Databázové systémy

Tomáš Skopal

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

Osnova přednášky

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

Algoritmus atributového uzávěru

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

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

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

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

Příklad – atributový uzávěr

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

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

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

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

Algoritmus příslušnosti

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

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

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

Testování redundancí

Algoritmus příslušnosti lze jednoduše použít k testu redundance

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

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

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

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

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

Minimální pokrytí

- použijeme postupně na všechny FZ testy redundance a ty odstraníme

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

decompose each dependency in F into elementary ones

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

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

endfor

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

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

endfor

return F;

Nalezení (prvního) klíče

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

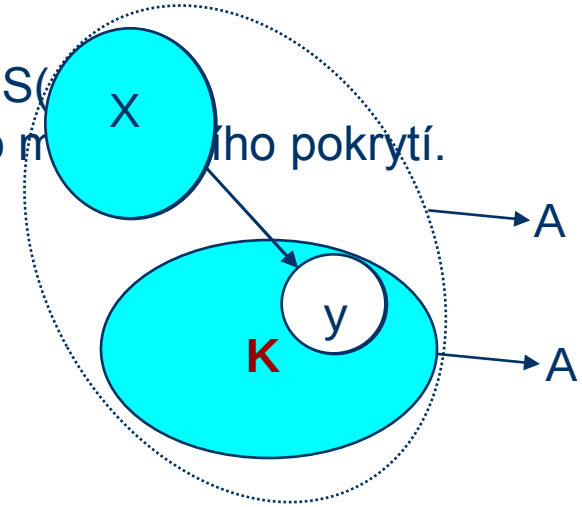
algorithm ***GetFirstKey***(set of deps. F, set of attributes A) : **returns a key** K;

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

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

Nalezení všech klíčů, princip

Máme schéma S(
Převeďme F do n ího pokrytí.



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

Nalezení všech klíčů, algoritmus

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

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

Příklad – nalezení všech klíčů

Contracts(A, F)

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

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

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

Testování normálních forem

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

Návrh schématu databáze

Dva způsoby modelování relační databáze:

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

Normalizace relačního schématu

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

Proč zachovávat integritu?

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

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

Firma
Sun
Oracle
Microsoft
IBM

Firma

Sídlo
Santa Clara
Redwood
Redmond
New York

Sídlo

Nadmořská výška
25 mnm
20 mnm
10 mnm
15 mnm

Nadmořská výška

Firma,
Sídlo → Nadmořská výška

Evidentně je ale s takovouto dekompozicí něco špatně...

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

Bezztrátovost

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

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

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

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

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

Příklad – ztrátová dekompozice

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

Firma, Používá DBMS

Firma	Používá DBMS
Sun	Oracle
Sun	DB2
Microsoft	MSSQL
Microsoft	Oracle

Firma, Používá DBMS

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

Firma, Spravuje dat

„rekonstrukce“
(přirozené spojení)

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

Firma, Používá DBMS, Spravuje dat

Příklad – bezztrátová dekompozice

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

Firma,
Sídlo → Nadmořská výška

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

Firma

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

Sídlo

„rekonstrukce“
(přirozené spojení)

Pokrytí závislostí

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

Příklad – pokrytí závislostí

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

pokrytí porušeno, ztratili jsme
Sídlo → Nadmořská výška

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

Firma

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

Sídlo

Firma,
Sídlo → Nadmořská výška

pokrytí zachováno

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

Firma

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

Sídlo

Algoritmus „Dekompozice“

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

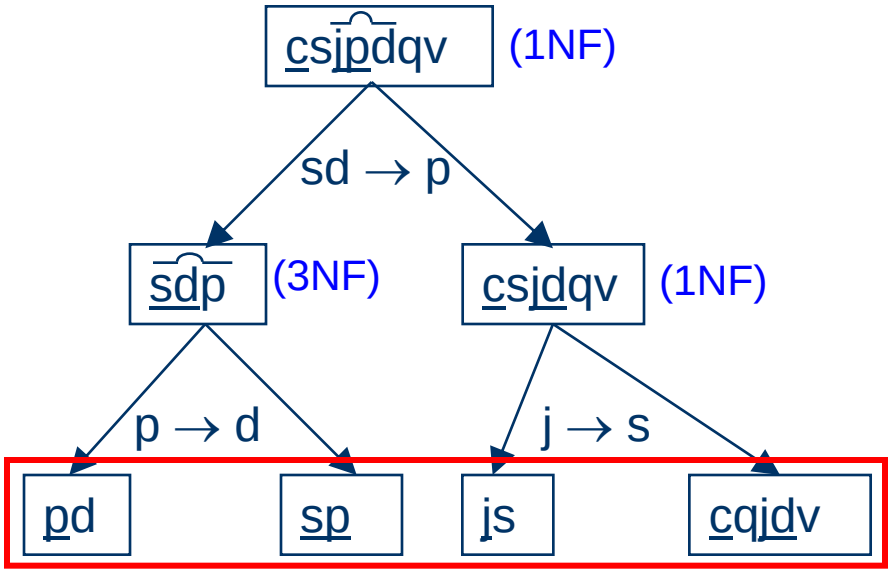
algorithm ***Decomposition***(set of elem. deps. F, set of attributes A) : **returns set** $\{R_i(A_i, F_i)\}$
 Result := {R(A, F)};
 Done := **false**;
 Create F^+;
 while not Done **do**
  if $\exists R_i(F_i, A_i) \in$ Result not being in BCNF **then** *// pokud ve výsledku je schéma porušující BCNF*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

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

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

Příklad – dekompozice

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



Nepokryté závislosti:
c → p
sd → p

cp (BCNF)

sdp (3NF)

Algoritmus „Syntéza“

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

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

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

Příklad – syntéza

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

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

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

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

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

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

Bernsteinovo rozšíření syntézy

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

Demo

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