iterácia

- Výpočet dotazov (vo všeobecnosti rekurzívnych) v Datalogu, algebre, kalkule či SQL zodpovedá **riešeniu systému rovníc** (vo všeobecnosti rekurzívnych)
- **Metódou riešenia je iterácia** (napríklad naivná či seminaivná)

Pozor: dotaz môže obsahovať viacero fixpoint operátorov. V takom prípade treba iterovať „všetky fixpoint operátory súčasne“:

```
do
{
    iteration_step( $\Phi_1$ );
    iteration_step( $\Phi_2$ );
    ...
    iteration_step( $\Phi_n$ );
} while a relation has changed; /* anywhere inside this loop */
```

Výpočet dotazov v Datalogu

Každému (IDB) predikátu s_i v programe prirad' reláciu S_i

Algoritmus výpočtu výsledku dotazu (naivná evaluácia):

1. Pre všetky i polož $S_i := \emptyset$

2. do

{

aplikuj každé pravidlo programu práve raz (t.j. pre každé pravidlo vypočítaj join resp. antijoin s použitím aktuálneho obsahu relácií v tele pravidla), výsledok prirad' relácii S_i prislúchajúcej hlave pravidla

} while (niektorá relácia S_i sa zmenila)

3. Aplikuj výslednú selekciu a projekciu danú dotazom

V kroku 2 sa počíta "výsledok celého programu" a **až v kroku 3 sa tento výsledok uložený v reláciách S_i zredukuje na výsledok daného dotazu** (ktorý sa týka len jednej z týchto relácií). Toto môže byť neefektívne, ale funguje aj pre rekurzívne programy

Problém: krok 2 v naívnej evaluácii nemusí vždy skončiť.

Príklad:

man(barber). man(mayor).

shaves(barber, X) \leftarrow man(X), not shaves(X, X).

Počítajme naivnou evaluáciou (relácia man je EDB, tá sa nemení). Predikátu shaves priradíme reláciu Shaves.

Iterujeme pravidlo pre Shaves (jediné):

Shaves = \emptyset

Shaves = {[barber, barber], [barber, mayor]}

Shaves = {[barber, mayor]}

Shaves = {[barber, barber], [barber, mayor]}

Shaves = {[barber, mayor]}

...

Problém: krok 2 v naívnej evaluácii nemusí vždy skončiť

shaves(barber, mayor) síce platí vo všetkých modeloch, ale dotaz
?- shaves(barber, mayor) neskončí, ak sa počíta (semi-)
naivnou evaluáciou

**Naivná evaluácia nie je jediný možný spôsob výpočtu
dotazov a nie je jednoduché vybrať ten „správny“ spôsob**

**Naštastie, pre istú podriedu dotazov je garantované, že
výpočet naivnou evaluáciou skončí**

Výpočet pevného bodu: Tarskeho veta

Tarskeho veta o pevnom bode pre úplné zväzy: ak f je neklesajúce zobrazenie nad úplným zväzom (t.j. $x \leq y \Rightarrow f(x) \leq f(y)$), tak potom f má aspoň jeden pevný bod p , pre ktorý platí $f(p) = p$

Z Tarskeho vety o pevnom bode vyplýva, že **iteratívny výpočet konverguje k pevnému bodu pre “obvyklé” (aj rekurzívne) dotazy**, keďže kartézsky súčin, join, zjednotenie, projekcia a premenovanie sú „neklesajúce“ operácie (počet riadkov v reláciách neklesá) a extenzionálne relácie sú konečné. K „neklesajúcim“ operáciám patrí aj *stratifikovaná negácia*

Problém je s nestratifikovanou negáciou a agregáciou

Stratifikovaná negácia

Definícia stratifikovaného programu: Ak graf závislostí predikátov neobsahuje cyklus s negovaným predikátom, tak je program stratifikovaný

Podmienky stratifikácie:

- Každému predikátu priradíme celé číslo s , tzv. stratum (vrstva)
- Ak p je definovaný pomocou q , tak musí platiť $S(p) \geq S(q)$
- Ak p je definovaný pomocou $\neg q$, tak musí platiť $S(p) > S(q)$

Platí: ak je program stratifikovaný a obsahuje n predikátov, tak tým predikátom možno priradiť strata od 1 po n

Algoritmus stratifikácie:

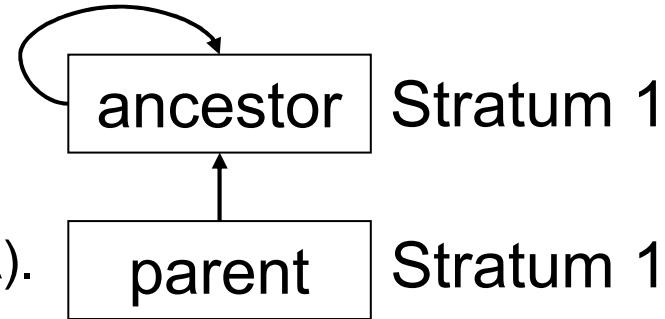
1. Všetkým predikátom prirad' stratum 1
2. Ak niektoré pravidlo porušuje podmienky stratifikácie, tak zvýš stratum hlavy niektorého predikátu na najmenšie celé číslo, ktoré tú podmienku splňa
3. Ak je niektoré stratum väčšie ako počet predikátov, tak program sa nedá stratifikovať

Stratifikovaná negácia

Príklady:

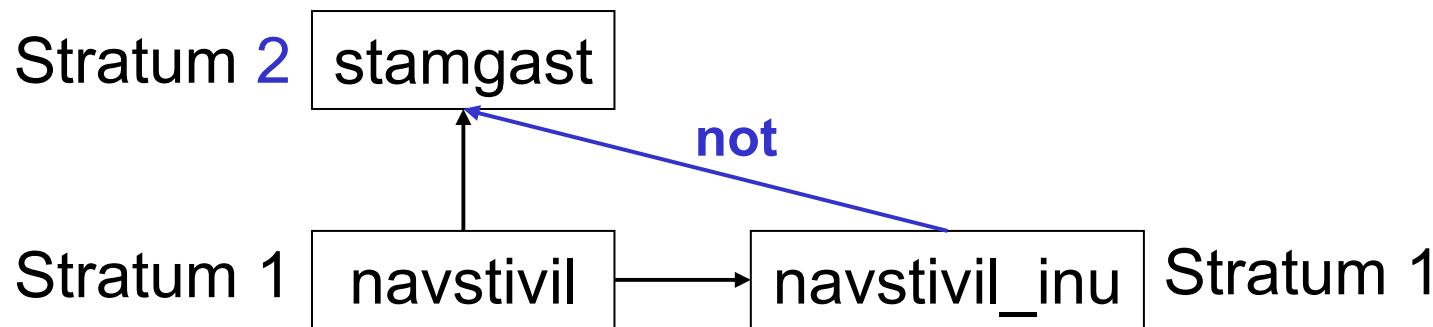
`ancestor(X, A) :- parent(X, A).`

`ancestor(X, A) :- parent(X, P), ancestor(P, A).`



`stamgast(P) :- navstivil(_, P, K), not navstivil_inu(P, K).`

`navstivil_inu(P, K) :- navstivil(_, P, K), navstivil(_, P, K2), K ≠ K2.`



`shaves(barber, X) ← man(X), not shaves(X, X).`



Nedá sa stratifikovať'

Výpočet stratifikovaných dotazov v Datalogu

Každému (IDB) predikátu s_i v programe prirad' reláciu S_i

Metóda výpočtu výsledku dotazu (naivná evaluácia):

Začni s prázdnymi reláciami S_i .

Vypočítaj relácie S_i priradené predikátom v stratum 1.

Vypočítaj relácie S_i priradené predikátom v stratum 2.

Vypočítaj relácie S_i priradené predikátom v stratum 3.

...

Poradie výpočtu S_i v rámci jedného strata:

- Pri výpočte každého strata preferuj najsôkôr predikáty, ktoré nezávisia od iných IDB predikátov. Relácie pre tieto predikáty stačí vypočítať len raz.
- V prípade rekurzívnej definície iteruj výpočet cez všetky predikáty v cykle grafu závislostí, až kým sa prislúchajúce relácie S_i prestanú meniť. Tieto relácie sa potom už nikdy nebudú meniť.

Výpočtová sila dotazovacích jazykov s rekurziou

Relačný kalkul, Datalog, relačná algebra a SQL **s rekurziou** sú (pri obmedzení na bezpečné dotazy) jazyky s rovnakou vyjadrovacou silou. Navyše sú to **úplné programovacie jazyky**, t.j. sú **rovnako expresívne ako Turingov stroj**

Dôkaz:

Prvé tvrdenie sa dokáže konštrukciou kompilátorov napr.

SQL → rel.algebra → rel.kalkul → Datalog → SQL

Druhá časť dôkazu (ekvivalencia s Turingovým strojom) sa urobí simuláciou Turingovho stroja v niektorom z tých jazykov, napr. v Datalogu. Idea: na kódovanie Turingovho stroja sa použijú relácie

paska(adresa, symbol),

program(stary_stav, stary_symbol, novy_stav, novy_symbol, pohyb),

pociatocny_stav(stav, adresa),

akceptacne_stavy(stav).

Iteráciou sa budú vyrábať dosažiteľné konfigurácie. Všimnite si, že v prípade nedeterminizmu sa simulujú všetky vetvy výpočtu “paralelne”.

Téma na ročníkový projekt

Výpočet pevného bodu: príklad (vrstovníci)

Relácia $p(\text{Child}, \text{Parent})$

Napr. $p = \{[\text{Claudia}, \text{Adam}], [\text{Dieter}, \text{Adam}], [\text{Dieter}, \text{Britta}], [\text{Erik}, \text{Britta}], [\text{Fridda}, \text{Claudia}], [\text{Gisela}, \text{Claudia}], [\text{Heinz}, \text{Dieter}], [\text{Ingo}, \text{Dieter}], [\text{Ingo}, \text{Erik}], [\text{Fridda}, \text{Erik}], [\text{Jürgen}, \text{Fridda}], [\text{Jürgen}, \text{Heinz}], [\text{Klaus}, \text{Gisela}], [\text{Klaus}, \text{Ingo}]\}$

Dvojice ľudí X a Y, ktorí sú v rovnakej generácii (**same generation**): bud' X a Y majú spolu dieťa, alebo ich rodičia sú v rovnakej generácii:

sg(X, Y) :- p(C, X), p(C, Y).

sg(X, Y) :- p(X, PX), p(Y, PY), sg(PX, PY).

Treba nájsť všetky dvojice X a Y, pre ktoré platí $sg(X, Y)$

Výpočet pevného bodu: iterácia (naivná evaluácia)

$sg(X, Y) :- p(C, X), p(C, Y).$

$sg(X, Y) :- p(X, PX), p(Y, PY), sg(PX, PY).$

Relačná algebra, fixpoint operátor (premenovania sú vyniechané):

$sg(X, Y) := \{\}$

$\Phi (sg(X, Y)) := \Delta (\Pi_{X,Y}(p(C, X) \bowtie p(C, Y)) \cup \Pi_{X,Y}(p(X, PX) \bowtie sg(PX, PY) \bowtie p(Y, PY)))$

Relačná algebra, naivná evaluácia:

$sg(X, Y) := \{\};$

do /* Tento cyklus reprezentuje operátor Φ */

{

$old_sg(X, Y) := sg(X, Y);$

$sg(X, Y) := \Delta (\Pi_{X,Y}(p(C, X) \bowtie p(C, Y)) \cup \Pi_{X,Y}(p(X, PX) \bowtie sg(PX, PY) \bowtie p(Y, PY)));$

}

$while old_sg(X, Y) <> sg(X, Y);$

Výpočet pevného bodu: iterácia (naivná evaluácia)

```
sg(X, Y) :- p(C, X), p(C, Y).  
sg(X, Y) :- p(X, PX), p(Y, PY), sg(PX, PY).
```

SQL:

```
WITH RECURSIVE sg AS (  
SELECT p1.Parent AS X, p2.Parent AS Y  
FROM p p1, p p2  
WHERE p1.Child = p2.Child  
UNION  
SELECT p1.Child AS X, P2.Child AS Y  
FROM p p1, p p2, sg s  
WHERE p1.Parent = s.X AND p2.Parent = s.Y)
```

```
SELECT DISTINCT s.X, s.Y  
FROM sg s
```

Výpočet pevného bodu: iterácia (naivná evaluácia)

```
sg(X, Y) :- p(C, X), p(C, Y).  
sg(X, Y) :- p(X, PX), p(Y, PY), sg(PX, PY).
```

SQL, naivná evaluácia:

```
sg := {};  
do /* Tento cyklus reprezentuje operátor  $\Phi$  */  
{  
    old_sg := sg;  
    sg :=  
        (SELECT p1.Parent AS X, p2.Parent AS Y  
         FROM p p1, p p2  
         WHERE p1.Child = p2.Child  
         UNION  
         SELECT p1.Child AS X, P2.Child AS Y  
         FROM p p1, p p2, sg s  
         WHERE p1.Parent = s.X AND p2.Parent = s.Y);  
}  
while (old_sg <> sg);
```

Výpočet pevného bodu: iterácia (naivná evaluácia)

$sg(X, Y) :- p(C, X), p(C, Y).$

$sg(X, Y) :- p(X, PX), p(Y, PY), sg(PX, PY).$

$p = \{ [Claudia, Adam], [Dieter, Adam], [Dieter, Britta], [Erik, Britta],$
 $[Fridda, Claudia], [Gisela, Claudia], [Heinz, Dieter], [Ingo, Dieter], [Ingo, Erik],$
 $[Fridda, Erik], [Jürgen, Fridda], [Jürgen, Heinz], [Klaus, Gisela], [Klaus, Ingo] \}$

i	sg	sg(X, Y)
1	19	AA, AB, BA, BB, CC, CE, DD, DE, EC, ED, EE, FF, FH, GG, GI, HF, HH, IG, II
2	29	AA, AB, BA, BB, CC, CE, DD, DE, EC, ED, EE, FF, FH, GG, GI, HF, HH, IG, II, CD, DC, FG, FI, GF, HI, IF, IH, JJ, KK
3	33	AA, AB, BA, BB, CC, CE, DD, DE, EC, ED, EE, FF, FH, GG, GI, HF, HH, IG, II, CD, DC, FG, FI, GF, HI, IF, IH, JJ, KK, GH, HG, JK, KJ
4	33	AA, AB, BA, BB, CC, CE, DD, DE, EC, ED, EE, FF, FH, GG, GI, HF, HH, IG, II, CD, DC, FG, FI, GF, GH, HG, HI, IF, IH, JJ, JK, KJ, KK

Výpočet pevného bodu: iterácia (seminaivná evaluácia)

Optimalizácia: nepočítať opakovane všetky dvojice v každom kroku. Stačí počítať (a pridávať) **len nové dvojice**

Naivná evaluácia: $19+29+33+33=114$ vypočítaných dvojíc

Seminaivná evaluácia: $19+10+4+0=33$ vypočítaných dvojíc

i	sgl	sg(X, Y)	\Delta sgl	\Delta sg(X, Y)
1	19	AA, AB, BA, BB, CC, CE, DD, DE, EC, ED, EE, FF, FH, GG, GI, HF, HH, IG, II	19	AA, AB, BA, BB, CC, CE, DD, DE, EC, ED, EE, FF, FH, GG, GI, HF, HH, IG, II
2	29	AA, AB, BA, BB, CC, CE, DD, DE, EC, ED, EE, FF, FH, GG, GI, HF, HH, IG, II, CD, DC, FG, FI, GF, HI, IF, IH, JJ, KK	10	CD, DC, FG, FI, GF, HI, IF, IH, JJ, KK
3	33	AA, AB, BA, BB, CC, CE, DD, DE, EC, ED, EE, FF, FH, GG, GI, HF, HH, IG, II, CD, DC, FG, FI, GF, HI, IF, IH, JJ, KK, GH, HG, JK, KJ	4	GH, HG, JK, KJ
4	33	AA, AB, BA, BB, CC, CE, DD, DE, EC, ED, EE, FF, FH, GG, GI, HF, HH, IG, II, CD, DC, FG, FI, GF, HI, IF, IH, JJ, KK, GH, HG, JK, KJ	0	

Výpočet pevného bodu: iterácia (seminaivná evaluácia)

```
sg(X, Y) :- p(C, X), p(C, Y).  
sg(X, Y) :- p(X, PX), p(Y, PY), sg(PX, PY).
```

Relačná algebra, seminaivná evaluácia (diferenčná schéma):

```
sg(X, Y) := ∅;  
Δsg(X, Y) := ∅;  
do /* Tento cyklus reprezentuje operátor Φ */  
{  
    Δsg(X, Y) := (ΠX,Y(p(C, X) ⋙ p(C, Y)) ∪  
                  ΠX,Y(p(X, PX) ⋙ p(Y, PY) ⋙ Δsg(PX, PY)))  
    - sg(X, Y);  
    sg(X, Y) := sg(X, Y) ∪ Δsg(X, Y);  
}  
while (Δsg(X, Y) <> ∅);
```

Výpočet pevného bodu: iterácia (seminaivná evaluácia)

```
sg(X, Y) :- p(C, X), p(C, Y).  
sg(X, Y) :- p(X, PX), p(Y, PY), sg(PX, PY).
```

Seminaivná evaluácia, optimalizovaná verzia (prvé pravidlo prispeje do Δsg iba raz):

```
sg(X, Y) := ∅;  
 $\Delta sg(X, Y) := \Pi_{X,Y}(p(C, X) \bowtie p(C, Y));$   
do /* Tento cyklus reprezentuje operátor  $\Phi$  */  
{  
     $\Delta sg(X, Y) := (\Pi_{X,Y}(p(X, PX) \bowtie p(Y, PY) \bowtie \Delta sg(PX, PY))$   
    - sg(X, Y);  
    sg(X, Y) := sg(X, Y) ∪  $\Delta sg(X, Y);$   
}  
while ( $\Delta sg(X, Y) \neq \emptyset$ );
```

Výpočet pevného bodu: iterácia (seminárná evaluácia)

SQL, seminárná evaluácia:

$sg := \emptyset;$

$\Delta sg := \emptyset;$

do /* Tento cyklus reprezentuje operátor Φ */

$\Delta sg :=$

SELECT newsg.X, newsg.Y

FROM

(SELECT p1.Parent AS X, p2.Parent AS Y

FROM p p1, p p2 J

WHERE p1.Child = p2.Child

UNION

SELECT p1.Child AS X, P2.Child AS Y

FROM p p1, p p2, sg s

WHERE p1.Parent = s.X AND p2.Parent = s.Y) newsg

WHERE [s.X, s.Y] NOT IN

(SELECT oldsg.X, oldsg.Y FROM sg oldsg);

$sg(X, Y) := sg(X, Y) \cup \Delta sg(X, Y);$

while ($\Delta sg(X, Y) <> \emptyset$);

Výpočet pevného bodu: magická transformácia

Ani seminaivná evaluácia nie je vždy výhodná. Napríklad pre dotaz

? sg(X, adam)

bude seminaivná evaluácia počítať vrstovníkov nielen pre Adama, ale aj pre všetky ostatné osoby v databáze (až na záver sa urobí finálna selekcia pre Adama) . Tomuto zbytočnému výpočtu sa vyhýba **magická transformácia** (v tomto kurze predstavená len na ilustračných príkladoch). Idea: presunúť selekciu čím hlbšie do rekurzie **transformáciou pravidiel programu pre daný dotaz**. Transformovaný program sa počíta seminaivnou iteráciou

Nasledujúce obrázky používajú trošku zmenenú definíciu sg (S. Boettcher):

sg(X, Y) :- flat(X, Y).

sg(X, Y) :- up(X, U), sg(U, V), down(V, Y).

up(X, U) :- down(U, X).

Výpočet pevného bodu: magická transformácia

? sg(X, adam)

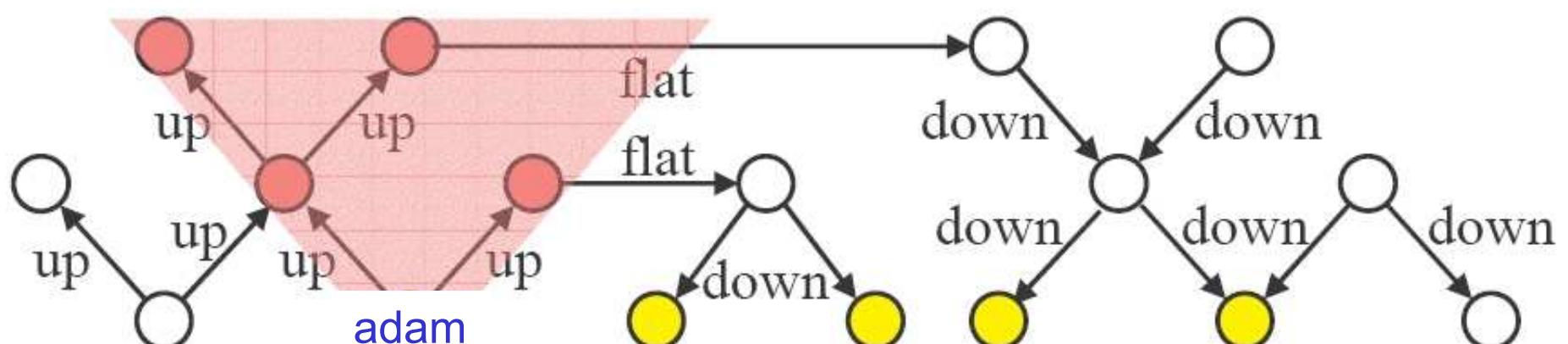
magic(adam).

magic(U) :- magic(X), up(X, U).

sg(X, Y) :- magic(X), flat(X, Y).

sg(X, Y) :- magic(X), up(X, U), sg(U, V), down(V, Y).

up(X, U) :- down(U, X).



(S. Boettcher)

Výpočet pevného bodu: magická transformácia

? sg(adam, Y) /* tu je fixovaný prvý argument */

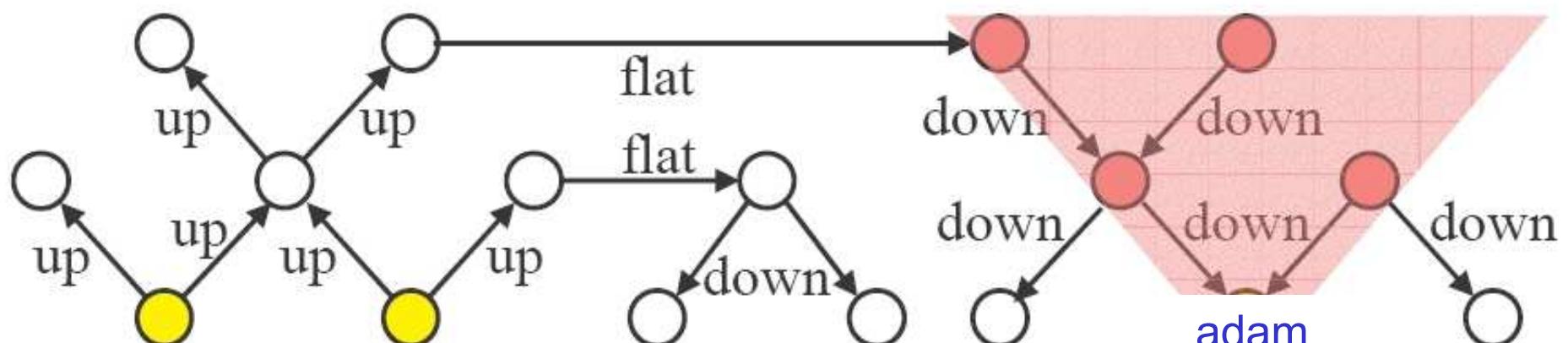
magic(adam).

magic(V) :- magic(Y), down(V, Y).

sg(X, Y) :- magic(Y), flat(X, Y).

sg(X, Y) :- magic(Y), up(X, U), sg(U, V), down(V, Y).

up(X, U) :- down(U, X).



(S. Boettcher)

Ešte príklad: emp(Id, Name, Mgr), Datalog

Najdite všetkých nadriadených zamestnanca BLAKE (priamych aj nepriamych).

Datalog:

```
answer(N) ← emp(Id, N, _), super_blake(Id).  
super_blake(Id) ← emp(_, blake, Id).  
super_blake(Id) ← emp(Sid, _, Id), super_blake(Sid).
```

Lenže obyčajne sa snažíme programovať všeobecné predikáty, (super_blake je príliš jednoúčelový predikát). Nasledujúca verzia je všeobecnejšia:

```
super(Id, Sid) ← emp(Id, _, Sid).  
super(Id, Sid) ← emp(Id, _, Sid2), super(Sid2, Sid).  
answer(N) ← emp(Sid, Sname, _), emp(Id, blake, _), super(Id, Sid).
```

Pred výpočtom dotazu „? answer(N)“ optimalizátor **automaticky** transformuje magickou transformáciou ten všeobecný program na program, ktorý je veľmi podobný tomu prvému programu!

Ešte príklad: emp(Id, Name, Mgr), relačná algebra

Najdite všetkých nadriadených zamestnanca BLAKE (priamych aj nepriamych).

$\text{super}(\text{Id}, \text{Sid}) \leftarrow \text{emp}(\text{Id}, _, \text{Sid}).$

$\text{super}(\text{Id}, \text{Sid}) \leftarrow \text{emp}(\text{Id}, _, \text{Sid2}), \text{super}(\text{Sid2}, \text{Sid}).$

$\text{answer}(\text{Sname}) \leftarrow \text{emp}(\text{Sid}, \text{Sname}, _), \text{emp}(\text{Id}, \text{blake}, _), \text{super}(\text{Id}, \text{Sid}).$

Relačná algebra:

$\Phi (\text{super} := \prod_{\text{Id}, \text{Mgr}} \text{emp} \cup \prod_{\text{emp.Id}, \text{super.Mgr}} \text{emp} \bowtie_{\text{emp.Mgr} = \text{super.Id}} \text{super})$

$\text{answer} := \sigma_{\text{Name}=\text{'blake'}} (P_{e2}(\text{emp})) \bowtie_{e2.\text{Id} = s.\text{Id}} P_s(\text{super}) \bowtie_{e1.\text{Id} = s.\text{Mgr}} P_{e1}(\text{emp})$

Ešte príklad: emp(Id, Name, Mgr), relačný kalkul

Nájdite všetkých nadriadených zamestnanca BLAKE (priamych aj nepriamych).

$\text{super}(\text{Id}, \text{Sid}) \leftarrow \text{emp}(\text{Id}, _, \text{Sid}).$

$\text{super}(\text{Id}, \text{Sid}) \leftarrow \text{emp}(\text{Id}, _, \text{Sid2}), \text{super}(\text{Sid2}, \text{Sid}).$

$\text{answer}(\text{Sname}) \leftarrow \text{emp}(\text{Sid}, \text{Sname}, _), \text{emp}(\text{Id}, \text{blake}, _), \text{super}(\text{Id}, \text{Sid}).$

Relačný kalkul:

{ Sname :

$(\exists \text{Sid} \exists \text{X1} \exists \text{X2} \exists \text{Id} \text{ emp}(\text{Sid}, \text{Sname}, \text{X1}) \wedge \text{emp}(\text{Id}, \text{blake}, \text{X2}) \wedge \text{super}(\text{Id}, \text{Sid})) \wedge$
 $(\forall \text{Id} \forall \text{Sid} \forall \text{N} (\text{emp}(\text{Id}, \text{N}, \text{Sid}) \Rightarrow \text{super}(\text{Id}, \text{Sid}))) \wedge$
 $(\forall \text{Id} \forall \text{Sid} \forall \text{Sid2} \forall \text{N} ((\text{emp}(\text{Id}, \text{N}, \text{Sid2}) \wedge \text{super}(\text{Sid2}, \text{Sid})) \Rightarrow \text{super}(\text{Id}, \text{Sid}))) \}$

Ešte príklad: emp(Id, Name, Mgr), WITH RECURSIVE

Najdite všetkých nadriadených zamestnanca BLAKE (priamych aj nepriamych).

```
super(Id, Sid) ← emp(Id, _, Sid).
super(Id, Sid) ← emp(Id, _, Sid2), super(Sid2, Sid).
answer(Sname) ← emp(Sid, Sname, _), emp(Id, blake, _), super(Id, Sid).
```

SQL s WITH RECURSIVE:

with recursive super as

(select e.Id, e.Mgr

from emp e)

union

(select e.Id, s.Mgr

from emp e, super s

where e.Mgr = s.Id),

select e1.Name

from emp e1, emp e2, super s

where e1.Id = s.Mgr and e2.Id = s.Id and e2.Name = 'blake'

Ešte príklad: emp(Id, Name, Mgr), SQL bez WITH

Najdite všetkých nadriadených zamestnanca BLAKE (priamych aj nepriamych).

```
super(Id, Sid) ← emp(Id, _, Sid).
super(Id, Sid) ← emp(Id, _, Sid2), super(Sid2, Sid).
answer(Sname) ← emp(Sid, Sname, _), emp(Id, blake, _), super(Id, Sid).
```

SQL bez WITH:

```
create temporary table super0 as
select e.Id, e.Mgr
from emp e;
```

```
create temporary table super1 as
((select e.Id, e.Mgr
from super0)
union
(select e.Id, s.Mgr
from emp e, super0 s
where e.Mgr = s.Id));
```

Ešte príklad: emp(Id, Name, Mgr), SQL bez WITH, pokr.

Najdite všetkých nadriadených zamestnanca BLAKE (priamych aj nepriamych).

```
super(Id, Sid) ← emp(Id, _, Sid).
super(Id, Sid) ← emp(Id, _, Sid2), super(Sid2, Sid).
answer(Sname) ← emp(Sid, Sname, _), emp(Id, blake, _), super(Id, Sid).
```

SQL bez WITH, pokračovanie:

...

```
create temporary table super10 as
((select e.Id, e.Mgr
from super9)
union
(select e.Id, s.Mgr
from emp e, super9 s
where e.Mgr = s.Id));
```

Ešte príklad: emp(Id, Name, Mgr), SQL bez WITH, pokr.

Najdite všetkých nadriadených zamestnanca BLAKE (priamych aj nepriamych).

```
super(Id, Sid) ← emp(Id, _, Sid).
super(Id, Sid) ← emp(Id, _, Sid2), super(Sid2, Sid).
answer(Sname) ← emp(Sid, Sname, _), emp(Id, blake, _), super(Id, Sid).
```

SQL bez WITH, finálny SELECT (pri hĺbke rekurzie 10):

```
select e1.Name
from emp e1, emp e2, super10 s
where e1.Id = s.Mgr and e2.Name = 'blake' and e2.Id = s.Id;
```

Pre ľubovoľnú fixnú hĺbku rekurzie je možné rekurziu simulovať aj bez použitia WITH RECURSIVE. Avšak ak hĺbka rekurzie nie je dopredu známa, tak takáto (ani žiadna iná) simulácia sa urobiť nedá

Vyskúšajte na databáze emp (stačí hĺbka rekurzie 3, netreba 10)

Ešte príklad: emp(Id, Name, Mgr), SQL bez WITH, Δ -opt

Najdite všetkých nadriadených zamestnanca BLAKE (priamych aj nepriamych).

```
super(Id, Sid) ← emp(Id, _, Sid).
super(Id, Sid) ← emp(Id, _, Sid2), super(Sid2, Sid).
answer(Sname) ← emp(Sid, Sname, _), emp(Id, blake, _), super(Id, Sid).
```

SQL bez WITH, seminaivná evaluácia (efektívnejšia ako predošlá naivná):

```
create temporary table super1 as
((select e.Id, e.Mgr
from super0)
union
(select e.Id, s.Mgr
from emp e, super0 s
where e.Mgr = s.Id and not exists (
    select *
    from super0 s0
    where s0.Id = e.Id and s0.Mgr = s.Mgr)));
... podobne pre super2 až super10
```

Úvod do databázových systémov

[http://www.dcs.fmph.uniba.sk/~plachetk
/TEACHING/DB2013](http://www.dcs.fmph.uniba.sk/~plachetk/TEACHING/DB2013)

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2013–2014

Cieľ a metodológia navrhovania databáz

- Návrhu databázy predchádza **plánovanie**, t.j. zber požiadaviek na aplikáciu. Metodológie špecifikácie systémov: SAD (Structured Analysis and Design), DFD (Data-Flow Diagrams), UML (Unified Modeling Language)
- Cieľom **návrhu** databázy (Database Design) je modelovanie časti reálneho sveta vo zvolenom dátovom modeli. Ak sa nepovie inak, implicitne sa predpokladá *relačný dátový model*. Začína sa s **koncepčným návrhom**, ktorého výsledkom je napr. UML class diagram alebo ER diagram (Entity-Relationship). Nasleduje **logický návrh** (stále nezávislý od hardwaru či softwaru), ktorého výsledkom sú relácie, typy atribútov, bezpečnostný model atď. (sem patrí aj proces normalizácie). Nasleduje **fyzický návrh**, ktorého cieľom je mapovanie logického návrhu na konkrétny DBMS a hardware. Výsledkom sú vytvorené tabuľky, kľúče, indexy, constrainty, užívateľské kontá, procedúry vkladania/vynechávania dát, pohľady (VIEWS), prístupové práva atď.
- Po návrhu nasleduje **správa a používanie** databázy

Jazyky na modelovanie reality (väčšinou vizuálnej reprezentácie modelu):

- Entitno-relačné diagramy (ER diagramy)
- UML diagramy tried (class diagrams)

Existujú softwarové nástroje, ktoré uľahčujú koncepčný návrh a čiastočne automatizujú následné fázy návrhu:

- Rational Rose (IBM)
- Visio (Microsoft)

Koncepčný návrh: ER diagramy

ER diagramy:

- popisujú aké veci (entity) v modeli vystupujú a ako spolu súvisia
- nemajú **žiadne operácie**, t.j. nepopisujú ako sa veci menia (napríklad ako vznikajú a zanikajú)
- dajú sa kresliť rôznymi spôsobmi, pričom však popisujú ten istý model („klasická“ syntax alebo UML syntax pre diagramy tried)

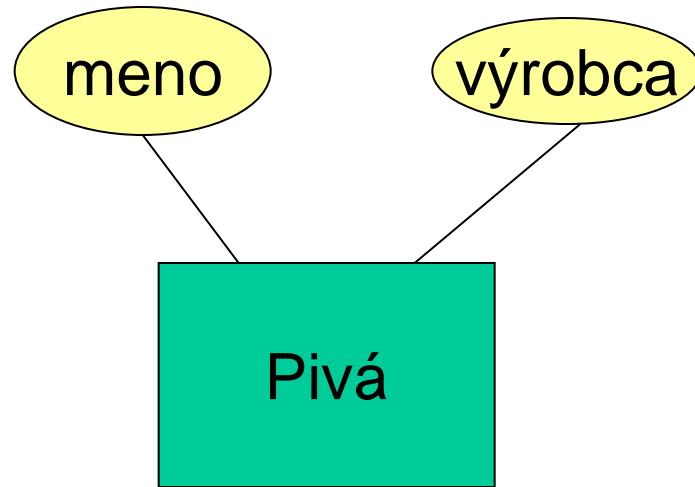
Entity sets

Pivá

Entity, ktoré modelujeme, sú pivá. Pod tým obdĺžnikom rozumieme množinu pív (entity set), nie jedno konkrétné pivo—preto ten plurál

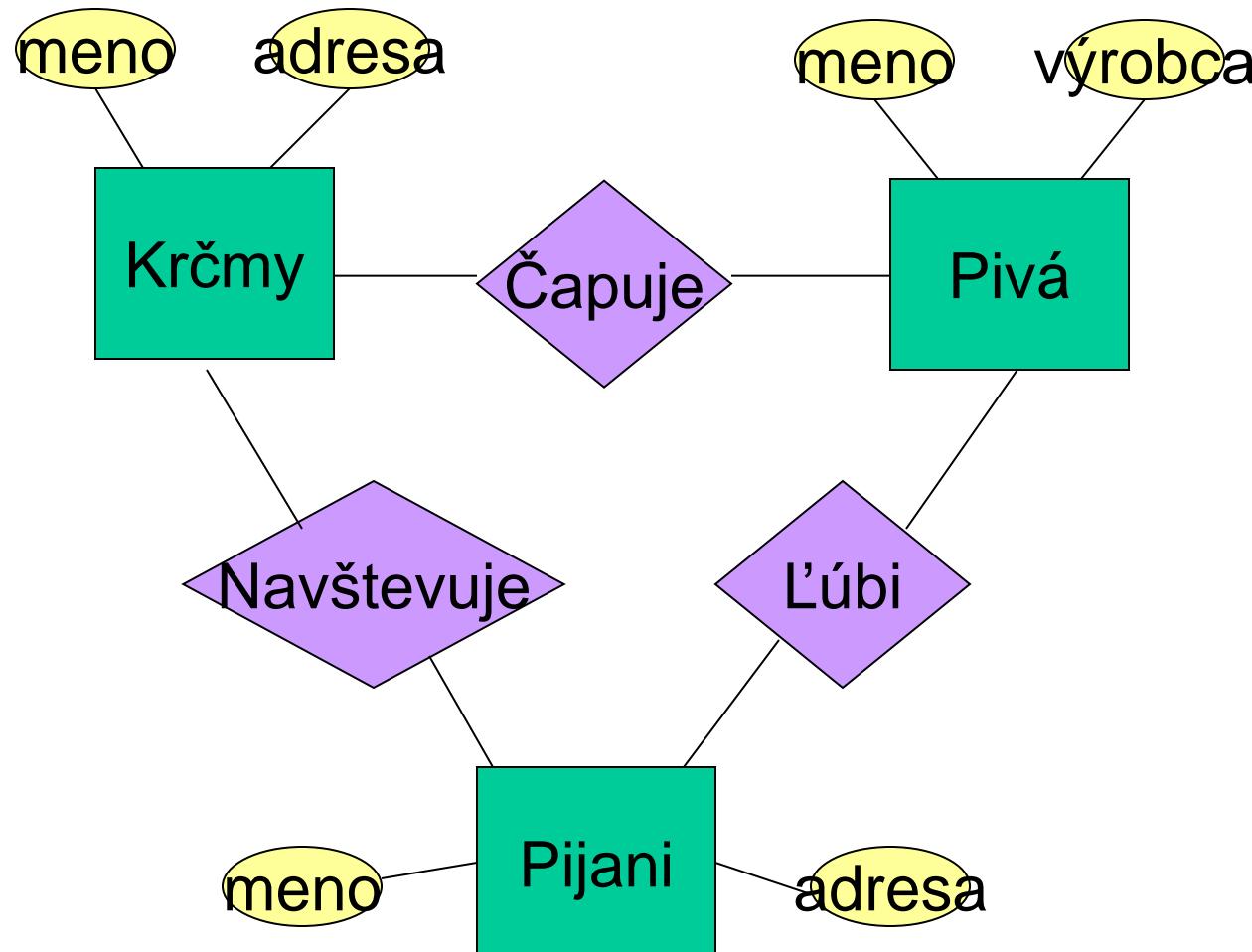
Koncepčný návrh: ER diagramy

Atribúty



Každé pivo má svoje meno a svojho výrobcu

Vzťahy (relationships)

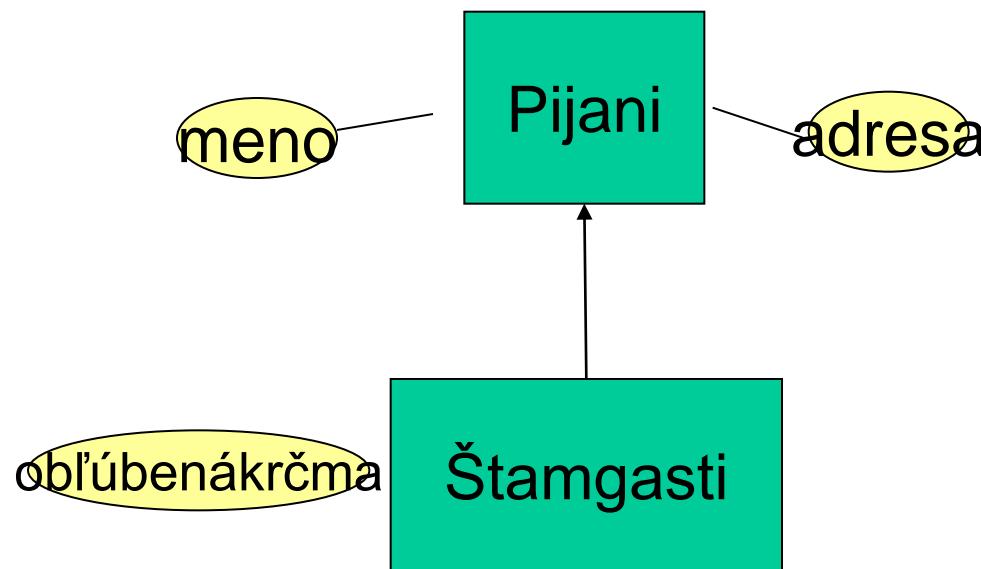


Krčmy čapujú
pivá

Pijani lúbia pivá

Pijani navštěvujú
krčmy

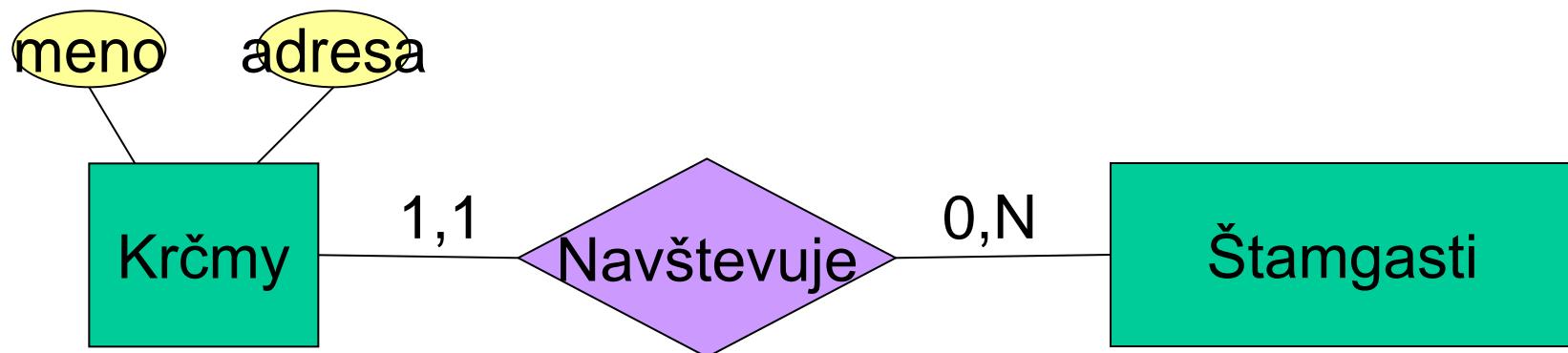
Špecializované entity (is a)



Štampasti sú pijani, ktorí majú svoje oblúbené krčmy. (Nie každý pijan má svoju oblúbenú krčmu.)

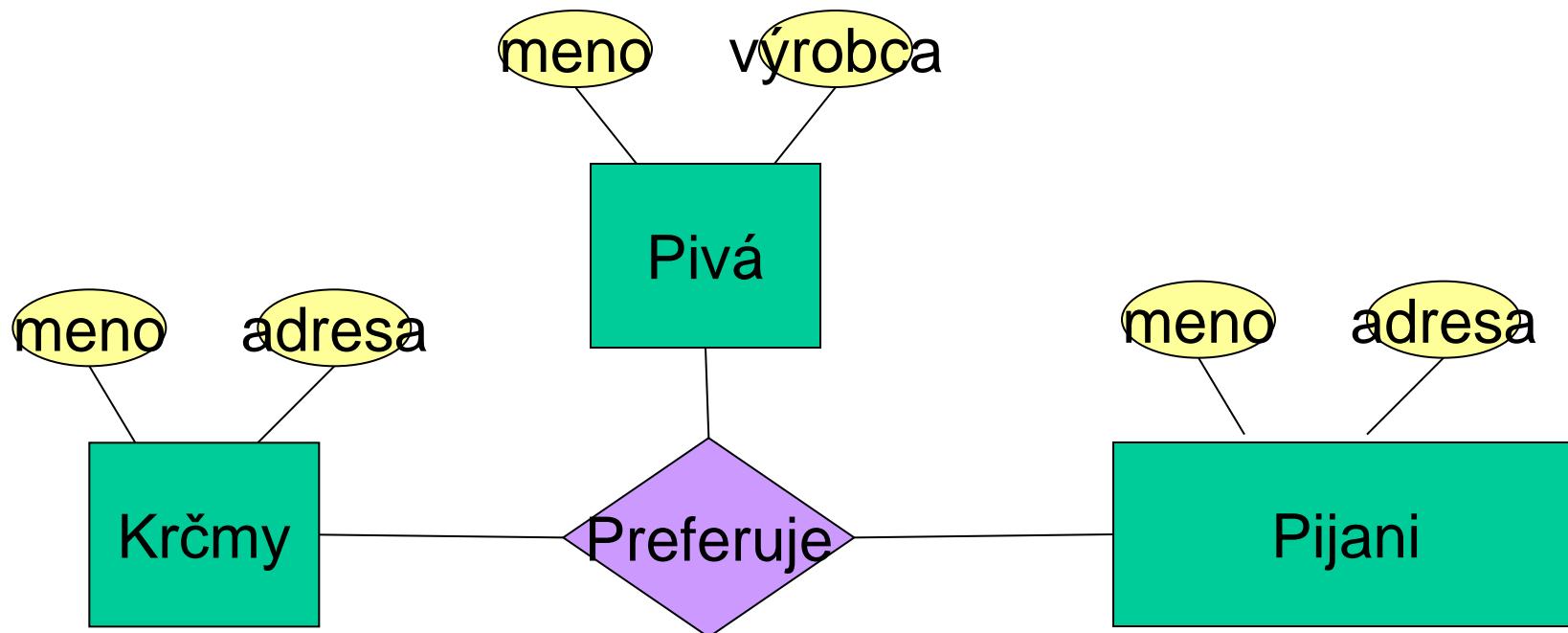
Rozdiel oproti O-O dedeniu: v O-O objekt štampast patrí iba do podriedy Štampasti. V ER objekt štampast patrí aj do Pijanov

Relácie (relationships) s multiplicitami výskytov



Štampasti sú pijani, ktorí navštevujú práve jednu krčmu. Tú istú krčmu môže navštěvovať ľubovoľne veľa (od 0 po N) štampastov

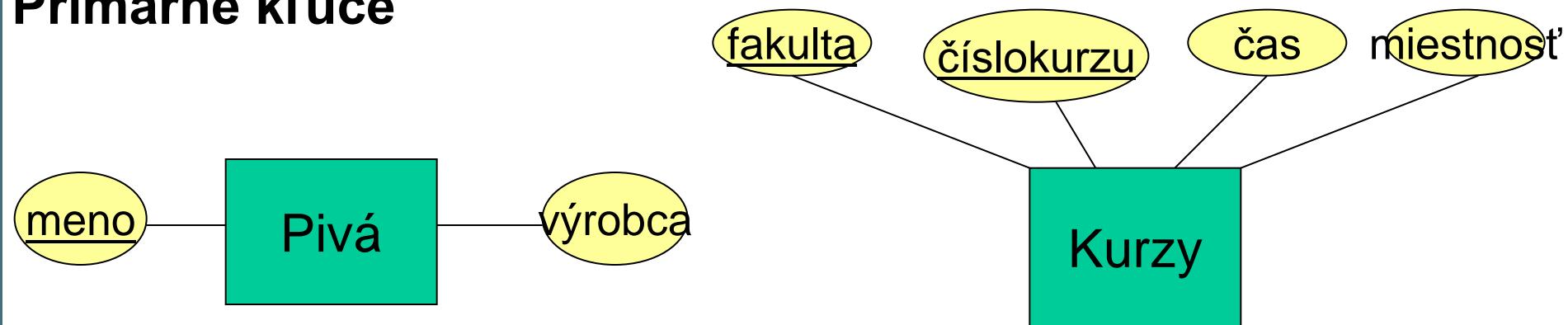
N-árne relácie



Niektoří pijani pijú Staropramen zásadne u Mamuta, ale v krčme U jeleňa pijú zásadne Stein

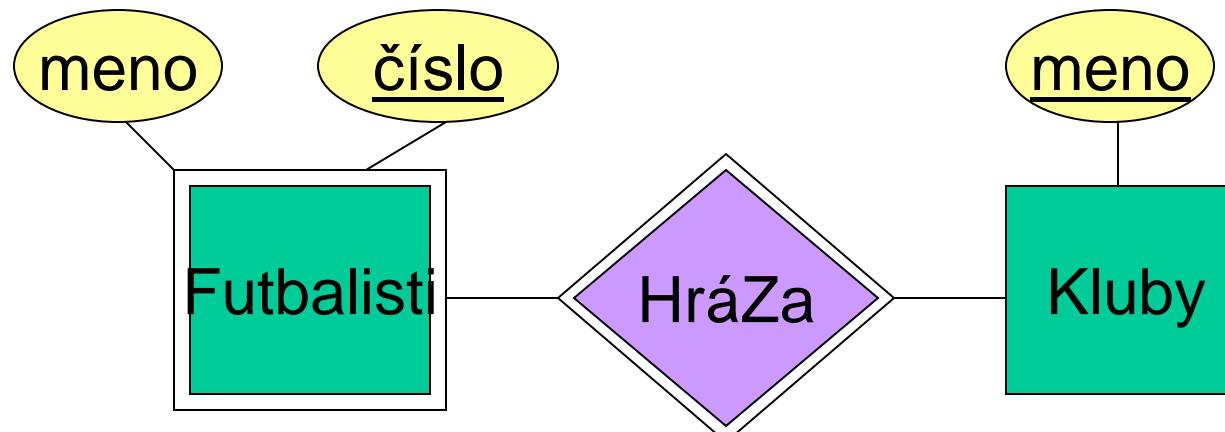
Koncepcný návrh: ER diagramy

Primárne klúče



Klúč je (neformálne) minimálna množina atribútov, ktorá jednoznačne identifikuje entitu. Klúčov môže byť viac. **Primárny klúč** je niektorý z klúčov a označuje sa počiarknutím atribútov, ktoré ho tvoria. V ER diagramoch sa vyžaduje, aby každý entity set mal nejaký klúč. V každom prípade je možné do entity set pridať tzv. **surrogate key**, ktorý slúži len pre ten účel, že je primárny klúčom. V špecializovaných entity sets (is-a hierarchia) musí byť primárny klúč v najvyššej triede zároveň klúčom vo všetkých podtriedach

Slabé entity sets



Meno futbalistu nie je kľúčom, lebo môžu existovať dva s rovnakým menom. Číslo tiež nie je kľúčom, lebo hráči rôznych klubov môžu mať na dresoch rovnaké číslo (a navyše môžu byť menovci). **Avšak číslo futbalistu spolu s menom klubu jednoznačne identifikujú futbalistu.** Kedže kľúč pre Futbalisti závisí od inej entity, Futbalisti je slabý entity set, ktorý musí byť podporený vzťahom Hráza a entity setom Kluby

Inými slovami, **existencia futbalistu je závislá na existencii klubu, za ktorý ten futbalista hrá**

- 1. Vyhýbať sa redundancii (Occamova britva)**
- 2. Vyhýbať sa slabým entity setom**
- 3. Nepoužívať entity set, keď sa dá nahradit' atribútom**

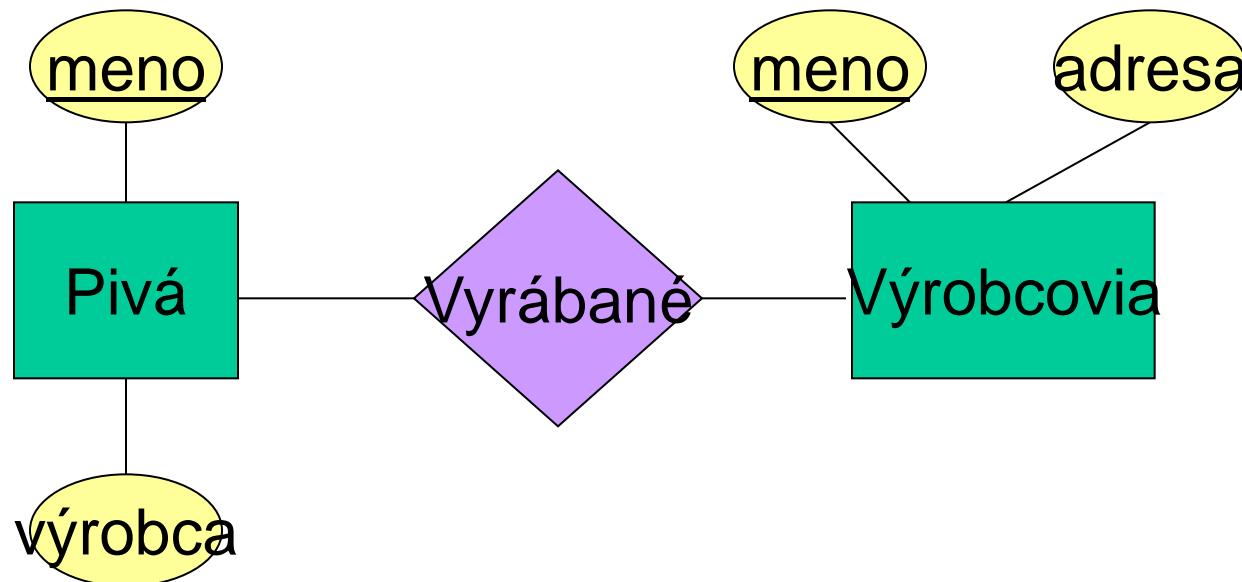
Redundancia znamená, že tá istá vec je vyjadrená niekoľkými rôznymi spôsobmi. Redundancia sa prejaví:

- plynvaním pamäťou (málo vadí)
- potrebou NULL hodnôt (málo vadí)
- **rizikom nekonzistencia (veľmi vadí)**

Slabé entity sets sa prejavia

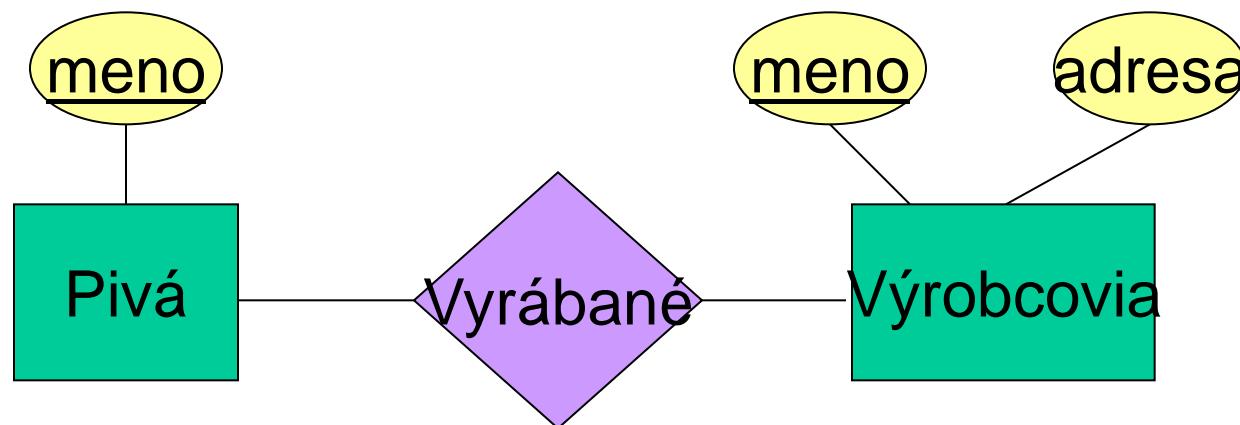
- **anomáliou pri vyniechávaní (veľmi vadí)**

Príklad chyby návrhu: redundantná entita



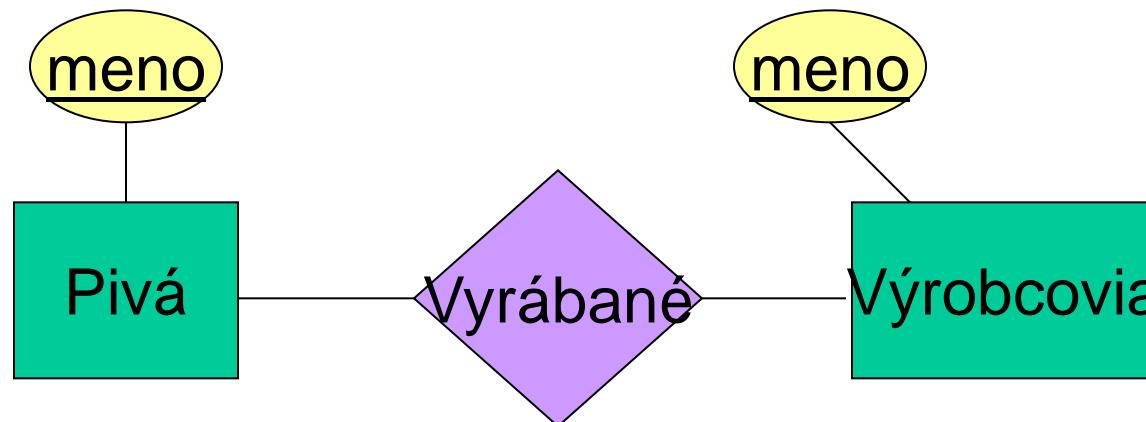
ZLE! Výrobca je aj atribút aj súvisiaci entity set

Príklad chyby návrhu: redundantná entita



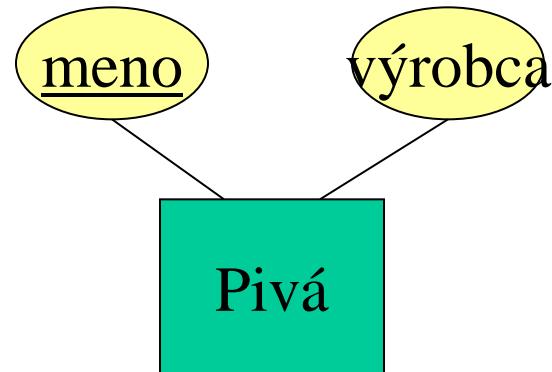
Správne

Príklad chyby návrhu: redundantná entita



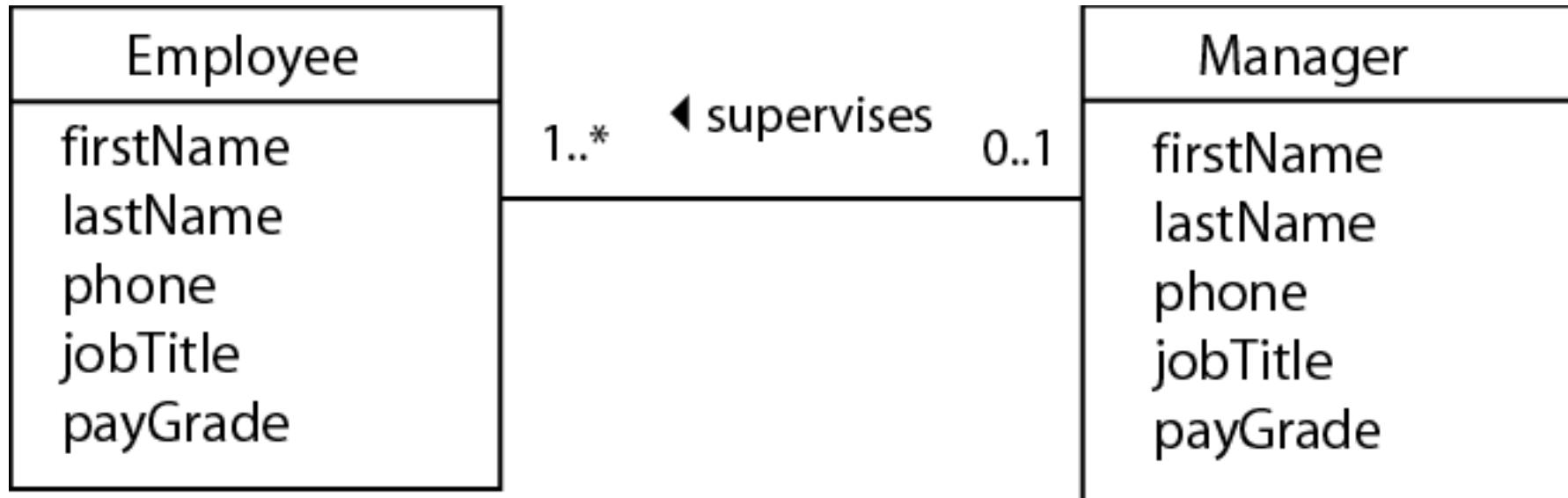
ZLE! Ak nás nezaujímajú adresy výrobcov, ale len ich mená, tak si Výrobcovia nezaslúžia byť entity setom.

Príklad chyby návrhu: redundantná entita



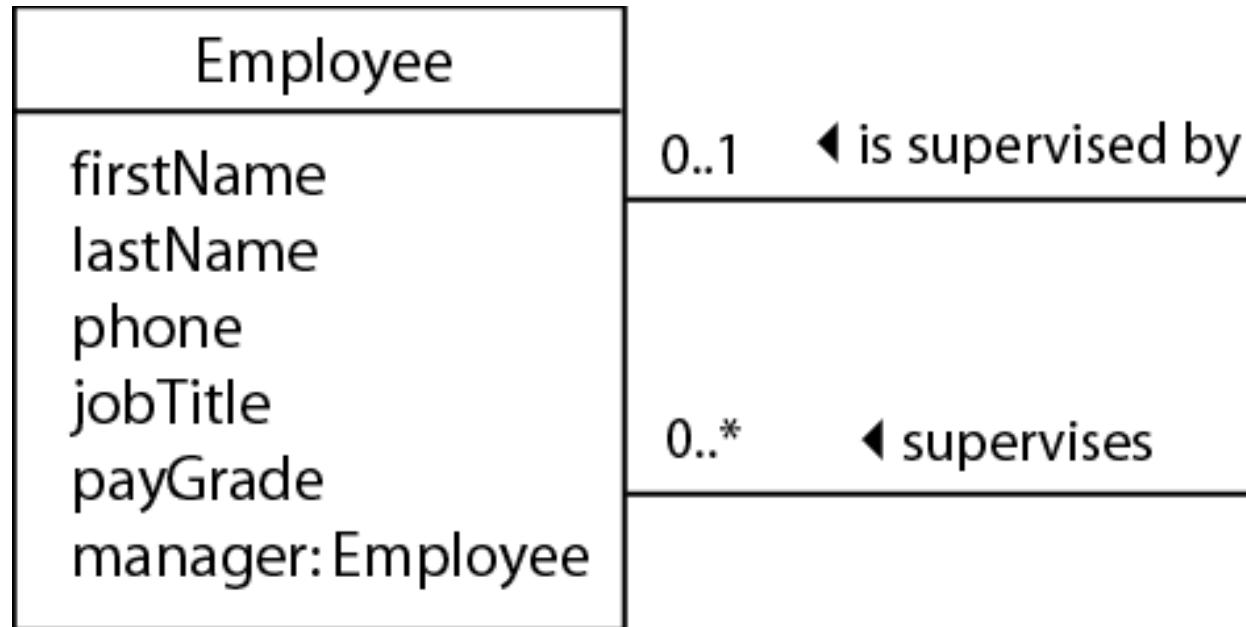
Správne

Príklad chyby návrhu: redundatná entita



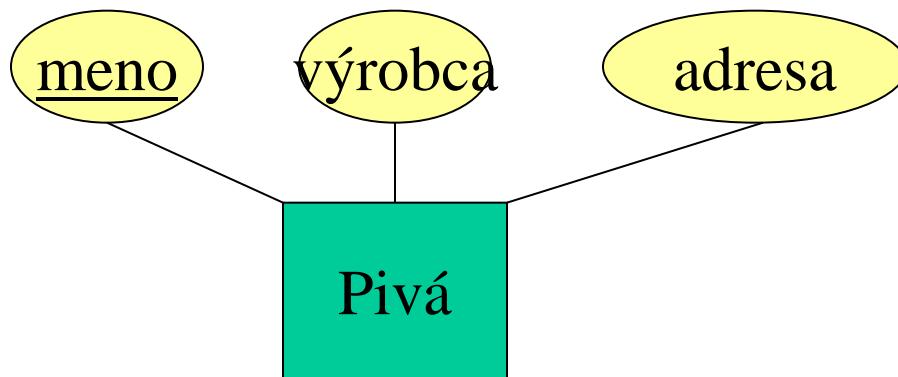
ZLE! Manager je tiež zamestnancom

Príklad chyby návrhu: redundantná entita



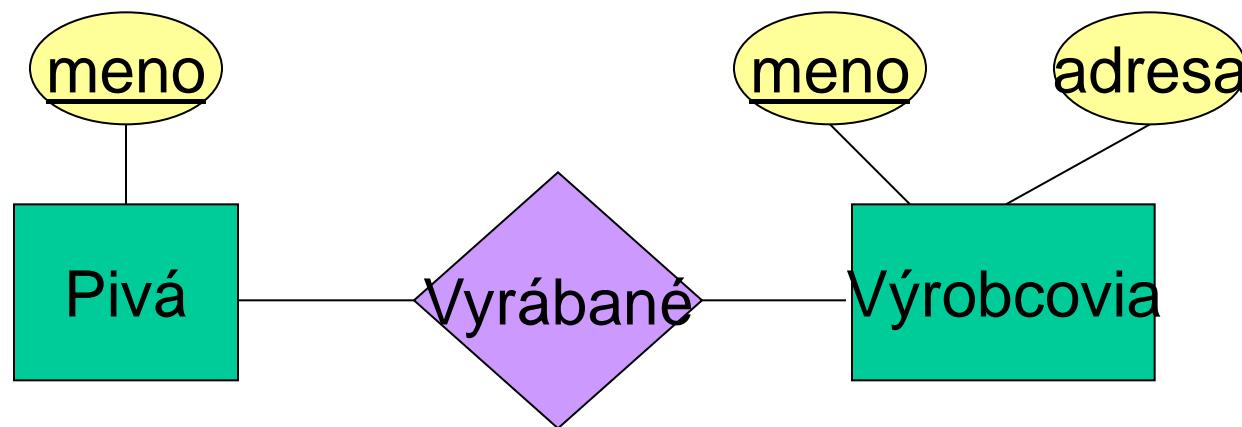
Správne

Príklad chyby návrhu: redundantné data



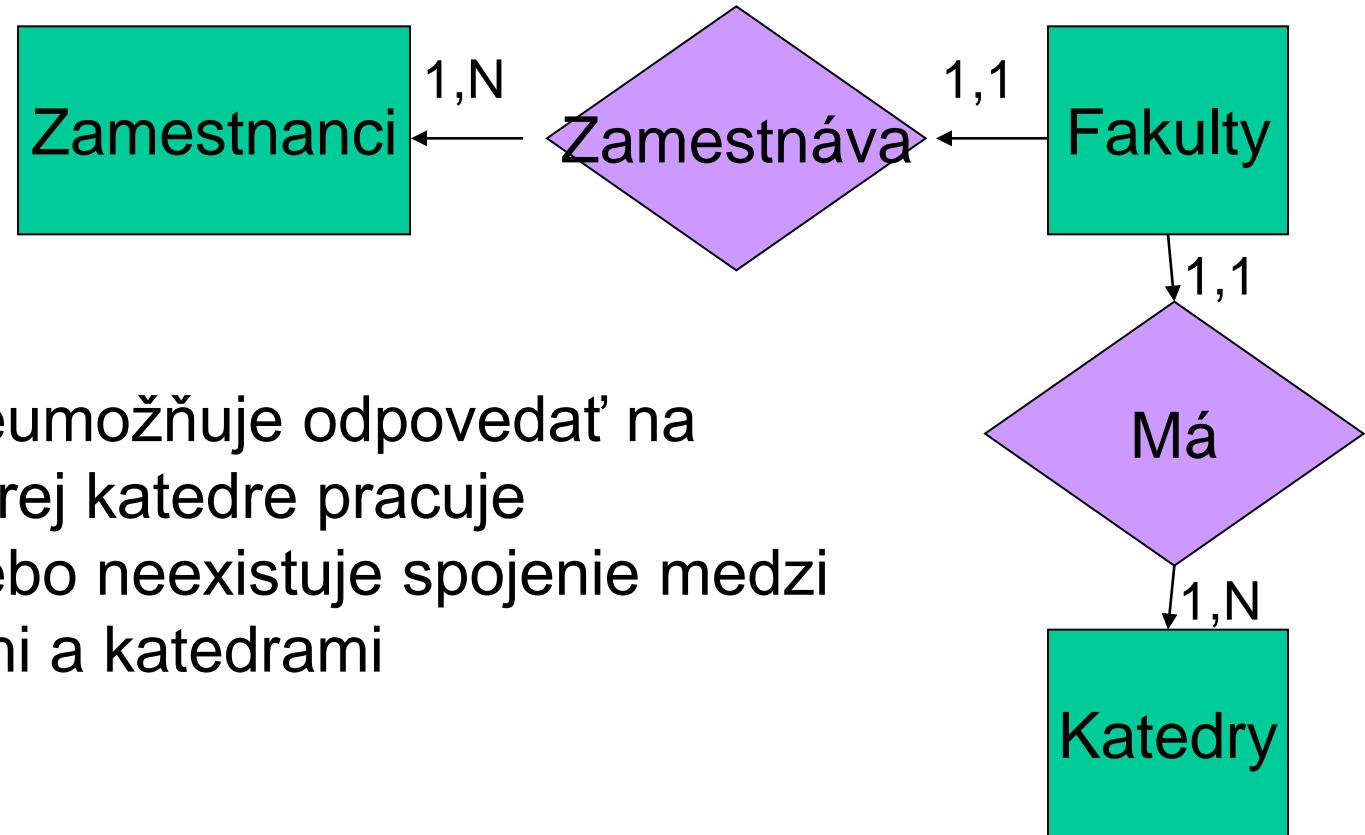
ZLE! Pre každé pivo, ktoré vyrába ten istý výrobca, sa opakuje adresa toho výrobcu. Ešte horšie je, že ak nejaký výrobca prestane na chvíľu vyrábať pivo, tak stratíme jeho adresu

Príklad chyby návrhu: redundantné data



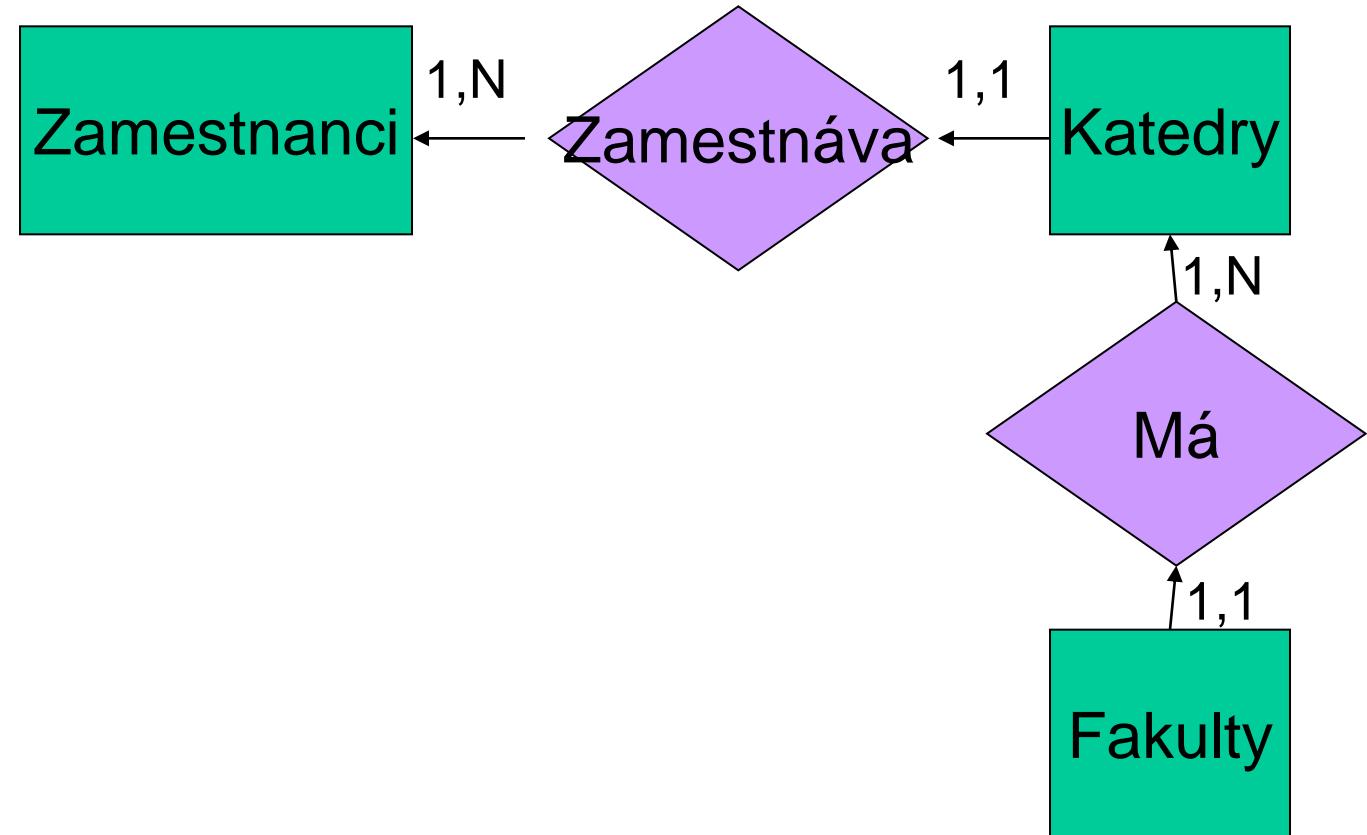
Správne

Príklad chyby návrhu: fan trap (vejár)



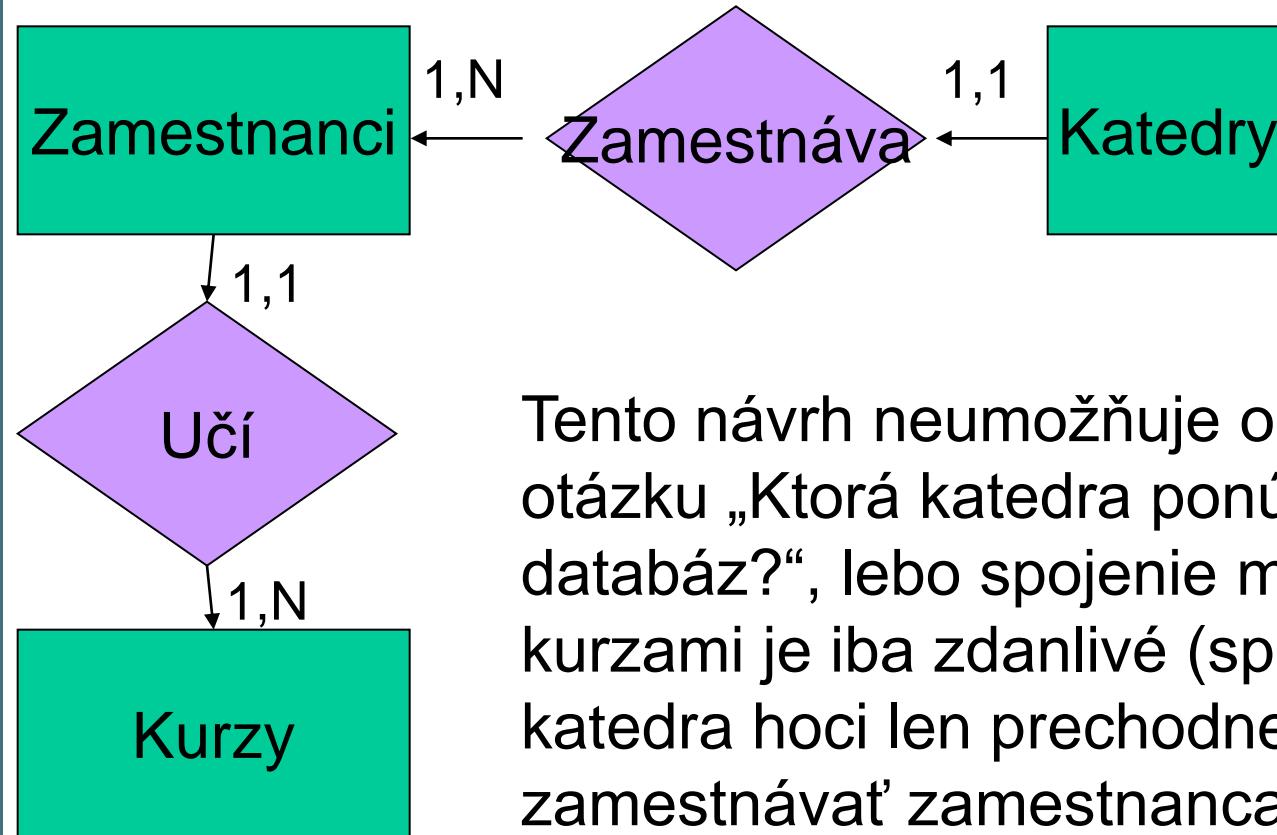
Tento návrh neumožňuje odpovedať na otázku „Na ktorej katedre pracuje Plachetka?“, lebo neexistuje spojenie medzi zamestnancami a katedrami

Príklad chyby návrhu: fan trap (vejár)



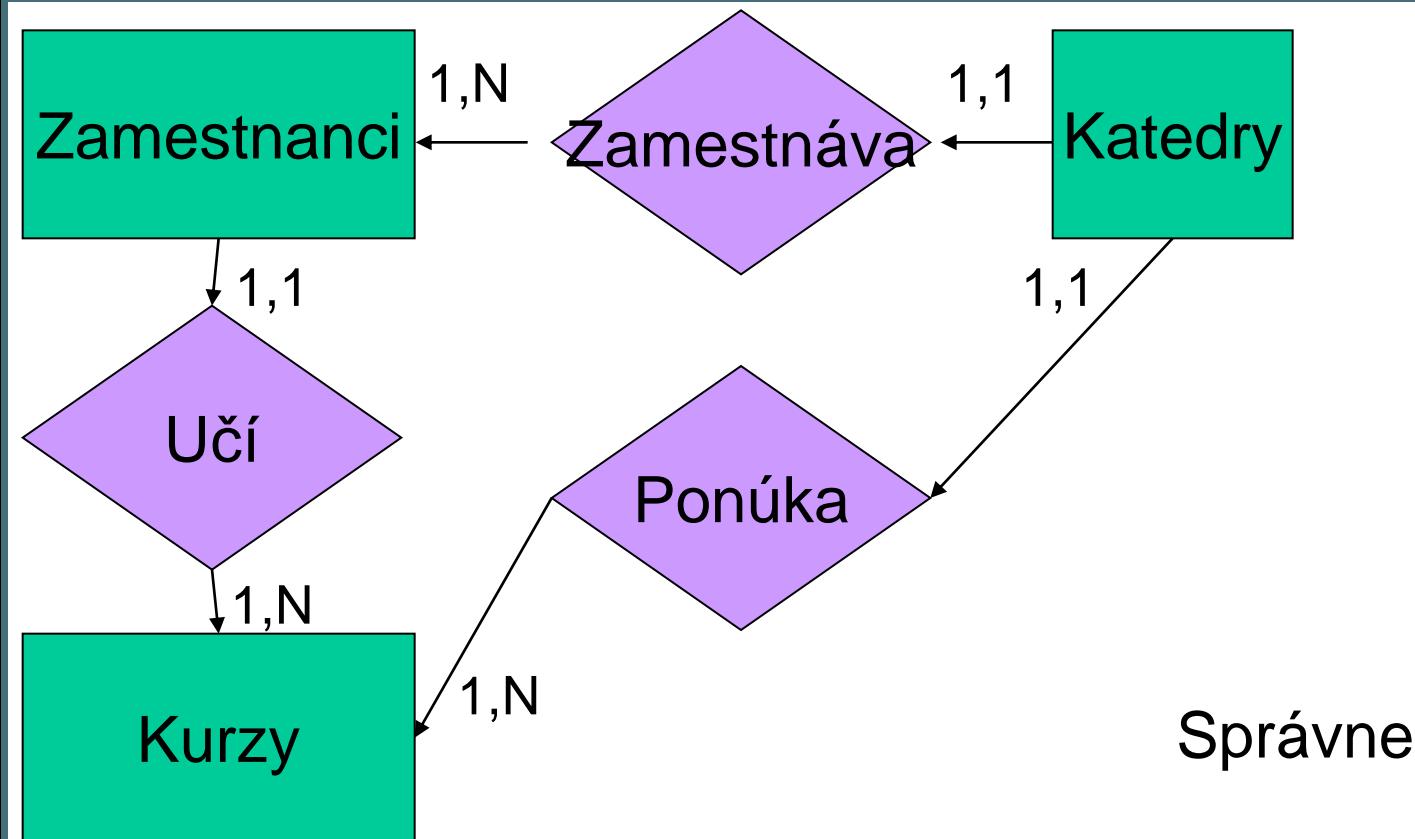
Správne

Príklad chyby návrhu: chasm trap (priepast')

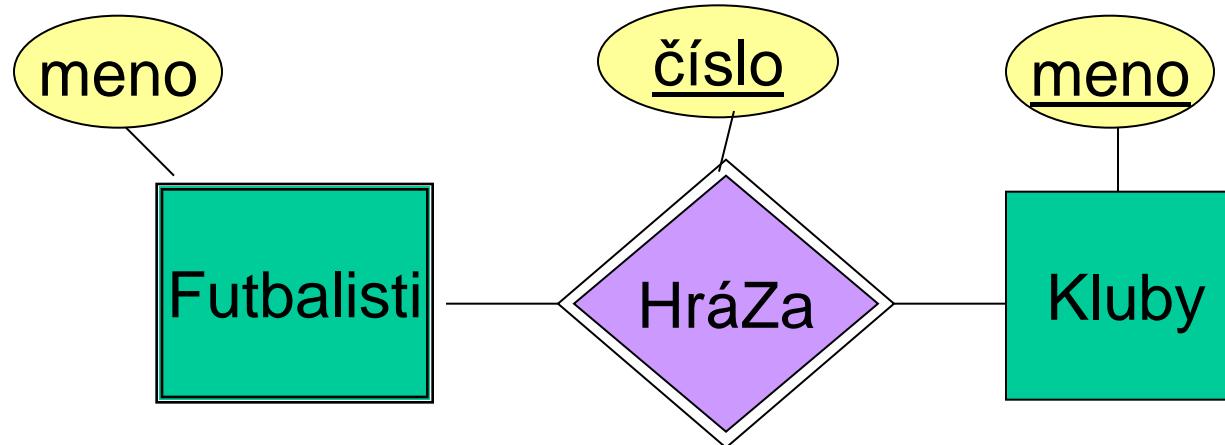


Tento návrh neumožňuje odpovedať na otázku „Ktorá katedra ponúka kurz Úvod do databáz?“, lebo spojenie medzi katedrami a kurzami je iba zdanlivé (spojenie sa stratí keď katedra hoci len prechodne prestane zamestnávať zamestnanca, ktorý ten kurz práve teraz učí)

Príklad chyby návrhu: chasm trap (priepast')



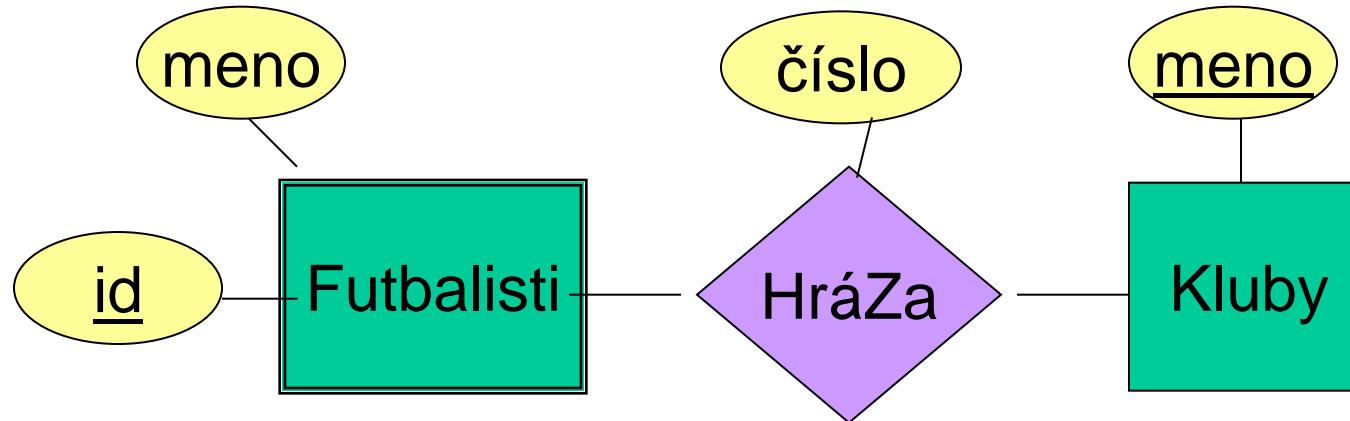
Príklad chyby návrhu: slabé entity sets



Kedže môžeme predpokladať, že kluby nevznikajú a nezanikajú príliš často, zdá sa byť použitie slabej entity set v tomto prípade celkom vhodné. Neexistenciu klubu vieme vyjadriť dodatočným atribútom, ak chceme zachovať údaje o jeho hráčoch.

Avšak tento návrh umožňuje len aktuálny pohľad. Ak futbalista prestúpi do iného klubu, tak stratíme informáciu o jeho pôvodnom klube. V podstate sa vtedy zmení **identita futbalistu**, hoci je to stále ten istý chlap. Ak na tom záleží, tak je rozumnejšie futbalistu identifikovať nezávisle na klube (napríklad surrogate kľúčom)

Príklad chyby návrhu: slabé entity sets



Ani použitie surrogate kľúča nerieši všetky problémy. Ak totiž hľadáme konkrétnego futbalistu, potrebujeme poznať jeho id. To id je sice jednoznačné, ale **odkiaľ sa dozvieme jeho hodnotu?** Toto je dôvod, prečo vzniká potreba **rozšíriť reálny svet o autoritu**, ktorá priraduje futbalistom jednoznačné identifikátory, napríklad na základe rodných čísel—takou autoritou je **futbalový zväz**. Futbalový zväz nesmie zverejniť rodné čísla, no v prípade nejednoznačného mena vie futbalový zväz rozlíšiť dvoch futbalistov podľa ďalších znakov (napr. podľa klubovej história alebo prinajhoršom podľa prezývky či rodného čísla)

Príklad chyby návrhu: nevhodný primárny kľúč

Príklad nevhodného primárneho kľúča: rodné číslo, PSČ

Výber primárneho kľúča treba starostlivo zvážiť. Často je vhodné použiť surrogate, aj keď prirodzený externý kľúč existuje. Dôvod: **databázový systém nedovoľuje modifikovať primárny kľúč**

Možnosti výberu primárneho kľúča:

- externý kľúč: napr. rodné číslo (ale pozor, ak hrozí omyl pri vkladaní rodného čísla do databázy, tak radšej surrogate key)
- surrogate key: generovaný automaticky databázovým systémom
- substitute primary key: jeden atribút, obvykle skratka (napr. 3-písmenové skratky letísk)

Návrhové vzory

- Oplatí sa poznať dobré riešenia typických situácií

- Zdroje:

- literatúra

- vlastné skúsenosti

- reverzné inžinierstvo fungujúcich systémov

Domáca úloha: pozrieť sa (vyhľadávať v Google), ako je navrhnutý formát GEDCOM, ktorý je v súčasnosti štandardom na uchovávanie genealogických dát

- Metódy v koncepčnej fáze návrhu sú iba **poloformálne**.

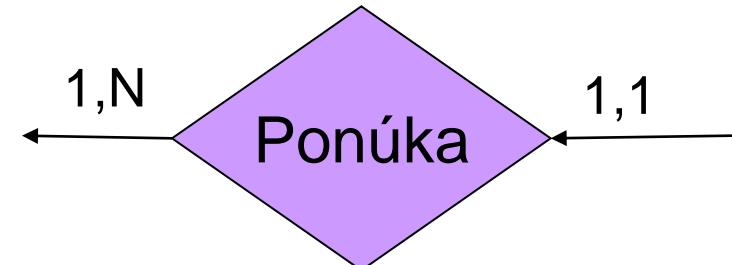
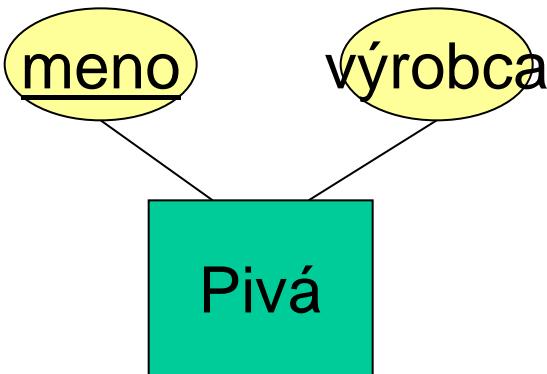
Niekedy je ľahké rozoznať dôležitý poznatok od (zbytočnej?) byrokracie. Lenže taký je reálny svet

Od diagramov k SQL: Data Definition Language (DDL)

Data Definition Language, DDL, slúži na vytvorenie databázy. Ked"že SQL je norma, možno preklad ER diagramov do SQL chápať ako prechod od koncepčného návrhu k logickému návrhu

- Entity sets sa prekladajú ako tabuľky
 - Problém je voľba primárneho kľúča—v prípade akýchkoľvek pochybností je vhodný surrogate key
 - Binárne vzťahy sa prekladajú do binárnych tabuliek (v prípade many-to-many sú kľúčom oba atribúty, v prípade many-to-one je kľúč na strane one). V prípade vzťahu many-to-one je možné binárnu reláciu nahradíť importovaním cudzieho kľúča do podriadenej (slabej) relácie
 - N-árne vzťahy sa prekladajú do n-árnych tabuliek

Preklad entity sets a vzťahov do SQL



1. Vytvorenie typov atribútov (SQL-99):

CREATE DOMAIN *name* [AS] *data_type*

[**DEFAULT** *expression*]

[*constraint* [...]]

where *constraint* is:

[**CONSTRAINT** *constraint_name*]

{ NOT NULL | NULL | **CHECK** (*expression*) }

2. Vytvorenie tabuľky:

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ]  
TABLE table_name (  
{ column_name data_type [ DEFAULT default_expr ] [  
column_constraint [ ... ] ]  
| table_constraint  
| LIKE parent_table [ { INCLUDING | EXCLUDING } DEFAULTS ]  
} [ , ... ]  
)  
[ INHERITS ( parent_table [ , ... ] ) ]  
[ WITH OIDS | WITHOUT OIDS ]  
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
```

OIDS je skratka pre Object Identifiers, t.j „WITH OIDS“ znamená „vytvor surrogate primary key“

Preklad entity sets a vzťahov do SQL

3. Constraints na stĺpce:

```
[ CONSTRAINT constraint_name ]  
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY |  
CHECK (expression) |  
REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH  
PARTIAL | MATCH SIMPLE ]  
[ ON DELETE action ] [ ON UPDATE action ] }  
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED  
| INITIALLY IMMEDIATE ]
```

action :: =

```
NO ACTION | SET NULL | SET DEFAULT | CASCADE
```

4. Constraints na celú tabuľku:

```
[ CONSTRAINT constraint_name ]  
{ UNIQUE ( column_name [, ...] ) |  
PRIMARY KEY ( column_name [, ...] ) |  
CHECK ( expression ) |  
FOREIGN KEY ( column_name [, ...] ) REFERENCES reftable [ ( refcolumn [, ...] ) ]  
[ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ] ]
```

action ::= =

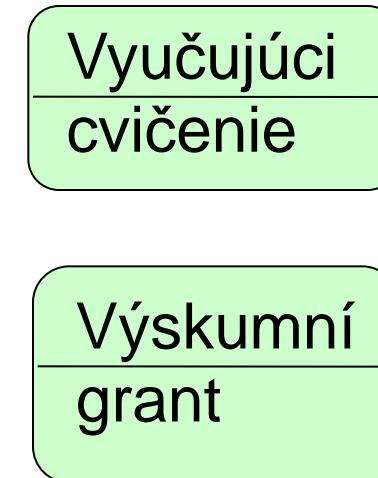
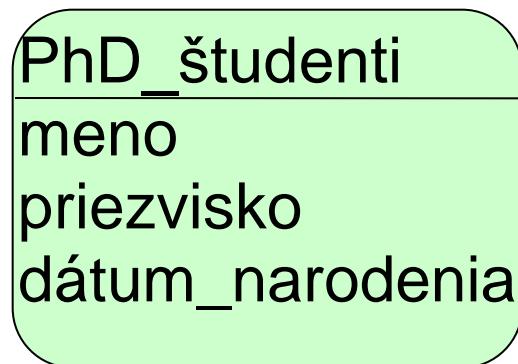
NO ACTION | **SET NULL** | **SET DEFAULT** | **CASCADE**

Preklad entity sets a vzťahov do SQL



Špecializácia sa preloží do samostatných tabuľiek s rovnakým primárnym kľúčom (ktorý je uložený v tabuľke Štampasti)

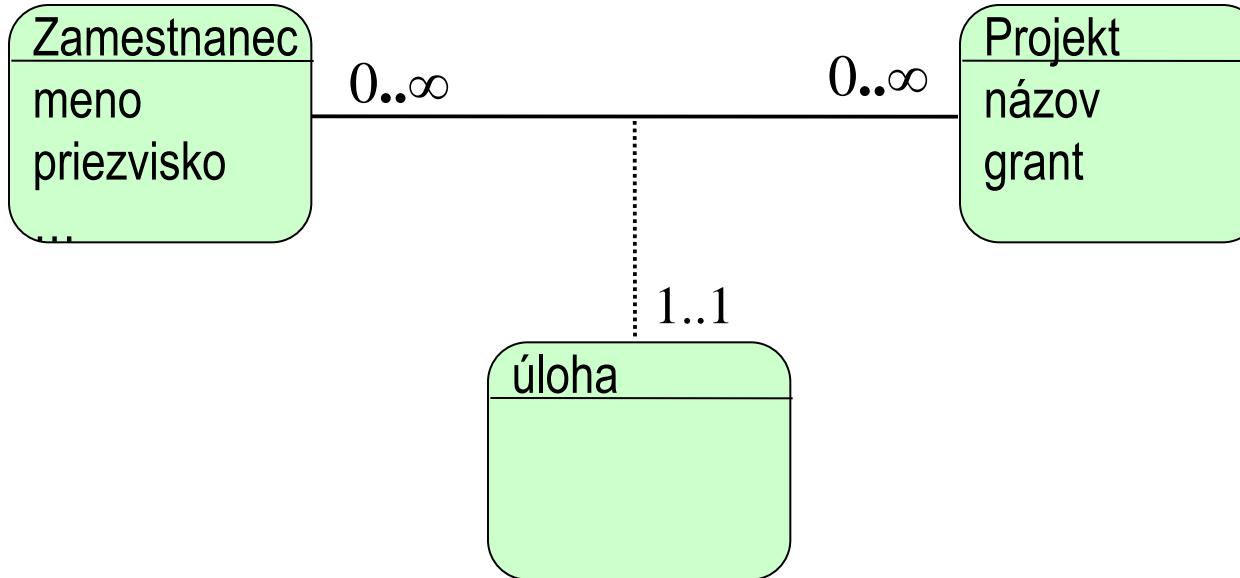
Príklad prekladu špecializácie



```
create table PhD_studenti (ids, meno, priezvisko,  
dátum_narodenia, constraint primary key (ids));  
create table Vyučujúci (ids, cvičenie,  
primary key (ids),  
check (exists (  
    select * from PhD_studenti P where P.ids =ids)));  
create table Výskumní (ids, grant,  
primary key(ids), check (exists  
(select * from PhD_studenti P where P.ids = ids)));
```

Preklad entity sets a vzťahov do SQL

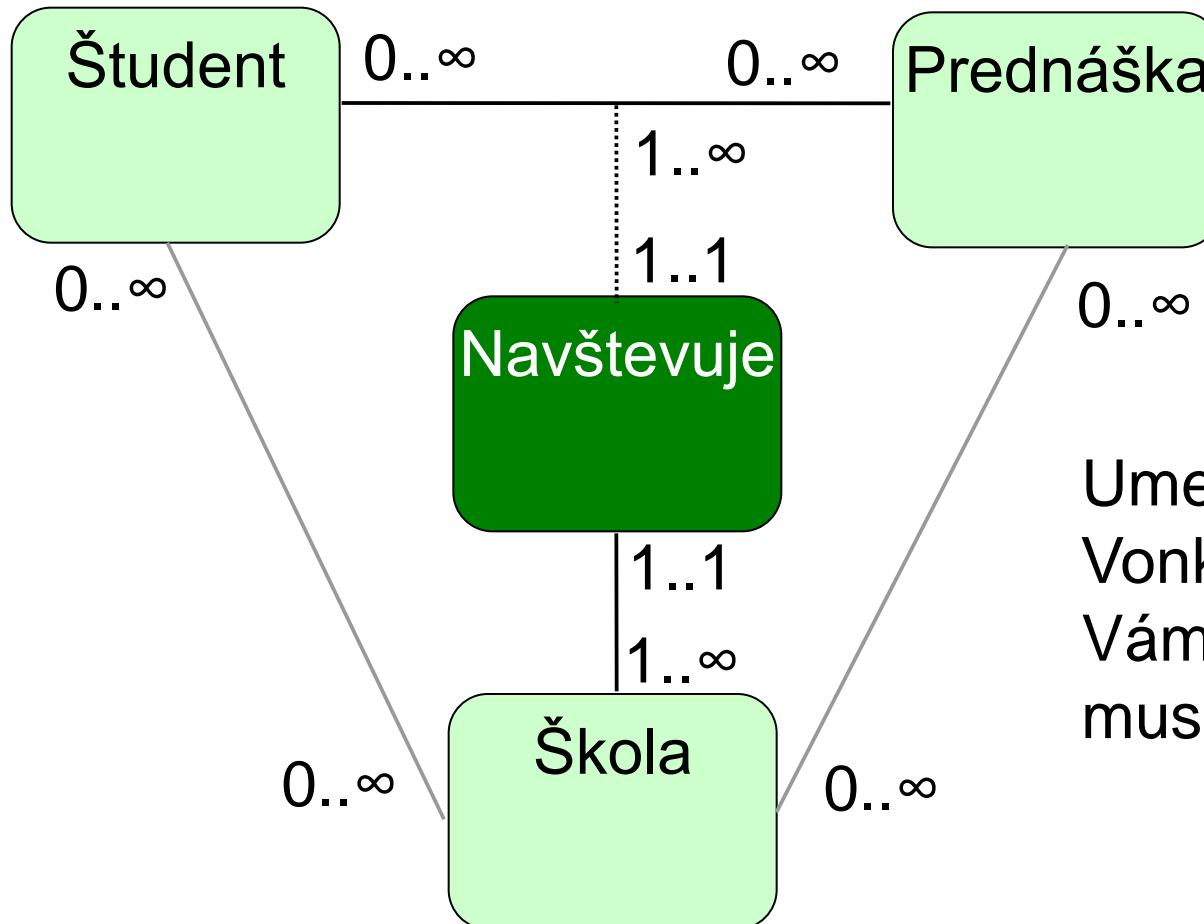
Príklad prekladu ternárneho vzťahu



```
create table Zamestnanec(idz, meno, priezvisko, ...,
    primary key (idz));
create table Projekt(čp, meno, názov, grant, ...,
    primary key (čp));
create table Úloha(idz, čp, ču, primary key(idz, čp, ču),
    foreign key (idz) references Zamestnanec,
    foreign key (čp) references Projekt
    on delete cascade);
```

Preklad entity sets a vztáhov do SQL

Ešte jeden príklad prekladu ternárneho vztáhu

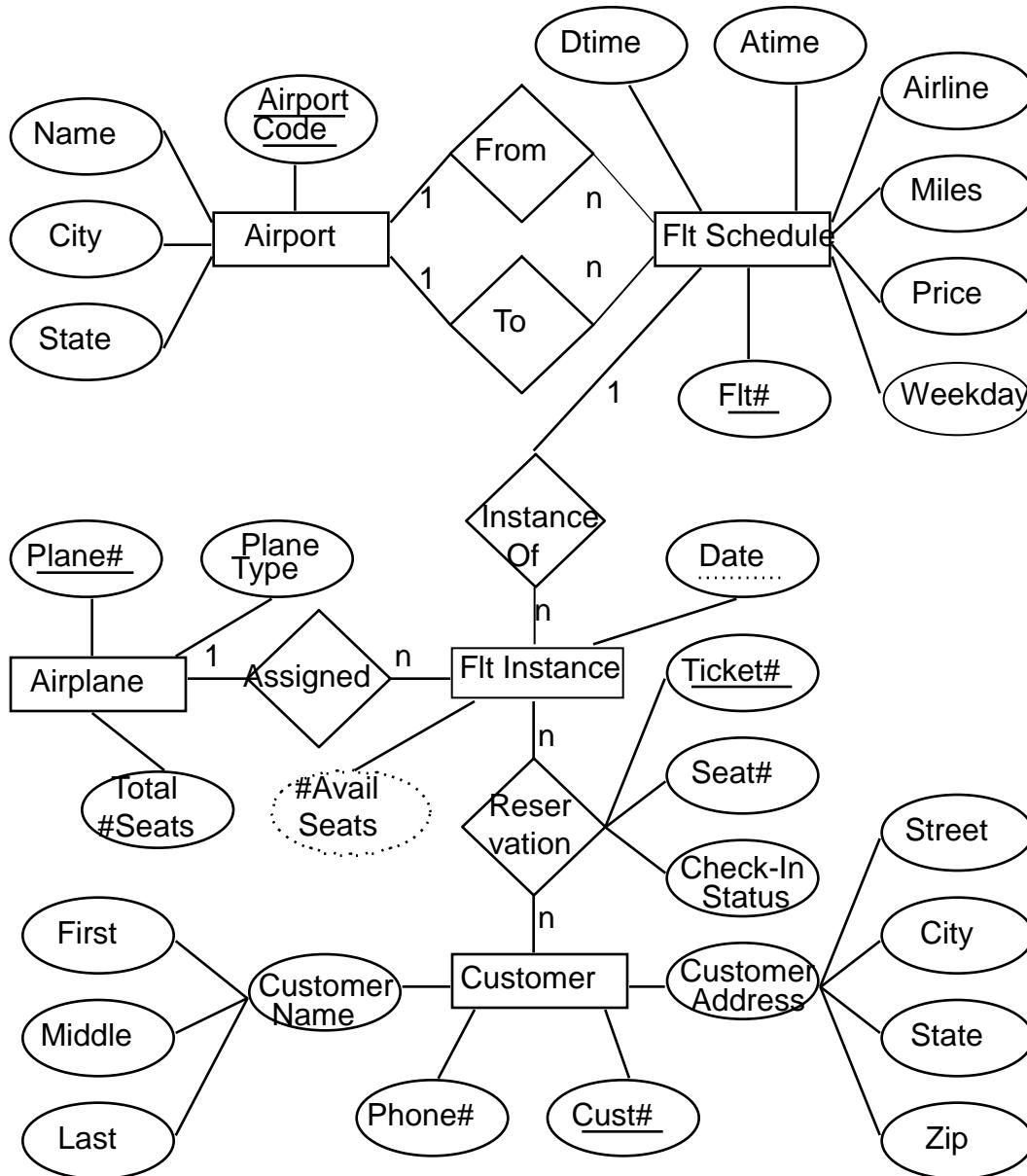


Umenie návrhu:
Vonkajší trojuholník
Vám hovoria, vnútro
musíte objaviť sami

Preklad entity sets a vzťahov do SQL

create table Študent (rč, meno, priezvisko, ... , **primary key**(rč));
create table Prednáška (názov, ... , **primary key**(názov));
create table Škola (IČO, ... , **primary key**(IČO));
create table Navštevuje (rč, názov, IČO,
 primary key(rč, názov, IČO),
 foreign key (rč) references Študent,
 foreign key (názov) references Prednáška,
 foreign key (IČO) references Škola);

Identifikácia funkčných závislostí (Lee Mark)



Funkčné závislosti v ER-diagrame

AIRPORT ↔ Airportcode

FLT-SCHEDULE ↔ Flt#

FLT-INSTANCE ↔ (Flt#, Date)

AIRPLANE ↔ Plane#

CUSTOMER ↔ Cust#

RESERVATION ↔ (Cust#, Flt#, Date)

RESERVATION ↔ Ticket#

Airportcode → Name, City, State

Flt# → Airline, Dtime, Atime, Miles, Price, (From) Airportcode, (To) Airportcode

(Flt#, Date) → Flt#, Date, Plane#

(Cust#, Flt#, Date) → Cust#, Flt#, Date, Ticket#, Seat#, CheckInStatus,

Ticket# → Cust#, Flt#, Date

Cust# → CustomerName, CustomerAddress, Phone#

Úvod do databázových systémov

[http://www.dcs.fmph.uniba.sk/~plachetk
/TEACHING/DB2013](http://www.dcs.fmph.uniba.sk/~plachetk/TEACHING/DB2013)

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2013–2014

- Teoreticky každá ľubovoľná relačná databáza sa dá reprezentovať jedinou tabuľkou. To však viedie k problémom: **riziko nekonzistencie, anomálie pri vynechávaní a modifikácii dát, potreba NULL hodnôt, plynvanie pamäťou**
- Produktom logického návrhu relačnej databázy sú **relácie, atribúty a funkčné závislosti** (pričom atribúty sú dôležitejšie než relácie)
- Na overenie kvality návrhu, resp. **automatické generovanie vhodných tabuliek** slúži **normalizácia** (formálne metódy)

Motivácia normalizácie (T. Conolly and C. Begg)

Príklad: StaffBranch

staffNo	sName	position	salary	branchNo	bAddress
SL21	John White	Manager	30000	B005	22 Deer Rd, London
SG37	Ann Beech	Assistant	12000	B003	163 Main St, Glasgow
SG14	David Ford	Supervisor	18000	B003	163 Main St, Glasgow
SA9	Mary Howe	Assistant	9000	B007	16 Argyll St, Aberdeen
SG5	Susan Brand	Manager	24000	B003	163 Main St, Glasgow
SL41	Julie Lee	Assistant	9000	B005	22 Deer Rd, London

Staff

staffNo	sName	position	salary	branchNo
SL21	John White	Manager	30000	B005
SG37	Ann Beech	Assistant	12000	B003
SG14	David Ford	Supervisor	18000	B003
SA9	Mary Howe	Assistant	9000	B007
SG5	Susan Brand	Manager	24000	B003
SL41	Julie Lee	Assistant	9000	B005

Branch

branchNo	bAddress
B005	22 Deer Rd, London
B007	16 Argyll St, Aberdeen
B003	163 Main St, Glasgow

Funkčné závislosti (functional dependencies)

Definícia. V relácii r platí funkčná závislosť $X \rightarrow Y$ (t.j. množina atribútov Y funkčne závisí od množiny atribútov X) vtedy a len vtedy ak

$$\forall X \forall Y_1 \forall Z_1 \forall Y_2 \forall Z_2 (r(X, Y_1, Z_1) \wedge r(X, Y_2, Z_2) \Rightarrow Y_1 = Y_2)$$

kde X je inštancia X , Y_1 a Y_2 sú inštancie Y

Inými slovami, $X \rightarrow Y$ v relácii r hovorí, že ak sa v r ľubovoľné dva riadky zhodujú na množine atribútov X , tak sa musia zhodovať aj na množine atribútov Y

Funkčné závislosti (functional dependencies)

Bar	Adresa	Pivo	Vyrobca	Cena
Janeway	Voyager	Bud	A.B.	3
Janeway	Voyager	WickedAle	Pete's	2
Spock	Enterprise	Bud	A.B.	3

Platí (vždy, nielen pre tieto tri konkrétné riadky!)

$\text{Bar} \rightarrow \text{Adresa}$

$\text{Bar}, \text{Pivo} \rightarrow \text{Cena}$

Ale neplatí napríklad

$\text{Bar} \rightarrow \text{Pivo}$

$\text{Pivo} \rightarrow \text{Cena}$ (hoci v tejto konkrétnej populácii databázy to platí)

Vlastnosti funkčných závislostí (Armstrongove axiómy)

(A1) $X \subseteq Y \Rightarrow Y \rightarrow X$	reflexívnosť
(A2) $\forall Z (X \rightarrow Y \Rightarrow XZ \rightarrow YZ)$	rozšírenie (augmentation)
(A3) $(X \rightarrow Y) \wedge (Y \rightarrow Z) \Rightarrow X \rightarrow Z$	tranzitívnosť

Dôkaz (z definície v relačnom kalkule):

$$(A1) \forall X \forall Y \forall Z_1 \forall Z_2 (X \subseteq Y \wedge r(Y, Z_1) \wedge r(Y, Z_2) \Rightarrow X = X)$$

Atd. Domáca úloha (bez bonusu): dokázať “axiómy“ A2 a A3

Ďalšie vlastnosti funkčných závislostí

(B1) $(X \rightarrow Y) \wedge (X \rightarrow Z) \Rightarrow X \rightarrow YZ$ union rule

(B2) $(X \rightarrow Y) \wedge (WY \rightarrow Z) \Rightarrow WX \rightarrow WZ$ pseudotransitivity

(B3) $(X \rightarrow Y) \wedge (Z \subseteq Y) \Rightarrow X \rightarrow Z$ decomposition

(B4) $(X \rightarrow Y) \wedge (X \subseteq Z) \Rightarrow Z \rightarrow Y$ left-hand side simplification

Dôkaz B1 (z Armstrongovych axióm):

$X \rightarrow Y \Rightarrow X \rightarrow XY$ podľa (A2)

$X \rightarrow Z \Rightarrow XY \rightarrow YZ$ podľa (A2)

Takže $(X \rightarrow Y) \wedge (X \rightarrow Z) \Rightarrow X \rightarrow YZ$ podľa (A3)

Dôkazy B2, B3 a B4 sa dajú urobiť podobným spôsobom použitím Armstrongovych axióm (**domáca loha, bez bonusu**)

Uzáver množiny atribútov

Definícia. Nech X je množina atribútov a F množina funkčných závislostí. Potom **uzáverom množiny atribútov X vzhľadom na F** rozumieme množinu X^+ všetkých atribútov Y takých, že $X \rightarrow Y$ je (logickým) dôsledkom funkčných závislostí F

Výpočet uzáveru:

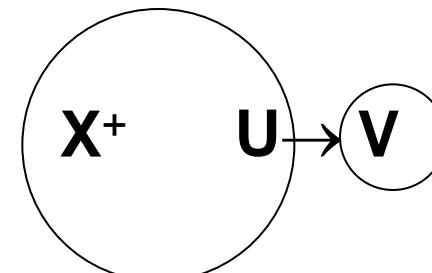
$X^+ := X;$

repeat

for each $U \rightarrow V \in F$ **do**

if $U \in X^+$ **then** $X^+ := X^+ \cup V;$

while niečo sa pridal do X^+ ;



Optimalizácia: každá závislosť sa použije práve raz, po použití ju možno vynechať. (Teda uzáver sa počíta v lineárnom čase.)

Výpočet uzáveru množiny atribútov

Príklad:

$$AB \rightarrow C$$

$$ACD \rightarrow B$$

$$CG \rightarrow BD$$

$$C \rightarrow A$$

$$D \rightarrow EG$$

$$CE \rightarrow AG$$

$$BC \rightarrow D$$

$$BE \rightarrow C$$

Nech $X = \{BD\}$. Potom X^+ sa počíta takto:

$$X^{(0)} = \{BD\}, X^{(1)} = \{BDEG\}, X^{(2)} = \{BCDEG\},$$

$$X^{(3)} = \{ABCDEG\} = X^{(4)} = X^+$$

Výpočet uzáveru množiny atribútov je potrebný napríklad pri
generovaní klúčov či hľadaní minimálneho pokrycia množiny funkčných závislostí

Veta. Funkčná závislosť $X \rightarrow Y$ sa dá odvodiť z F pomocou Armstrongovych axióm práve vtedy, keď $X \rightarrow Y$ je logickým dôsledkom F

Dôkaz:

Ked'že Armstrongove axiómy sú formuly, ktoré sa dajú dokázať z definície funkčnej závislosti, dajú sa z nich odvodiť len platné závislosti

Ostáva dokázať, že ak nejaká funkčná závislosť platí (t.j. je dôsledkom F), tak sa dá odvodiť z Armstrongovych axióm. To dokážeme sporom

Úplnosť Armstrongovych axióm

Predpokladajme, že $X \rightarrow Y$ platí a nedá sa odvodiť z Armstrongovych axióm. Uvažujme dvojriadkovú reláciu r:

	C	ostatné atribúty
$r_0:$	1 1 ... 1	0 0 ... 0
$r_1:$	1 1 ... 1	1 1 ... 1

kde **C** je množina atribútov podobná X^+ až na to, že tu uzáver robíme len s použitím z Armstrongovych axióm, teda nie predošlým algoritmom (platí sice, že $C = X^+$, lenže to sme zatiaľ nedokázali, takže **C** a X^+ musíme považovať za rôzne). Platí $X \in C$ (podľa A1) a $Y \notin C$ (lebo predpokladáme, že $X \rightarrow Y$ sa nedá odvodiť). Teda $X \rightarrow Y$ nie je splnená v r (pre X sú v r rôzne Y).

Úplnosť Armstrongovych axióm

Stačí dokázať, že v r je splnená každá funkčná závislosť, ktorá je dôsledkom F (spor s tvrdením, že $X \rightarrow Y$ nie je splnená v r).

Nech niektorá funkčná závislosť $S \rightarrow T$ je dôsledkom F, ale nie je splnená v r. Sú dve možnosti: 1. $S \subseteq C$ a 2. $S \not\subseteq C$.

1. Ak $S \subseteq C$, tak sa dá odvodiť $C \rightarrow S$ (podľa A1) a tiež $C \rightarrow T$ (podľa A3). Takže ak $S \subseteq C$, tak aj $T \subseteq C$ (C podľa definície obsahuje všetky odvoditeľné atribúty). Lenže v tom prípade závislosť $S \rightarrow T$ je splnená v r, lebo riadky r_0 a r_1 sa zhodujú na T .

2. Ak $S \not\subseteq C$, tak potom riadky r_0 a r_1 v r majú rôzne hodnoty na S , takže funkčná závislosť $S \rightarrow T$ je splnená triviálne (v takom prípade nezáleží na tom, či sa riadky r_0 a r_1 zhodujú na T).

Takže v r je splnená každá závislosť, ktorá je dôsledkom F. **QED**

Uzáver množiny funkčných závislostí

Definícia. Označme F^+ je množinu všetkých funkčných závislostí, ktoré sú dôsledkom funkčných závislostí z F (t.j. ktoré sa dajú odvodiť z F použitím Armstrongovych axióm). Množinu F^+ budeme nazývať **uzáverom množiny funkčných závislostí F**

Množina F^+ je príliš rozsiahla. Stačí však uvádzat len maximálne závislosti. Definícia: **funkčná závislosť je maximálna**, ak nemôžeme vyniechať žiadnen atribút z jej ľavej strany ani pridať nejaký atribút k pravej strane bez porušenia jej platnosti

K charakterizácii F^+ je stačí dokonca uviesť len pravé strany maximálnych funkčných závislostí. Týmto pravým stranám sa hovorí **nasýtené množiny**

Pokrytie množiny funkčných závislostí

Definícia. Hovoríme, že množina funkčných závislostí G pokrýva množinu funkčných závislostí F práve vtedy, ak $G^+ \supseteq F^+$

Testovanie pokrytia podľa tejto definície je exponenciálne zložité.
(lebo uzávery G^+ , resp. F^+ môžu obsahovať exponenciálne veľa funkčných závislostí).

Stačí však testovať, či každú funkčnú závislosť z F možno odvodiť v G a naopak. Testovanie pokrytia je teda polynomiálne

Príklad:

$\{AB \rightarrow AC, B \rightarrow A, C \rightarrow B\}$ pokrýva $\{AB \rightarrow C, C \rightarrow A\}$, ale nie naopak

Minimálne pokrytie množiny funkčných závislostí

Definícia. Funkčná závislosť sa nazýva kanonická práve vtedy, ak má na pravej strane práve jeden atribút a na ľavej strane nemožno vynechať žiadnen atribút bez porušenia jej platnosti

Definícia. Minimálne pokrytie množiny funkčných závislostí je pokrytie množinou kanonických funkčných závislostí, z ktorých žiadna sa nedá vynechať bez toho, aby sa porušila vlastnosť byť pokrytím

Algoritmus výpočtu (nejakého) minimálneho pokrycia je polynomiálny vzhľadom na počet vstupných závislostí

Minimálne pokrytie množiny funkčných závislostí

Príklad:

$AB \rightarrow C, D \rightarrow E, CG \rightarrow B, C \rightarrow A, D \rightarrow G, CG \rightarrow D, BC \rightarrow D,$
 $BE \rightarrow C, CE \rightarrow A, ACD \rightarrow B, CE \rightarrow G$

Minimálne pokrycia (vždy existuje aspoň jedno minimálne pokrytie, no môže ich existovať viac ako jedno):

$AB \rightarrow C, C \rightarrow A, BC \rightarrow D, D \rightarrow E, D \rightarrow G, BE \rightarrow C, CE \rightarrow G,$
 $CD \rightarrow B, CG \rightarrow D$

$AB \rightarrow C, C \rightarrow A, BC \rightarrow D, D \rightarrow E, D \rightarrow G, BE \rightarrow C, CE \rightarrow G,$
 $CG \rightarrow B$

Výpočet minimálneho pokrycia množiny funkč. závislostí

Algoritmus výpočtu minimálneho pokrycia množiny funkčných závislostí F:

1. Nahrad' $X \rightarrow Y$ množinou $\{X \rightarrow A, A \in Y, A \text{ je jednoduchý atribút}\}$
2. Vynechaj všetky redundantné atribúty na ľavých stranách $X \rightarrow A$ (každý atribút treba testovať práve raz)
3. Vynechaj všetky redundantné závislosti $X \rightarrow A$ (opakuj tento krok s redukovanou množinou funkčných závislostí, kým žiadna závislosť nie je redundantná, každú závislosť treba testovať práve raz)

Krok 1 je triviálny

V kroku 2, pre každý atribút $B \in X$ sa vypočíta uzáver $(X-B)^+$ s použitím F. Ak $A \in (X-B)^+$, tak atribút B je redundantný

V kroku 3 sa vypočíta uzáver X^+ , ale len s použitím závislostí $F - \{X \rightarrow A\}$. Ak $A \in X^+$, tak závislosť $X \rightarrow A$ je redundantná

Výpočet minimálneho pokrycia množiny funkč. závislostí

F:

$$\begin{array}{llll} AB \rightarrow C & ACD \rightarrow B & \textcolor{red}{CG \rightarrow BD} & C \rightarrow A \\ \textcolor{blue}{D \rightarrow EG} & \textcolor{teal}{CE \rightarrow AG} & BC \rightarrow D & BE \rightarrow C \end{array}$$

Krok 1:

$$\begin{array}{llll} AB \rightarrow C & ACD \rightarrow B & \textcolor{red}{CG \rightarrow B} & \textcolor{red}{CG \rightarrow D} \\ C \rightarrow A & \textcolor{blue}{D \rightarrow E} & \textcolor{blue}{D \rightarrow G} & \textcolor{teal}{CE \rightarrow A} \\ \textcolor{teal}{CE \rightarrow G} & BC \rightarrow D & BE \rightarrow C & \end{array}$$

Krok 2:

- Závislosť $ACD \rightarrow B$ sa nahradí $CD \rightarrow B$, lebo $B \in \{CD\}^+$ (kedže platí $C \rightarrow A$ a $ACD \rightarrow B$)
- Závislosť $CE \rightarrow A$ sa nahradí $C \rightarrow A$, lebo $A \in \{C\}^+$ (kedže platí $C \rightarrow A$)
- Žiadna ľavá strana sa už nedá skrátiť

Krok 3:

$AB \rightarrow C, CD \rightarrow B, CG \rightarrow B, CG \rightarrow D, C \rightarrow A, D \rightarrow E, D \rightarrow G,$
 $C \rightarrow A, CE \rightarrow G, BC \rightarrow D, BE \rightarrow C$

- Závislosť $\mathbf{C} \rightarrow A$ je redundantná, lebo $C \rightarrow A$ je tam dvakrát
- Závislosť $\mathbf{CG} \rightarrow B$ je redundantná, lebo $CG \rightarrow D, C \rightarrow A, CD \rightarrow B$, takže $B \in \{CG\}^+$, aj keď k výpočtu uzáveru nepoužijeme závislosť $CG \rightarrow B$

Minimálne pokrytie:

$AB \rightarrow C$	$CD \rightarrow B$	$CG \rightarrow D$
$C \rightarrow A$	$D \rightarrow E$	$D \rightarrow G$
$CE \rightarrow G$	$BC \rightarrow D$	$BE \rightarrow C$

(Iné minimálne pokrytie dostaneme, ak v kroku 3 vynecháme
 $CD \rightarrow B, CG \rightarrow D, CE \rightarrow A.$)

Definícia. Nech r je relácia nad množinou atribútov **U**. Potom množinu atribútov **K** takú, že $K \rightarrow U$, nazývame **nadklúč relácie r** (superkey, candidate key). Minimálny nadklúč v zmysle množinovej inklúzie sa nazýva **kľúč (key)**

Príklad: Nech v $r(A, B, C, D, E, F, G, H)$ platia funkčné závislosti

$A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow GH$, $ACDF \rightarrow EG$

Jediným kľúčom je ACDF, lebo ACDF nie sú na pravej strane žiadnej závislosti (takže musia patriť do každého kľúča) a všetky ostatné atribúty patria do uzáveru $\{ACDF\}^+$

Výpočet všetkých kľúčov

Algoritmus zdola nahor:

Generuj lexikograficky vzostupne (počínajúc prázdnou množinou atribútov) **všetky podmnožiny atribútov, pre každú podmnožinu over, či je kľúčom** (t.j. over, či uzáver tej podmnožiny obsahuje všetky atribúty). Ak áno, vypíš kľúč a ďalej túto podmnožinu nerozširuj

Algoritmus zhora nadol je efektívnejší (*lebo pri nájdení kľúča už netreba overovať či sú podmnožiny kľúčom*):

Generuj lexikograficky zostupne (počínajúc celou množinou atribútov) **všetky podmnožiny atribútov**. Ak uzáver niektornej podmnožiny obsahuje všetky atribúty, vypíš kľúč a ďalej túto podmnožinu neredučuj

Pre nájdenie všetkých kľúčov neexistuje lepší algoritmus ako exponenciálny, lebo kľúčov môže byť exponenciálne veľa

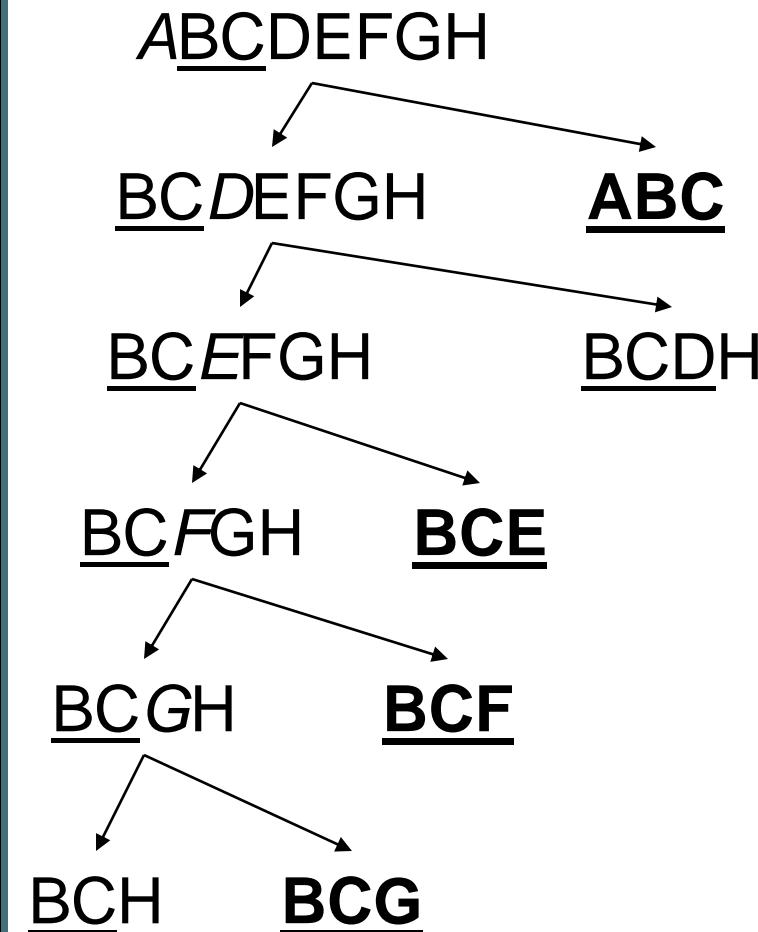
Praktický tip:

Atribúty, ktoré nie sú na pravej strane žiadnej funkčnej závislosti, musia byť v každom kľúči

Výpočet všetkých kľúčov zhora nadol: príklad

$\{G \rightarrow F, F \rightarrow A, AC \rightarrow E, F \rightarrow H, AH \rightarrow F, E \rightarrow G, H \rightarrow D, BF \rightarrow G\}$

Atribúty B a C musia byť v každom kľúči, lebo nie sú na žiadnej pravej strane



Prehľadávanie do hĺbky (backtrack):

V ľavej vetve sa odstráni 1 atribút, v pravej vetve je ten atribút v každom kľúči (je podčiarknutý)

Optimalizácie:

- Pred prehľadávaním stromu sa oplatí urobiť uzáver z množiny podčiarknutých atribútov—ak sú podčiarknuté atribúty nadkľúčom, netreba strom vôbec prehľadávať
- V ľavej vetve treba overiť, či uzáver nadľalej obsahuje ten atribút, ktorý sa v tej vetve vynecháva
- Po nájdení nejakého kľúča netreba ďalej prehľadávať podstromy, ktoré ten kľúč obsahujú

Dekompozícia relačnej schémy

Definícia. Množinu atribútov relácie r spolu s množinou funkčných závislostí, ktoré platia v r nazývame **relačná schéma**.

Definícia. **Dekompozícia relačnej schémy** $(r(U), F)$ je množina $(r_1, F_1), \dots, (r_n, F_n)$, kde každá z relácií r_1, \dots, r_n je projekciou r na nejakej podmnožine atribútov r , pričom zjednotenie atribútov r_1, \dots, r_n je **U**, a zároveň F pokrýva všetky F_1, \dots, F_n (t.j. dekompozíciou nevznikajú žiadne nové funkčné závislosti).

Definícia. **Dekompozícia** $(r_1, F_1), \dots, (r_n, F_n)$ relačnej schémy (r, F) je **bezstratová** (spája sa bezstratovo) práve vtedy, ak platí

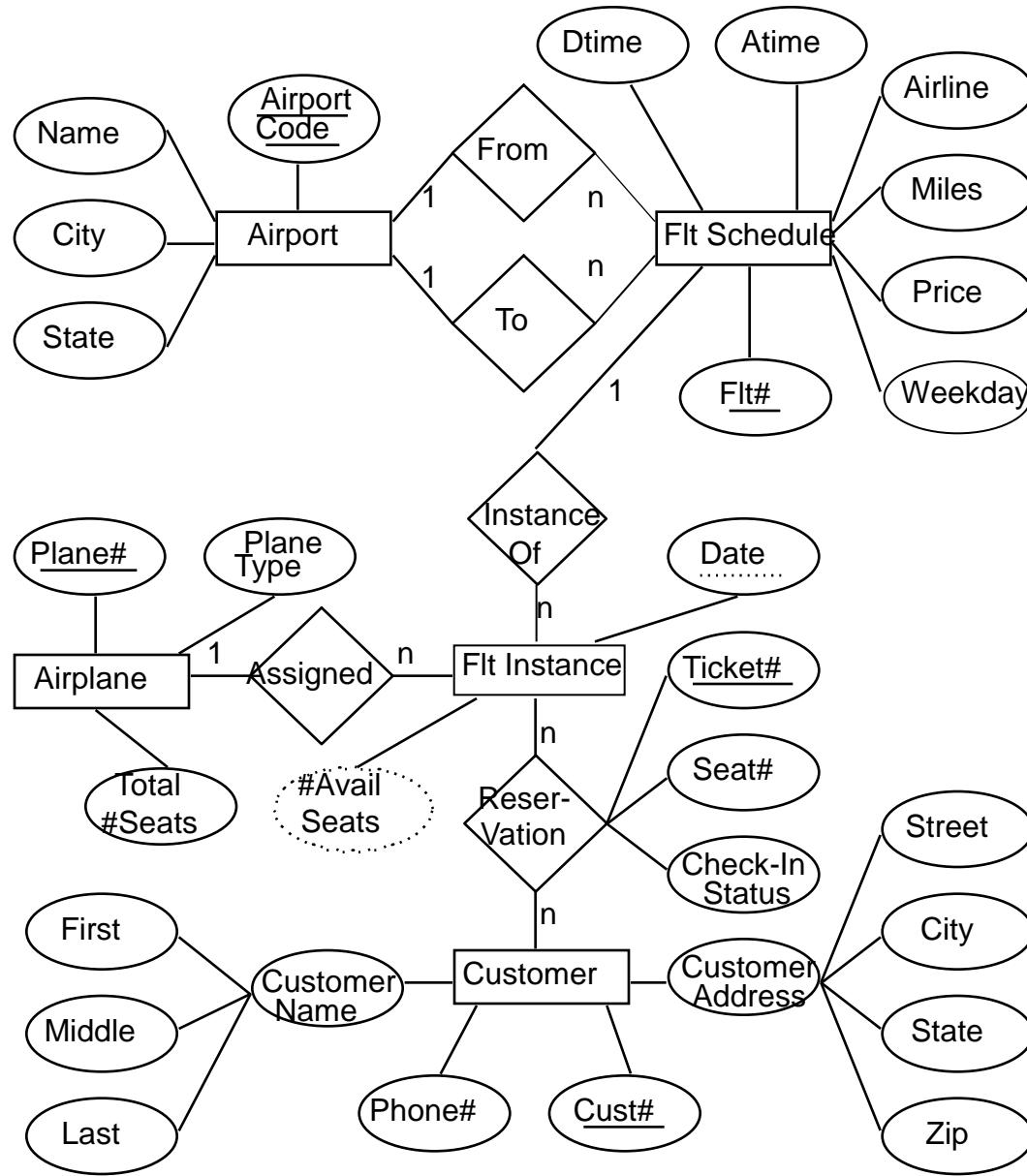
$$r = \Pi_{r_1}(r) \bowtie \Pi_{r_2}(r) \bowtie \dots \bowtie \Pi_{r_n}(r) \text{ pre každú populáciu relácie } r$$

The Four Commandments:

- Thou Shalt Commit No Redundancy of Fact
- Thou Shalt Clutter No Facts
- Thou Shalt Preserve Information
- Thou Shalt Preserve Functional Dependencies



Identifikácia funkčných závislostí (Lee Mark)



AIRPORT \leftrightarrow Airportcode

FLT-SCHEDULE \leftrightarrow Flt#

FLT-INSTANCE \leftrightarrow (Flt#, Date)

AIRPLANE \leftrightarrow Plane#

CUSTOMER \leftrightarrow Cust#

RESERVATION \leftrightarrow (Cust#, Flt#, Date)

RESERVATION \leftrightarrow Ticket#

Airportcode \rightarrow name, City, State

Flt# \rightarrow Airline, Dtime, Atime, Miles, Price, (from) Airportcode, (to) Airportcode

(Flt#, Date) \rightarrow Flt#, Date, Plane#

(Cust#, Flt#, Date) \rightarrow Cust#, Flt#, Date, Ticket#, Seat#, CheckInStatus,

Ticket# \rightarrow Cust#, Flt#, Date

Cust# \rightarrow CustomerName, CustomerAddress, Phone#

Normalizácia (Lee Mark)

BAD

FLT-SCHEDULE

flt#	weekday	airline	dtime	from	atime	to
DL242	MO WE FR	DELTA	10:40	ATL	12:30	BOS
SK912	SA SU	SAS	12:00	CPH	15:30	JFK
AA242	MO FR	AA	08:00	CHI	10:10	ATL

Attributes must be defined over domains with atomic values (1NF)

FLT-SCHEDULE

flt#	weekday	airline	dtime	from	atime	to
DL242	MO	DELTA	10:40	ATL	12:30	BOS
DL242	WE	DELTA	10:40	ATL	12:30	BOS
DL242	FR	DELTA	10:40	ATL	12:30	BOS
SK912	SA	SAS	12:00	CPH	15:30	JFK
SK912	SU	SAS	12:00	CPH	15:30	JFK
AA242	MO	AA	08:00	CHI	10:10	ATL
AA242	FR	AA	08:00	CHI	10:10	ATL

BETTER

Normalizácia (Lee Mark): Redundancy of Fact

BAD

FLIGHTS

flt#	date	airline	plane#
DL242	10/23/00	Delta	k-yo-33297
DL242	10/24/00	Delta	t-up-73356
DL242	10/25/00	Delta	o-ge-98722
AA121	10/24/00	American	p-rw-84663
AA121	10/25/00	American	q-yg-98237
AA411	10/22/00	American	h-fe-65748

- **redundancy:** airline name repeated for same flight
- **inconsistency:** when airline name for a flight changes, it must (perhaps) be changed in many places

Normalizácia (Lee Mark): Fact Clutter

BAD

FLIGHTS			
flt#	date	airline	plane#
DL242	10/23/00	Delta	k-yo-33297
DL242	10/24/00	Delta	t-up-73356
DL242	10/25/00	Delta	o-ge-98722
AA121	10/24/00	American	p-rw-84663
AA121	10/25/00	American	q-yg-98237
AA411	10/22/00	American	h-fe-65748

- **insertion anomalies:** how do we represent that SK912 is flown by Scandinavian without there being a date and a plane assigned?
- **deletion anomalies:** when we cancel AA411 on 10/22/00, we lose information that AA411 is flown by American (in other weeks).
- **update anomalies:** if DL242 is flown by Sabena, we must change it everywhere.

Normalizácia (Lee Mark): Information Loss

DATE-AIRLINE-PLANE

date	airline	plane#
10/23/00	Delta	k-yo-33297
10/24/00	Delta	t-up-73356
10/25/00	Delta	o-ge-98722
10/24/00	American	p-rw-84663
10/25/00	American	q-yg-98237
10/22/00	American	h-fe-65748

BAD

FLIGHTS-AIRLINE

flt#	airline
DL242	Delta
AA121	American
AA411	American

FLIGHTS, original

flt#	date	airline	plane#
DL242	10/23/00	Delta	k-yo-33297
DL242	10/24/00	Delta	t-up-73356
DL242	10/25/00	Delta	o-ge-98722
AA121	10/24/00	American	p-rw-84663
AA121	10/25/00	American	q-yg-98237
AA411	10/22/00	American	h-fe-65748

•information

loss: we polluted the database with false facts; we can't find the true facts.

FLIGHTS, joined

flt#	date	airline	plane#
DL242	10/23/00	Delta	k-yo-33297
DL242	10/24/00	Delta	t-up-73356
DL242	10/25/00	Delta	o-ge-98722
AA121	10/24/00	American	p-rw-84663
AA121	10/25/00	American	q-yg-98237
AA211	10/22/00	American	h-fe-65748
AA411	10/24/00	American	p-rw-84663
AA411	10/25/00	American	q-yg-98237
AA411	10/22/00	American	h-fe-65748

Normalizácia (Lee Mark): Dependency Loss

BAD

FLIGHTS-AIRLINE	
flt#	airline
DL242	Delta
AA121	American
AA411	American

DATE-AIRLINE-PLANE		
date	airline	plane#
10/23/00	Delta	k-yo-33297
10/24/00	Delta	t-up-73356
10/25/00	Delta	o-ge-98722
10/24/00	American	p-rw-84663
10/25/00	American	q-yg-98237
10/22/00	American	h-fe-65748

- **dependency loss:** we lost the fact that
 $(\text{flt}\#, \text{date}) \rightarrow \text{plane}\#$

Normalizácia (Lee Mark): Good Database Design

GOOD

FLIGHTS-AIRLINE	
flt#	airline
DL242	Delta
AA121	American
AA411	American

FLIGHTS-DATE-PLANE		
flt#	date	plane#
DL242	10/23/00	k-yo-33297
DL242	10/24/00	t-up-73356
DL242	10/25/00	o-ge-98722
AA121	10/24/00	p-rw-84663
AA121	10/25/00	q-yg-98237
AA411	10/22/00	h-fe-65748

- **no redundancy of *FACT* (!)**
- **no inconsistency**
- **no insertion, deletion or update anomalies**
- **no information loss**
- **no dependency loss**

Redundancia: anomálie pri vkladaní

student

Snumber	Sname	Pnumber	Pname
s1	tamara	p1	tomas
s2	jozef	p2	jan

Ako pridáme profesora, ktorý (zatiaľ) nemá žiadnych študentov?
Anomália: toto sa nedá bez použitia NULLs

Redundancia: anomálie pri vyneschávaní

student

<u>Snumber</u>	Sname	Pnumber	Pname
s1	tamara	p1	tomas
s2	jozef	p2	jan

Ked' vynescháme študenta, máme zmazať riadok alebo nahradíť informácie o študentovi NULLs?

Ak zmažeme riadok, stratíme (občas) kompletnú informáciu o profesorovi

Ak nahradíme študenta NULLs, tak začnú vznikať duplikáty (hoci v skutočnosti to nie sú doslova duplikáty, lebo hodnoty NULL sú navzájom neporovnatelné)

Redundancia: anomálie pri modifikácii

student

Snumber	Sname	Pnumber	Pname
s3	peter	p1	toms
s1	tamara	p1	tamas
s2	jozef	p2	tamas
s2	jozef	p1	tamas

Ak zmeníme meno profesora (napríklad chceme odstrániť preklep a premenovať profesora toms, resp. tamas na tomas), tak musíme zmenu urobiť vo veľa riadkoch

A čo je horšie, **niekedy nesmieme tú zmenu urobiť**. Napríklad Tamas môže byť správne meno profesora s Pnumber=2, takže pri opravovaní preklepov v Pname sa musíme dívať tiež na pNumber

Úloha funkčných závislostí pri detekcii redundancie

Uvažujme $r(A, B, C)$

- Ak v r neplatí žiadna funkčná závislosť, tak v r nie je žiadna redundancia vzhľadom na funkčné závislosti
- Ak v $r(A, B, C)$ platí $A \rightarrow B$ (ale neplatí $A \rightarrow C$), tak niektoré riadky môžu mať rovnakú hodnotu A . Lenže potom budú mať aj rovnakú hodnotu B (ale rôzne hodnoty v C). Toto je problém
 - Ten problém sa dá odstrániť **dekompozíciou do $r_1(A, B)$ a $r_2(A, C)$** . V r_1 už závislosť $A \rightarrow B$ nevadí, keďže do r_1 nikdy nepridávame duplikát už existujúceho riadku. Ideou normalizácie je „správna“ dekompozícia (nasledujúca prednáška hovorí o tom, čo znamená „správna“)

Úvod do databázových systémov

[**http://www.dcs.fmph.uniba.sk/~plachetk**](http://www.dcs.fmph.uniba.sk/~plachetk)
/TEACHING/DB2013

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2013–2014

Dekompozícia zachovávajúca funkčné závislosti

Definícia. Relačná schéma (r, F) je relácia r spolu s množinou funkčných závislostí F , ktoré platia v r

(Množina F indukuje ďalšie funkčné závislosti platné v r .)

Definícia. Hovoríme že dekompozícia $(r_1, F_1), \dots, (r_n, F_n)$ zachováva funkčné závislosti schémy (r, F) , ak každá platná funkčná závislosť $X \rightarrow Y$ z F je v uzávere platných funkčných závislostí z F_i , t.j. $F^+ = (\cup F_i)^+$

Aby sme dekompozíciu vôbec považovali za rozumnú, tak sa musí **spájať bezstratovo**; a zároveň by mala (ak je to možné), zachovávať funkčné závislosti pôvodnej relačnej schémy

Test bezstratovosti rozkladu pre dve relácie

Veta. Rozklad do 2 tabuľiek je bezstratový, ak platí, že joinovacia množina atribútov je nadklúč v r_1 alebo v r_2

Algoritmus testovania bezstratovosti rozkladu (polynomiálna časová zložitosť):

- Over, či platí v r funkčná závislosť $r_1 \cap r_2 \rightarrow r_1$ alebo $r_1 \cap r_2 \rightarrow r_2$. Ak platí aspoň jedna z nich, tak je rozklad bezstratový, inak je stratový

Pozor! Tento test nefunguje pre dekompozíciu do 3 či viac relácií!

Pre dekompozíciu do 3 a viac relácií treba použiť komplikovanejší test (chase), avšak stále s polynomiálnou časovou zložitosťou

Test bezstratovosti rozkladu pre N relácií (chase)

Input: Dekompozícia schémy r , F do r_1, \dots, r_m

1. Vyrob maticu S s 1 riadkom pre každú podreláciu r_1, \dots, r_m
a s 1 stĺpcom pre každý atribút r

2. Nastav počiatočné hodnoty, $S[i, j] := „b_{i,j}“$

3. for ($i = 1 \dots m$)

 for ($j = 1 \dots n$)

 if (atribút A_j patrí do r_i) then $S[i, j] := „a_j“$

4. Opakuj kým sa niečo mení:

 for all ($X \rightarrow Y \in F$)

 for (všetky riadky S s rovnakými symbolmi v tých stĺpcoch, ktoré zodpovedajú atribútom v X)

 Nastav všetky symboly v každom stĺpci, ktorý zodpovedá atribútu z Y takto:

 Ak niektorý riadok obsahuje aspoň jeden symbol „ a_* “, tak nastav hodnoty v celom stĺpci na tento (ľubovoľný 1) symbol.

 Inak vyber z riadku ľubovoľný symbol „ $b_{*,*}$ “ a nastav hodnoty v celom stĺpci na tento symbol.

5. Ak jeden riadok obsahuje len symboly „ a_* “, rozklad je bezstratový, inak nie.

Test bezstratovosti rozkladu: príklad

Príklad: $r(A, B, C, D, E, F, G, H)$

$A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow GH$, $ACDF \rightarrow EG$

Dekompozícia:

$r_1(AB)$, $r_2(ACDE)$, $r_3(EFG)$, $r_4(EFH)$.

Spája sa táto dekompozícia bezstratovo?

Ked'že ide o dekompozíciu do viac ako 2 tabuľiek, treba použiť tabuľkovú metódu (tzv. chase)

	A	B	C	D	E	F	G	H
AB	a1	a2	b13	b14	b15	b16	b17	b18
ACDE	a1	b22	a3	a4	a5	b26	b27	b28
EFG	b31	b32	b33	b34	a5	a6	a7	b38
EFH	b41	b42	b43	b44	a5	a6	b47	a8

Test bezstratovosti rozkladu: príklad, pokračovanie

$A \rightarrow B$

A	B	C	D	E	F	G	H
a1	a2	b13	b14	b15	b16	b17	b18
a1	a2	a3	a4	a5	b26	b27	b28
b31	b32	b33	b34	a5	a6	a7	b38
b41	b42	b43	b44	a5	a6	b47	a8

$ABCD \rightarrow E$

A	B	C	D	E	F	G	H
a1	a2	b13	b14	b15	b16	b17	b18
a1	a2	a3	a4	a5	b26	b27	b28
b31	b32	b33	b34	a5	a6	a7	b38
b41	b42	b43	b44	a5	a6	b47	a8

Test bezstratovosti rozkladu: príklad, pokračovanie

$EF \rightarrow GH$

A	B	C	D	E	F	G	H
a1	a2	b13	b14	b15	b16	b17	b18
a1	a2	a3	a4	a5	b26	b27	b28
b31	b32	b33	b34	a5	a6	a7	a8
b41	b42	b43	b44	a5	a6	a7	a8

$ACDF \rightarrow EG$

A	B	C	D	E	F	G	H
a1	a2	b13	b14	b15	b16	b17	b18
a1	a2	a3	a4	a5	b26	b27	b28
b31	b32	b33	b34	a5	a6	a7	a8
b41	b42	b43	b44	a5	a6	a7	a8

Dekompozícia AB, ACDE, EFG, EFH sa nespája bezstratovo

Test bezstratovosti rozkladu: iný príklad

Iný príklad: $r(A, B, C, D, E, F, G, H)$

$A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow GH$, $ACDF \rightarrow EG$

Dekompozícia:

$r_1(AB)$, $r_2(ACDE)$, $r_3(EFG)$, $r_4(EFH)$, $r_5(ACDF)$.

Spája sa táto dekompozícia bezstratovo?

	A	B	C	D	E	F	G	H
AB	a1	a2	b13	b14	b15	b16	b17	b18
ACDE	a1	b22	a3	a4	a5	b26	b27	b28
EFG	b31	b32	b33	b34	a5	a6	a7	b38
EFH	b41	b42	b43	b44	a5	a6	b47	a8
ACDF	a1	b52	a3	a4	b55	a6	b57	b58

Test bezstratovosti rozkladu: iný príklad, pokračovanie

$A \rightarrow B$

A	B	C	D	E	F	G	H
a1	a2	b13	b14	b15	b16	b17	b18
a1	a2	a3	a4	a5	b26	b27	b28
b31	b32	b33	b34	a5	a6	a7	b38
b41	b42	b43	b44	a5	a6	b47	a8
a1	a2	a3	a4	b55	a6	b57	b58

$ABCD \rightarrow E$

A	B	C	D	E	F	G	H
a1	a2	b13	b14	b15	b16	b17	b18
a1	a2	a3	a4	a5	b26	b27	b28
b31	b32	b33	b34	a5	a6	a7	b38
b41	b42	b43	b44	a5	a6	b47	a8
a1	a2	a3	a4	a5	a6	b57	b58

Test bezstratovosti rozkladu: iný príklad, pokračovanie

$EF \rightarrow GH$

A	B	C	D	E	F	G	H
a1	a2	b13	b14	b15	b16	b17	b18
a1	a2	a3	a4	a5	b26	b27	b28
b31	b32	b33	b34	a5	a6	a7	a8
b41	b42	b43	b44	a5	a6	a7	a8
a1	a2	a3	a4	a5	a6	a7	a8

**Dekompozícia AB, ACDE, EFG, EFH, ACDF
sa spája bezstratovo, lebo posledný riadok obsahuje iba
symboly a***

Prvá normálna forma (1NF)

Definícia. Relačná schéma (r , F) je v prvej normálnej forme (1NF) práve vtedy ak žiadene z jej atribútov r nie je zložený atribút a databáza neobsahuje duplikáty

Pokladáme za samozrejmosť, že atribúty nemajú štruktúru, t.j. žiadene atribúty nie sú zložené z viacerých podatribútov

Príklad schémy,
ktorá **nie je v 1NF**

FLT-SCHEDULE

flt#	weekday	airline	dtime	from	atime	to
DL242	MO WE FR	DELTA	10:40	ATL	12:30	BOS
SK912	SA SU	SAS	12:00	CPH	15:30	JFK
AA242	MO FR	AA	08:00	CHI	10:10	ATL

Druhá normálna forma (2NF)

Definícia. Relačná schéma (r, F) je v druhej normálnej forme (2NF) práve vtedy ak je v 1NF a ak v nej neexistuje platná funkčná závislosť $X \rightarrow Y$, kde X je striktná podmnožina nejakého klúča a Y nepatrí do žiadneho klúča.

Príklad schémy, ktorá **nie je v 2NF**. Jediný klúč je dvojica $[flt\#, date]$, ale airline a plane# funkčne závisia len od flt#.

FLIGHTS				
flt#	date	airline	plane#	
DL242	10/23/00	Delta	k-yo-33297	
DL242	10/24/00	Delta	t-up-73356	
DL242	10/25/00	Delta	o-ge-98722	
AA121	10/24/00	American	p-rw-84663	
AA121	10/25/00	American	q-yg-98237	
AA411	10/22/00	American	h-fe-65748	

Tretia normálna forma (3NF)

Definícia. Relačná schéma (r, F) je v tretej normálnej forme (3NF) práve vtedy, ak je v 2NF a zároveň pre každú platnú funkčnú závislosť $X \rightarrow Y$ (okrem prípadov $Y \subseteq X$) platí, že buď X je nadklúč v r alebo Y je časťou nejakého kľúča v r .

“Everything depends on the key, the whole key and nothing but the key (so help me Codd).”

Platí:

- Každá binárna relácia je v 3NF
- Ak r nie je v 3NF, tak existujú atribúty A a B také, že
 $r - AB \rightarrow A$

Tretia normálna forma (3NF)

Testovanie, či je schéma v 3NF je NP-ťažké. Avšak rozkladanie schémy do nejakej 3NF, ktoré je bezstratové a zachováva všetky funkčné závislosti, má polynomiálnu časovú zložitosť (vzhľadom na počet atribútov a počet závislostí relačnej schémy).

(Daňou za redukciu časovej zložitosťi je zbytočná resp.
„neprirodzená“ dekompozícia tabuliek.)

Príklad schémy, ktorá **nie je v 3NF**.
planetype funkčne závisí od plane#,
ale plane# nie je nadklúč, a zároveň
planetype nie je časťou žiadneho
klúča (jediným klúčom je flt#).

FLT-PLANE-PLANETYPE

flt#	plane#	planetype
DL242	k-yo-33297	A320
DL242	k-yo-33297	A320
DL242	o-ge-98722	LC4
AA121	k-yo-33297	A320
AA121	q-yg-98237	B747
AA411	q-yg-98237	B747

Boyce-Coddova normálna forma (BCNF)

Definícia. Relačná schéma (r, F) je v Boyce-Coddovej normálnej forme (BCNF) práve vtedy (ak je v 2NF a zároveň) ak pre každú platnú funkčnú závislosť $X \rightarrow Y$ platí, že X je nadklúč v r

Platí:

- Každá binárna relácia je v BCNF.
- Ak r nie je v BCNF, tak existujú atribúty A a B také, že
 $r - AB \rightarrow A$

Boyce-Coddova normálna forma (BCNF)

Testovanie, či je schéma v BCNF, je NP-ťažké. Bezstratový rozklad do nejakej BCNF sa dá nájsť v polynomiálnom čase s použitím nasledujúceho algoritmu, ale negarantuje zachovanie všetkých funkčných závislostí.

Je NP-ťažké rozhodnúť, či vôbec existuje nejaká BCNF dekompozícia, ktorá zachováva všetky funkčné závislosti.

Relačná schéma, ktorá je v BCNF, neobsahuje žiadnu redundanciu vzhľadom na funkčné závislosti.

Naivná dekompozícia do 3NF, resp. BCNF

1. Nájdi funkčnú závislosť z F^+ , ktorá porušuje podmienku $*NF$.
2. Minimalizuj jej ľavú stranu.
3. Maximalizuj jej pravú stranu (nie je nutné).
4. Nech výsledok je $X \rightarrow Y$.
5. Vytvor dve nové relácie: $r - Y$ a XY .
6. So vzniknutými reláciami proces opakuj, až kým všetky nie sú v požadovanej $*NF$.

Tento algoritmus má pre 3NF a BCNF **exponenciálnu časovú zložitosť** (vzhľadom na počet atribútov a počet závislostí relačnej schémy).

Iná naivná dekompozícia do 3NF, resp. BCNF

1. Nájdi atribúty A a B také, že $r - AB \rightarrow A$.
2. Vytvor dve nové relácie: $r - AB$ a $r - B$.
3. So vzniknutými reláciami proces opakuj, až kým všetky nie sú v požadovanej *NF.

Tento algoritmus má pre 3NF a BCNF **polynomiálnu časovú zložitosť** (vzhľadom na počet atribútov a počet závislostí relačnej schémy), ale niekedy **zbytočne dekomponuje** aj reláciu, ktorá už je v 3NF resp. BCNF.

Dekompozícia do 3NF zachovávajúca funkčné závislosti

Vstup: Relačná schéma r a **minimálne pokrytie** závislostí F

Výstup: Relačné schémy bestratovej dekompozície do 3NF, pričom všetky funkčné závislosti ostanú zachované

Dekompozícia do 3NF zachovávajúca funkčné závislosti:

Ak F obsahuje závislosť, ktorá obsahuje všetky atribúty r, potom r je už v 3NF

Inak každej funkčnej závislosti v F zodpovedá jedna relačná schéma

Ak je niektorá podschéma podmnožinou inej, treba ju vyniechať

Ak žiadna z tých relačných podschém neobsahuje klúč r, treba ešte pridať schému s nejakým klúčom. (Testovanie, či podschéma obsahuje nejaký klúč r, nevyžaduje výpočet klúčov. Stačí overiť, či atribúty niektornej podschémy sú nadklúčom v r. Ak áno, tak tá podschéma obsahuje nejaký klúč r, inak neobsahuje.)

Dekompozícia do 3NF zachovávajúca funkčné závislosti

Príklad:

V relácii $r(A, B, C, D, E, F, G, H)$ platia funkčné závislosti
 $A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow GH$, $ACDF \rightarrow EG$.

Minimálne pokrytie: $A \rightarrow B$, $ACD \rightarrow E$, $EF \rightarrow G$, $EF \rightarrow H$.

r nie je v 3NF, lebo platí $A \rightarrow B$ a A nie je nadklúč ani B nie je časťou žiadneho klúča (jediným klúčom je $ACDF$).

Bezstratová dekompozícia do 3NF zachovávajúca funkčné závislosti:

$r_1=\{AB\}$ /* zachováva $A \rightarrow B$ */

$r_2=\{ACDE\}$ /* zachováva $ACD \rightarrow E$ */

$r_3=\{EFG\}$ $r_4=\{EFH\}$ /* zachováva $EF \rightarrow G$ a $EF \rightarrow H$ */

$r_5=\{ACDF\}$ /* jediný klúč $ACDF$ sa nevyskytuje v žiadnej predošej relácii */

**r₅ treba “umelo“ pridať k reláciám indukovaným minimálnym pokrytím,
inak sa dekompozícia nebude spájať bezstratovo!**

Dekompozícia do BCNF

Pri dekompozícii sa treba snažiť zachovať funkčné závislosti
(i keď nie vždy sa to dá), kvôli automatizácii integrity

Algoritmus dekompozície relácie R do BCNF (je prirodzené začať s dobrou 3NF dekompozíciou):

1. Dekomponuj r do 3NF so zachovaním funkčných závislostí.
2. Over, či každá relácia dekompozície je v BCNF. Ak nie, nájdi v nej funkčnú závislosť $X \rightarrow Y$ ktorá porušuje BCNF a dekomponuj reláciu r_i do dvoch relácií, $r_i - Y$ a XY .
3. Opakuj overovanie a rozkladanie až kým sú všetky relácie v BCNF.

Dekompozícia do BCNF

Príklad:

V relácii $r(A, B, C, D, E, F, G, H)$ platia funkčné závislosti

$A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow GH$, $ACDF \rightarrow EG$.

Minimálne pokrytie: $A \rightarrow B$, $ACD \rightarrow E$, $EF \rightarrow G$, $EF \rightarrow H$.

Bezstratová dekompozícia do 3NF zachovávajúca funkčné závislosti:

$r_1 = \{AB\}$, $r_2 = \{ACDE\}$, $r_3 = \{EFGH\}$, $r_4 = \{ACDF\}$

Pre overenie BCNF treba najskôr vypočítať všetky netriviálne funkčné závislosti (ktorých ľavé strany neobsahujú zbytočné atribúty), ktoré platia v r a ktorých ľavé strany nie sú nadklúčom v r :

$A \rightarrow B$, $ACD \rightarrow E$, $EF \rightarrow G$, $EF \rightarrow H$.

Ľavé strany všetkých týchto funkčných závislostí z r sú nadklúčmi v r_1 , r_2 , r_3 , r_4 , preto je tá dekompozícia nielen v 3NF, ale aj v BCNF

Porušenie BCNF vs. zlomenie funkčných závislostí

Príklad:

adresy(Mesto, Ulica_Císlo, PSČ)

PSČ → Mesto

Mesto, Ulica_Císlo → PSČ

Táto schéma nie je v BCNF, lebo platí PSČ → Mesto, pričom PSČ nie je nadklúč. Dekompozícia do BCNF

(Mesto, PSČ), (Ulica_Císlo, PSČ) láme funkčnú závislosť

Mesto, Ulica_Císlo → PSČ

Lenže pozor, tá prvá funkčná závislosť nemusí platiť v každej krajine, napríklad na Slovensku neplatí celkom. **V tomto prípade je rozumné nerobiť žiadnu dekompozíciu**

Porušenie BCNF vs. zlomenie funkčných závislostí

Ešte jeden príklad schémy v 3NF, ale nie v BCNF (Elmasri, Navathe):

Student, Course → Instructor, Instructor → Course

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

Figure 10.13

A relation TEACH that
is in 3NF but not
BCNF.

Rozumná BCNF dekompozícia za predpokladu, že tá druhá závislosť skutočne platí:

(Student, Instructor), (Instructor, Course)

Ešte jeden príklad rozkladu do 3NF a BCNF

$r(A, B, C, D, E, F, G, H)$

$F \rightarrow A, A \rightarrow E, E \rightarrow B, E \rightarrow D, D \rightarrow H, BG \rightarrow F, CD \rightarrow A, BD \rightarrow E$

Toto je už minimálne pokrytie.

Kľúče v r sú CGA, CGB, CGD, CGE, CGF.

3NF rozklad z tohto min. pokrytia:

FA, AE, EB, ED, DH, BGF, CDA, BDE, **CGA**

Iný 3NF rozklad (všetky atribúty okrem H sú kľúčové, takže môžu byť v jednej tabuľke):

ABCDEFG, DH.

BCNF rozklad (pozor! platí $F \rightarrow B$ a tiež $A \rightarrow D$):

FA, AE, EB, ED, DH, **GF**, **BF**, **CA**, **AD**, BDE, CGA

(niektoré závislosti treba rozbit')

4NF a vyššie normálne formy

Vyššie normálne formy (4NF, 5NF, ...), odstraňujú redundanciu vzhľadom na **funkčné multizávislosti**

Konštrukcia 4NF a vyšších má **exponenciálnu časovú zložitosť**

Proces normalizácie konverguje k rozkladu do binárnych relácií, čo zvyčajne nechceme

Rozhodnutie kedy normalizáciu zastaviť, vyžaduje dlhodobú prax. Niekedy je dokonca vhodné databázu **denormalizovať**, t.j. spojiť niektoré rozbité tabuľky do jednej

S normalizáciou treba narábať veľmi opatrne, lebo zmena schémy znamená zmenu všetkého. „Zdravý“ počiatočný návrh je BCNF, vyrobená z 3NF, ktorá zachováva „čo najviac“ funkčných závislostí. Výnimkou z BCNF môžu byť pochybné závislosti typu $PSC \rightarrow Mesto$. (Ak BCNF líame funkčné závislosti, je možno rozumné ostať pri 3NF, prípadne snažiť sa vrobiť lepšiu 3NF/BCNF dekompozíciu.)

Úvod do databázových systémov

<http://www.dcs.fmph.uniba.sk/~plachetk>

/TEACHING/DB2013

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2013–2014

Niekoľko organizačných vecí

Odporúčaná literatúra k tejto prednáške (link na elektronickú verziu je na web stránke prednášky):

P.A. Bernstein, V. Hadzilacos, N. Goodman: Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987

Transakcie (z pohľadu používateľa)

Definícia (neformálna). Transakcia je program, ktorý pristupuje k databáze

Transakcií môže v tom istom čase bežat' viac, pričom jedna o druhej nevedia. Napríklad k bankovým kontám (dokonca aj k rovnakému kontu) môže pristupovať naraz operátor v banke aj klient pri bankomate. Alebo viacero klientov či operátorov môže naraz narábať s rezervačným systémom leteckej. Alebo viacero klientov môže narábať s knižničnou databázou

Pre jednoduchosť v tejto prednáške **budeme predpokladat', že databáza je centralizovaná.** (Z pohľadu transakcií je jedno, či je databáza centralizovaná alebo distribuovaná, avšak z hľadiska systému to jedno nie je. Implementácia distribuovaného systému prináša kvalitatívne nové problémy.)

Transakcie (z pohľadu databázového systému)

Definícia. **Transakcia** je postupnosť nasledujúcich operácií. Pritom **START** musí byť v tej postupnosti práve raz, a to na začiatku. **COMMIT** resp. **ABORT** musí byť v tej postupnosti práve raz, a to na konci (ak to tak nie je, systém transakciu abortuje)

READ: číta objekt (napr. záznam) z databázy

WRITE: zapisuje objekt do databázy

INSERT: vkladá objekt do databázy

DELETE: odstraňuje objekt z databázy

START: začína transakciu

COMMIT: úspešne končí transakciu

ABORT: neúspešne končí transakciu

Transakcie

Príklad: bankový prevod sumy S z účtu A na účet B
transfer(S, A, B)

{

float SA, SB;

START(transfer);

SA = **READ(A);**

SB = **READ(B);**

SA = SA – S;

SB = SB + S;

WRITE(A, SA);

WRITE(B, SB);

COMMIT(transfer);

}

Zjednodušený zápis:

s1, r1(A), r1(B), w1(A), w1(B), c1

- Súčasne môže bežať viacero inštancií transakcie *transfer*. Z hľadiska systému sú rôzne, takže systém im prideluje rôzne IDs (táto transakcia má ID 1)
- V prípade READ systém nevidí lokálnu premennú, do ktorej sa ukladá hodnota prečítaná z databázy, vidí len jej hodnotu
- V prípade WRITE systém nevidí lokálnu premennú, ktorej hodnota sa ukladá do databázy, vidí len tú hodnotu

Požiadavky na transakčný databázový systém: ACID

Atomicity: transakcia sa vykoná buď celá alebo vôbec

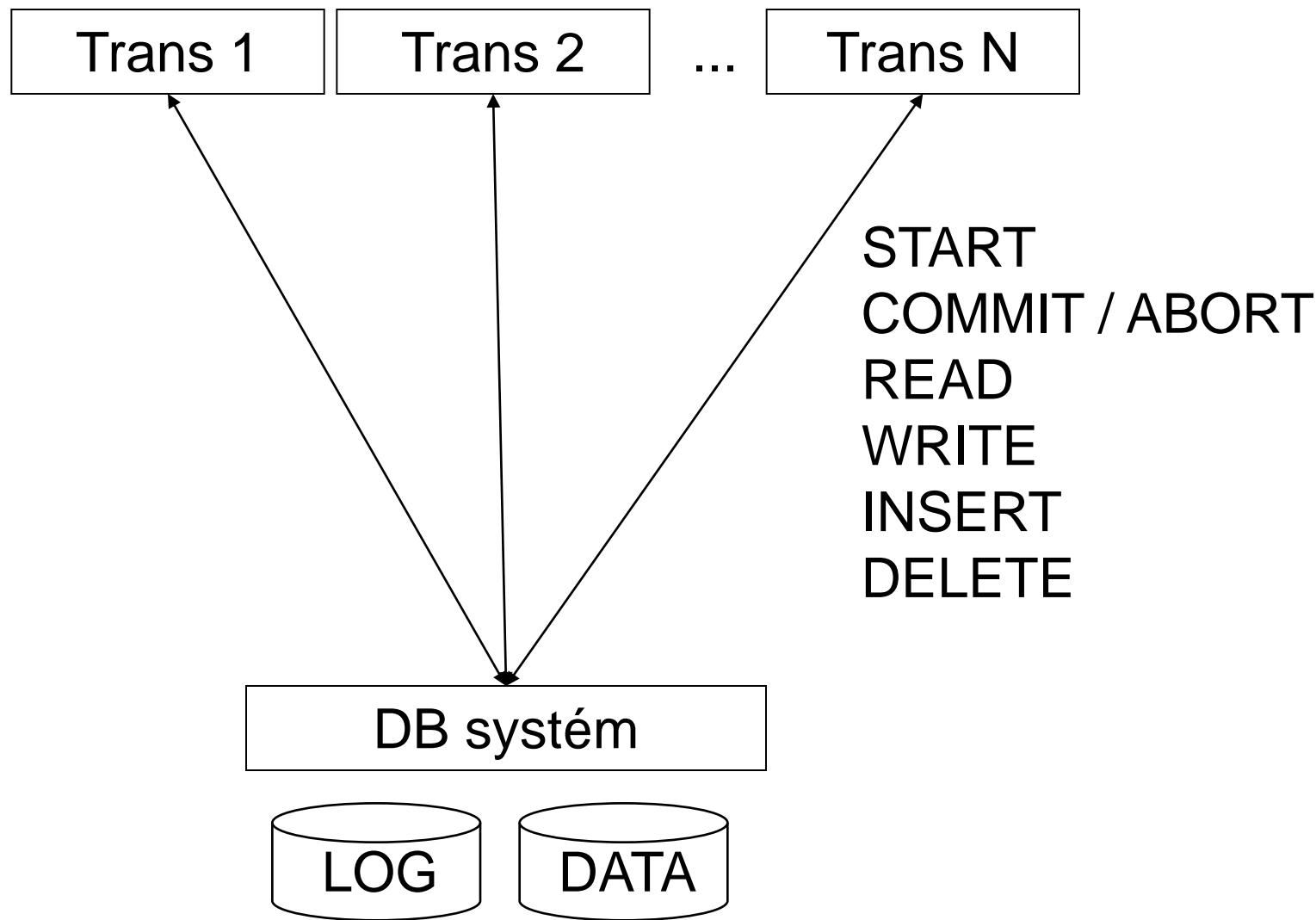
Consistency: vykonanie transakcie znamená prechod od konzistentného stavu databázy opäť ku konzistentnému stavu
(toto nie je požiadavka na systém, ale na implementorov transakcií)

Isolation: hoci systém môže vykonávať viacero transakcií „paralelne“, výsledný efekt musí byť taký, ako keby sa vykonávali celé transakcie sériovo (jedna po druhej)

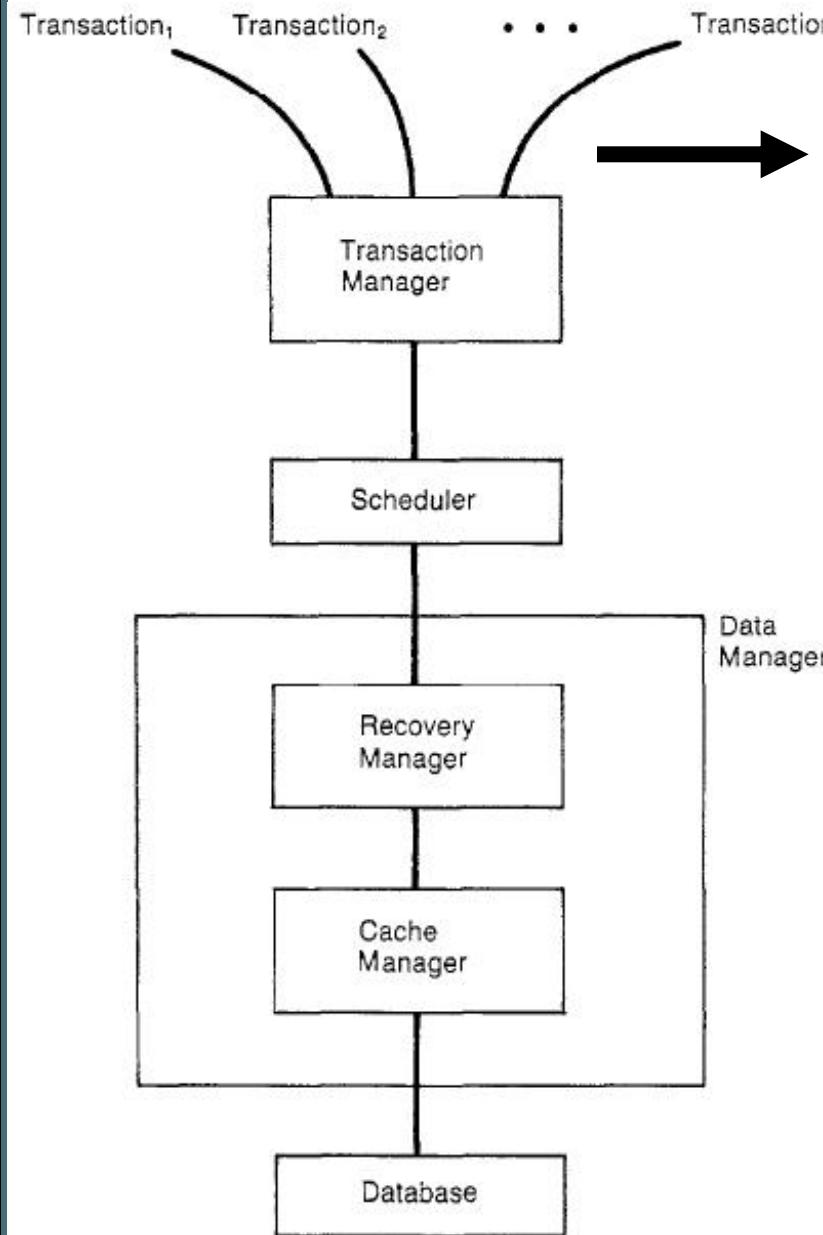
Durability: ak transakcia úspešne skončí (t.j. ak systém vykoná COMMIT), tak všetky zmeny, ktoré vykonala v databáze, budú navždy zachované

Toto všetko musí systém garantovať aj v prípade havárií, ako je napríklad nečakaný výpadok v ľubovoľnom momente (na strane klienta alebo servera alebo oboch súčasne)

Transakčný databázový systém: 2-tier architektúra



Transakčný databázový systém



- READ: číta objekt
WRITE: zapisuje do objektu
START: začína transakciu
COMMIT: úspešne končí transakciu
ABORT: neúspešne končí transakciu
INSERT: vkladá nový objekt
DELETE: odstraňuje objekt

Transaction manager sa stará o čítanie a bufferovanie operácií od transakcií a o autorizáciu transakcií

Scheduler rozhoduje o poradí vykonávania operácií. (Operácie nemusia byť vykonávané v tom poradí ako boli prečítané.)

Recovery manager vedie log-file a stará sa o to, aby v databáze boli aj v prípade výpadkov zapísané zmeny spôsobené commitovanými transakciami. A naopak, zabraňuje aby boli do databázy zapísané zmeny spôsobené abortovanými transakciami

Cache manager sa stará o to, aby nejaká časť disku (spoločlivá, veľká, pomalá pamäť) bola uložená v operačnej pamäti (nespoločlivá, malá, rýchla pamäť) a stará sa o synchronizáciu medzi diskom a operačnou pamäťou

Toto je abstraktný model: implementácia môže vyzerat' inak!

Dôvody pre ABORT transakcie

- **Výpadok na strane klienta** spôsobí ABORT transakcie
- **Výpadok na strane servera** spôsobí ABORT všetkých transakcií, ktoré sú v tom momente aktívne
- **Výpadok spojenia medzi klientom a serverom** spôsobí ABORT transakcie
- Užívateľ na strane klienta stlačí tlačítko „cancel“, čo spôsobí, že **transakcia sa sama rozhodne pre ABORT**
- **Systém (presnejšie, scheduler)** vykoná ABORT transakcie (napr. keď zistí, že COMMIT nebude môcť nikdy vykonat', lebo by tým porušil niektorú z ACID požiadaviek). **Samozrejme, cieľom systému je ABORTovať čo najmenej transakcií**

Rozvrhy (plány, histórie)

Definícia. Rozvrh (plán, história) je postupnosť, ktorá vznikne premiešaním operácií niekoľkých transakcií, vo všeobecnosti nekompletných. Toto premiešanie nie je ľubovoľné—zachováva poradie operácií jednotlivých transakcií (projekcia rozvrhu na individuálnu transakciu je postupnosť operácií tej transakcie) Rozvrh je kompletný, ak v ňom všetky transakcie končia buď operáciou COMMIT alebo ABORT

Transakcia je v danom čase aktívna, ak v tom čase už začala (t.j. bola vykonaná jej operácia START) a zároveň v tom čase ešte neskončila (t.j. nebola vykonaná jej operácia COMMIT či ABORT)

Scheduler transformuje vstupnú postupnosť operácií do rozvrhu. Na to používa operácie execute, delay a reject

„Dobré“ vs. „zlé“ rozvrhy

Príklad (Garcia-Molina): 2 transakcie

T1:

Read(A)
 $A \leftarrow A + 100$
Write(A)
Read(B)
 $B \leftarrow B + 100$
Write(B)

T2:

Read(A)
 $A \leftarrow A \times 2$
Write(A)
Read(B)
 $B \leftarrow B \times 2$
Write(B)

Constraint: $A=B$

Constraint: $A=B$

„Dobré“ vs. „zlé“ rozvrhy

Rozvrh A: dobrý

T1

Read(A); $A \leftarrow A + 100$

Write(A);

Read(B); $B \leftarrow B + 100;$

Write(B);

Commit;

T2

Read(A); $A \leftarrow A \times 2;$

Write(A);

Read(B); $B \leftarrow B \times 2;$

Write(B);

Commit;

A	B
25	25
125	
	125
250	
	250
250	250

„Dobré“ vs. „zlé“ rozvrhy

Rozvrh B: dobrý

T1

Read(A); A \leftarrow A+100
Write(A);
Read(B); B \leftarrow B+100;
Write(B);
Commit;

T2

Read(A); A \leftarrow A×2;
Write(A);
Read(B); B \leftarrow B×2;
Write(B);
Commit;

A	B
25	25
	50
	50
	150
	150
150	150

„Dobré“ vs. „zlé“ rozvrhy

Rozvrh C: dobrý

T1

Read(A); $A \leftarrow A + 100$

Write(A);

Read(B); $B \leftarrow B + 100$;

Write(B);

Commit;

T2

Read(A); $A \leftarrow A \times 2$;

Write(A);

Read(B); $B \leftarrow B \times 2$;

Write(B);

Commit;

A	B
25	25
	125
125	250
	125
250	250
	250
250	250

„Dobré“ vs. „zlé“ rozvrhy

Rozvrh D: zlý

T1

Read(A); $A \leftarrow A + 100$

Write(A);

Read(B); $B \leftarrow B + 100$;

Write(B);

Commit;

T2

Read(A); $A \leftarrow A \times 2$;

Write(A);

Read(B); $B \leftarrow B \times 2$;

Write(B);

Commit;

A	B
25	25
	125
	250
	50
	150
250	150

„Dobré“ vs. „zlé“ rozvrhy

Rozvrh E (rovnaký ako D, s mierne zmenenou T2): zlý, hoci...

T1

T2'

Read(A); A \leftarrow A+100

Write(A);

Read(B); B \leftarrow B+100;

Write(B);

Commit;

A	B
25	25
125	
125	
25	
125	
125	125

„Dobré“ vs. „zlé“ rozvrhy

Intuitívne, „dobré“ rozvrhy sú A, B, C

Rozvrh D nezachováva izoláciu

Rozvrh E zachováva izoláciu, ale len náhodou—spolieha na vnútornú štruktúru transakcie, resp. momentálny stav databázy

Rozvrhy A, B sú **sériové**

Rozvrh C je **sériovateľný** (presnejšie, **konflikt-sériovateľný**)

Rozvrhy D a E sú „zlé“, lebo nie sú konflikt-sériovateľné (E je len matematicky sériovateľný)

Konfliktné operácie a konflikt-ekvivalencia rozvrhov

Definícia. V histórii sú dve **operácie konfliktné**, ak patria rôznym transakciám, ich operandom je rovnaký objekt a aspoň jedna z tých operácií je *write*

Idea generovania „dobrých“ rozvrhov: sériu nekonfliktných operácií môžeme ľubovoľne premiešať a rozvrh ostane „dobrý“; ale sériu konfliktných operácií musíme zachovať

Definícia. Dve **histórie** sú **konflikt-ekvivalentné** práve vtedy, ak

- pozostávajú z rovnakých operácií a
- relatívne poradie každých dvoch konfliktných operácií je rovnaké v oboch históriách

Sériové a sériovateľné rozvrhy

Definícia. Rozvrh je sériový práve vtedy, ak je kompletný (t.j. každá transakcia v tom rozvrhu končí commitom alebo abortom) a pre každú dvojicu transakcií T1, T2 platí, že buď všetky operácie T1 v tom rozvrhu predchádzajú operáciám T2 alebo naopak

Definícia. Rozvrh je sériovateľný (konflikt-sériovateľný) práve vtedy, ak jeho projekcia na commitované transakcie je konflikt-ekvivalentná niektorému sériovému rozvrhu tých commitovaných transakcií

Prečo projekcia na commitované transakcie? Lebo len pre commitované transakcie dáva systém nejaké garancie!

Testovanie sériovateľnosti rozvrhov: precedenčný graf

Definícia. Nech S je rozvrh, ktorý obsahuje commitované transakcie T_1, \dots, T_n . **Precedenčný graf** je orientovaný graf s vrcholmi T_1, \dots, T_n , v ktorom je hrana $T_i \rightarrow T_j$ práve vtedy, ak O_i, O_j sú konfliktné operácie a O_i je v rozvrhu skôr ako O_j .

Príklad (rozvrh E): $r1(A), w1(A), r2(A), w2(A), r2(B), w2(B), c2, r1(B), w1(B), c1$

Read(A); $A \leftarrow A+100$

Write(A);

Read(A); $A \leftarrow A \times 1$;

Write(A);

Read(B); $B \leftarrow B \times 1$;

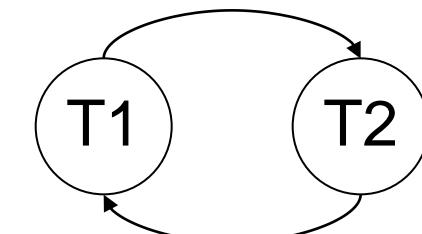
Write(B);

Commit;

Read(B); $B \leftarrow B+100$;

Write(B);

Commit;



Testovanie sériovateľnosti rozvrhov: precedenčný graf

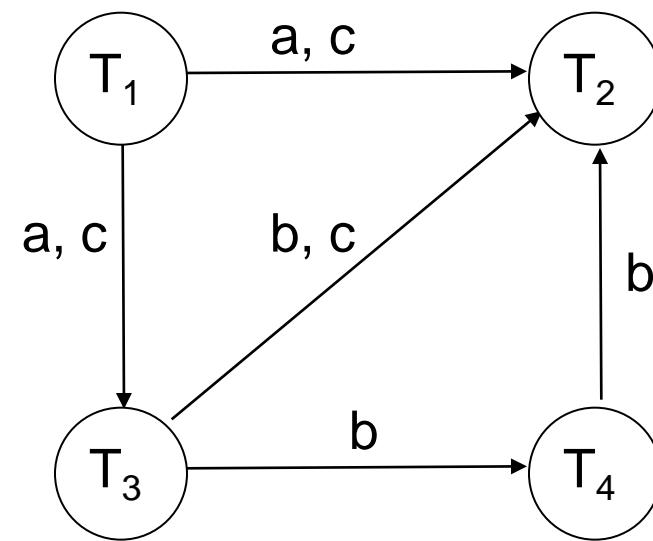
Veta. Rozvrh je sériovateľný práve vtedy, ak jeho precedenčný graf je acyklický

Veta. Ak je rozvrh sériovateľný, tak topologické usporiadanie jeho precedenčného grafu hovorí, **ktorému** sériovému rozvrhu je ten rozvrh ekvivalentný

Testovanie sériovateľnosti rozvrhov: precedenčný graf

Príklad

T_1	T_2	T_3	T_4
		read(b) write(b)	
read(a)	read(b)		
read(c)			write(b)
write(a)			
write(c)			
		read(a) write(c)	
		read(a)	
		write(c)	

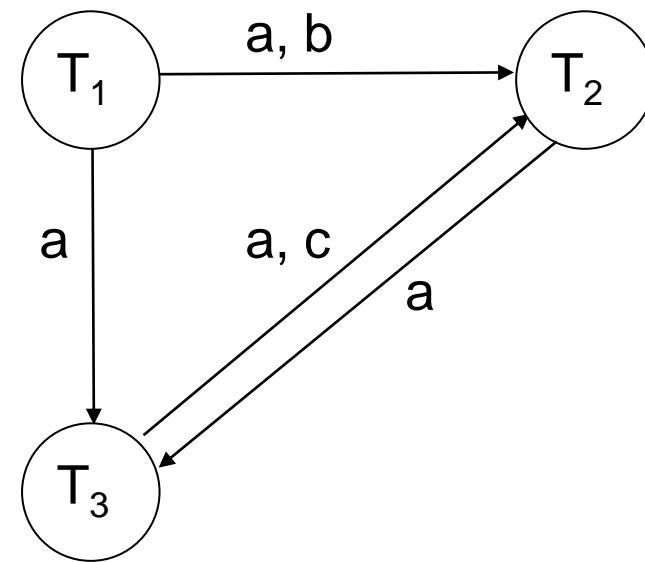


Precedenčný graf neobsahuje cyklus, takže rozvrh je konflikt-sériovateľný. Rozvrh je ekvivalentný sériovému rozvrhu
 $T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_2$

Testovanie sériovateľnosti rozvrhov: precedenčný graf

Príklad

T_1	T_2	T_3
read(a)		
read(b)		
write(a)		
	read(a)	
	read(b)	
		write(c)
	read(c)	
	write(b)	
	read(a)	
		write(a)
		write(c)
		write(a)



Precedenčný graf obsahuje cyklus $T_2 \rightarrow T_3 \rightarrow T_2$, takže rozvrh nie je konflikt-sériovateľný

View-sériovateľnosť

Ekvivalenciu rozvrhov je možné definovať aj iným spôsobom:

Definícia. Hovoríme, že v rozvrhu **transakcia T_2 číta X od transakcie T_1** práve vtedy, ak v tom rozvrhu existuje operácia $w_1(X)$ a neskôr operácia $r_2(X)$, pričom T_1 je v čase operácie $r_2(X)$ stále aktívna

Definícia. **Dve histórie H a H' sú view-ekvivalentné**, ak (sú definované nad tými istými transakciami a zároveň)

- pre každú dvojicu operácií v H , kde nejaká transakcia T_1 číta X od T_2 existuje taká istá dvojica operácií v H' , kde T_1 tiež číta X od T_2 , a zároveň

- pre každý dátový objekt X, ak transakcia T_i je posledná transakcia ktorá píše do X v H , tak aj v H' je T_i je posledná transakcia ktorá píše do X (final write)

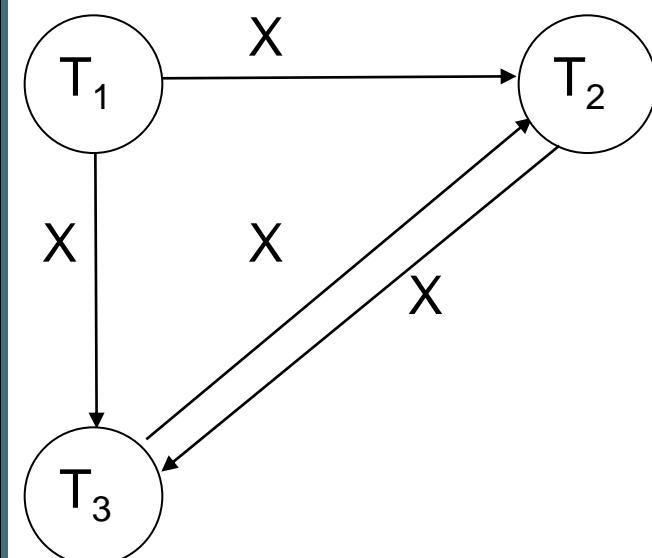
(Intuitívne, rozvrhy sú view-ekvivalentné, ak majú rovnaký efekt.)

View-sériovateľnosť

Definícia: rozvrh je **view-sériovateľný**, ak commitovaná projekcia každého jeho prefixu je view-ekvivalentná niektorému sériovému rozvrhu

Príklad: $w_1(X)$, $r_2(X)$, $w_3(X)$, $w_2(X)$, c_3 , c_2 , c_1

Tento rozvrh nie je konflikt-sériovateľný, ale je **view-sériovateľný** (je view-ekvivalentný sériovému rozvrhu $T_1 \rightarrow T_3 \rightarrow T_2$)



Platí: **ak je rozvrh konflikt-sériovateľný, tak je aj view-sériovateľný** (ale nie vždy to platí naopak)

Možno sa to nezdá, ale **testovanie, či je rozvrh view-sériovateľný, je NP-ťažký problém** (hoci testovanie view-ekvivalentnosti **dvoch** rozvrhov je ľahké, dokonca lineárne)

Vieme **testovať**, či je rozvrh konflikt-sériovateľný. Ale vieme zestrojiť online algoritmus, ktorý **generuje** iba konflikt-sériovateľné rozvrhy?

Odpoveď je áno. Triviálna možnosť je generovať *len sériové rozvrhy*, t.j. vykonávať len operácie jednej transakcie, kým tá transakcia neskončí (scheduler operácie ostatných transakcií dovtedy zdržiava a ukladá niekam do buffera). Táto možnosť je však zjavne nepraktická, lebo príliš redukuje priepustnosť systému. Generovať view-sériovateľné rozvrhy je ľahšie. Ale existuje hned niekoľko spôsobov pre generovanie konflikt-sériovateľných rozvrhov (presnejšie, nejakých podried konflikt-sériovateľných rozvrhov)

- **Algoritmus zámkov:** transakcie zamykajú dátové objekty pred každým čítaním či zápisom, aby iné transakcie s nimi nemohli interferovať
- **Algoritmus časových pečiatok:** na základe časových pečiatok sa pri každej operácii systém rozhoduje, či transakcia môže pokračovať alebo abortuje
- **Multiversion algoritmus:** každá transakcia zapisuje svoje zmeny do svojej lokálnej kópie databázy namiesto do ostrej databázy. Systém rozhodne až pri žiadosti o COMMIT, či transakcia commituje a lokálne zmeny sa prenesú do ostrej databázy, alebo či transakcia abortuje. Systém udržiava pre jeden dátový objekt viacero lokálnych verzií a pri operácii READ rozhoduje, ktorá verzia sa bude čítať
- **Validačný algoritmus:** systém **optimisticky** vykonáva operácie tak ako idú, ale prísne kontroluje, či dovolí COMMIT

Izolácia: Zamykanie (locking)

Idea: do rozvrhu sa pridajú operácie lock a unlock, ktorými sa zamykajú resp. odomykajú dátové objekty. Transakcia smie dátový objekt čítať/písat' len v momente, keď vlastní potrebný zámok na ten objekt. Commit je triviálny, netreba nič kontrolovať

Základné typy zámkov:

- Read-lock (RL): dovoľuje iba čítať'
- Write-lock (WL), exclusive-lock: dovoľuje **čítať aj písat'**

Granularita zámkov:

- (na jeden atribút jedného záznamu)
- na jeden záznam
- na celú tabuľku
- (na diskový blok)

Izolácia: Zamykanie (locking)

Kompatibilita zámkov:

	RL	WL
RL	true	false
WL	false	false

Inými slovami, transakcia smie získať RL na X aj vtedy, ak iná transakcia má v tej chvíli RL na X. Akákoľvek iná kombinácia zámkov v jednom momente je neprípustná a transakcia žiadajúca o zámok musí v takom prípade počkať, kým tá druhá transakcia svoj zámok uvoľní (operáciou unlock). To čakanie zabezpečí scheduler tak, že operáciu žiadosti o zámok (t.j. operáciu RL, resp. WL) jednoducho nevykoná, ale odloží jej vykonanie na neskôr

Dvojfázové zamykanie (two-phase locking)

Pravidlá:

1. Ak transakcia už niekedy urobila unlock, nesmie už nikdy žiadať o ďalší lock. (T.j. v prvej fáze každá transakcia získava zámky, v druhej fáze ich odovzdáva späť systému.) Scheduler si udržuje stav zámkov, ktoré pridelil a ľahko vie otestovať, či niektorá transakcia porušuje toto dvojfázové pravidlo
2. Transakcia musí vlastniť potrebný zámok keď sa pokúša o čítanie/písanie dátového objektu. Aj toto vie scheduler ľahko overiť

Ak niektorá transakcia poruší pravidlá hry, scheduler ju hned' abortuje (reject)

Po ukončení transakcie uvoľní scheduler zámky tej transakcie

Dvojfázové zamykanie (two-phase locking)

Veta: Two-phase locking generuje len (konflikt-) sériovateľné rozvrhy

Dôkaz: nepriamo. Nech je rozvrh dvojfázový a nech nie je sériovateľný. To druhé znamená, že precedenčný graf obsahuje cyklus $T_i \rightarrow \dots \rightarrow T_i$, kde T_i je prvá transakcia v rozvrhu, ktorá je v nejakom cykle. Lenže to znamená, že transakcia T_i musela niečo zamykať po tom, čo nejaký zámok uvoľnila, inak by sa nedostala do toho druhého konfliktu, ktorý spôsobil ten cyklus. To je spor s dvojfázovosťou rozvrhu. QED

Naopak to neplatí. Nie každý sériovateľný rozvrh sa dá generovať two-phase locking algoritmom. Napríklad:
 $r1(X), r2(X), r1(X), w2(X), c1, c2$

Iná (tiež pesimistická) metóda izolácie: časové pečiatky

Každá transakcia T_i dostane pri svojom štarte timestamp $TS(T_i)$.

Každý **objekt X** má dve pečiatky: jedna sa aktualizuje keď sa X číta, druhá sa aktualizuje keď sa do X zapisuje

Keď transakcia číta resp. píše do X, tak nechá v X svoju časovú pečiatku, ak je pečiatka transakcie novšia ako príslušná pečiatka pri X

Pravidlá (Bernstein a iní ich trošku zoslabujú) :

1. Transakcia nesmie čítať hodnoty, ktoré písala neskôr začatá transakcia
2. Transakcia nesmie písat' hodnoty, ktoré čítala alebo písala neskôr začatá transakcia

Zámky vs. časové pečiatky

	T_1	T_2
1		read b
2	read a	
3	write c	
4		write c

	T_1	T_2	T_3
1	read a		
2		read a	
3			read d
4			write d
5			write a
6		read c	
7		write b	
8			write b

Rozvrh S je sériovateľný zámkami, ale nie je sériovateľný časovými razítkami. Rozvrh T naopak.

Optimistická metóda izolácie: validácia

Systém vykonáva operácie tak, ako prichádzajú. Pre každú transakciu T sleduje tri časy: $TS(T)$, $TVAL(T)$, $TF(T)$ (pečiatka udalosti, ktorá ešte nenastala, je ∞), a dve množiny dát: read-set $RS(T)$ a write-set $WS(T)$. Transakcie nepíšu priamo do databázy, ale len do svojej lokálnej kópie. Vo chvíli, keď T požiada o commit, dostane pečiatku $TVAL(T)$ a začne sa jej validačná fáza. Ak je validácia úspešná, T dostane pečiatku $TF(T)$ a začne sa jej finálna fáza, t.j. T začne písat' do databázy. Inak je T abortovaná. Validácia T je neúspešná práve vtedy, ak existuje transakcia T_c , pre ktorú platí jedna z nasledujúcich podmienok:

$$(RS(T) \cap WS(T_c) \neq \emptyset) \wedge TVAL(T_c) < TVAL(T) \wedge TF(T_c) > TS(T),$$

$$(WS(T) \cap WS(T_c) \neq \emptyset) \wedge TVAL(T_c) < TVAL(T) \wedge TF(T_c) > TVAL(T)$$

Multiversion concurrency control (MVCC)

Každá transakcia T dostane pri svojom štarte timestamp $TS(T)$.

Každý **objekt** X (presnejšie, každá verzia X) má pečiatky X_r a X_w

Ked' transakcia T píše do X , tak vytvorí novú verziu objektu X , s $X_w = TS(T)$. Systém pamätá aspoň 2 posledné verzie každého objektu.

Ked' transakcia T číta z X , scheduler „podhodí“ na čítanie poslednú takú verziu X , pre ktorú platí $X_w < TS(T)$. Ďalej, ak pre túto verziu platí $TS(T) > X_r$, tak do X_r sa priradí $TS(T)$

Pri zápise objektu X od transakcie T systém kontroluje, či existuje nejaká verzia X s $X_w < TS(T)$ a zároveň $X_r > TS(T)$. Ak existuje, tak systém abortuje transakciu T (lebo čítajúca transakcia mala prečítať až verziu od T).

Multiversion concurrency control (MVCC)

Postgres a Oracle používajú MVCC, ktorý sa dostal do módy (lebo systém musí tak či tak pamätať aspoň 2 posledné verzie objektu). Pozor na úskalia! Citácia z manuálu PostgreSQL:

In fact PostgreSQL's Serializable mode does not guarantee serializable execution in mathematical sense. As an example, consider a table mytab, initially containing

class	value
-----+-----	
1	10
1	20
2	100
2	200

Suppose that serializable transaction A computes

`SELECT SUM(value) FROM mytab WHERE class = 1;`

and then inserts the result (30) as the value in a new row with class = 2.

Concurrently, serializable transaction B computes

`SELECT SUM(value) FROM mytab WHERE class = 2;`

and obtains the result 300, which it inserts in a new row with class = 1. Then both transactions commit. None of the listed undesirable behaviors have occurred, yet we have a result that could not have occurred in either order serially. If A had executed before B, B would have computed the sum 330, not 300, and similarly the other order would have resulted in a different sum computed by A.

Izolácia v praktických systémoch

Štandard ANSI/ISO SQL používa rétoriku odlišnú od teórie, na ktorej buduje. Definuje 4 stupne izolácie transakcií:
READ_UNCOMMITTED, READ_COMMITTED,
REPEATABLE_READ, **SERIALIZABLE** (najvyšší stupeň izolácie)
Ideou je umožniť aplikačným programátorom obetovať ACID garancie v záujme urýchlenia aplikácií.

H.Berenson, P.Bernstein, J.Gray, J.Melton, E.O'Neil, P.O'Neil, A Critique of ANSI SQL Isolation Levels, ACM SIGMOD'95

<http://research.microsoft.com/apps/pubs/default.aspx?id=69541>

Módny trend v IT reprezentuje rôznorodá skupina **NoSQL** (Not-Only-SQL) systémov, ktoré sa spravidla vyhýbajú použitiu relačného dátového modelu, resp. jazyka SQL. Majú skôr charakter distribuovaných file-systémov, špecializovaných na prácu s dátami typu „key-value“. Dôraz sa obvykle kladie na „agilitu“, konzistencia dát je druhoradá.

<http://nosql-database.org/>

<http://www.igvita.com/2010/03/01/schema-free-mysql-vs-nosql/>

Sériovateľnosť nestáčí. Príklad:

$r1(A)$, $w1(A)$, $r2(A)$, $w2(C)$, $c2 \downarrow c1$

Tento rozvrh je sériovateľný. Smie však scheduler vôbec vygenerovať takýto rozvrh? Predpokladajme, že nastane výpadok T_1 v momente, na ktorý ukazuje šípka—to nemôžeme vylúčiť.

Transakcia T_2 už bola commitovaná, takže jej efekt bol už navždy zapísaný do databázy (efekt operácie $w2(C)$ ostane v databáze aj po výpadku). Lenže transakcia T_1 , od ktorej T_2 čítala, musela byť abortovaná a efekt $w1(A)$ bol zo systému navždy odstránený. Tým pádom transakcia T_2 mohla prečítať neplatnú hodnotu objektu A, od ktorej mohla závisieť zapísaná hodnota objektu C.

Hrozila nekonzistencia! Len šťastnou náhodou aj T_1 commituje...

Cieľom recovery algoritmov je garantovať ACID aj v prípade neočakávaných výpadkov

Predpoklady:

- non-volatilné fyzické médiá (disky, RAID) sú spoľahlivé—ak nie, treba siahnuť k záložnej kópii (backup)
- zápis do non-volatilných médií je atomický aspoň na úrovni blokov (bloky majú konštantnú veľkosť, ale sú dostatočne veľké)
- transakcie sú vnútorne konzistentné
- výpadky je možné detektovať

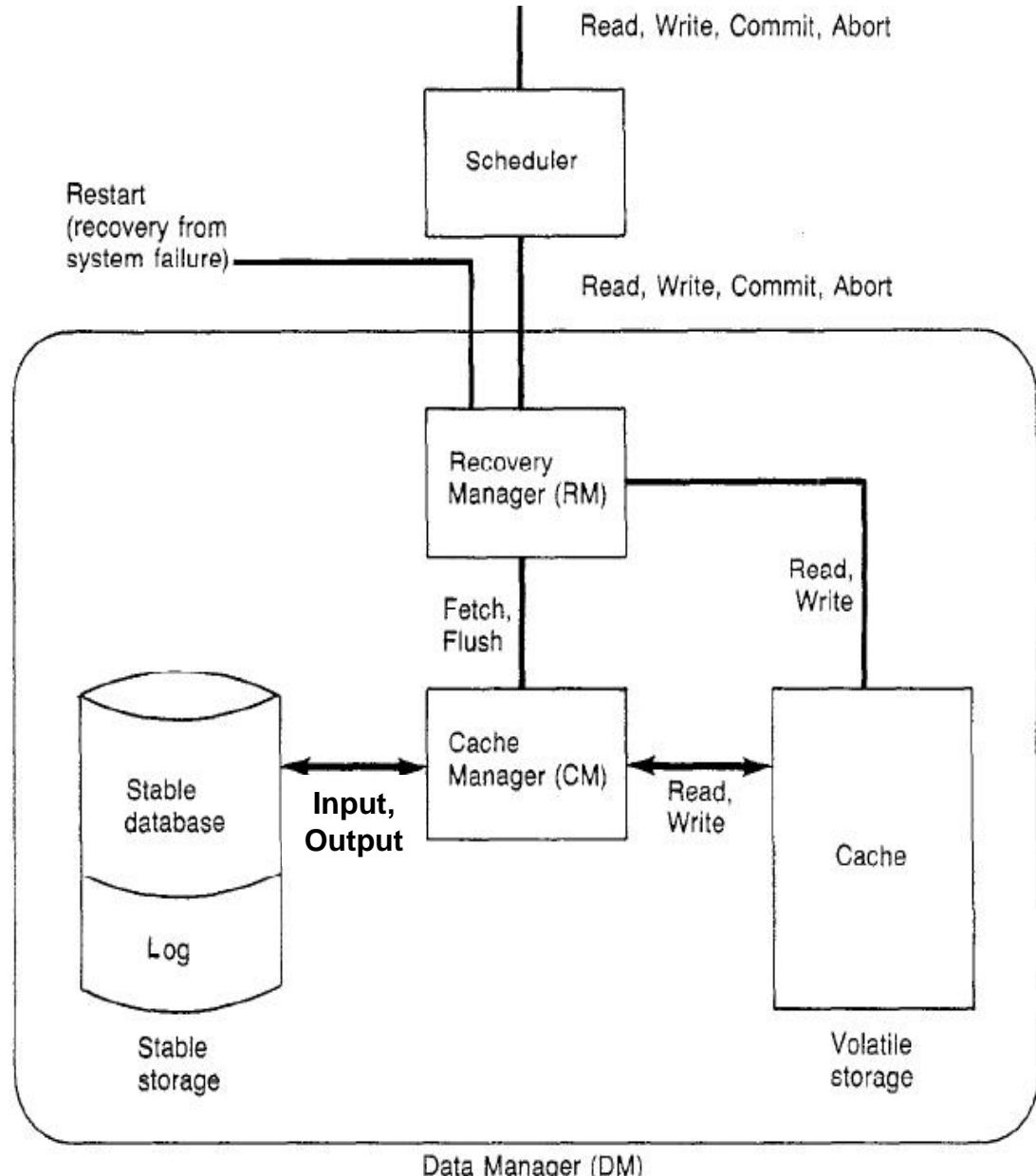
Recovery algoritmy sa skladajú z 2 častí:

- 1.Zbieranie informácií počas normálneho behu (**log**)
- 2.Reakcia na výpadok (**idempotentné operácie UNDO resp. REDO pre abortované resp. commitované transakcie**)

Obnova (recovery): Cache Manager

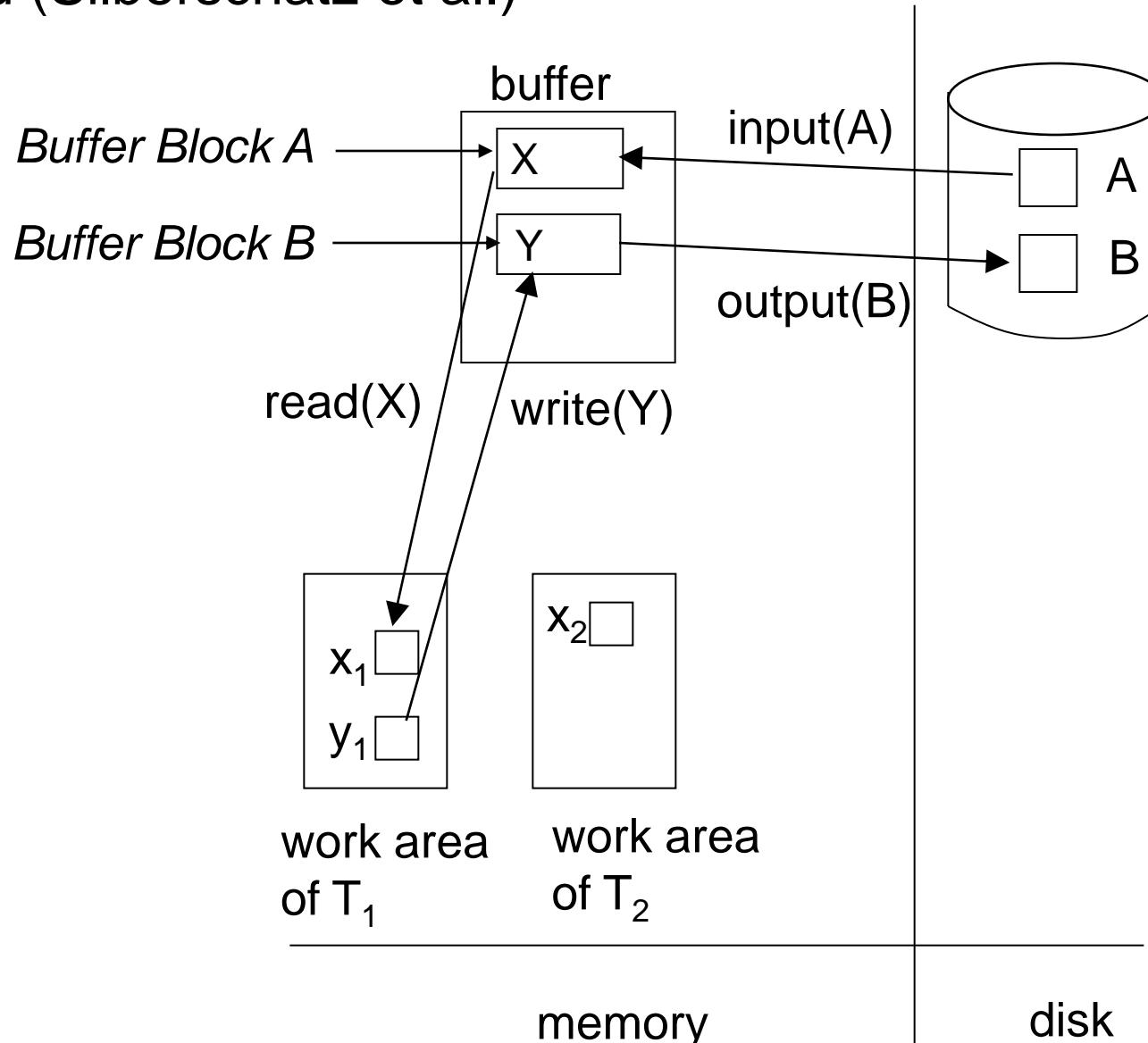
Fyzická optimalizácia transakčných operácií:

- Prvý $\text{read}(X)$ spôsobí čítanie X z disku (fetch/input), ďalšie $\text{read}(X)$ spôsobí čítanie z RAM (fetch/read)
- $\text{write}(X)$ nemusí spôsobiť okamžité písanie na disk (flush/output), ale iba do RAM (flush/write)



Organizácia prístupu k dátam: Cache Manager

Príklad (Silberschatz et al.)



Buffer management: Cache Manager

Blok v cache sa nesmie aktualizovať (a niekedy ani čítať) v momente, keď je zapisovaný na disk (output). Obyčajne sa o zápis bloku stará hardware (DMA). Je potrebná synchronizácia medzi hardwarom a softwarom (cache manager)

Riešenie: **pred prístupom k bloku je potrebný exkluzívny zámok, ktorý stačí držať na veľmi krátku chvíľu.** Moderný hardware ponúka takéto zámky, hovorí sa im **latch**

Niektoré operácie vyžadujú okamžitý zápis bloku na disk. Ostatné bloky ostávajú v pamäti až kým sa pamäť nenaplní. Potom sa pre výmenu blokov používajú techniky známe z operačných systémov (napr. LRU, FIFO)

„Transakcia prebehne buď úplne, alebo vôbec“

Stačí sa sústredit' **iba na operácie write** (read nemení stav dát).

Dokonca (teoreticky), pre každý objekt sa stačí sústredit' na posledný write poslednej commitovanej transakcie

Dva prístupy k implementácii atomicity:

1. **Log**
2. Shadow-paging

Log-file je sekvenčný súbor uložený na disku, ktorý odráža zmeny spôsobené transakciami. Zmeny v log-file sa okamžite zapíšu na disk (output). Garancia atomicity diskových prístupov je dôležitá najmä pre log-file

Filozofia „**write-ahead**“: skôr ako transakcia urobí zmenu v dátach, zamýšľaná zmena sa zapíše do log-file

Obsah log-file:

- Začiatok transakcie T_i : $\langle T_i \text{ start} \rangle$
- Write-operácie transakcie T_i : $\langle T_i, X, \text{old_value}, \text{new_value} \rangle$
- Commit transakcie T_i : $\langle T_i \text{ commit} \rangle$
- atď. (napr. **CHECKPOINT**, **DUMP**)

Okamžitý zápis dát (všeobecná, flexibilná stratégia):

- `write(X)` sa zapíše aj do log-file, aj do databázy. Cache manager rozhodne o tom, kedy sa objekt X skutočne zapíše do dát na disku, ale smie tak urobiť hoci aj hned'. Cache manager taktiež rozhoduje o *poradí* zápisov dát na disk
- Pred vykonaním operácie WRITE je dôležité zapísat' do log-file aj starú aj novú hodnotu zapisovaného objektu

Okamžitý zápis dát (immediate data modification)

- V prípade obnovy treba najprv robiť UNDO pre tie transakcie T_i , pre ktoré existuje záznam $\langle T_i \text{ start} \rangle$, ale neexistuje $\langle T_i \text{ commit} \rangle$. UNDO sa robí pri **zostupnom** čítaní log-file (od konca po začiatok). Počas UNDO prechodu systém zapisuje do databázy staré hodnoty objektov uložené v logu pri operáciách write; a vytvára 2 zoznamy identifikátorov transakcií: undo_list a redo_list (v redo_list sú commitované transakcie, v undo_list necommitované)
- Potom treba urobiť REDO pre tie transakcie v redo_list. REDO sa robí pri **vzostupnom** čítaní log-file (od začiatku po koniec)

Aj počas obnovy môže znova nastat' výpadok, ale to nevadí (vdľaka idempotentnosti REDO / UNDO). Obnova sa jednoducho zopakuje pri opäťovnom štarte systému

Okamžitý zápis dát (immediate data modification)

Príklad (Silberchatz et al.):

T0	T1
read(A)	read (C)
A = A - 50	C = C - 100
write(A, 950)	write (C)
read (B)	
B = B + 50	
write (B)	

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$	$\langle T_0, A, 1000, 950 \rangle$
$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$	$\langle T_0, B, 2000, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 700, 600 \rangle$	$\langle T_1, C, 700, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$

(a)

(b)

(c)

- UNDO(T_0): B:=2000, A:=1000
- UNDO(T_1): C:=700,
REDO(T_0): A:=950, B:=2050
- REDO(T_0): A:=950, B:=2050
REDO(T_1): C:=600

Oneskorený zápis dát (deferred data modification)

Oneskorený zápis dát (efektívna stratégia, ale s predpokladmi na cache):

- write(X) sa síce zapíše do log-file aj do databázy, ale nie na disk
- do databázy na disku sa zapisuje **vždy** až niekedy po commit transakcie
- **Do log-file netreba písat' staré hodnoty dát (before-images)**
- **Operáciu UNDO netreba vôbec implementovať**

Operáciu REDO treba vykonať len pre tie transakcie, pre ktoré sú v log-file záznamy $\langle T_i, \text{start} \rangle$ aj $\langle T_i, \text{commit} \rangle$. Algoritmus **REDO prechádza log-file raz, zosupne** (toto je rozdiel oproti všeobecnej stratégii). Pri tomto jedinom prechode sa vytvárajú dva zoznamy: redo_list a redone_list (s identifikátormi **objektov**). Systém pre každý záznam $\langle \text{commit } T_i \rangle$ pridá T_i do redo_list. Pre každý záznam $\langle T_i, X, \text{new_value} \rangle$ urobí systém toto:

```
if ( $T_i \in \text{redo\_list}$ ) AND ( $X \notin \text{redone\_list}$ ) then {  
    write(X, new_value);  
    redone_list = redone_list  $\cup$  X;  
}
```

Oneskorený zápis dát (deferred data modification)

Príklad (Silberchatz et al.):

T0	T1
read(A)	read (C)
A = A - 50	C = C - 100
write(A, 950)	write (C)
read (B)	
B:= B + 50	
write (B)	

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$
$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 600 \rangle$	$\langle T_1, C, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$

(a)

(b)

(c)

- (a) Žiadne REDO akcie
- (b) REDO(T_0) je potrebné, lebo log obsahuje $\langle T_0 \text{ commit} \rangle$: B:=2050, A:=950
- (c) REDO(T_1): C:=600 REDO(T_0): B:=2050, A:=950

Checkpointing je technika, ktorá pomáha redukovať dĺžku log-files a skracuje čas potrebný na obnovu systému. Nezávisle od spracovania transakčných operácií sa dá kedykoľvek spustiť nasledujúca **atomická** procedúra:

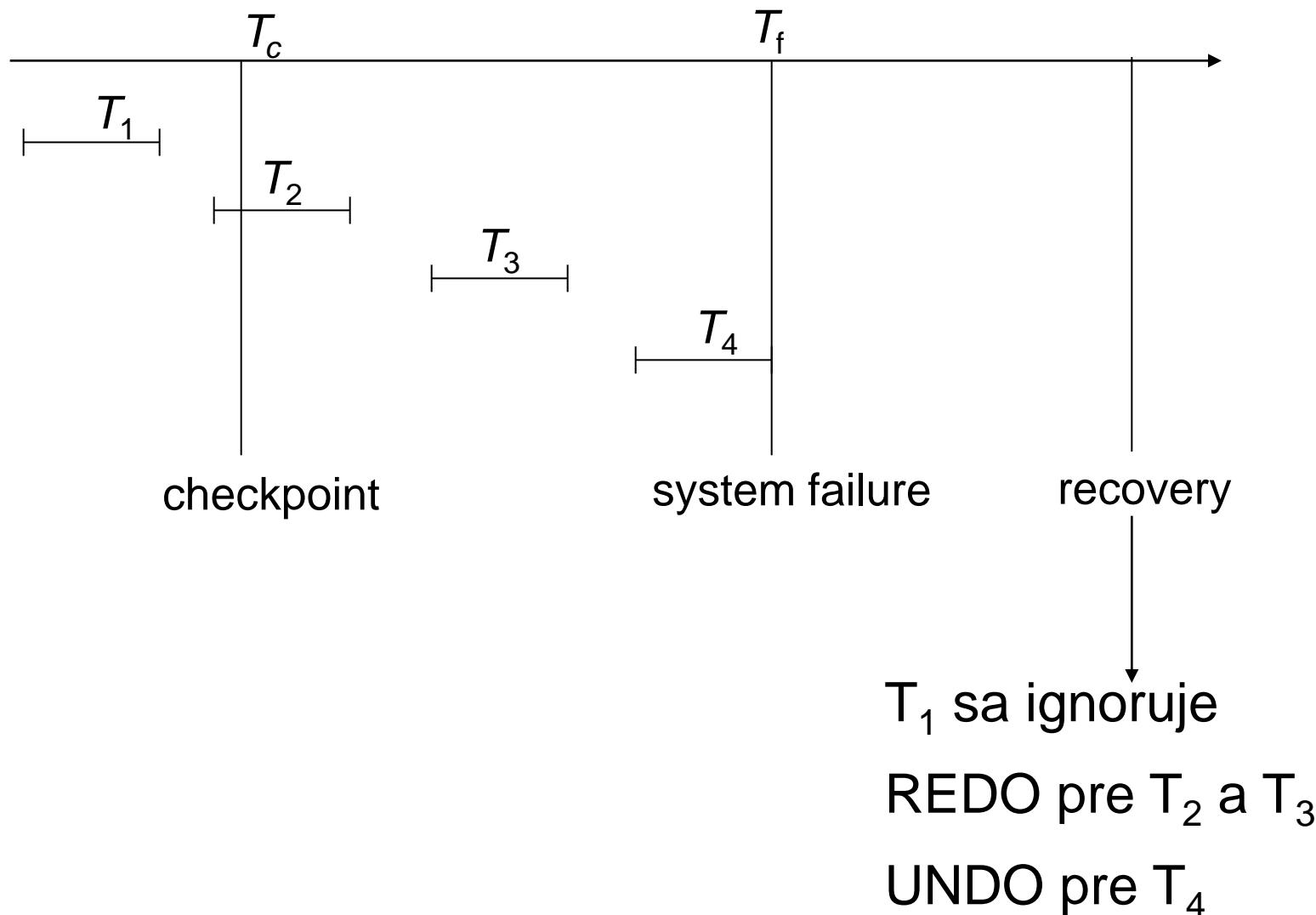
1. Zapíš všetky log-záznamy na disk
2. Zapíš všetky dátá na disk
3. Zapíš <CHECKPOINT> do log-file

Checkpointing: recovery (všeobecný algoritmus)

0. Nájdi (zostupne) v log-file posledný záznam <CHECKPOINT>, pritom vytváraj undo_list a redo_list a vykonávaj UNDO akcie
1. Nájdi zostupne najstarší záznam < T_i start> pred <CHECKPOINT> pre transakcie, ktoré boli aktívne v čase CHECKPOINT (tiež sa zastav ked' nájdeš iný CHECKPOINT, resp. začiatok logu), pritom vykonávaj UNDO akcie
2. Spusti (vzostupne) REDO-prechod, počínajúc záznamom nájdeným v kroku 1

Všimnite si, že časť log-file pred záznamom nájdeným v kroku 1 nie je pre obnovu dôležitá a možno ju zahodiť (garbage collection)

Checkpointing: príklad (Silberschatz et al.)



Checkpointing so zoznamom aktívnych transakcií

Systém tak či onak udržiava v pamäti zoznam aktívnych transakcií (napr. kvôli autentifikácii), takže je prirodzené rozšíriť záznam `<CHECKPOINT>` na `<CHECKPOINT L>`, kde L je zoznam aktívnych transakcií

Rekonštrukcia zoznamu aktívnych transakcií z log-file:

1. `undo_list:=∅`, `redo_list:=∅`
2. Prechádzaj zostupne log-file. Ked' nájdeš `<CHECKPOINT L>`, chod' na krok 3. Ked' nájdeš `<Ti commit>`, pridaj T_i do `redo_list`. Ked' nájdeš inú operáciu od T_i , a $T_i \notin redo_list$, tak pridaj T_i do `undo_list`
3. Pre všetky $T_i \in L$: ak $T_i \notin redo_list$, pridaj T_i do `undo_list`

Recovery algoritmus (všeobecný):

1. Prechádzaj log zostupne, vytváraj redo_list a undo_list a vykonávaj UNDO pre všetky transakcie v undo_list
2. Ak nájdeš <CHECKPOINT L>, aktualizuj zoznamy. Potom pokračuj v zostupe a vykonávaj UNDO pre všetky transakcie v undo_list. Zastav sa keď nájdeš $\langle T_i \text{ start} \rangle$ pre každú $T_i \notin \text{undo_list}$, alebo keď nájdeš iný CHECKPOINT
3. Prechádzaj log vzostupne až po koniec log-file. Pri tomto prechode rob vykonávaj REDO pre všetky transakcie v redo_list

Backup (dump)

Backup je procedúra, ktorá zvyšuje odolnosť voči výpadkom médií. Nezávisle od spracovania transakčných operácií sa dá kedykoľvek spustiť nasledujúca **atomická** procedúra:

1. (Spusti procedúru CHECKPOINT, skrát log-file)
2. Vytvor kópiu log-file a kópiu databázy na novom médiu
3. Zapíš <DUMP> do log-file a zapíš log na disk

Teória obnoviteľnosti

Definícia: Rozvrh je obnoviteľný (recoverable, RC) práve vtedy, keď pre každú commitovanú transakciu T2 čítajúcu necommitovanú hodnotu od T1 (dirty read) platí, že T1 commituje tiež—a zároveň commit T1 je v tom rozvrhu skôr ako commit T2

Motivácia je zrejmá. Napríklad R1: w1(X) r2(X) w2(Y) c2 c1 nie je obnoviteľný rozvrh, lebo po c2 mohol nasledovať abort transakcie T1. Ani toto nie je obnoviteľný rozvrh: R2: w1(X) r2(X) w2(Y) c2 a1

Rozvrh R3: w1(X) r2(X) w2(Y) c1 c2 je obnoviteľný. V rozvrhu R3 však hrozí iná nepríjemnosť: kaskádový abort (cascading abort). Ak sa v rozvrhu R2 transakcia T1 rozhodne pre abort, tak to spôsobí *následný vynútený abort* T2 (transakcia T2 nemôže už nikdy skončiť commitom, ak má byť ten rozvrh obnoviteľný)

Kaskádový abort môže byť vskutku „reťazová reakcia“, je lepšie sa mu vyhnúť:

T1 r(X) w(X) abort

T2 r(X) w(Y)

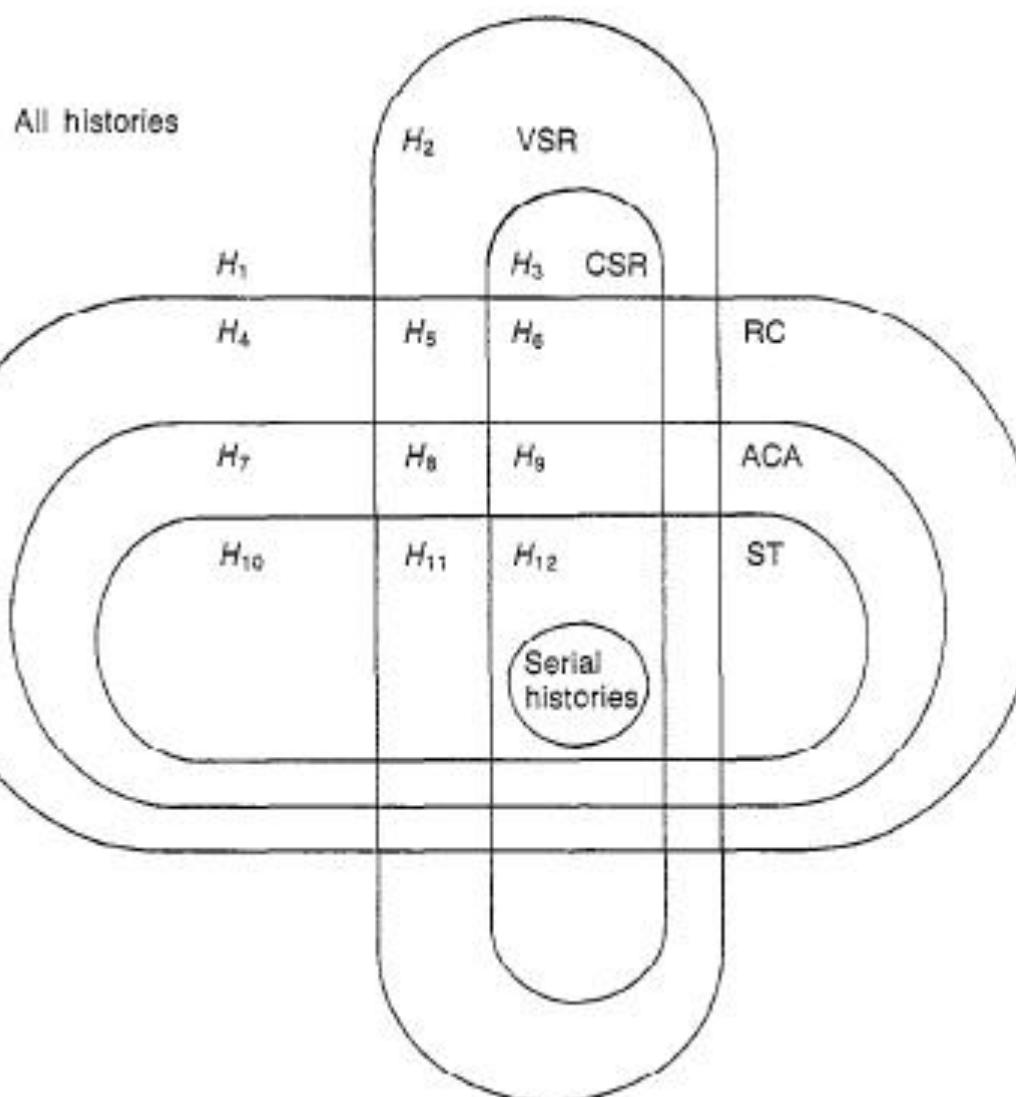
T3 r(Y) w(Z)

Definícia: Rozvrh sa vyhýba kaskádovým abortom (avoids cascading aborts, ACA) práve vtedy, ak neobsahuje dirty read (dirty read je čítanie necommittedovej hodnoty zapísanej inou transakciou, ktorá v momente čítania nebola committedovaná)

Rozvrh $r2(Y)$ $w1(X)$ $w2(X)$ a2 c1 sa vyhýba kaskádovým abortom, ale v prípade výpadku vyžaduje recovery algoritmus dva prechody cez log-file (ten druhý je kvôli REDO). Aj tomu sa možno oplatí vynhnúť. Ak je garantované, že scheduler generuje len striktné rozvrhy, dá sa algoritmus obnovy urýchliť, resp. zjednodušiť

Definícia: Rozvrh je striktný (strict) práve vtedy, ak neobsahuje dirty read ani dirty write
(dirty write je prepisovanie necommittedovej hodnoty zapísanej inou transakciou, ktorá v momente zápisu nebola committedovaná)

Dve ortogonálne hierarchie (Bernstein, Hadzilacos)



Minimálna praktická požiadavka:
 $VSR \cap RC$ (viewsériovateľné a obnoviteľné rozvrhy)

Rozumná praktická požiadavka:
 $CSR \cap ACA$, resp.
 $CSR \cap ST$ (konflikt-sériovateľné rozvrhy vyhýbajúce sa kaskádovým abortom, resp. striktné)

Ako generovať rozumné rozvrhy: Strict 2PL

Dvojfázový zamykací protokol garantuje sériovateľnosť, ale nie obnoviteľnosť

Príklad: $wl1(X)$, $w1(X)$, $ul1(X)$, $rl2(X)$, $r2(X)$, $ul2(X)$, $c2$, $c1$

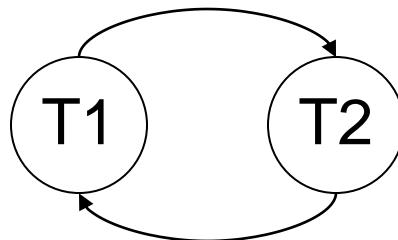
Strict 2PL je dvojfázový zamykací protokol, v ktorom je dovolené zámky uvoľňovať iba spolu s commitom (resp. s abortom). Striktný 2PL garantuje, že generovať sa budú len sériovateľné rozvrhy, ktoré sú zároveň striktné: $CSR \cap ST$

Deadlock

Bohužiaľ, 2PL aj striktný 2PL majú ešte jeden problém ktorý treba vyriešiť: **deadlock (uviaznutie)**

Príklad: $wl1(X)$, $w1(X)$, $rl2(Y)$, $rl2(X)$, $wl1(Y)$

posledné dve operácie spôsobia, že T1 ani T2 nemôže ďalej pokračovať, lebo čakajú na zámky, ktoré nikdy nedostanú



Wait-for graph: T1 čaká na uvoľnenie zámku ktorý drží T2 a naopak

Veta. Transakcie (niektoré) sú v deadlocku práve vtedy, ak **wait-for graf** obsahuje cyklus. V deadlocku sú tie transakcie, ktoré sú v cykle

Optimistická stratégia: nevyhýbať sa deadlocku. Pravidelne spustiť detekciu cyklov vo WFG a rozbitiť cykly abortovaním niektorých transakcií, resp. použiť timeout v žiadostiach o zámok (dobrovoľný abort transakcie, ktorá detektuje timeout pri žiadosti o zámok)

Pesimistická stratégia: systém udržuje wait-for graf (WFG). Ak vidí, že práve vykonaná operácia pridala do WFG hranu, ktorá spôsobila cyklus, tak abortuje niektorú zo zacyklených transakcií (Dajú sa použiť tiež rôzne modifikácie bankárskeho algoritmu známeho z operačných systémov, napr. prostriedky sú očíslované a každá transakcia ich musí zamykať vo vzostupnom poradí.)

Riešenie deadlock (pesimistické stratégie)

Wait-die stratégia: každá transakcia T_i dostane pri svojom štarte timestamp $TS(T_i)$. Ak transakcia T_1 žiada o zámok, ktorý drží T_2 :

If $TS(T_1) < TS(T_2)$

then T_1 čaká kým T_2 neskončí

else abort T_1

Wound-wait (kill-wait) stratégia: každá transakcia T_i dostane pri svojom štarte timestamp $TS(T_i)$. Ak transakcia T_1 žiada o zámok, ktorý drží T_2 :

If $TS(T_1) < TS(T_2)$

then kill T_2 /* ak T_2 práve necommituje, tak bude abortovaná */

else T_1 čaká kým T_2 neskončí

Úvod do databázových systémov

[http://www.dcs.fmph.uniba.sk/~plachetk
/TEACHING/DB2013](http://www.dcs.fmph.uniba.sk/~plachetk/TEACHING/DB2013)

Tomáš Plachetka

Fakulta matematiky, fyziky a informatiky,
Univerzita Komenského, Bratislava

Zima 2013–2014

- Dáta sú uložené na **trvácnych (externých) médiách**, ktoré možno rozdeliť do dvoch kategórií:
 - médiá **so sekvenčným prístupom** (napr. magnetické pásy), práca s nimi pripomína prácu s linkovaným zoznamom
 - médiá **s ľubovoľným prístupom** (napr. magnetické disky), práca s nimi pripomína prácu s poľom
- Dáta v tabuľkách sú organizované po riadkoch (records), **dáta na externých médiách sú organizované v blokoch**
 - predpokladá sa, že **jeden blok obsahuje viacero záznamov**
 - pre jednoduchosť predpokladáme, že dáta v operačnej pamäti (RAM) sú organizované po rovnakých blokoch
 - driver v operačnom systéme abstrahuje od fyzických detailov médií (napr. cylindre, sektory, stopy) a ponúka adresáciu blokov a čítanie/písanie bloku z/do operačnej pamäte (operácie input/output v Cache Manager)

Literatúra:

- H. Garcia-Molina, J.D. Ullman, J. Widom: Database System Implementation, Prentice Hall, 2000 (Chapters 6–7)
- R. Treat: Explaining Explain
- Postgres manual,
<http://www.postgresql.org/docs/current/static/performance-tips.html>
- Y.E. Ioannidis: Query Optimization,
<http://www-db.stanford.edu/~widom/cs346/ioannidis.pdf>
- S. Chaudhuri: An Overview of Query Optimization in Relational Systems, <http://www-db.stanford.edu/~widom/cs346/chaudhuri.pdf>

Fyzická optimalizácia dotazov: fyzická algebra

Query (SQL)

Query parser

Initial tree

Query optimiser (rewriter)

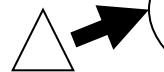
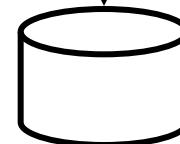
Logical plan

Executor (compiler)

Physical plan

Query processor

I/O plan



Logical operators:

join, selection, projection,
grouping, sorting, ...

Physical operators:

table-scan, filter, subplan,
...

Fyzická optimalizácia dotazov: fyzická algebra

Možnosti implementácie fyzických operátorov:

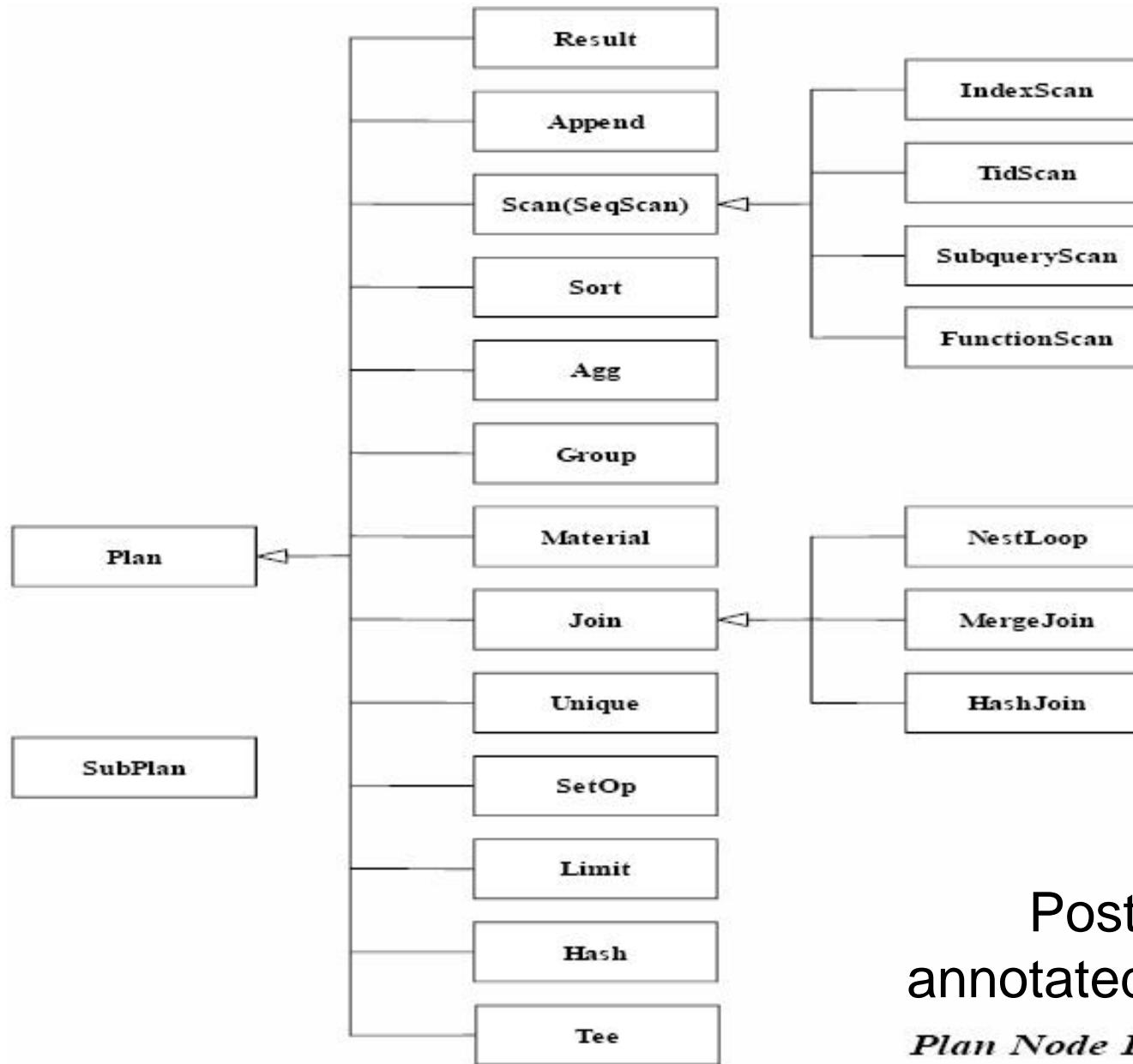
- **Materializácia:** medzivýsledky operácií sa reprezentujú tabuľkou.
Nevýhody: medzivýsledky môžu byť obrovské, medzivýsledky treba ukladať a čítať viackrát ako raz, ...
- **Iterátory (pipelining):** medzivýsledky sa reprezentujú vhodným previazaním pointerov na záznamy (riadky relácií). V jednej iterácii sa generuje jeden riadok výslednej relácie, výsledok iterátora môže byť vstupom do ďalšieho iterátora

Metódy iterátorov: **open**, **next**, **close** (iterátor je class v objektovej filozofii)

Postgres, ako aj prakticky všetky súčasné DB systémy, používa iterátory na implementáciu fyzických operátorov. Materializácia je implementovaná ako špeciálny iterátor

Cieľom návrhu a implementácie fyzických operátorov je minimalizovať počet diskových prenosov, t.j. počet input/output operácií

Fyzická optimalizácia dotazov: fyzická algebra



PostgreSQL,
annotated source code
Plan Node Hierarchy Chart

Fyzická optimalizácia dotazov: zložitosť fyz. operátorov

Miery zložitosti relácií (relácia je „bag“, nie set)

B(R): počet blokov R

T(R): počet tuples (záznamov) R, vrátane duplikátov

V(R, a₁, a₂, ..., a_N): počet rôznych tuples $\pi_{a_1, a_2, \dots, a_N}(R)$

Miery zložitosti fyzických operátorov

počet I/O prenosov, čas

POSTGRESQL Cost Parameters

Parameter	Description	Default	vs. page read
page read	Cost to read fetch one page of data, by definition	1.00	-
cpu_tuple_cost	Cost of typical CPU time to process a tuple	0.01	100x quicker
cpu_index_tuple_cost	Cost of typical CPU time to process an index tuple	0.001	1000x quicker
cpu_operator_cost	Cost of CPU time to process a typical WHERE operator	0.0025	400x quicker
random_page_cost	Cost of a non-sequential page fetch	4	4x slower
effective_cache_size	Amount of cache = likelihood of finding a page in cache	1000	N/A

Iterator SeqScan(R)

```
open(R) {  
    R.open();  
}  
close(R) {  
    R.close();  
}  
next(R) {  
    return R.next();  
}
```

Seq Scan Operator : Example

```
oscon=# explain select oid from pg_proc;  
QUERY PLAN
```

```
-----  
Seq Scan on pg_proc  (cost=0.00..87.47  
rows=1747 width=4)
```

- Most basic, simply scans from start to end
- Checks each row for any conditionals as it goes
- Large tables prefer index scans
- Cost (no startup cost), rows (tuples), width (oid)

Absolútна miera zložitosť fyzických operátorov: čas

V konfigurácii Postgres je konštanta 0.01, ktorá hovorí že RAM je 100x rýchlejšia ako disk. Postgres meria zložitosť v počte I/O prenosov, ale zahŕňa do nej aj čas výpočtov v RAM. V tomto príklade $B(R)=70$ a $T(R)=1747$, tak odhad zložosti $\text{seq_scan}(R)$ je $70+1747 \cdot 0.01 = 87.47$ I/O prenosov. (To 0.00 je odhad „startup cost“, t.j. zložitosť „open“. V tomto prípade nestojí startup nič.)

Iterator SeqScan(R)

```
open(R) {  
    R.open();  
}  
close(R) {  
    R.close();  
}  
next(R) {  
    return R.filter().next();  
}
```

```
oscon=# explain select oid from pg_proc where oid<>0;  
          QUERY PLAN  
-----  
Seq Scan on pg_proc  (cost=0.00..91.84 rows=1746 width=4)  
  Filter: (oid <> 0::oid)  
IOcost+CPUcost=B(R)+T(R)*(0.01+0.0025)=91.835  
(0.0025 je cena filtrovacej podmienky)
```

```
oscon=# explain select oid from pg_proc where oid<>0  
and oid<100;
```

QUERY PLAN

```
-----  
Seq Scan on pg_proc  (cost=0.00..96.20 rows=582  
width=4)
```

Filter: ((oid <> 0::oid) AND (oid < 100::oid))

IOcost+CPUcost=B(R)+T(R)*(0.01+2*0.0025)=96.205
(cenu za testovanie podmienky platíme dvakrát)

Algoritmus sort (externý merge-sort)

Triedenie dlhého súboru, ktorý sa nezmestí celý do operačnej pamäte. Nech je v operačnej pamäti miesto pre M diskových blokov

1. fáza, vytvorenie utriedených behov:

$i=0;$

Opakuj pre celú reláciu:

- Prečítaj M blokov do operačnej pamäte
- Utried' bloky v operačnej pamäti (heapsort, resp. quicksort)
- Zapíš utriedené bloky do behu R_i ; $i=i+1$;

Nech N je počet behov

Algoritmus sort (externý merge-sort)

2.fáza, spájanie behov do väčších behov (pre jednoduchosť predpokladajme, že $N < M$). Použi $M-1$ blokov v pamäti pre vstup, 1 blok pre výstup:

Načítaj 1 blok z každého behu.

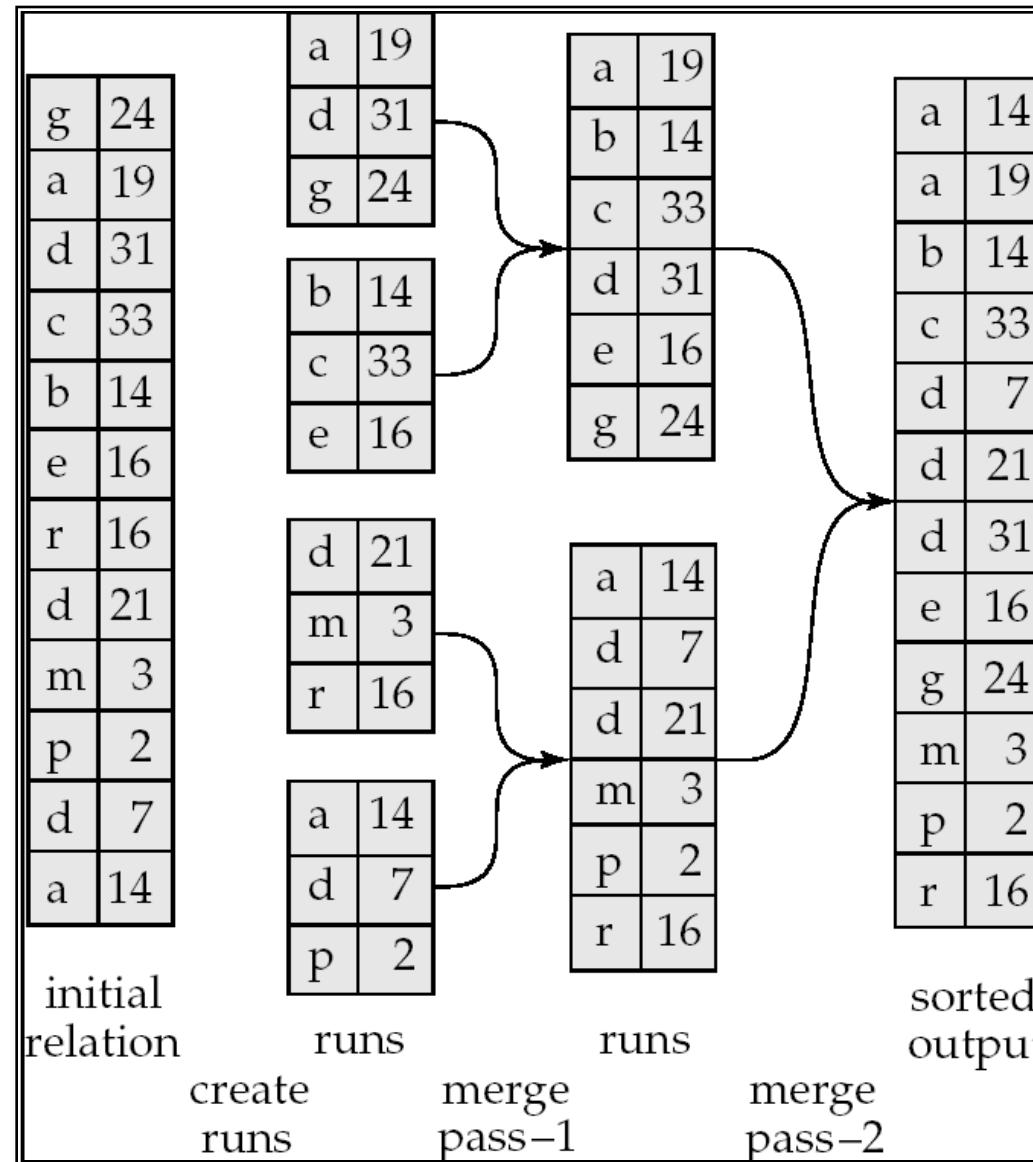
Opakuj kým sú vstupné bloky neprázdne:

- Vyber najmenší záznam spomedzi všetkých $M-1$ behov v pamäti
- Zapíš ten záznam do výstupného bloku. Ak je výstupný blok plný, zapíš ho na disk
- Zmaž ten najmenší záznam. Ak je ten vstupný blok prázdny, načítaj ďalší blok z toho behu

Ak $N \geq M$, druhá fáza sa musí niekoľko krát opakovat'

Vybrané fyzické operátory: merge-sort

Príklad (Silberschatz et al.), M=3:



Vybrané fyzické operátory: merge-sort

Zložitosť:

- Počet opakovaní druhej fázy: $\lceil \log_{M-1} (B(R) / M) \rceil$
- Počet diskových operácií v 1. fáze aj v 2. fáze je $2 B(R)$, ak počítame aj výstup na disk
- Celkový počet diskových operácií je (približne)
$$\text{cost} = 2 B(R) (1 + \lceil \log_{M-1} (B(R) / M) \rceil)$$

Vybrané fyzické operátory: merge-sort

Iterator Sort(R)

```
private table oldR, newR;  
open(R) {  
    R.open();  
    mergesort(R, newR);  
    R.close();  
    oldR=R;  
    R=newR;  
    R.open();  
}  
  
next(R) {  
    return R.next();  
}  
  
close(R) {  
    R.close();  
    R.drop();  
    R=oldR;  
}
```

```
oscon=# explain select oid from pg_proc order by oid;  
QUERY PLAN  
-----  
Sort (cost=181.55..185.92 rows=1747 width=4)  
  Sort Key: oid  
    -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747  
width=4)  
  
• Explicit: ORDER BY clause  
• Implicit: Handling Unique, Sort-Merge Joins, and other operators  
• Has startup cost: cannot return right away
```

Takmer celú prácu iterátora sort treba urobiť počas open(). V tomto príklade je odhad startup cost 185.92, čo zodpovedá internému volaniu mergesort. Ako už vieme, mergesort je 2-fázový (resp. multifázový, ak je málo RAM):

1. Číta postupne všetky bloky, utriedi každý z nich v RAM a zapíše na disk.
2. Spojí utriedené bloky do jednej tabuľky.

Algoritmus join (nested-loop join)

$R \bowtie_c S$

Ulož menšiu z relácií, R, do M-2 blokov v pamäti (resp. koľko sa zmestí). 1 blok rezervuj pre vstup S a 1 blok pre výstup.
Postupne čítaj všetky bloky druhej relácie, S, a pre každý blok aplikuj join všetkých záznamov R na záznamy v práve načítanom bloku S. Ak joinovacia podmienka c platí, zapíš výsledný záznam do výstupného bloku. Ak je výsledný blok plný, zapíš ho na disk

$$\text{cost} = \lceil B(R) / (M-2) \rceil * B(S)$$

(Cena písania na disk tu nie je zahrnutá, lebo veľkosť joinu závisí od joinovacej podmienky.)

Výpočet kartézskeho súčinu je špeciálny prípad tohto algoritmu, keď je joinovacia podmienka splnená pre každú dvojicu záznamov

Vybrané fyzické operátory: nested-loop join

Iterator NestedLoopJoin(R,S)

```
private tuple r,s;  
open(R,S) {  
    R.open();  
    S.open();  
    s=S.next();  
}  
  
close(R,S) {  
    R.close();  
    S.close();  
}  
  
next(R,S) {  
    if (s==EOF)  
        return EOF;  
    do {  
        r=R.next();  
        if (r==EOF)  
            R.close();  
        s=S.next();  
        if (s==EOF)  
            return EOF;  
        R.open();  
    } while not (r and s join);  
    return join of r and s;  
}
```

```
oscon# select * from pg_foo join pg_namespace on  
(pg_foo.pronamespace=pg_namespace.oid);  
-----  
          QUERY PLAN  
-----  
Nested Loop (cost=1.05..39920.17 rows=5867 width=68)  
  Join Filter: ("outer".pronamespace = "inner".oid)  
    -> Seq Scan on pg_foo  (cost=0.00..13520.54 rows=234654 width=68)  
    -> Materialize (cost=1.05..1.10 rows=5 width=4)  
      -> Seq Scan on pg_namespace  (cost=0.00..1.05 rows=5 width=4)
```

- Joins two tables (two input sets)
- USED with INNER JOIN and LEFT OUTER JOIN
- Scans 'outer' table, finds matches in 'inner' table
- No startup cost
- Can lead to slow queries when not indexed, especially when functions are in the select clause

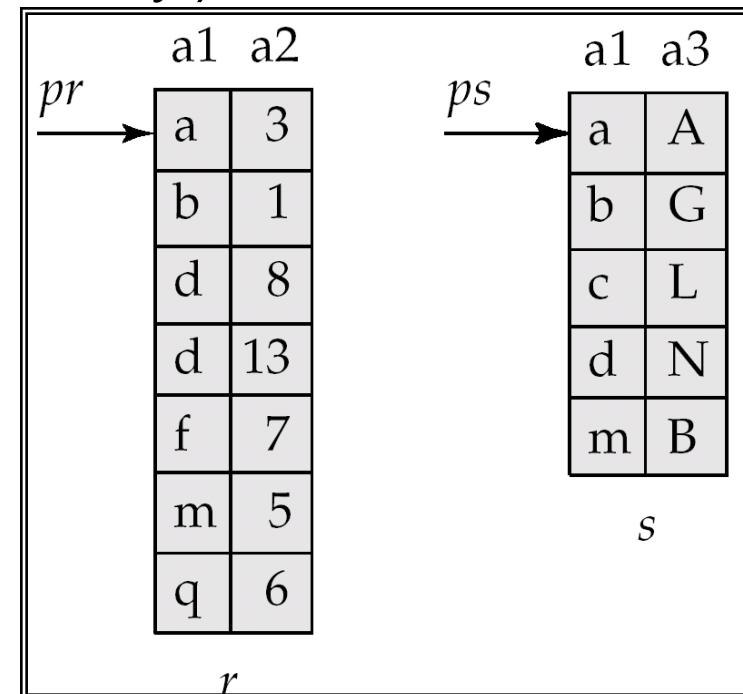
Algoritmus join (merge join)

Tento algoritmus dovoľuje len joinovaciu podmienku na rovnosť (equijoin, resp. natural join) a predpokladá, že obe relácie sú utriedené

$$\text{cost} = B(R) + B(S)$$

(Cena písania na disk tu nie je zahrnutá, lebo veľkosť joinu závisí od joinovacej podmienky.)

Podobný 2. fáze merge-sort algoritmu. Ale nie celkom rovnaký: rozdiel je v spracovaní duplikátov!



Iterator **IndexScan(R)**

```
open(R) {      oscon=# explain select oid from pg_proc where oid=1;  
    R.open();          QUERY PLAN  
}  
-----  
close(R) {     Index Scan using pg_proc_oid_index on pg_proc  (cost=0.00..5.99  
    R.close();    rows=1 width=4)  
}  
next(R) {       Index Cond: (oid = 1::oid)  
    return R.index().next();  
}
```

**IndexScan je pri veľkých reláciách
oveľa rýchlejší ako SeqScan**

Motivácia: pri hľadaní záznamu napr. podľa kľúča sa chceme vyhnúť sekvenčnému prehľadávaniu (sequential scan)

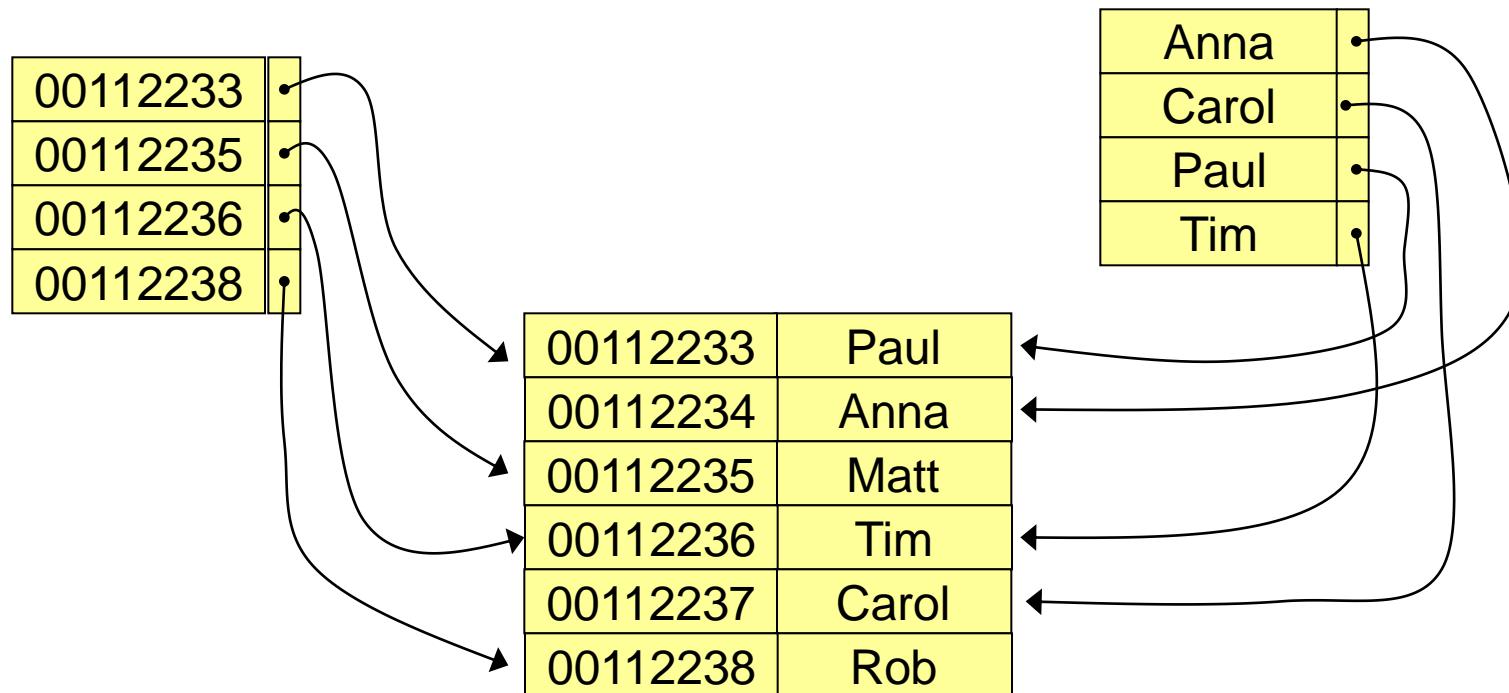
Jedna z možností je udržiavať tabuľku na disku utriedenú podľa toho kľúča. Toto sice redukuje čas vyhľadávania, ale spôsobuje **problémy pri vkladaní a vyniechávaní**, podobné problémom pri vkladaní a vyniechávaní do poľa v RAM. Navyše, čo ak chceme vyhľadávať podľa viacerých rôznych kľúčov?

Idea: **index v nezávislom súbore** (ISAM, Index Sequential Access Method). Záznamy v indexovom súbore sú dvojice [kľúč, pointer]. Pôvodná tabuľka je neusporiadaná, ale **index je usporiadaný podľa nejakého kľúča**. K jednej tabuľke môže byť viacero indexov, ktoré sú usporiadané podľa rôznych kľúčov

Pozor, „vyhľadávací kľúč“ je nový pojem!

Primárny sekvenčný index

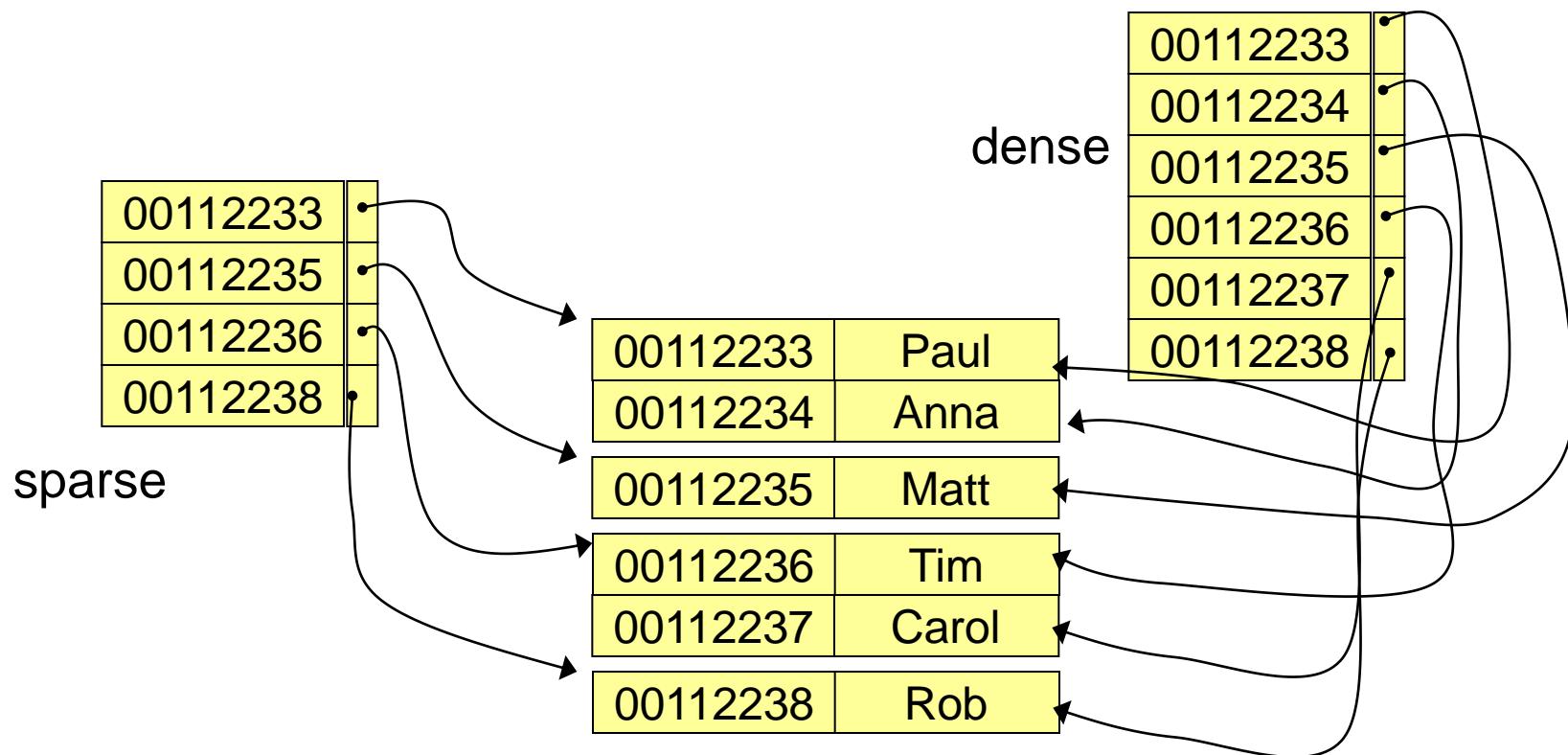
Primárny sekvenčný index je usporiadaný podľa primárneho kľúča (primary key). Je však možné pridať ďalšie indexy podľa iných atribútov (dokonca nie nutne kľúčových—vyhľadávací kľúč nemusí byť kľúč, resp. nadkľúč relácie)



Typy sekvenčných indexov: dense, sparse

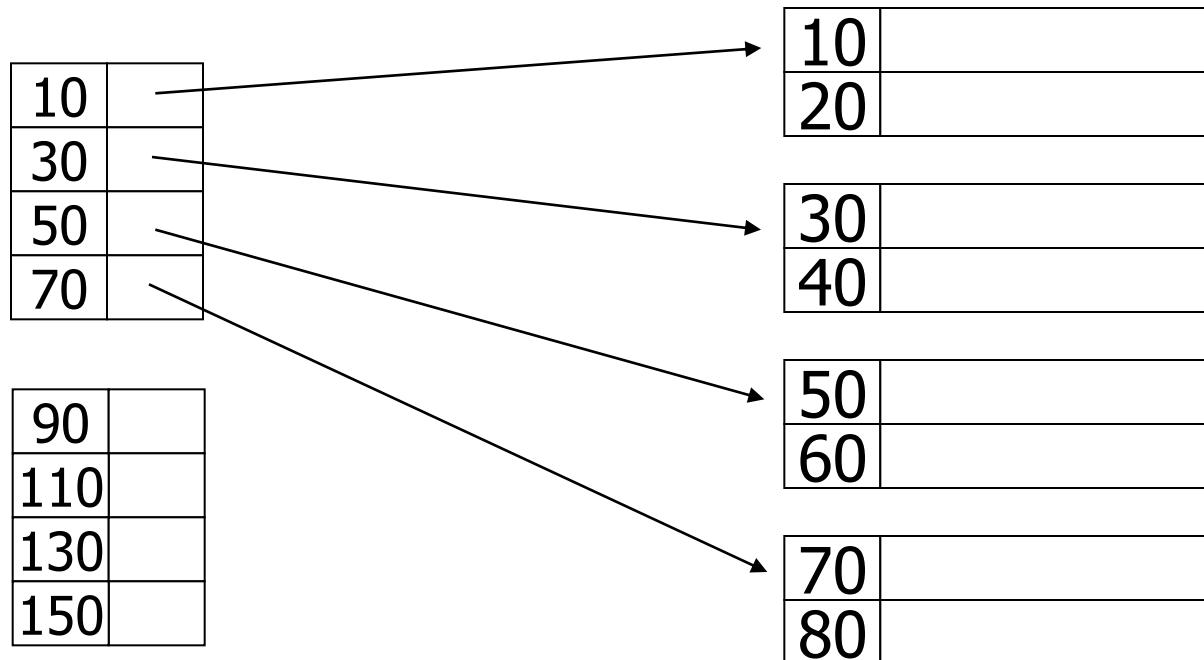
Dense (hustý) index obsahuje adresy všetkých (existujúcich) záznamov

Sparse (riedky) index obsahuje len adresy niektorých záznamoch, napr. adresy začiatkov blokov



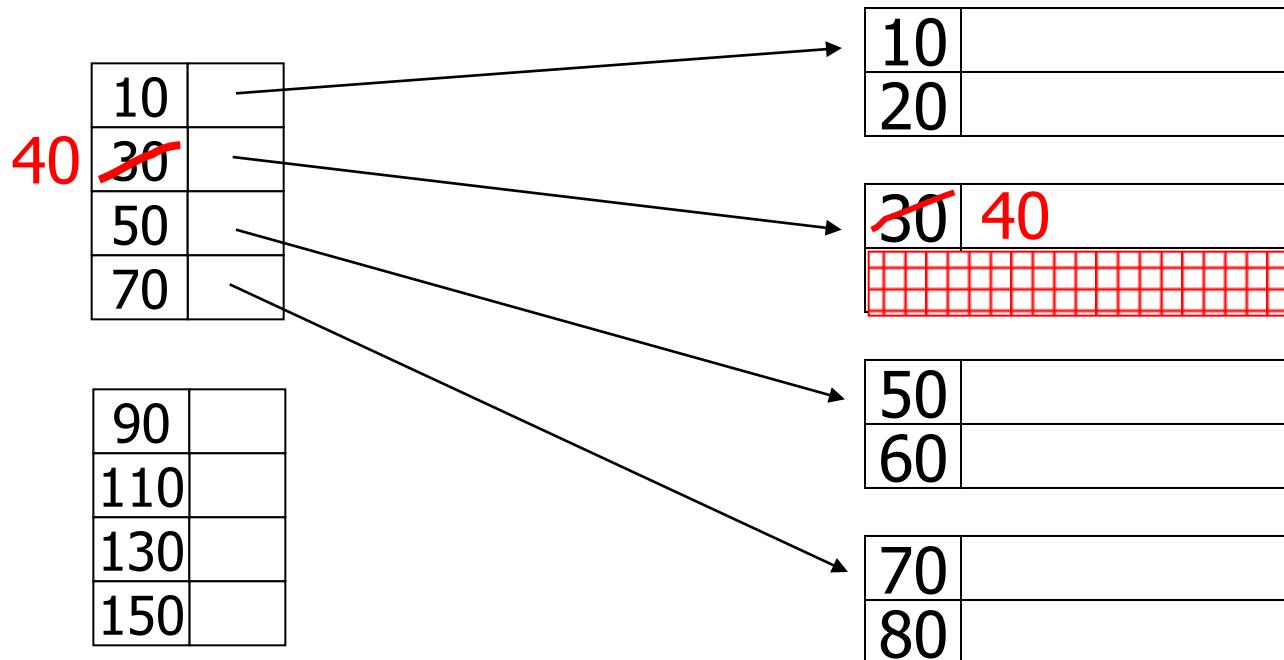
Vynechávanie z riedkeho sekvenčného indexu

Príklad (Gupta)



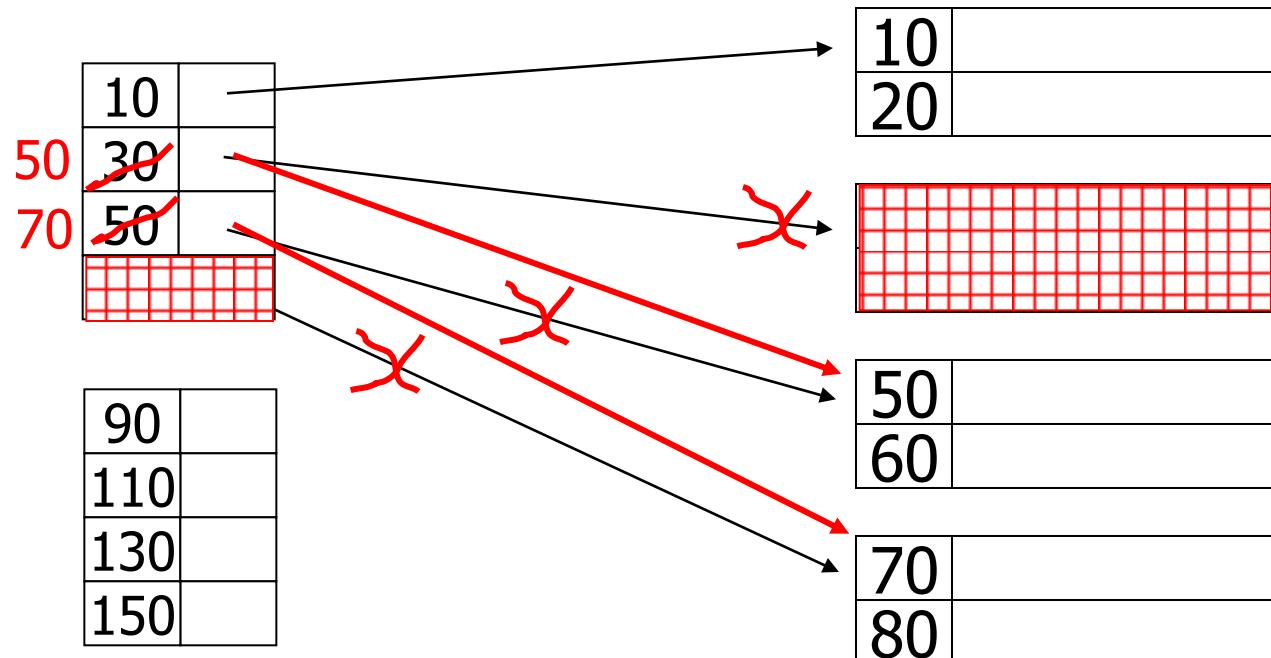
Vynechávanie z riedkeho sekvenčného indexu

Príklad (Gupta): vyniechanie záznamu 30



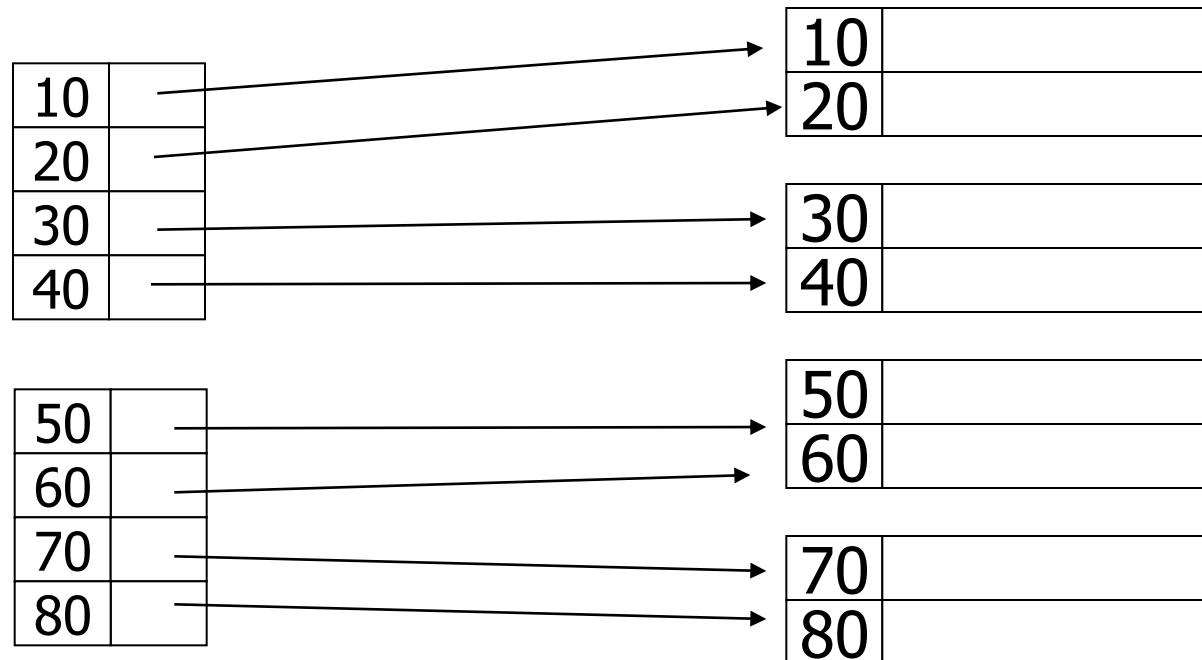
Vynechávanie z riedkeho sekvenčného indexu

Príklad (Gupta): následné vynechanie záznamu 40. Vzniká voľný blok



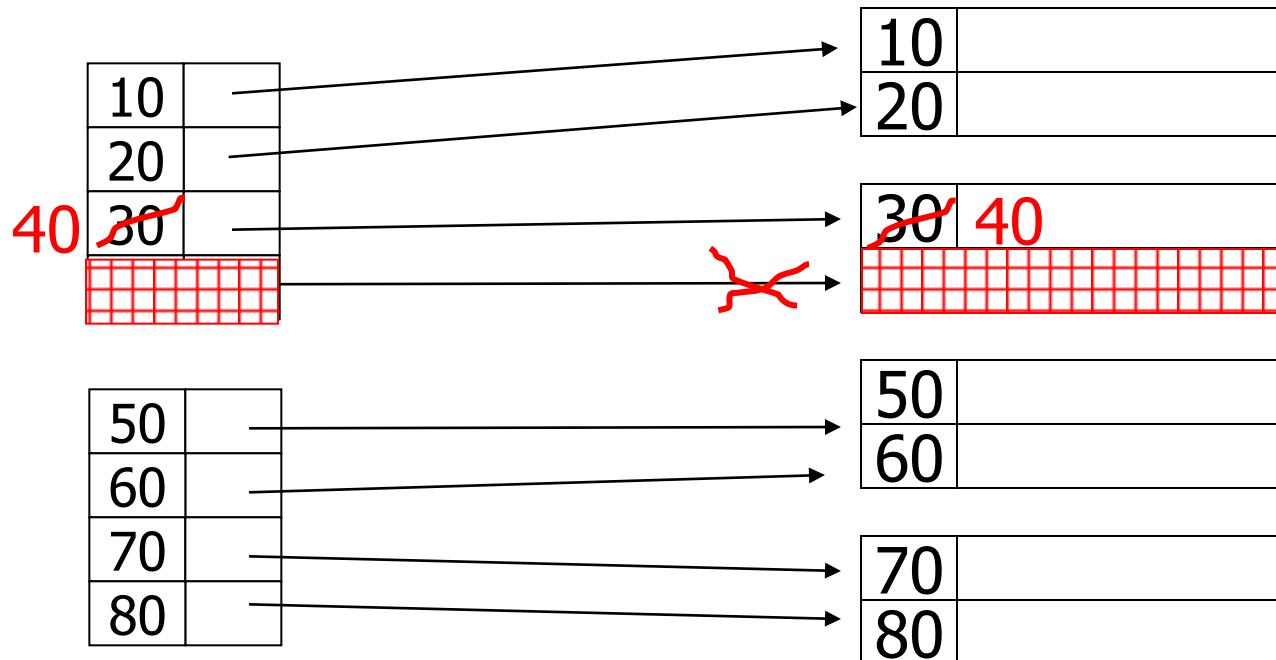
Vynechávanie z hustého sekvenčného indexu

Príklad (Gupta)



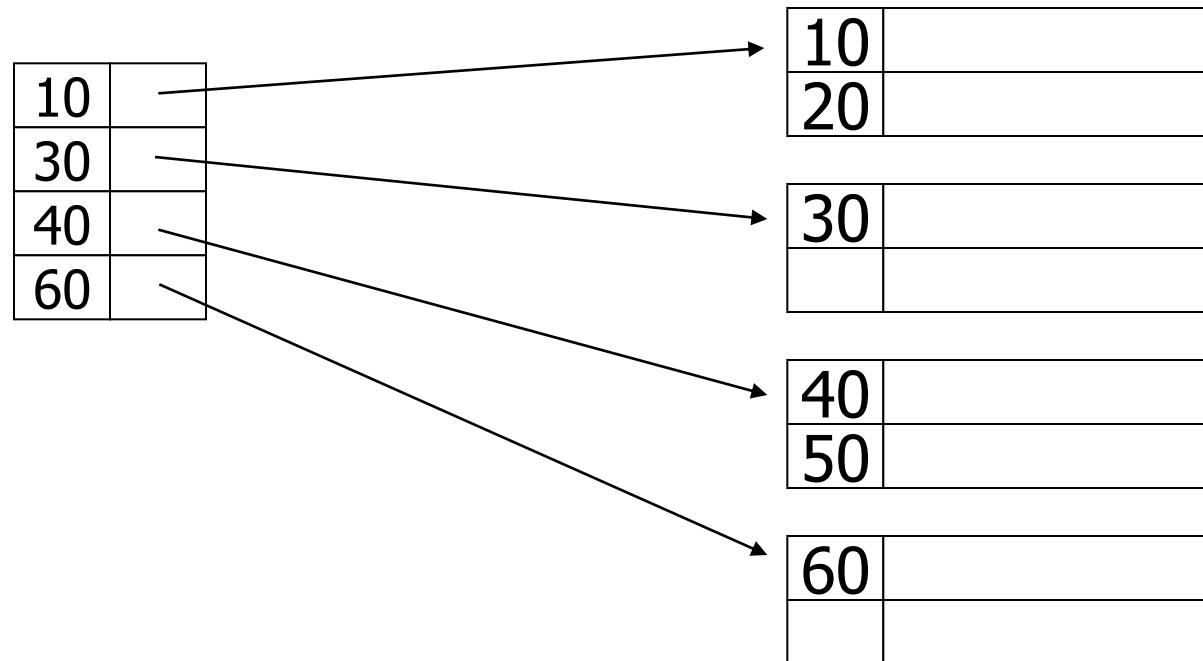
Vynechávanie z hustého sekvenčného indexu

Príklad (Gupta): vyniechanie záznamu 30



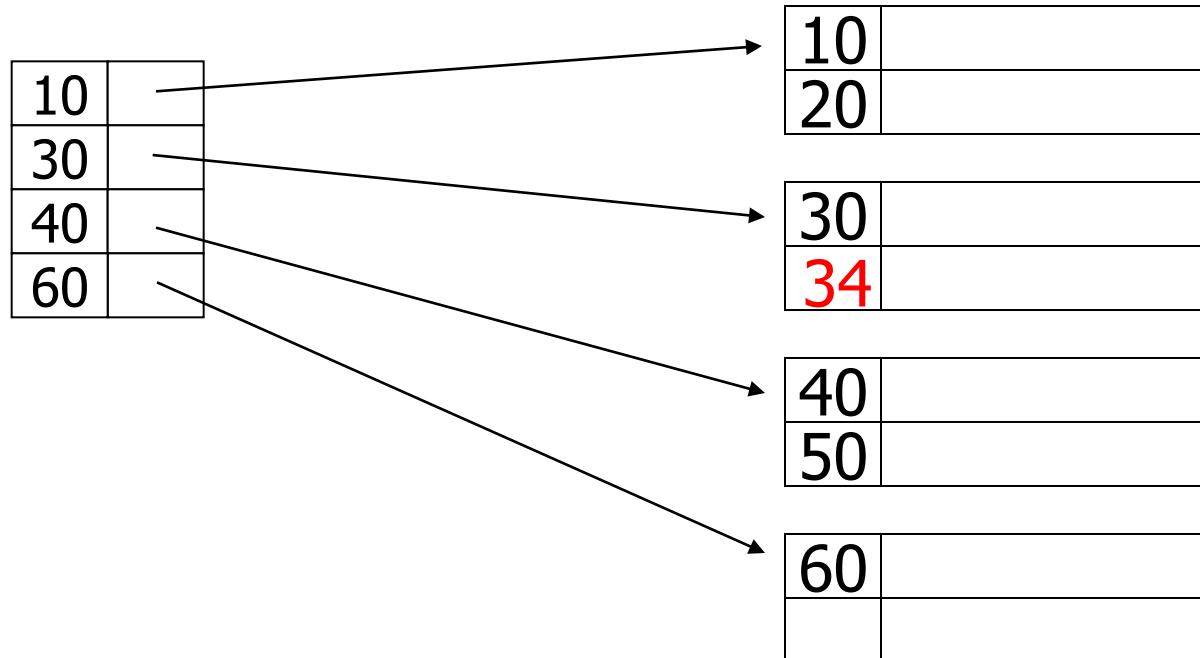
Vkladanie do riedkeho sekvenčného indexu

Príklad (Gupta): vloženie záznamu 34



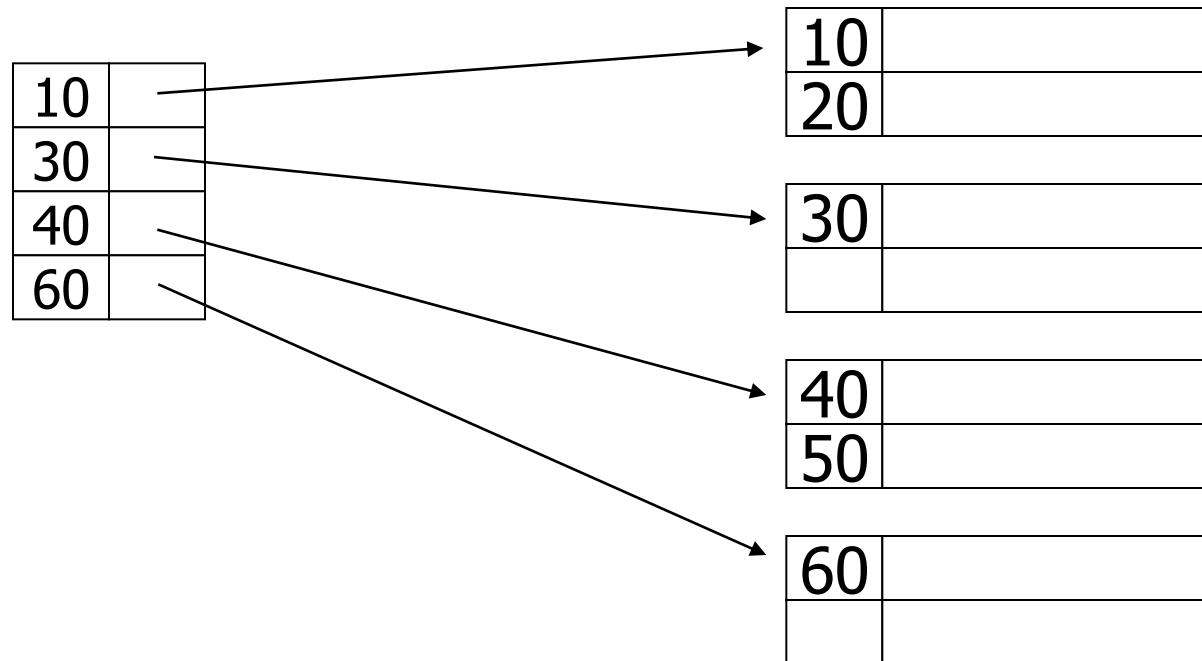
Vkladanie do riedkeho sekvenčného indexu

Príklad (Gupta): vloženie záznamu 34. Ak je v bloku miesto, jednoducho vložíme záznam



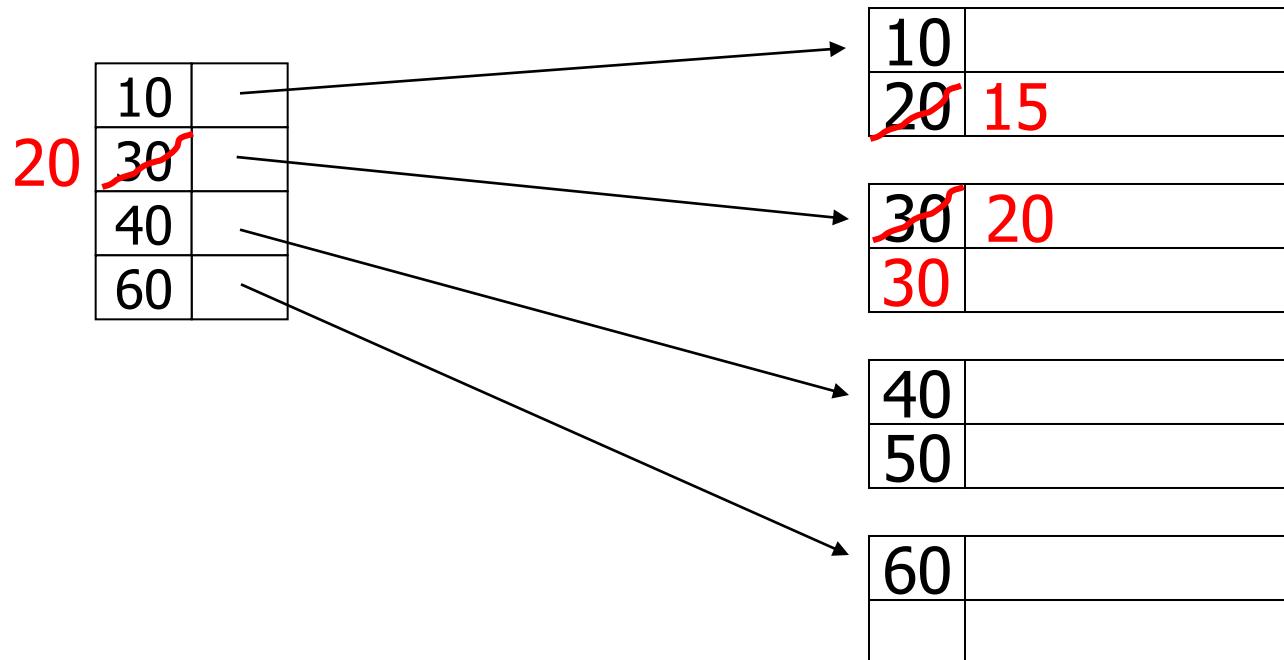
Vkladanie do riedkeho sekvenčného indexu

Príklad (Gupta): vloženie záznamu 15. Ak nie je v bloku miesto, máme dve alternatívy: 1.bud' okamžite reorganizujeme index alebo 2.použijeme blok preplnenia



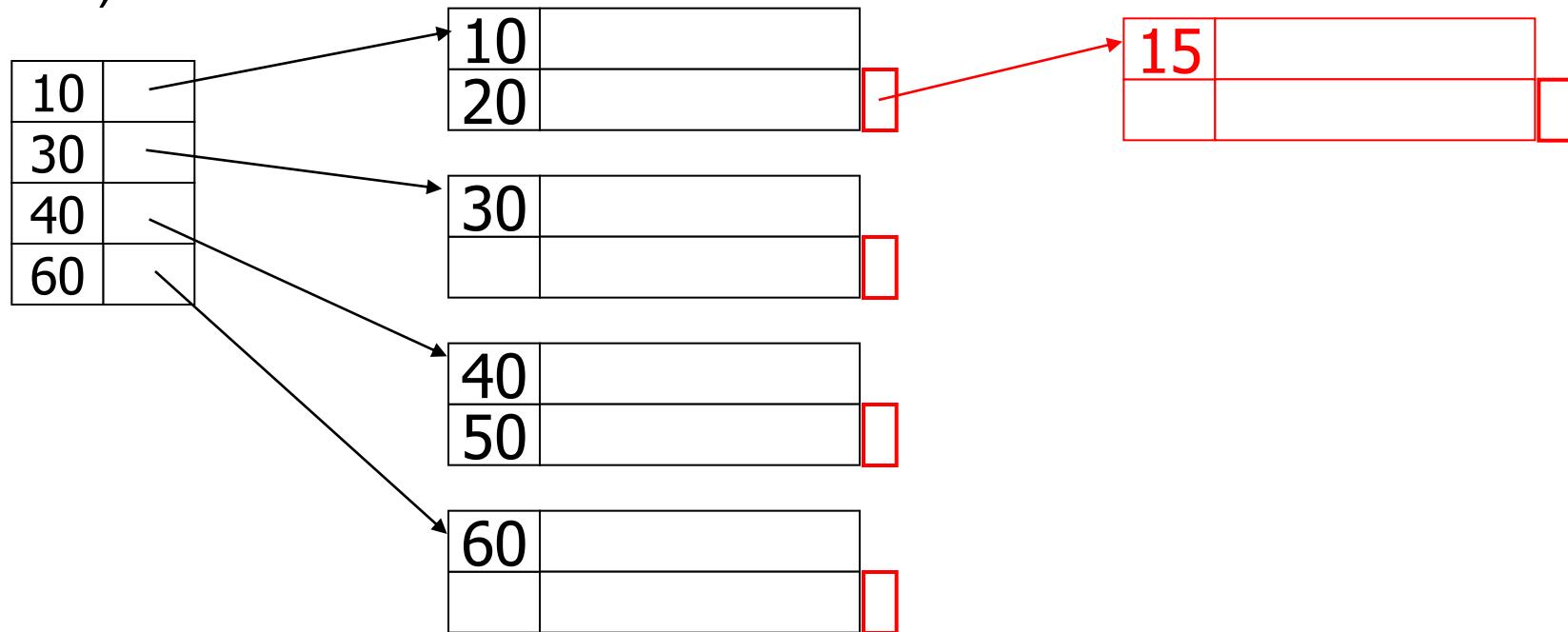
Vkladanie do riedkeho sekvenčného indexu

Príklad (Gupta): vloženie záznamu 15, reorganizácia indexu

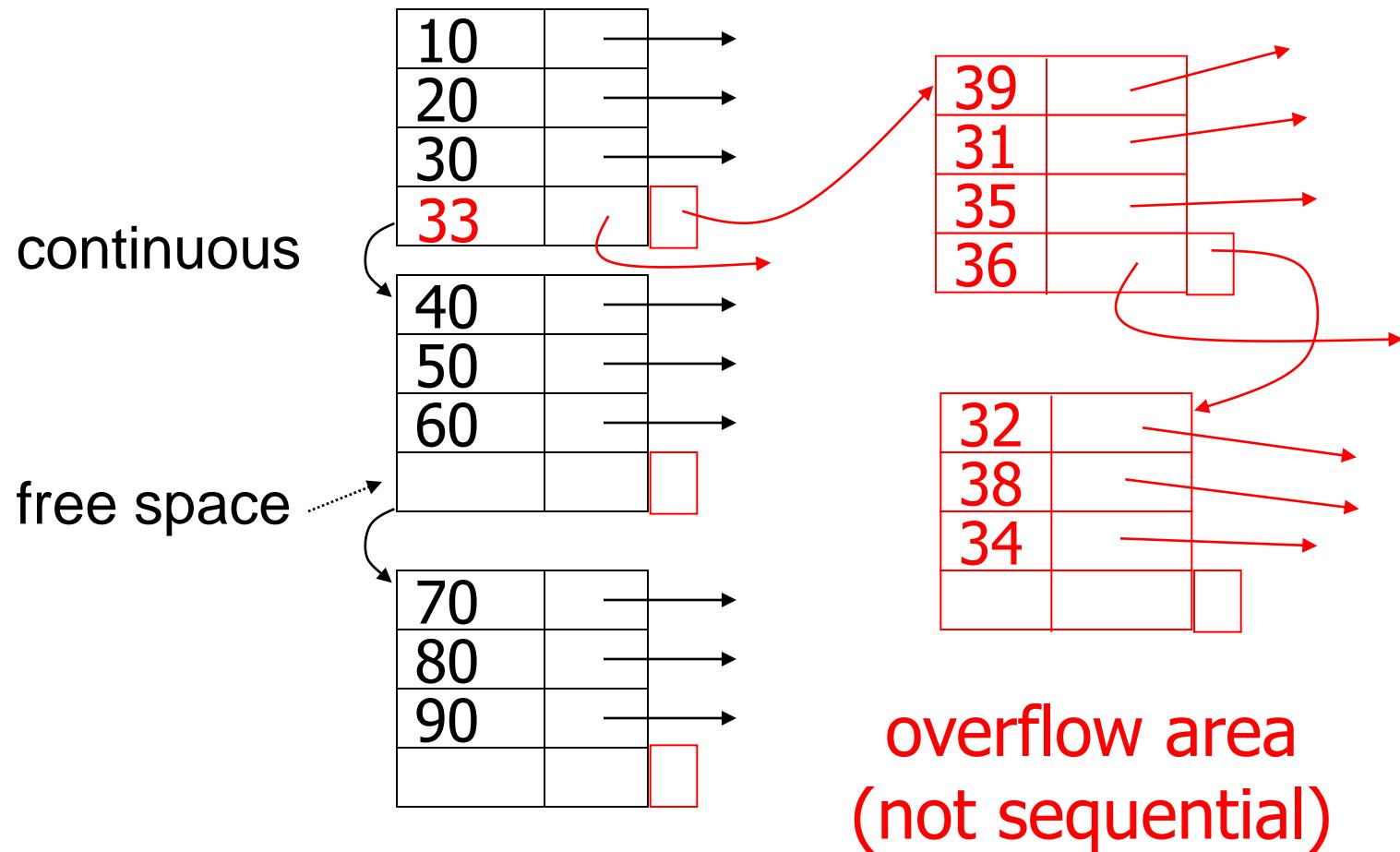


Vkladanie do riedkeho sekvenčného indexu

Príklad (Gupta): vloženie záznamu 15 s použitím bloku preplnenia. (Ak v budúcnosti vznikne príliš veľa blokov preplnenia, bude treba index preorganizovať alebo vytvoriť nanovo.)

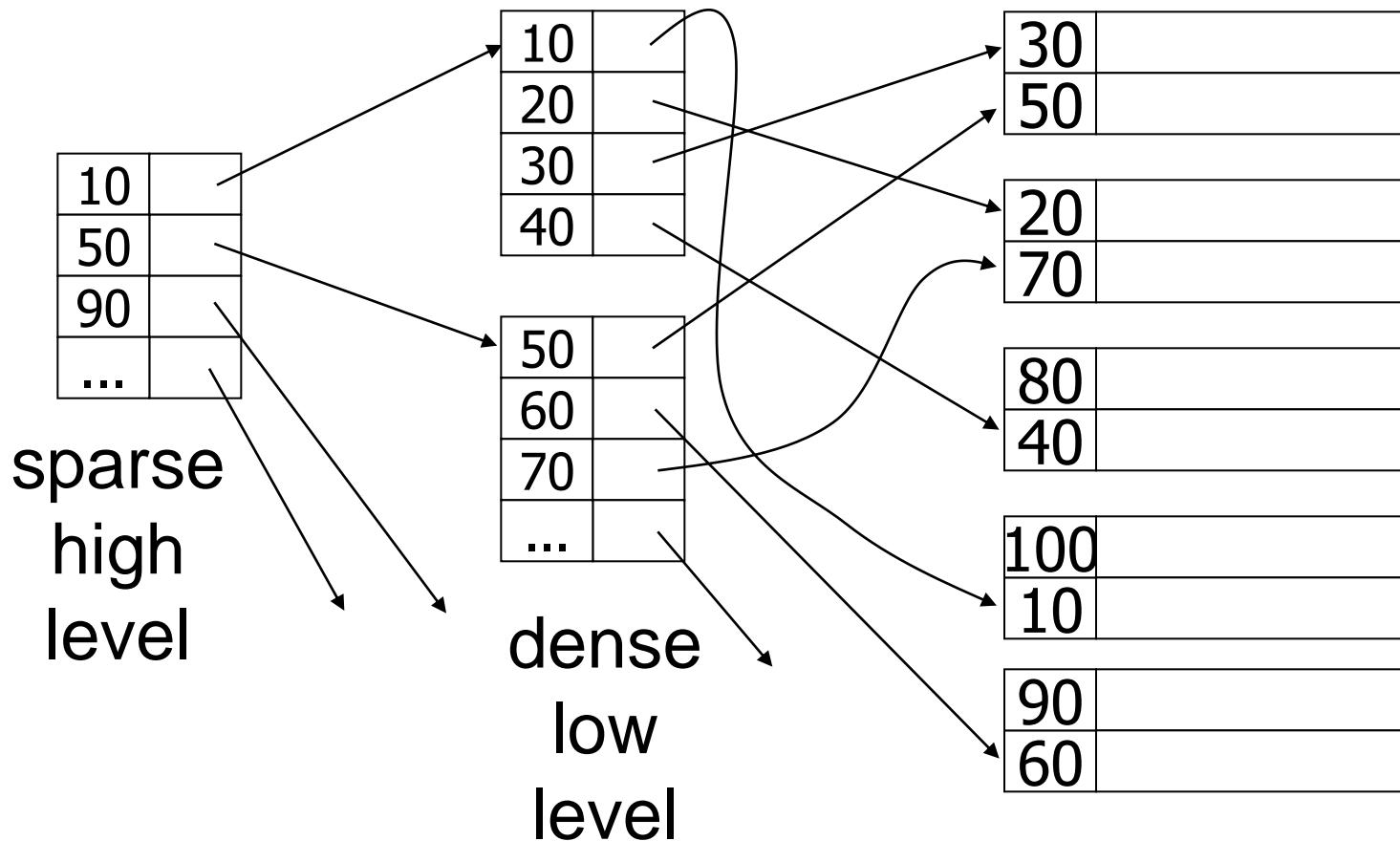


Index (sequential)



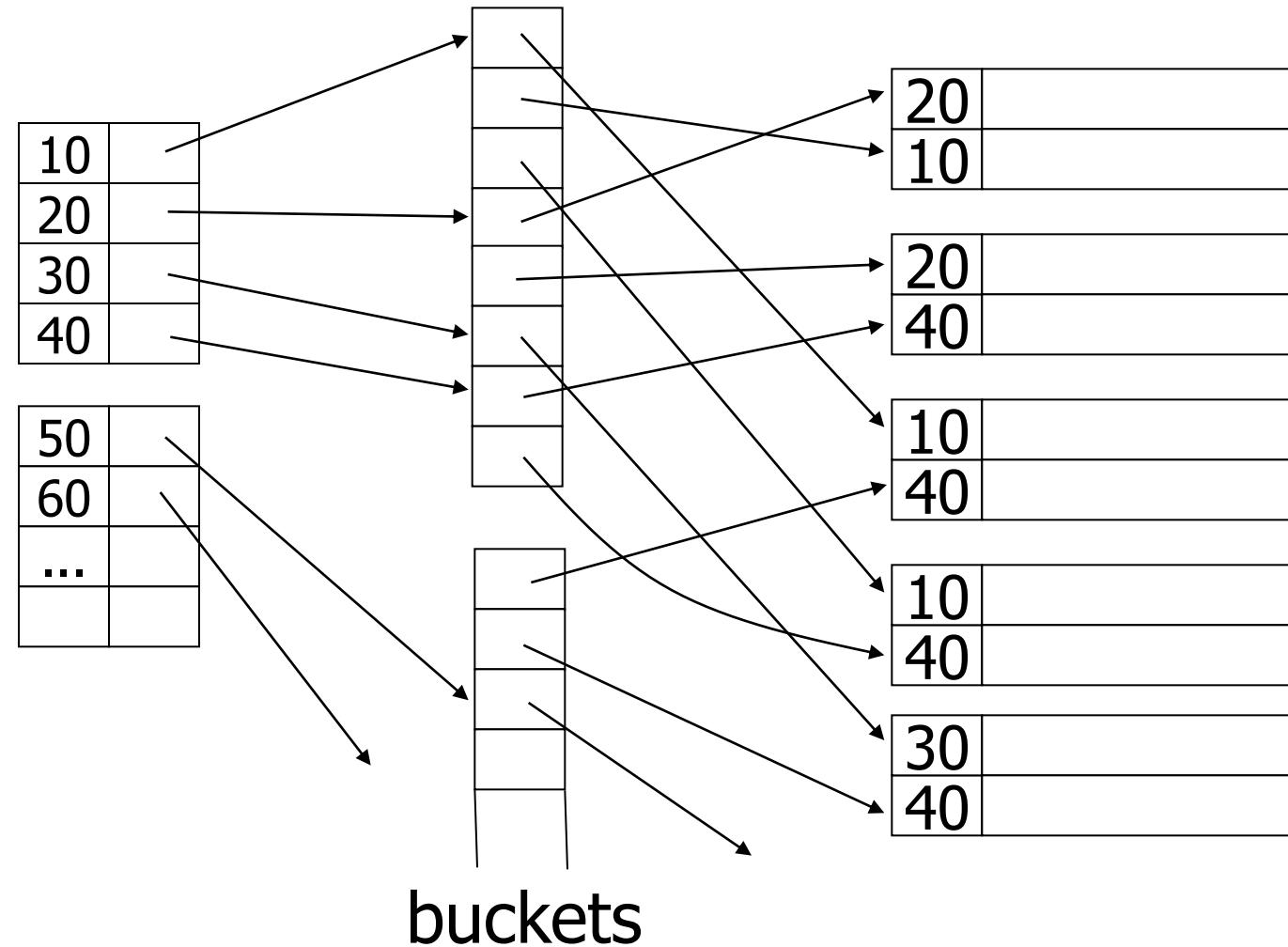
Multilevel indexy (index indexu)

Pri multilevel indexoch musí byť
najvnútornejší index vždy hustý!



Multilevel indexy (index indexu)

Pri multilevel indexoch musí byť
najvnútornejší index vždy hustý! Toto platí aj pre hashovanie



Výhody:

- jednoduchá implementácia
- vhodné pre sekvenčné prehľadávanie: v prípade hustého indexu vieme rozhodnúť o existencii záznamu bez toho, aby sme pristupovali k dátovému súboru

Nevýhody:

- drahé vkladanie
- **drahé vyhľadávanie**

Ked'že pre každú reláciu treba implementovať aspoň SeqScan, každá relácia sa indexuje podľa primárneho klúča

Riešenie nevýhod: vyhľadávacie stromy (B^+ stromy)

B stromy a B⁺ stromy

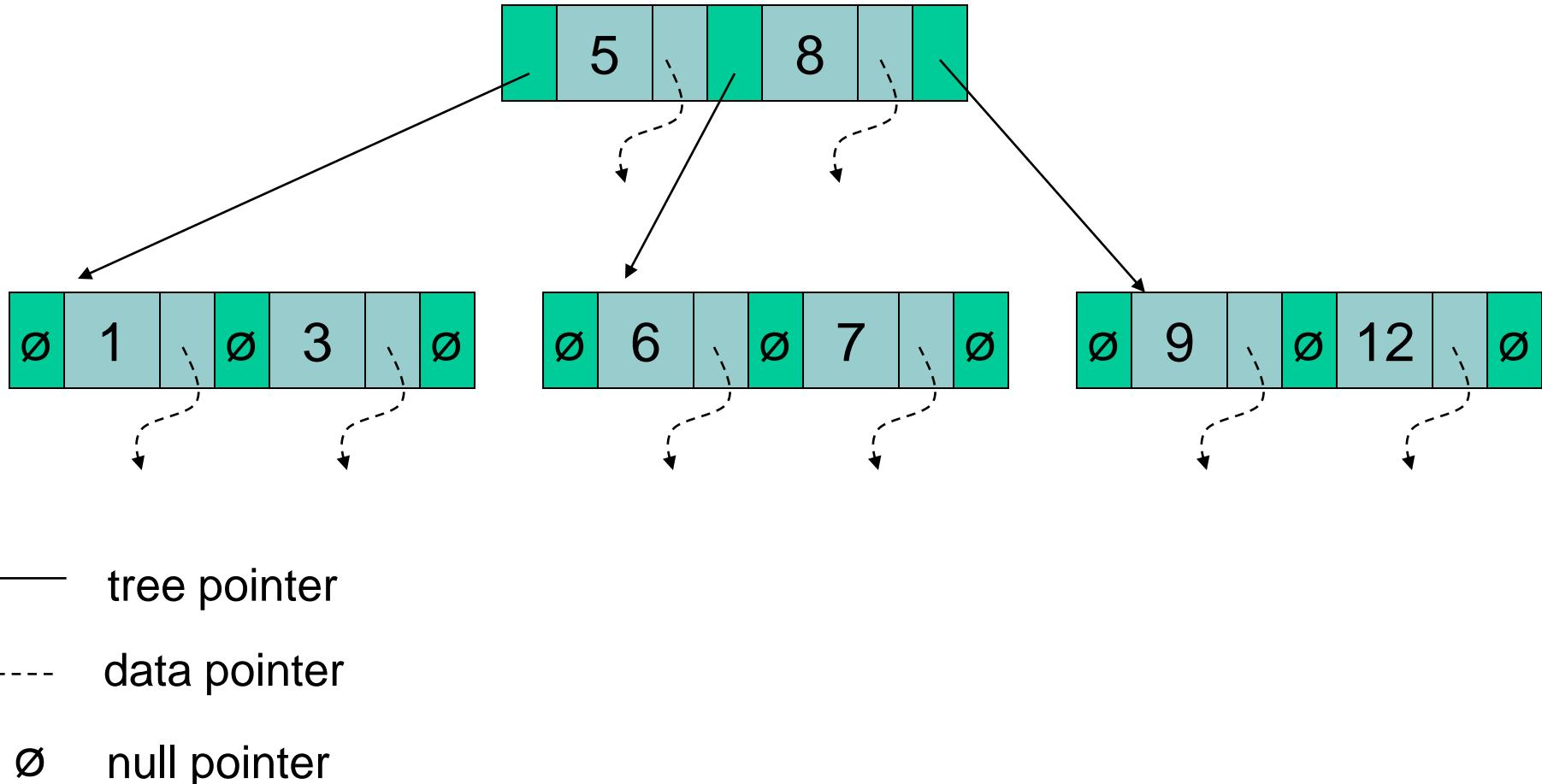
Vyhľadávacie AVL stromy nie sú pre blokovú organizáciu vhodné, lebo každý uzol AVL obsahuje len 1 záznam

Definícia: **B(m)-strom** je strom, pre ktorý platí:

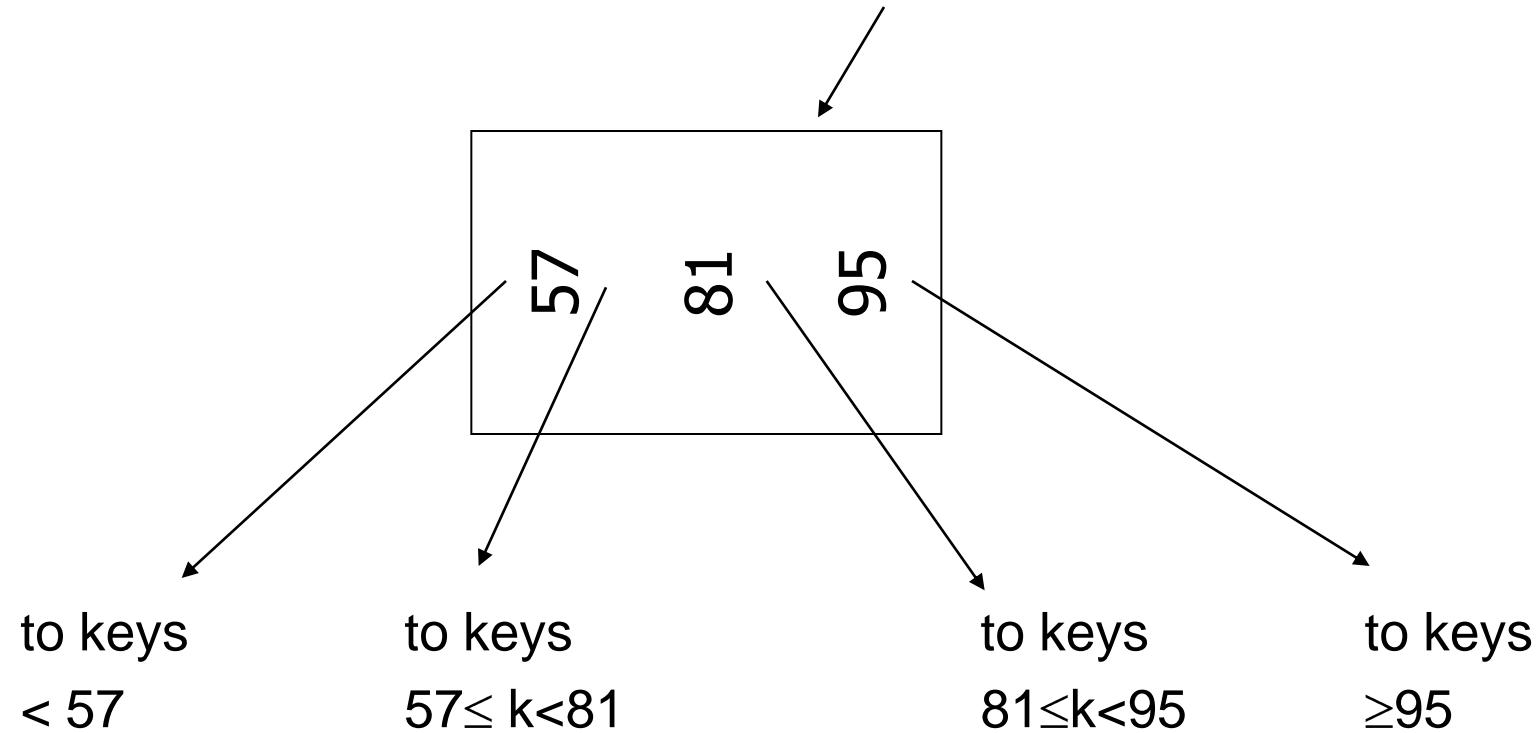
- Každý uzol okrem koreňa obsahuje k záznamov, kde $\lfloor m/2 \rfloor \leq k \leq m$. (Dá sa vyžadovať aj naplnenie $\lfloor 2m/3 \rfloor \leq k \leq m$.)
- Koreň, ak nie je listom, obsahuje aspoň 1 a najviac m záznamov.
- Vnútorný uzol s k záznamami má $k+1$ synov.
- Všetky listy sú na tej istej úrovni.
- **B⁺ stromy** majú vo vnútorných uzloch iba kľúče (index) a dátové záznamy sú iba v listoch.

2/3 -stromy sú teoreticky výhodnejšie ako 1/2-stromy, lebo majú menšiu hĺbku, ale 2/3 sa ľahšie implementujú

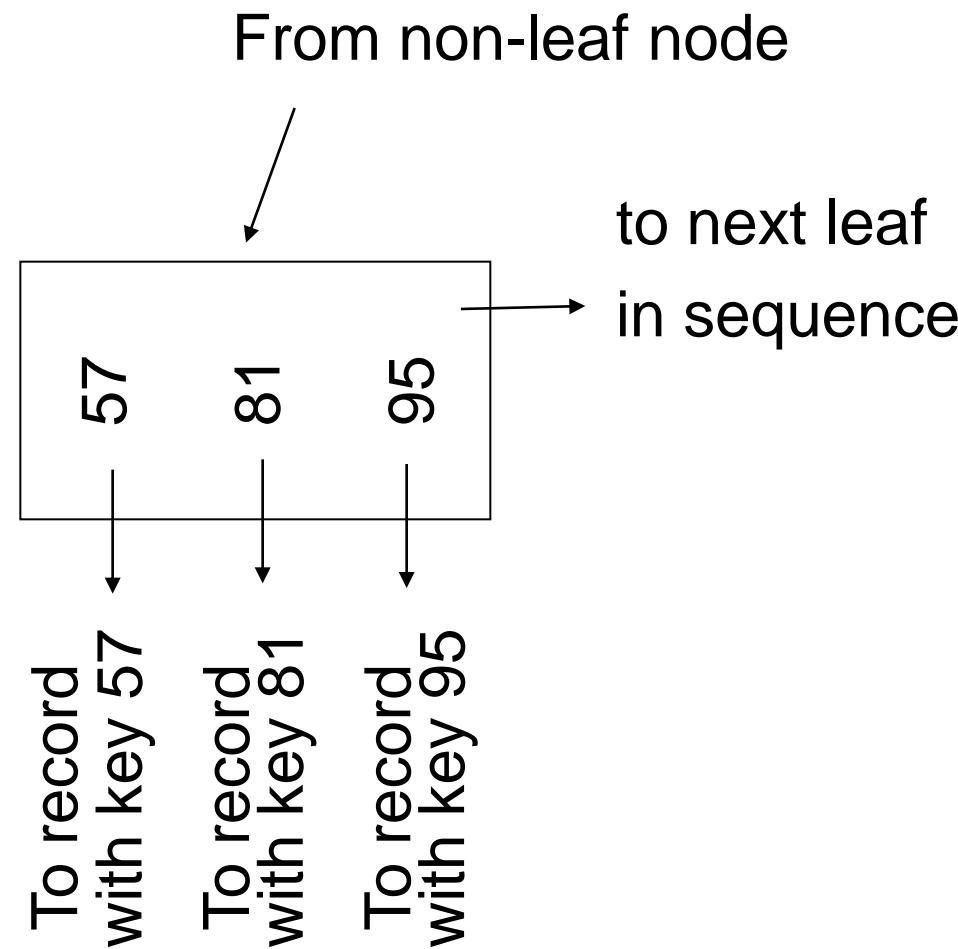
Príklad B(2) stromu



Príklad nelistového (vnútorného) uzla (Gupta)



Príklad listového uzla (Gupta)



B⁺ stromy: vkladanie

1. Nájdi list do ktorého záznam patrí a vlož
2. Ak sa list nepreplnil, skonči (A)
3. Ak je list preplnený ($m+1$ záznamov), tak skontroluj či sa nedajú redistribuovať záznamy medzi susednými listami. Ak áno, urob to a uprav index (dve možnosti: spoločný otec, rôzni otcovia). Potom rozdel' list na dva a index pre druhú časť vlož do nadradeného uzla (otca) (B)

Vloženie do vnútorného uzla je podobné vloženiu do listu, ale vkladá sa aj príslušný smerník. Pri redistribúcii sa redistribujú aj príslušné smerníky. (C)

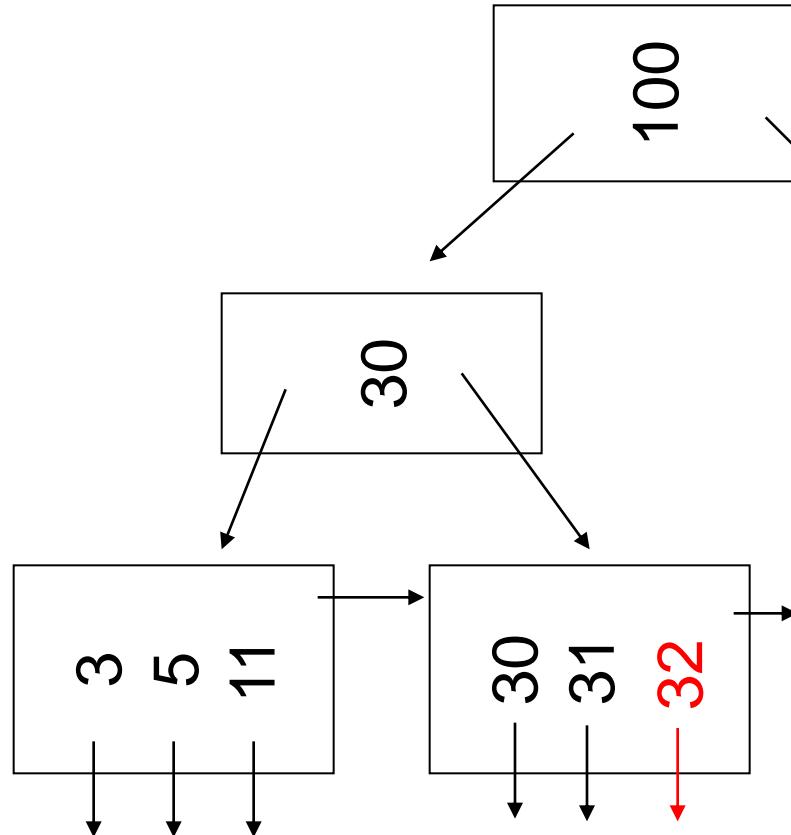
Ak sa preplnil koreň, vznikne nový koreň s jednou hodnotou a smerníkmi na ľavý a pravý podstrom (D)

Nasledujúce obrázky zodpovedajú definícii B⁺(m) stromu s naplnením $\lfloor m/2 \rfloor \leq k \leq m$ (niektoré ukazujú len podstatné fragmenty stromu)

B⁺ stromy: vkladanie (A) simple case

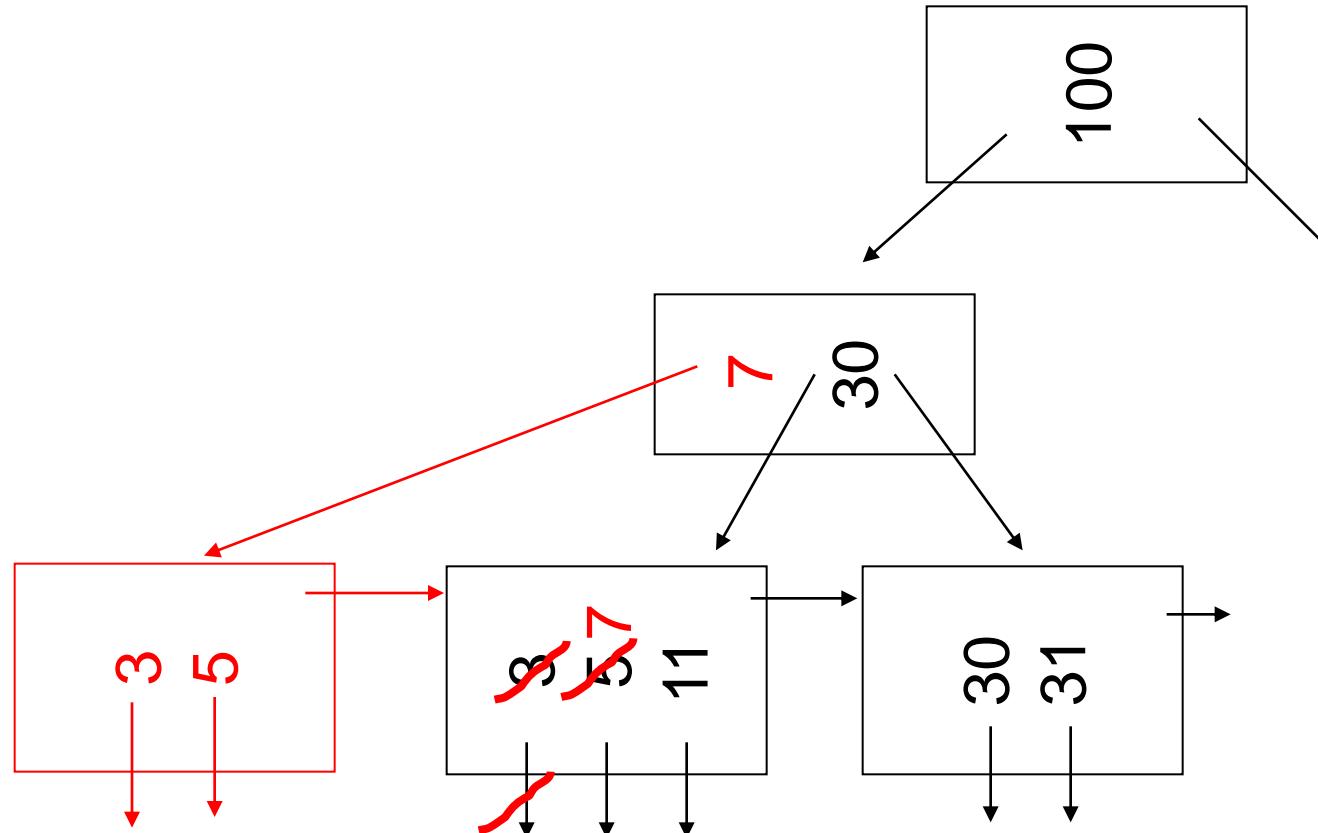
Insert 32

n=3



Insert 7

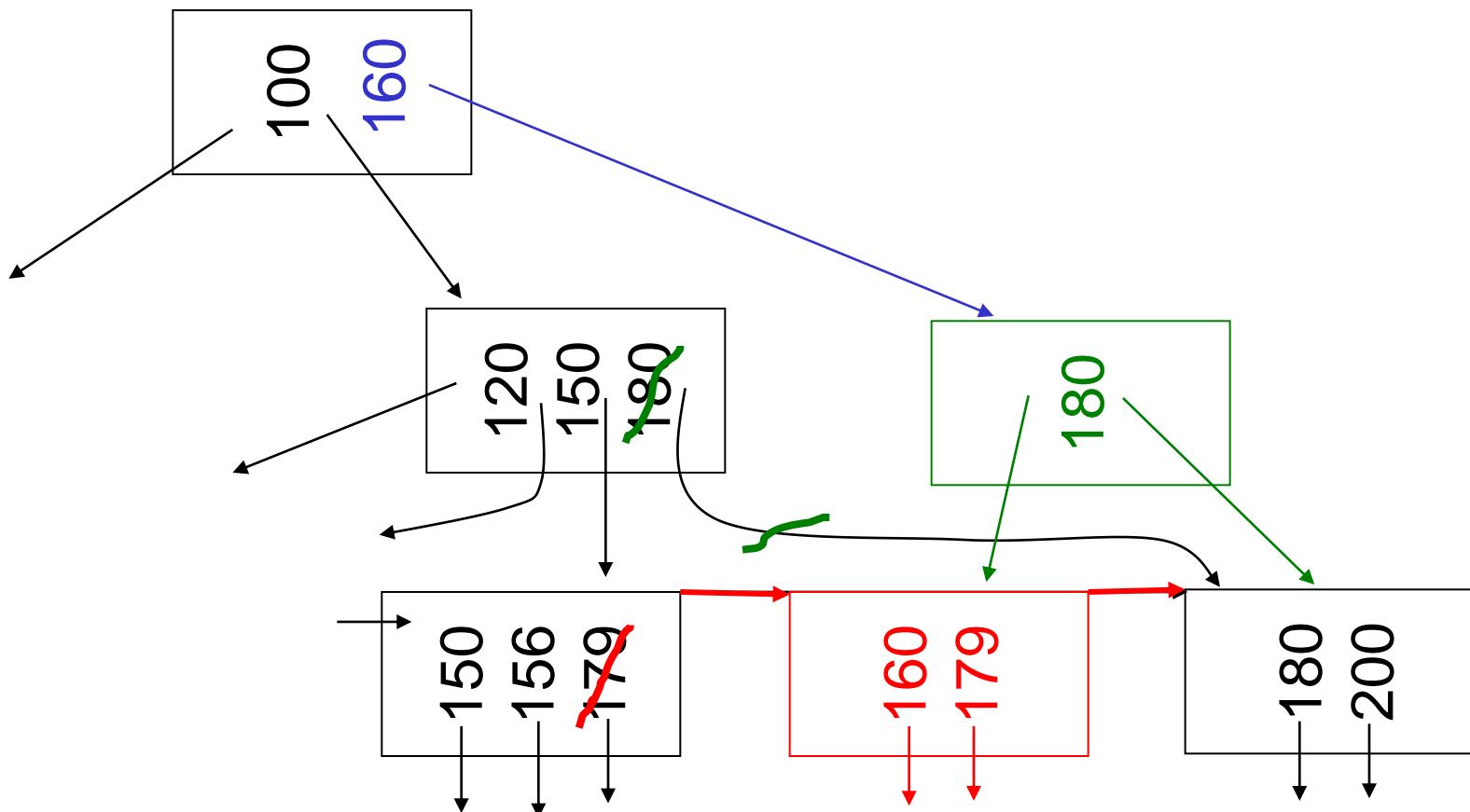
n=3



B⁺ stromy: vkladanie (C) non-leaf overflow

Insert 160

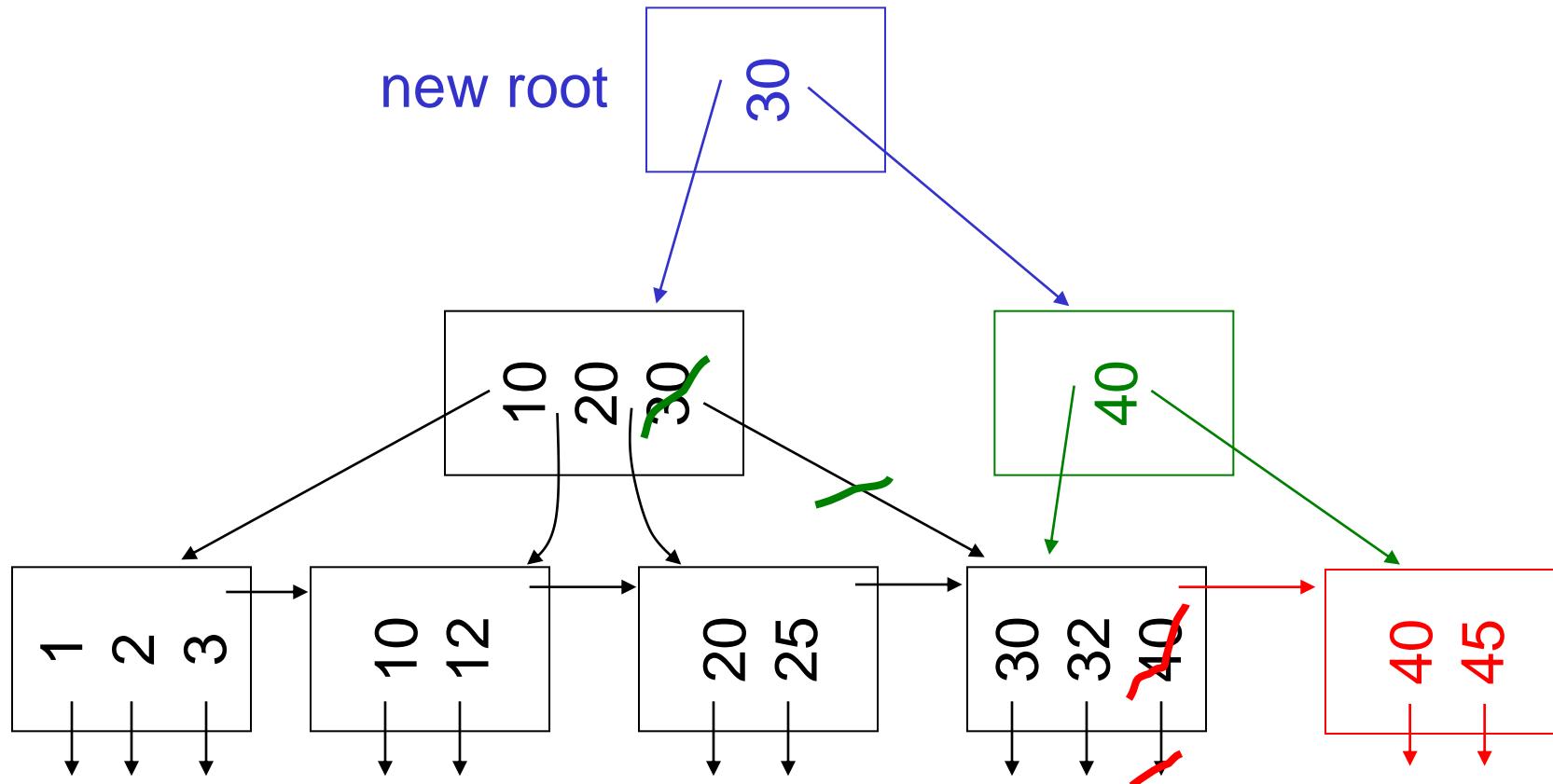
n=3



B⁺ stromy: vkladanie (D) new root

Insert 45

n=3



B⁺ stromy: vyniechanie

1. Nájdi a vyniechaj záznam
2. Ak list nie je podplnený, skonči (A)
3. Ak je list podplnený a ak je brat dostatočne plný, tak redistribuuju záznamy medzi listom a bratom a aktualizuj otca.
(C) Inak zlúč list s bratom a vyniechaj smerník a index z otca
(B)

Vyniechanie z vnútorného uzla je podobné ako vyniechanie z listu. Môže sa propagovať až ku koreňu.

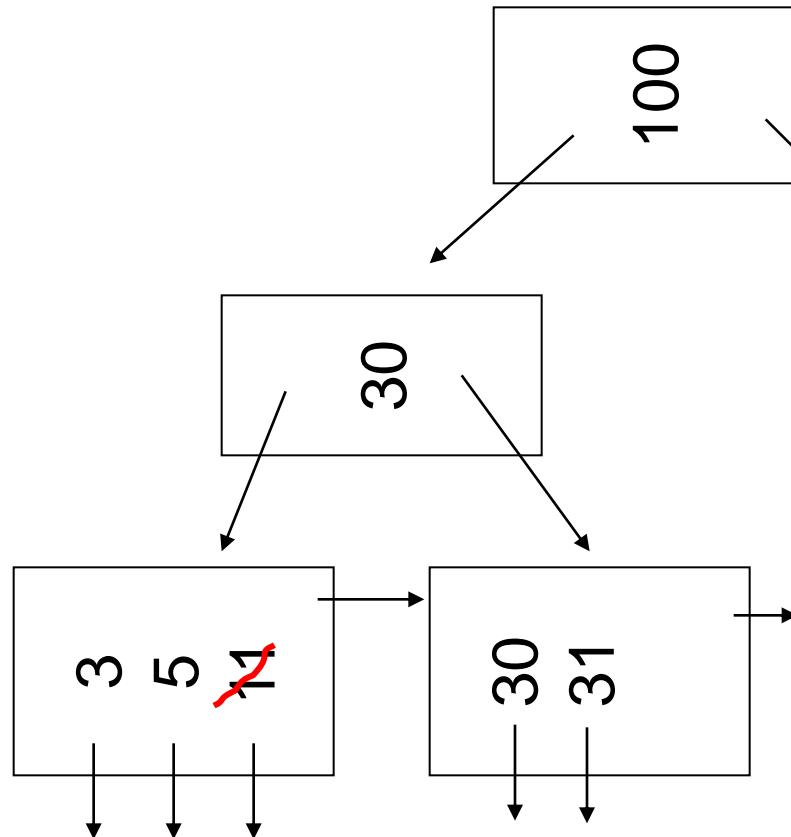
Ak vyniecháme poslednú hodnotu z koreňa, tento zanikne.
Novým koreňom sa stane jeho jediný syn. (D)

Zlučovanie uzlov je náročné na implementáciu, takže sa často neimplementuje. Preto indexové súbory majú po opakovanom vkladaní a vyniechávaní tendenciu rásť.

B⁺ stromy: výnechanie (A) simple case

Delete 11

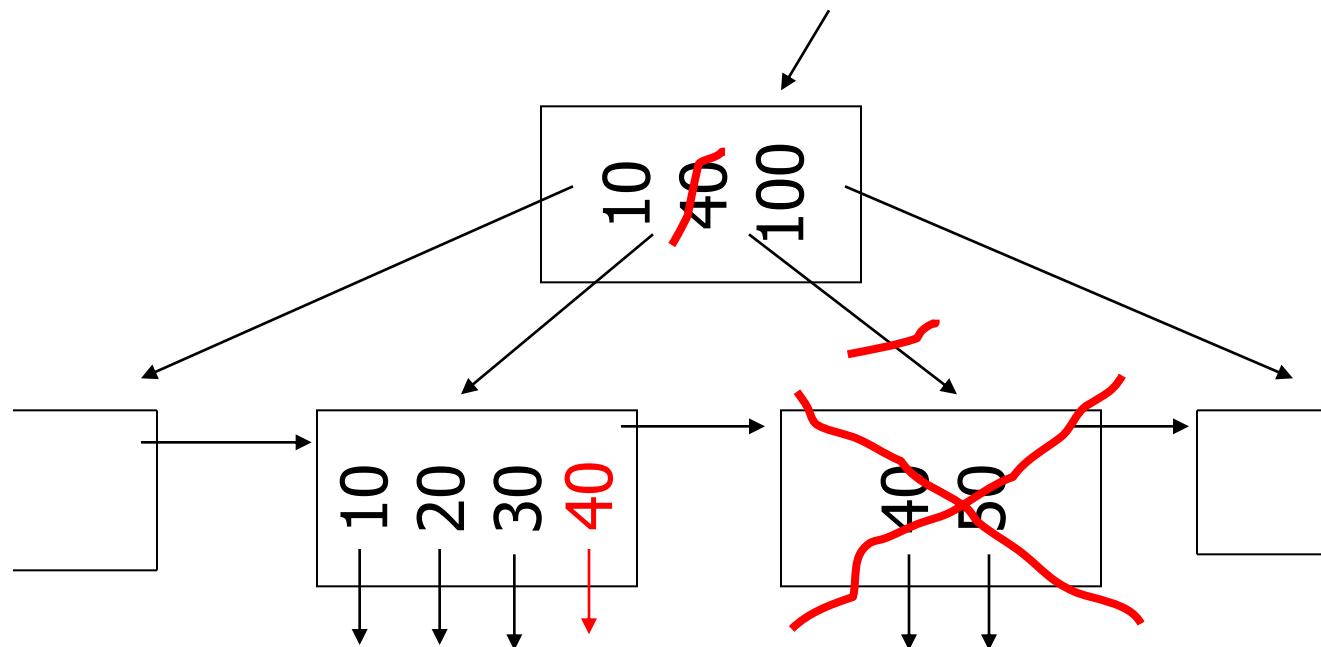
n=3



B⁺ stromy: vyniechanie (B) coalesce with sibling

Delete 50

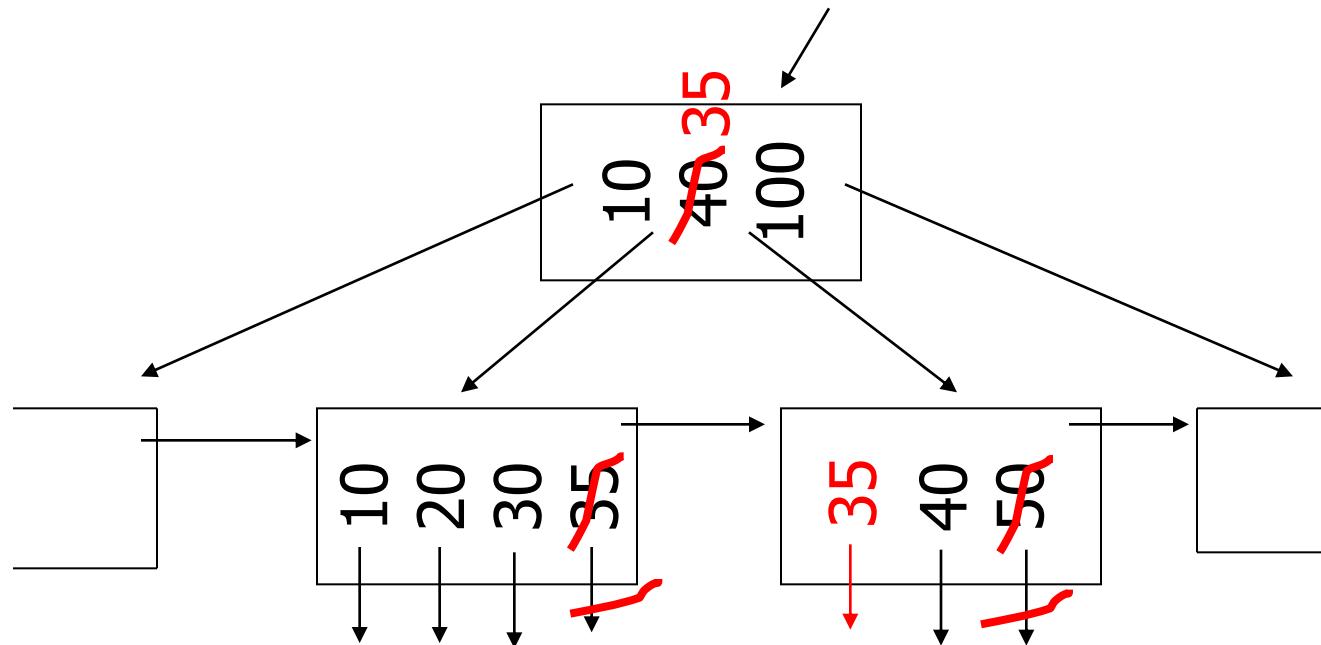
n=4



B⁺ stromy: vyniechanie (C) redistribute keys

Delete 50

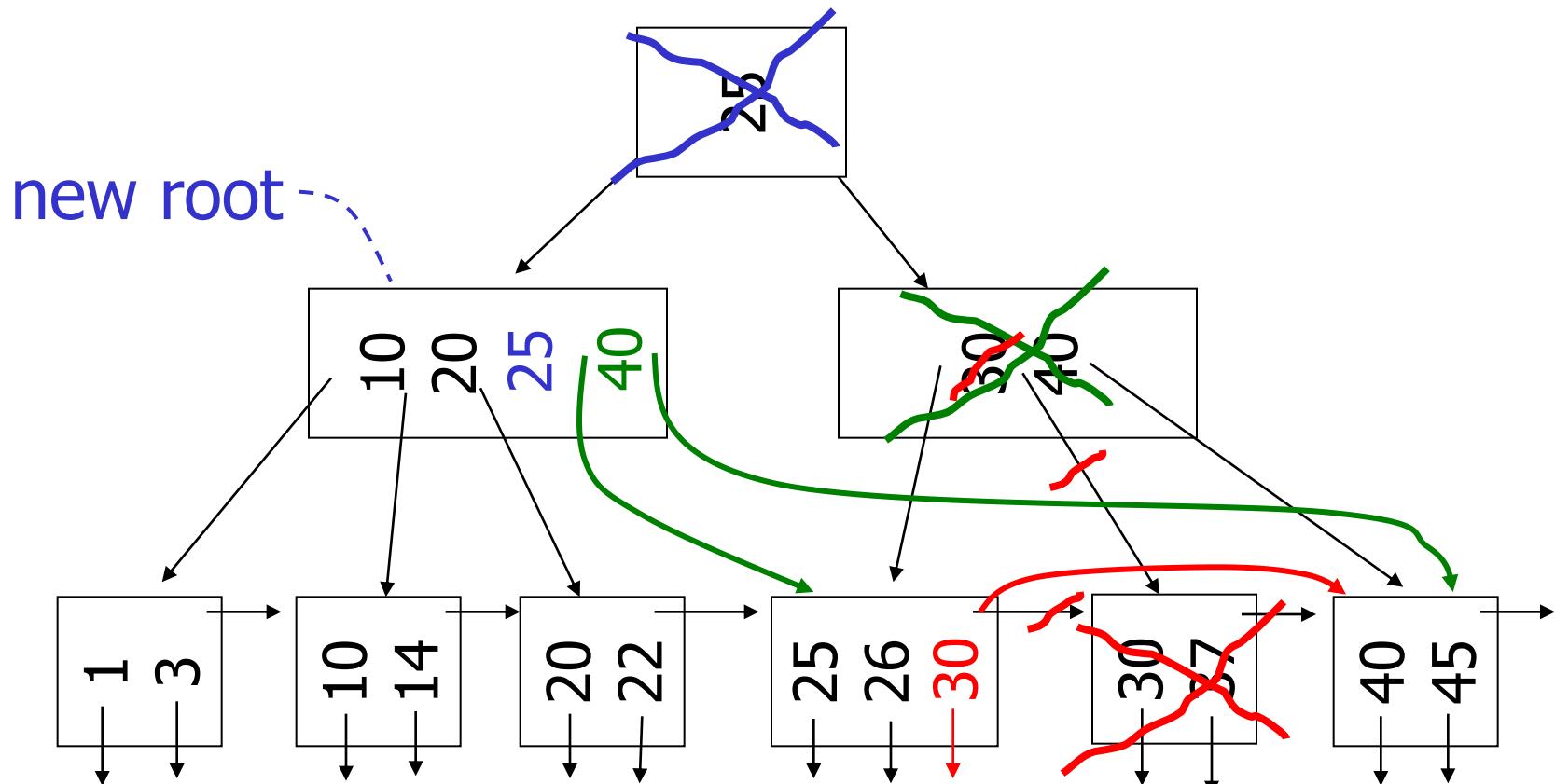
n=4



B⁺ stromy: vyniechanie (D) non-leaf coalesce

Delete 37

n=3



Výhody:

- vhodné pre sekvenčné prehľadávanie aj pre prehľadávanie podľa kľúča (3-4 diskové operácie)
- rýchle vkladanie a vynechávanie (zriedka vzniká potreba reorganizácie stromu)

Nevýhody:

- relatívne náročná implementácia (hoci pravdepodobne jednoduchšia ako implementácia B stromov)
- pamäťový overhead (oproti sekvenčným indexom treba pridať dodatočné stupne)

Idea: transformácia kľúča $\text{key} \rightarrow h(\text{key})$

Príklad hashovacej funkcie

- Key = ' $x_1 x_2 \dots x_n$ ' môžeme považovať za pole bytov dĺžky n
- Všetky možné stringy rozdelíme do B bucketov hashovacou funkciou $(x_1 + x_2 + \dots + x_n) \bmod B$

Iný príklad (hashovacia funkcia zachovávajúca usporiadanie):

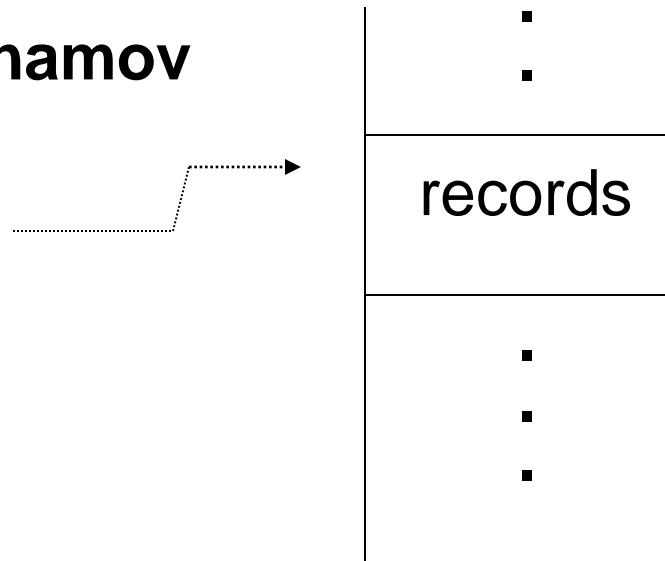
$$h(k) = \lfloor B^*(k-x)/(b-a) \rfloor, \text{ kde } a \leq k \leq b$$

Definícia (inžinierska). **Ideálna hashovacia funkcia** je taká, pre ktorú očakávaný počet záznamov v každom buckete je rovnaký (cez všetky okamžité stavy databázy)

Hashovanie: priama vs. nepriama adresácia záznamov

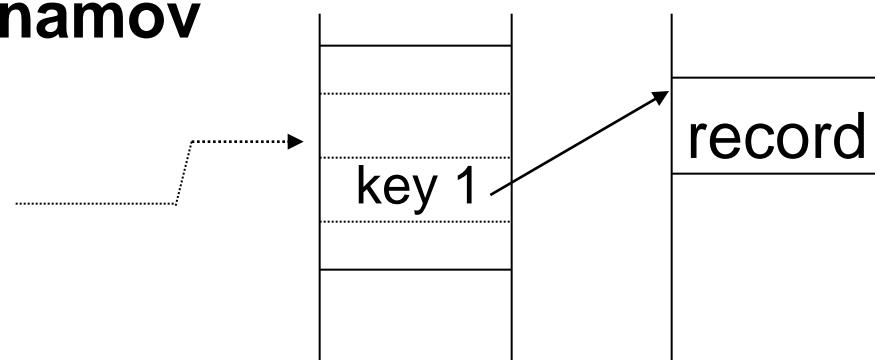
Priama adresácia záznamov

$\text{key} \rightarrow h(\text{key})$



Nepriama adresácia záznamov

$\text{key} \rightarrow h(\text{key})$



Index

Hashovaný súbor: directory, preplnenie

INSERT:

$$h(a) = 1$$

$$h(b) = 2$$

$$h(c) = 1$$

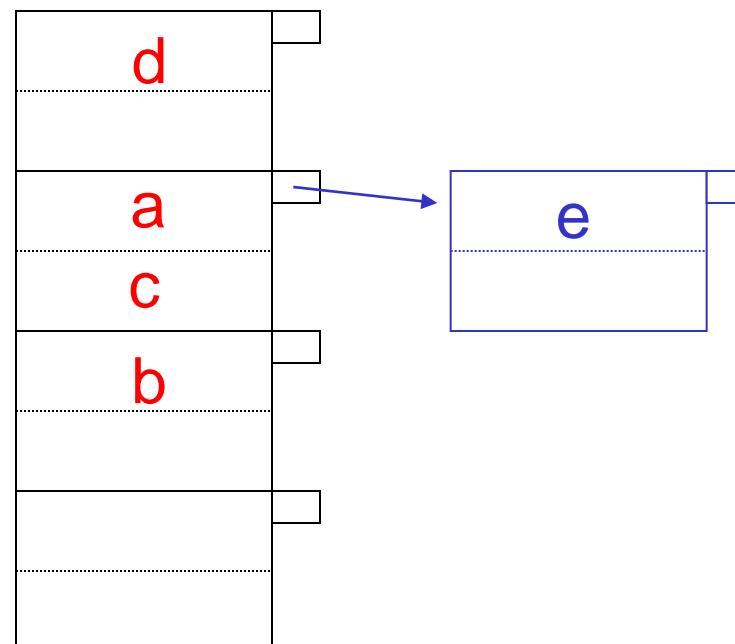
$$h(d) = 0$$

$$h(e) = 1$$

Directory
(adresár)



Základné
bloky



Bloky
preplnenia

Dynamické hashovanie: využitie priestoru

Využitie priestoru v percentách:

$$U = \# \text{ použitých blokov} / \# \text{ všetkých blokov}$$

U by malo byť medzi 50% a 80%

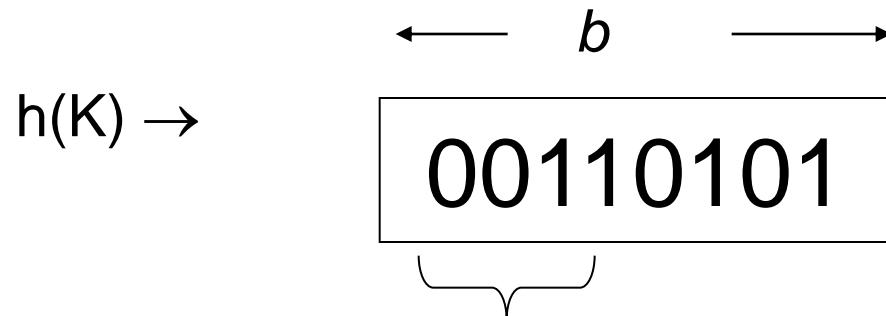
- $U < 50\%$ znamená plytvanie pamäťou, a plytvanie časom (pri diskových prenosoch sa prenášajú skoro prázdne bloky)
→ **reorganizácia (spájanie blokov)**
- $U > 80\%$ znamená dlhé sekvencie blokov preplnenia. Efektivita hashovania je vtedy porovnateľná s efektivitou sekvenčného indexu
→ **reorganizácia (zdvojnásobenie pamäťovej štruktúry)**

Dva spôsoby reorganizácie:

- **Rozšíritelné hashovanie**
- **Lineárne hashovanie**

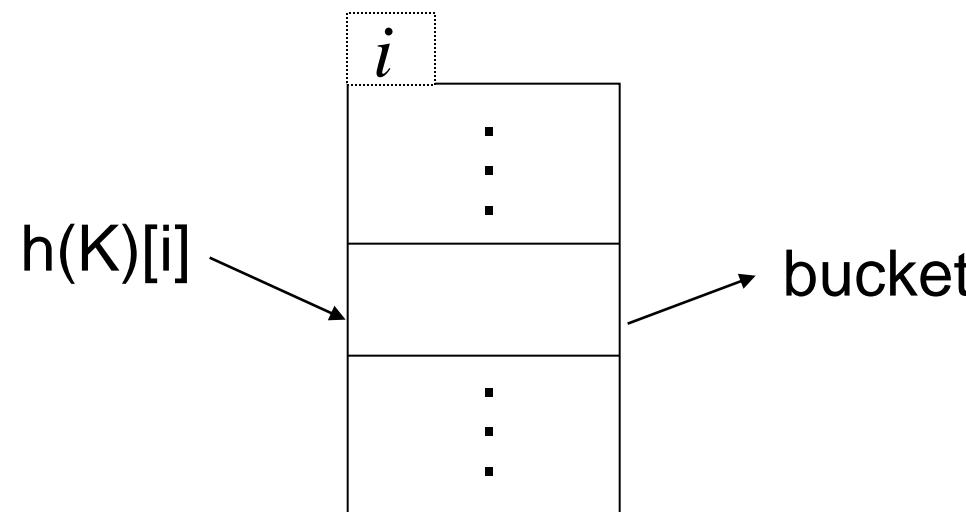
Rozšíriteľné hashovanie

(a) Použi len prvých i bitov z b bitov, ktoré vracia hashovacia funkcia



použi prvých i bitov → a pri preplnení zvýš i

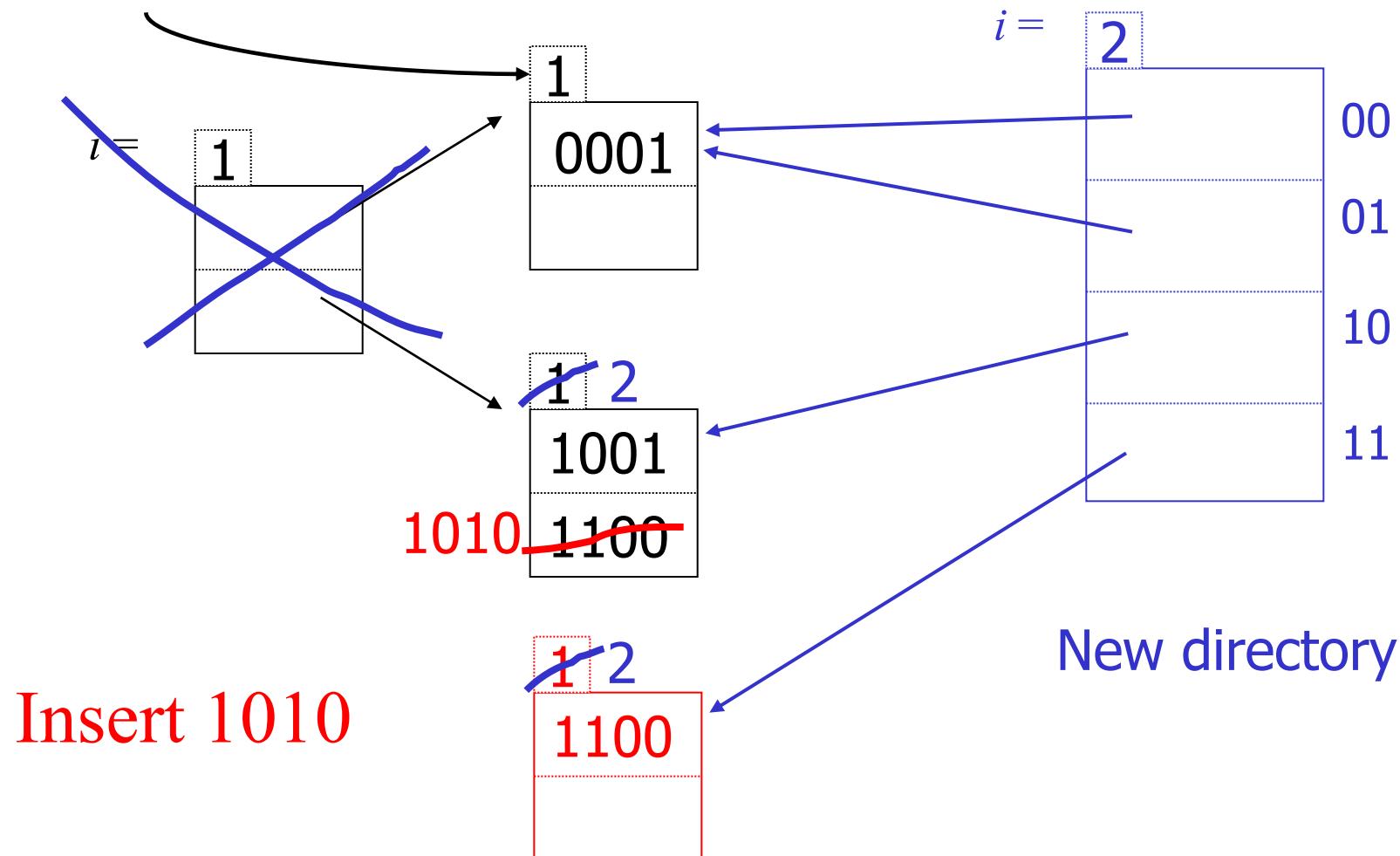
(b) Použi adresár



Rozšíriteľné hashovanie: vkladanie

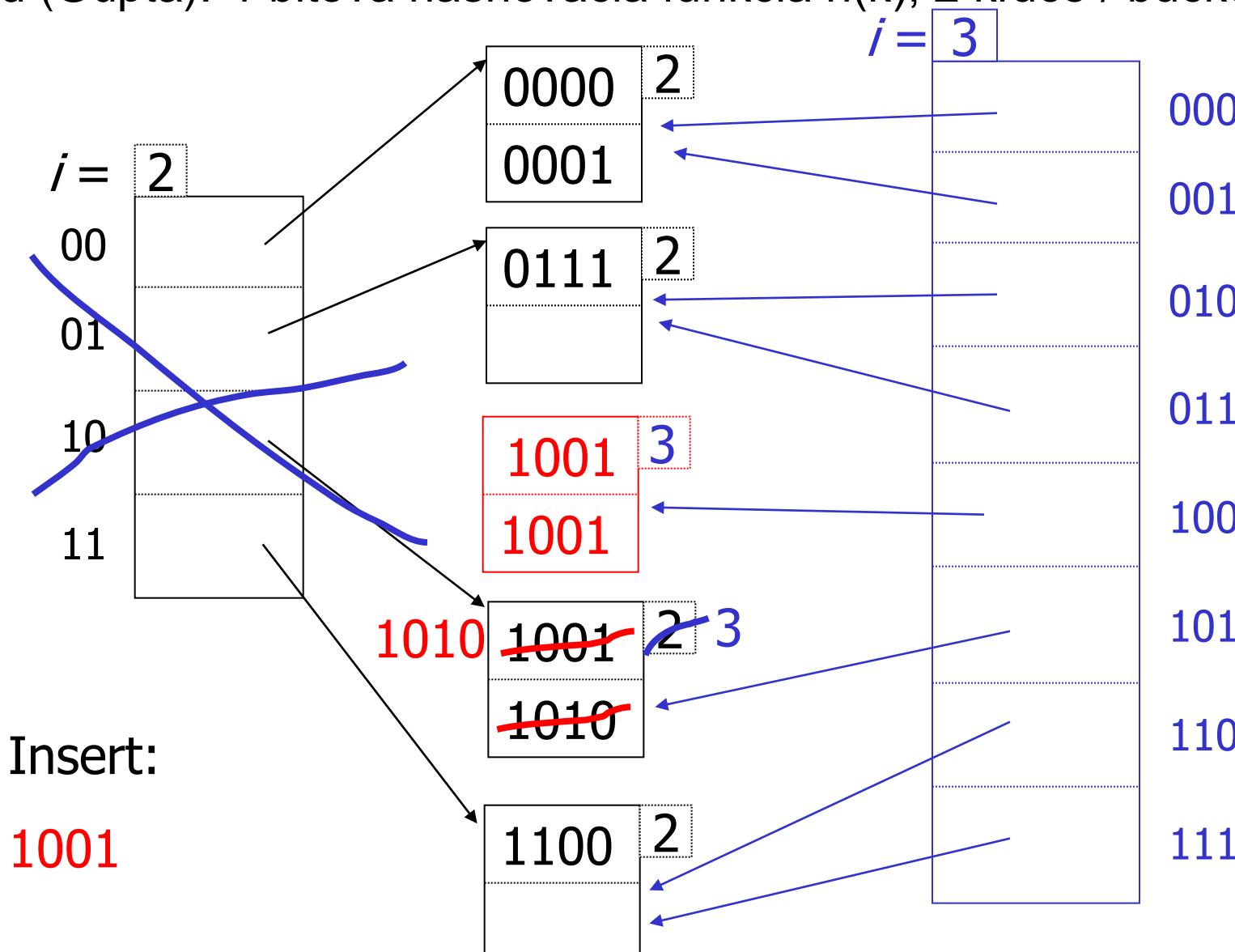
Príklad (Gupta): 4-bitová hashovacia funkcia $h(k)$, 2 kľúče / bucket

'Nub' -> Number of bits použitých
na určenie bloku



Rozšíriteľné hashovanie: vkladanie

Príklad (Gupta): 4-bitová hashovacia funkcia $h(k)$, 2 kľúče / bucket



Lineárne hashovanie

Základná myšlienka: vyhnúť sa rézii s adresárom:

- na adresár stačí jeden bit (existuje/neexistuje blok)
- adresár si netreba pamätať, stačí si pamätať adresu posledného použitého bloku (t.j. momentálnu maximálnu adresu)

Nekonečná trieda hashovacích funkcií: $h_{p,q}(K) = K \bmod (2^q * p)$

Nech $N = 2^q * p$. Potom $h_{p,q}(K) = h_N(K) = K \bmod (N)$

Adresár: $A = \text{array}[1..N]$ of bits

Platí (dôležité pre vyhľadávanie a vkladanie):

$$K \bmod 2N = \text{buď } K \bmod N, \text{ alebo } (K \bmod N) + N$$

Vyhľadanie bloku (resp. kľúča):

```
n=N;  
k=K mod N; /* hashovacia funkcia */  
while ((k>p) and (not A[k]))  
{  
    n = n / 2; /* celočíselné delenie */  
    k = k - n;  
}
```

Vloženie záznamu:

Ak blok k nie je plný:
vlož záznam do bloku k;

Inak, ak $n < N$:
vlož záznam do bloku $k+n$;

Inak zdvojnásob adresár A:

for $i=N$ downto 0

{

 A[n+i] = FALSE;

}

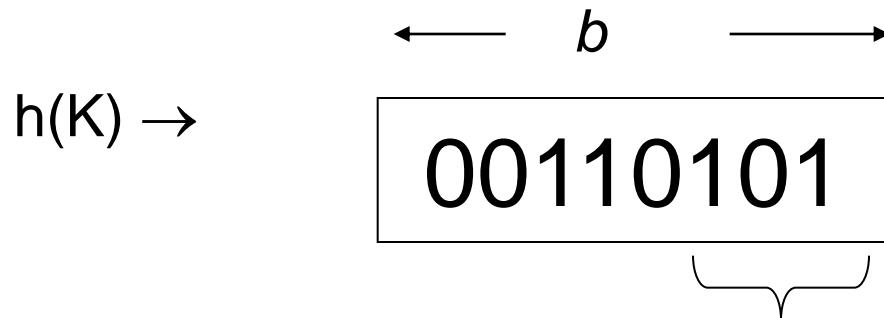
 A[k+n]=TRUE;

 vlož záznam do bloku $k+n$;

 N = 2 * N; /* modifikácia hashovacej funkcie */

Lineárne hashovanie

(a) Použi len posledných m bitov z b bitov, ktoré vracia hashovacia funkcia

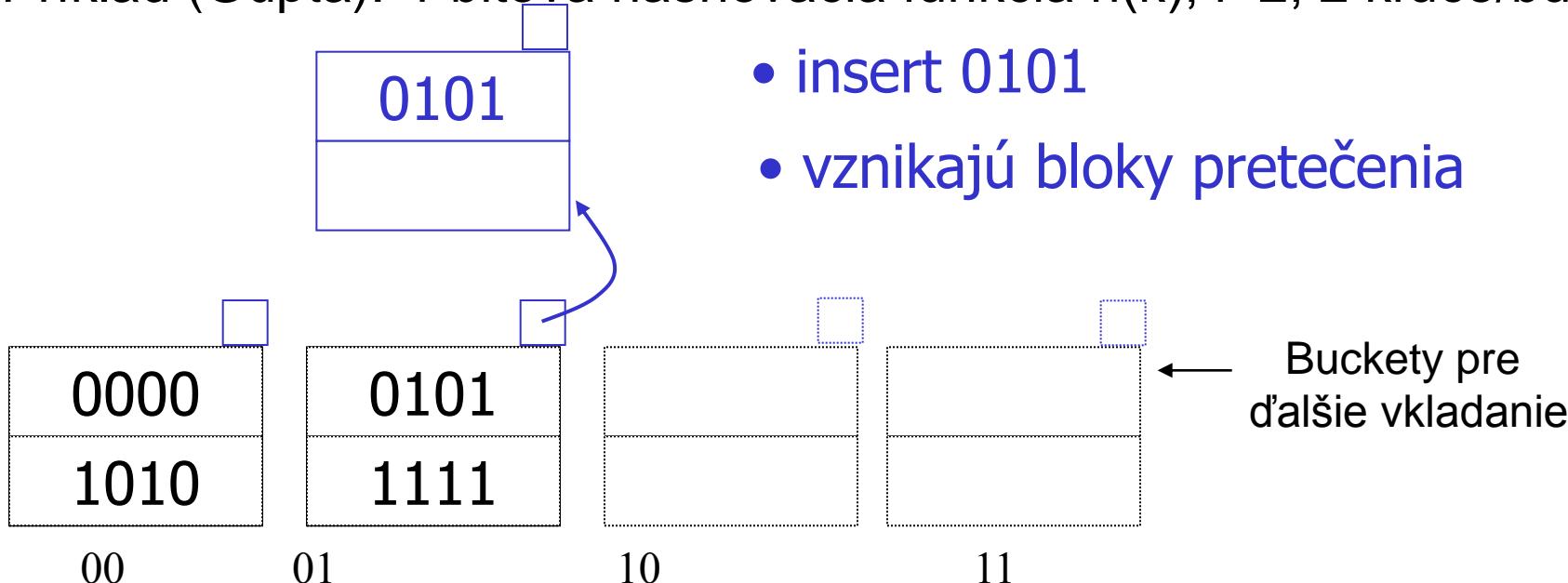


použi prvých m bitov \rightarrow a pri preplnení zvýš m

(b) Pamätaj si len adresu posledného bloku

Lineárne hashovanie

Príklad (Gupta): 4-bitová hashovacia funkcia $h(k)$, $i=2$, 2 kľúče/bucket

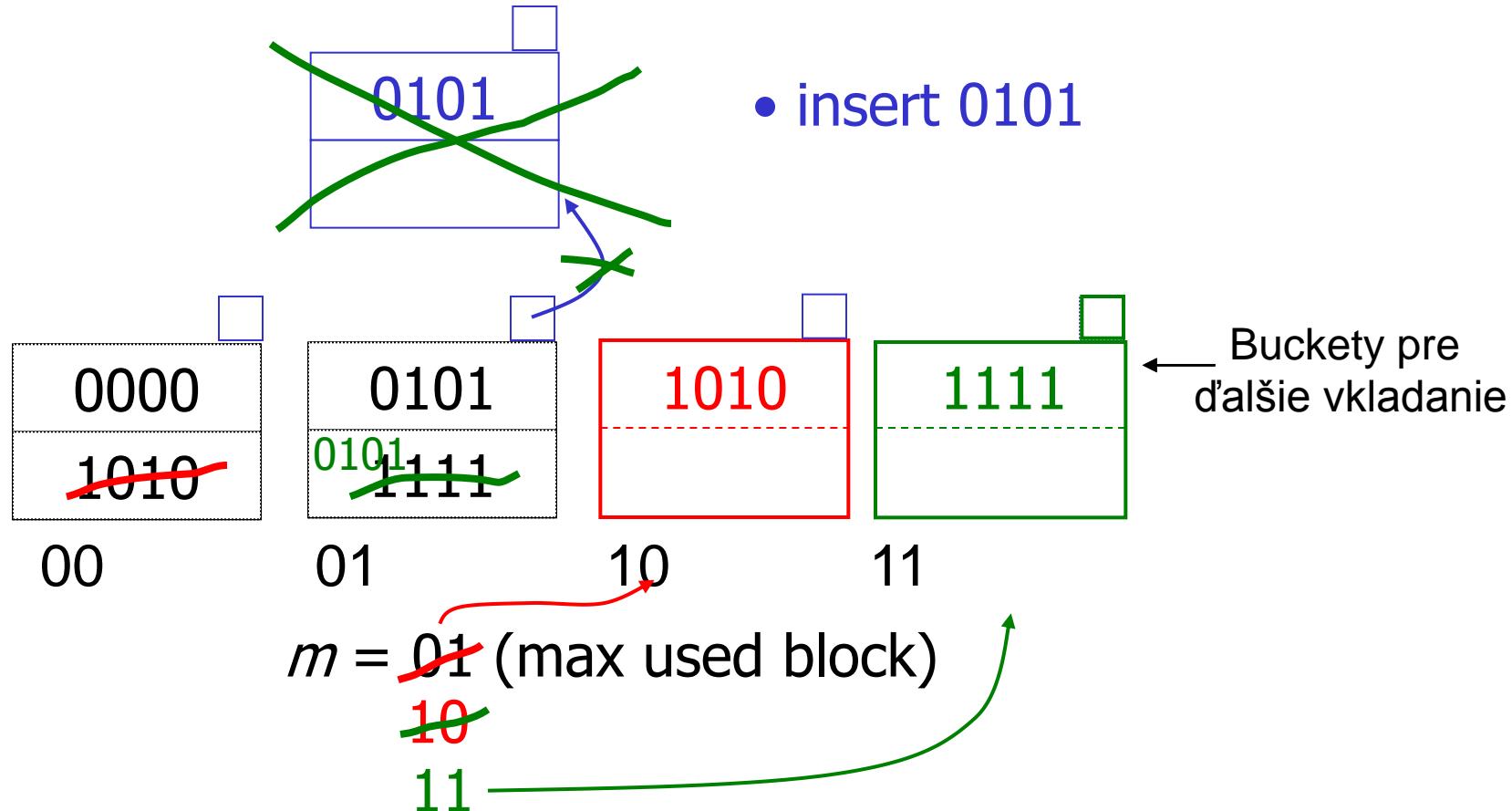


$$m = 01 \text{ (max used block)}$$

```
if  $h(k)[i] \leq m$ , then  
    hľadaj v buckete  $h(k)[i]$   
else  
    hľadaj v buckete  $h(k)[i] - 2^{i-1}$ 
```

Lineárne hashovanie

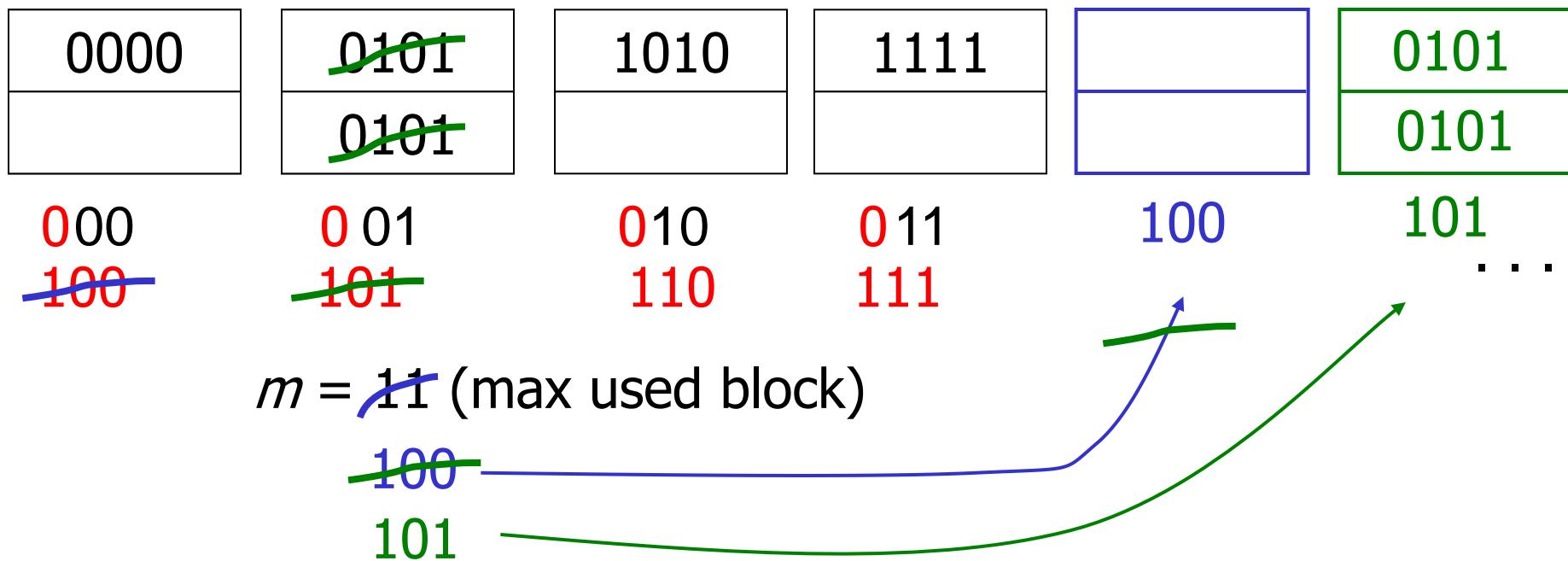
Príklad (Gupta): 4-bitová hashovacia funkcia $h(k)$, $i=2$, 2 kľúče/bucket



Lineárne hashovanie

Príklad (Gupta): 4-bitová hashovacia funkcia $h(k)$, $i=2$, 2 kľúče/bucket

$$i = \cancel{2} \ 3$$



Cena reorganizácie

Predpokladajme, že dátová štruktúra veľkosti n sa dá vytvoriť v čase $O(n \log(n))$. Počas existencie štruktúry (bez reorganizácie) sa vykoná $O(n)$ operácií, pričom každá operácia trvá $O(\log(n))$. Potom treba štruktúru reorganizovať. Reorganizácia zväčší veľkosť štruktúry na qn (kde $q > 1$, napr. $q=2$).

Potom sa tá reorganizácia dokázateľne oplatí, lebo amortizovaná cena operácií ostane konštantná!

$$\lim_{n \rightarrow \infty} \frac{\sum_{i=0}^n (q \log(q))^i}{n q^n} = \lim_{n \rightarrow \infty} \frac{\log(q)}{n q^n} \sum_{i=0}^n i q^i = \text{const} \quad /* \text{Amortizovaná cena operácií} */$$

$$\sum_{i=0}^n i q^i = q \sum_{i=0}^n i q^{i-1} = q \frac{\partial}{\partial q} \left(\frac{q^n - 1}{q - 1} \right) = \frac{(q-1) \cancel{q^{n+1} - q^n}}{\cancel{(q-1)^2}}$$

$$\lim_{n \rightarrow \infty} \frac{\log(q)}{n q^n} \frac{(q-1) \cancel{q^{n+1} - q^n}}{\cancel{(q-1)^2}} = \lim_{n \rightarrow \infty} \frac{(q-1) \log q}{n \cancel{(q-1)}} = \frac{\log q}{q-1}$$

/* Výpočet tej sumy */

/* Výpočet tej limity */

QED

Hashovanie vs. B⁺ stromy

Hashovanie je výhodnejšie, ak sú časté dotazy na rovnosť kľúča

SELECT ...

FROM R, S

WHERE R.A = 42

Teoreticky je očakávaný čas vyhľadávania hashovaného záznamu konštantný! (Bohužiaľ, iba teoreticky, lebo nájst' dobrú a zároveň rýchlu hashovaciu funkciu je problém.)

B⁺ stromy sú výhodnejšie, ak sú časté intervalové dotazy na kľúč, (ale pomáhajú aj pri dotazoch na rovnosť a poskytujú veľmi dobré garancie aj pre najhorší prípad)

SELECT ...

FROM R, S

WHERE R.A > S.B

Vybrané fyzické operátory: index scan

Algoritmus index scan (vyžaduje existenciu index-file podľa testovaného klúča)

Nájdi prvý vyhovujúci záznam, ďalšie stačí hľadať v nasledujúcich blokoch.

$\text{cost} = \text{height} + b + 1$, kde height je hĺbka indexového stromu a b je počet blokov obsahujúcich vyhovujúce záznamy

Vybrané fyzické operátory: index scan

Iterator **IndexScan(R)**

```
open(R) {      oscon=# explain select oid from pg_proc where oid=1;  
               QUERY PLAN  
   R.open();  
}  
-----  
close(R) {     Index Scan using pg_proc_oid_index on pg_proc  (cost=0.00..5.99  
   R.close();    rows=1 width=4)  
}  
               Index Cond: (oid = 1::oid)  
next(R) {  
  return R.index().next();  
}
```

Algoritmus join (hash join)

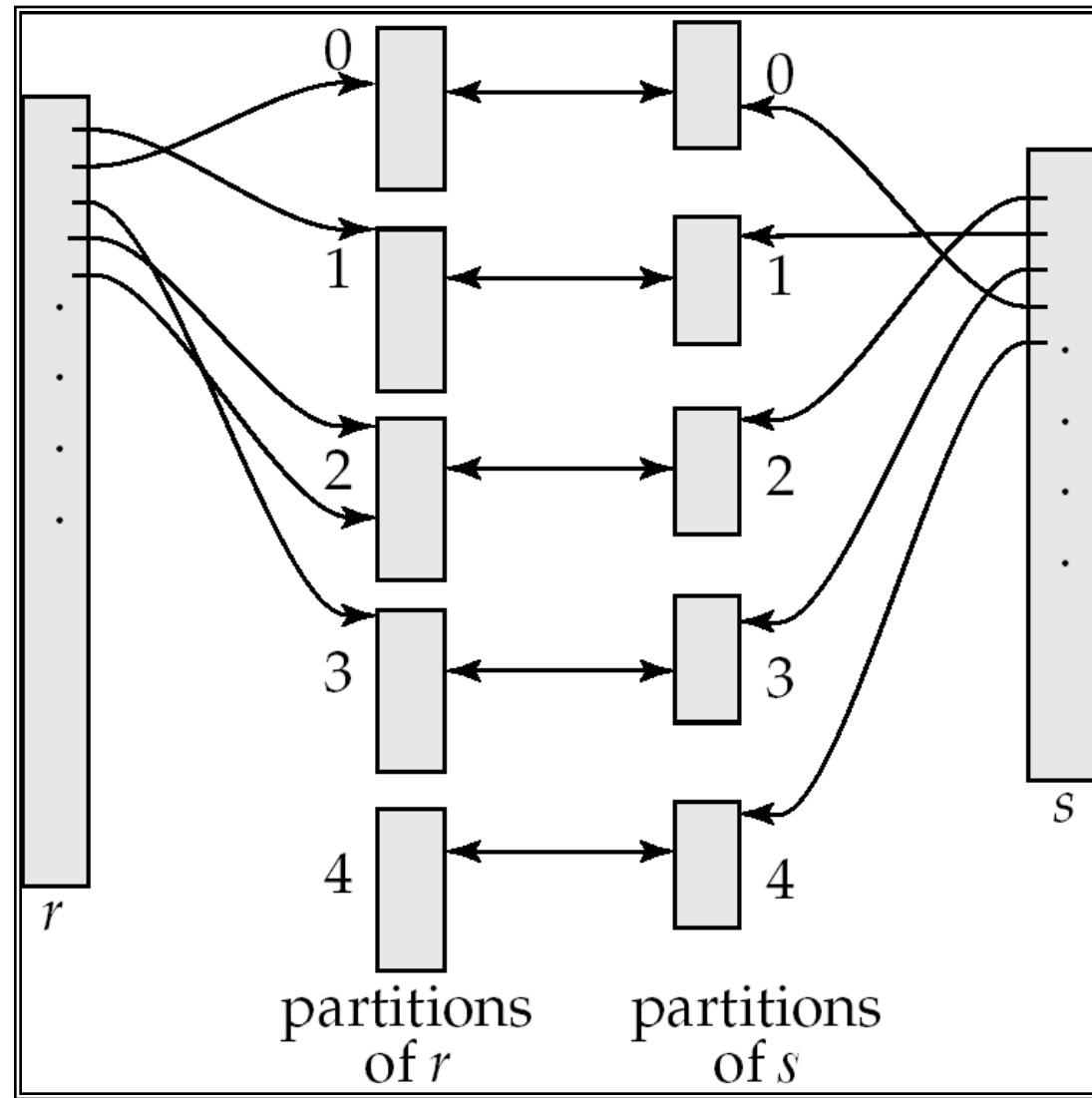
Tento algoritmus dovoľuje len joinovaciu podmienku na rovnosť (equijoin, resp. natural join)

Základná myšlienka: predpokladajme, že každý hashovaný bucket sa zmestí do pamäte. (V hashovanom buckete sú záznamy s rovnakou hodnotou hashovacej funkcie.) V takom prípade stačí opakovane ukladať do pamäte hashované buckety R a S, lebo *len* pre záznamy s rovnakým hashovacím kľúčom *môže byť* joinovacia podmienka splnená

(Ak nie je pravda, že každý hashovaný bucket zmestí do pamäte, treba toto opakovat'.)

Vybrané fyzické operátory: hash join

Príklad (Silberschatz et al.):



Algoritmus join (hash join):

1. Vytvor hashované buckety R_1, R_2, \dots, R_p z relácie R , a podobne vytvor hashované buckety S_1, S_2, \dots, S_p
2. Pre každý bucket i , $i=1,\dots,p$:
 - načítaj jeden bucket R_i a vytvor preň nový hash (index)
 - pre každý záznam bucketu S_i otestuj na základe novej hashovacej funkcie, či sa môže nachádzať v R_i . Ak áno, nájdi záznamy, ktoré joinujú a zapíš do výstupného bloku. Ak je výstupný blok plný, zapíš ho na disk

$$\text{cost} = 3 * (B(R) + B(S)) + 4 * p$$

(Cena písania na disk tu nie je zahrnutá, lebo veľkosť joinu závisí od joinovacej podmienky.)

- Paradigma rozdeľuj a panuj
("rozdeľovanie = neporiadok, usporiadavanie = poriadok")
 - Stromy: fyzické rozdeľovanie, logické usporiadavanie
 - Hashovanie: logické rozdeľovanie, fyzické usporiadavanie
- Spracovanie dlhých vstupov (väčších ako operačná pamäť)
 - Stromy: viacfázové spájanie
 - Hashovanie: viacfázové rozdeľovanie
- Vstupno-výstupné vzory (I/O patterns)
 - Stromy: sekvenčný zápis, náhodné čítanie (merge)
 - Hashovanie: náhodný zápis, sekvenčné čítanie (buckety)