21121350
Database System

Lecture 6: Relational Database Design

Lu Chen (陈璐)
College of Computer Science
Zhejiang University
Spring & Summer 2023
luchen@zju.edu.cn/18868818726

Outline

- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce-Codd Normal Form
- Third Normal Form
- Multivalued Dependencies
- Fourth Normal Form

First Normal Form

- Domain is atomic if its elements are considered to be indivisible units.
  - Examples of non-atomic domains:
    - Composite attributes --- set of names
    - Multi-value attribute --- a person’s phones
    - Complex data type--- object-oriented

- A relational schema *R* is in first normal form (1NF) if the domains of all attributes of *R* are atomic.

- For the relational database, it’s required that all relations are in 1NF.

First Normal Form (Cont.)

- How to deal with non-atomic values =>
  - For composite attributes: use a number of attributes.
  - For multi-value attributes:
    - Use multi fields, e.g., *person*(*pname*, …, *phon1*, *phon2*, *phon3*, …);
    - Use a separate table, e.g., *phone*(*pname*, *phone*);
    - Use a single field, e.g., *person*(*pname*, …, *phones*, …)

- Drawbacks of non-atomic strategy:
  - Complicate storage
  - Encourage redundant storage of data
  - Complicated to query

First Normal Form (Cont.)

- Atomicity is actually a property of how the elements of the domain are used.
  - E.g., Strings would normally be considered indivisible.
  - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*.
  - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
  - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

Outline

- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce-Codd Normal Form
- Third Normal Form
- Multivalued Dependencies
- Fourth Normal Form

Pitfalls in Relational Database Design

- Relational database design requires that we find a “good” collection of relation schemas.
- A bad design may lead to:
  - Redundant storage, insert / delete / update anomalies --- inability to represent certain information.
- Example: Consider the relation schema: *Lending-schema* = (*branch-name*, *branch-city*, *assets*, *customer-name*, *loan-number*, *amount*)

Note: There are two design methods:

(1) Top-down

(2) Button-up:

Universal relation (泛关系) ➡ decomposition

➡ good database schema

Sample *Lending* Relation

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Key: (*branch-name*, *customer-name*, *loan-number*)

Deficiencies for the *Lending* Relation

- Redundancy:
  - Data for *branch-name*, *branch-city*, and *assets* are repeated for each loan that a branch makes
  - Drawback: Wastes space, may result in inconsistency.
- Updating anomaly: Complicates updating, introducing possibility of inconsistency, e.g., modify *assets* value, many tuples need be changed.
- Insert / delete anomalies. (if have a key: (*branch-name*, *customer-name*, *loan-number*))
  - Or use Null values: (If have no key)
    - To store information about a branch if no loans exist, can use null values, but they are difficult to handle.

Decomposition

- Main refinement technique: decomposition, e.g., replacing (*ABCD*) with, (*AB*) and (*BCD*), or (*ACD*) and (*ABD*), or (*ABC*) and (*CD*), or (*AB*), (*BC*), and (*CD*), or (*AD*) and (*BCD*).

- Example: Decompose the *Lending-schema* into: *Branch-schema* = (*branch-name*, *branch-city*, *assets*), and *Loan-info-schema* = (*branch-name*, *customer-name*, *loan-number*, *amount*)

- All attributes of an original schema (R) must appear in the decomposition (R_1, R_2), i.e., $R = R_1 \cup R_2$
- Lossless-join decomposition (无损连接分解), i.e., for all possible relations r on schema R,

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

Requirement for decomposition

Example of Non Lossless-Join Decomposition

- Decomposition of $R = (A, B)$

  $R_1 = (A)$, $R_2 = (B)$

- E.g.:

$r =$

A	B
α	1
α	2
β	1

$\Rightarrow$

A
α
β

$\Pi_A(r)$

$+$

B
1
2

$\Pi_B(r)$

$\Pi_A (r) \bowtie \Pi_B (r) =$

A	B
α	1
α	2
β	1
β	2

$\neq r$

A bad decomposition!

Thus, it's not lossless-join decomposition. It's illegal!

Goal: Devise a Theory for the Following

- Decide whether a particular relation *R* is in "good" form. --- No redundant
- In the case that a relation *R* is not in "good" form, decompose it into a set of relations $\{R_1, R_2, ..., R_n\}$ such that
  - Each relation is in good form.
  - The decomposition is a lossless-join decomposition.
- Our theory is based on:
  - Functional dependencies (函数依赖)
  - Multivalued dependencies (多值依赖)

Outline

- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce-Codd Normal Form
- Third Normal Form
- Multivalued Dependencies
- Fourth Normal Form

Functional Dependencies

- Let R be a relation schema, α and β be attributes, i.e., $\alpha \subseteq R$ and $\beta \subseteq R$

借用了数学上的函数概念: $y \rightarrow f(x)$

- The functional dependency $\alpha \rightarrow \beta$ holds on R if and only if for any legal relations $r(R)$, whenever any two tuples t_1 and t_2 of r agree on the attributes α, they also agree on the attributes β, i.e.,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

β is functionally dependent on α, α functionally determines β.

β函数依赖于α, α函数决定β.

α	β	γ
a	f	1
b	h	2
a	f	3
c	f	4

Functional Dependencies (Cont.)

- Functional dependency --- a kind of integrity constraints, which express the relationship of values on specific attributes, can be used to judge schema normalization and to suggest refinements.

- Example: Consider $r(A, B)$ with the following instance of r.
  - On this instance, $A \rightarrow B$ does NOT hold,
  - But $B \rightarrow A$ may hold.

∵若B属性值确定了，则A属性值也唯一确定了；因此$B \rightarrow A$成立。

A	*B*
1	4
1	5
3	7

Functional Dependencies (Cont.)

- Functional dependency vs. key
  - A functional dependency is a generalization of the notion of a key.
  - K is a superkey for the relation schema R if and only if $K \to R$.
  - K is a candidate key for R if and only if
    - $K \to R$, and
    - No $\alpha \subset K$, $\alpha \to R$ (不存在K的真子集α，使之满足$\alpha \to R$)
  - Functional dependencies allow us to express constraints that cannot be expressed using keys.
    - Consider the schema: *Loan-info-schema* = (<u>*customer-name*, *loan-number*</u>, *branch-name*, *amount*). We expect this set of functional dependencies to hold:

    函数依赖集 F = {*loan-number* → *amount*, *loan-number* → *branch-name*,
    (*customer-name*, *loan-number*) → *amount*,
    (*customer-name*, *loan-number*) → *branch-name*}

    but would not expect the following to hold:

    loan-number → *customer-name*

The Use of Functional Dependencies

- We use functional dependencies to:
  - (1) Test relations to see if they are legal under a given set of functional dependencies F.
    - If a relation r is legal under a set F of functional dependencies, we say that r satisfies F.

$r =$

A	B	C	D
a1	b1	c1	d1
a1	b2	c1	d2
a2	b2	c2	d2
a2	b3	c2	d3
a3	b3	c2	d4

$F = \{$

$A \rightarrow C$,

$AB \rightarrow D$, $\Rightarrow$ $\{A, B\} \rightarrow D$

$ABC \rightarrow D$

$\}$, but

$A \nrightarrow B$, $A \nrightarrow D$, $B \nrightarrow A$, $C \nrightarrow A$, $C \nrightarrow D$,

$B \nrightarrow C$, $C \nrightarrow B$, $B \nrightarrow D$, …

The Use of Functional Dependencies (Cont.)

- We use functional dependencies to:
  - (2) Specify constraints (F) on the set of legal relations --- schema.
    - We say that F holds on R (F在R上成立) if all legal relations r on R satisfy the set of functional dependencies F.

$r_1(R)$ =

A	B	C	D
a1	b1	c1	d1
a1	b2	c1	d2
a2	b2	c2	d2
a2	b3	c2	d3
a3	b3	c2	d4

F = {

A → C,

$AB \rightarrow D$,

$ABC \rightarrow D$

}

$r_2(R)$ = …

$r_3(R)$ = …

……

C.f.: a relation r satisfies F?
F hold on schema R?

Note: 容易判别一个r是否满足给定的F;
不易判别F是否在R上成立。不能仅由某个r推断出F。
R上的函数依赖F, 通常由定义R的语义决定。

We can say the relation $r(R)$ satisfies F, but we cannot, in accord with only a $r(R)$, say: F holds on schema R.

Student(*sno*, *sname*, *ssex*, *sage*)
F = {*sno* → {*sname*, *ssex*, *sage*},
? *sname* → *sno*, *sname* → *sage* , …}

Definition of Trivial and Non-Trivial Dependency

- *A* functional dependency is trivial (平凡的) if it is satisfied by all relations
  - E.g., $A \rightarrow A$, $AB \rightarrow A$
    - (*customer-name*, *loan-number*) $\rightarrow$ *customer-name*
    - *customer-name* $\rightarrow$ *customer-name*
  - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$, otherwise, is non-trivial, i.e.,

    Trivial: $\alpha \rightarrow \beta$, if $\beta \subseteq \alpha$ (平凡的函数依赖)

    Non-trivial: $\alpha \rightarrow \beta$, if $\beta \nsubseteq \alpha$ (非平凡的函数依赖)

Closure of a Set of Functional Dependencies

- Given a set of functional dependencies F, there are certain other functional dependencies that are logically implied by F.
  - E.g., If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- Definition: The set of all functional dependencies logically implied by F is the *closure of F*, denoted by F^+ (函数依赖集F的闭包)
  - E.g., $F = \{A \rightarrow B, B \rightarrow C\}$, $F^+ = \{A \rightarrow B, B \rightarrow C, A \rightarrow C, A \rightarrow A, AB \rightarrow A, AB \rightarrow B, AC \rightarrow C, A \rightarrow BC, \ldots\}$
- How to find F^+?
  - E.g., $R = (A, B, C, G, H, I)$

    $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$

    $F^+ = ?$

Closure of a Set of Functional Dependencies (Cont.) - Armstrong's Axioms

- Armstrong's Axioms provide inference rules to find F^+.
- We can find all of F^+ by applying Armstrong's Axioms:
  - If $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (reflexivity, 自反律) --- trivial
  - If $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$ (augmentation, 增补律)

    $\gamma\alpha \rightarrow \beta$
  - If $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (transitivity, 传递律)
- These rules are
  - Sound (保真的, generate only functional dependencies that actually hold).
  - Complete (完备的, generate all functional dependencies that hold).

Example

- $R = (A, B, C, G, H, I)$
  $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
- Some members of F^+
  - $A \rightarrow H$ by transitivity from $A \rightarrow B$ and $B \rightarrow H$
  - $AG \rightarrow I$ $\underline{A \rightarrow C} \xrightarrow{aug} AG \rightarrow CG$, Given $\underline{CG \rightarrow I}$, $\xrightarrow{trans} AG \rightarrow I$
  - $AG \rightarrow H$ $\underline{A \rightarrow C} \xrightarrow{aug} AG \rightarrow CG$, Given $\underline{CG \rightarrow H}$, $\xrightarrow{trans} AG \rightarrow H$

(pseudotransitivity, 伪传递律)

  - $CG \rightarrow HI$ $\underline{CG \rightarrow H} \xrightarrow{aug} CG \rightarrow CGH$, $\underline{CG \rightarrow I} \xrightarrow{aug} CGH \rightarrow HI$, $\xrightarrow{trans} CG \rightarrow HI$
  - $A \rightarrow BC$ $\underline{A \rightarrow B} \xrightarrow{aug} A \rightarrow AB$, $\underline{A \rightarrow C} \xrightarrow{aug} AB \rightarrow BC$, $\xrightarrow{trans} A \rightarrow BC$

(union, 合并律)

Armstrong's Axioms的补充定律

- We can further simplify manual computation of F^+ by using the following additional rules.
  - If $\alpha \to \beta$ and $\alpha \to \gamma$ holds, then $\alpha \to \beta\gamma$ holds (union, 合并律)
  - If $\alpha \to \beta\gamma$ holds, then $\alpha \to \beta$ and $\alpha \to \gamma$ holds (decomposition, 分解律)
  - If $\alpha \to \beta$ and $\gamma\beta \to \delta$ holds, then $\alpha\gamma \to \delta$ holds (pseudotransitivity, 伪传递律)

- The above rules can be inferred from Armstrong's axioms.

Proof: Rule1: $\beta\gamma \to \beta$, $\beta\gamma \to \gamma$, given $\alpha \to \beta\gamma$, by Rule3, then $\alpha \to \beta$ and $\alpha \to \gamma$

Procedure for Computing F^+

- To compute the closure of a set of functional dependencies *F*:

  $F^+ = F$

  Repeat

  For each functional dependency *f* in F^+

  Apply reflexivity and augmentation rules on *f*

  Add the resulting functional dependencies to F^+

  For each pair of functional dependencies f_1 and f_2 in F^+

  If f_1 and f_2 can be combined using transitivity

  Then add the resulting functional dependency to F^+

  Until F^+ does not change any further

- Note: The maximum number of possible Functional Dependencies (FDs) is $2^n \times 2^n$, for *n* attributes. --- We will see an alternative procedure for this task later.

Closure of Attribute Sets

- How to test whether *a* is a superkey?
  - Method 1: First find F^+, and then for all $a \rightarrow \beta_i$ in F^+, see whether $\{\beta_1, \beta_2, \beta_3, \ldots\} = R$? --- But it is not easy to compute F^+.
  - Method 2: Find the closure of *a*.

- Definition: Given a set of attributes *a*, the closure of *a* under *F*, denoted by a^+, is the set of attributes that are functionally determined by *a* under *F* (在*F*下由*a*所直接和间接函数决定的属性的集合称为a^+).

- To test $a \rightarrow \beta$ is in $F^+ \Leftrightarrow \beta \subseteq a^+$
- To test *a* is a superkey

  $a \rightarrow R$ is in $F^+ \Leftrightarrow R \subseteq a^+$

The use of attribute set closure

Closure of Attribute Sets (Cont.)

- How to get a^+?
  - Algorithm for computing a^+, the closure of a under F

    result := *a*;
    while (changes to *result*) do
        for each $\beta \rightarrow \gamma$ in F do
            begin
                If $\beta \subseteq$ *result* then *result* := *result* $\cup\ \gamma$
            end;
        a^+ := *result*

例如

已知
$\alpha \rightarrow \beta$,
$\beta \rightarrow \gamma$,
$\alpha^+ = ?$

$\alpha \rightarrow$ a, result={a},
$\alpha \rightarrow \beta$, result={a β},
$\beta \rightarrow \gamma$, result={a β γ} = a^+,

? a $\rightarrow \gamma$,
{a β γ} = a^+, $\therefore$(a $\rightarrow \gamma$)

避免了找F^+(反复使用公理)的麻烦.

Example of Attribute Set Closure

- Example 1: $R = (A, B, C, G, H, I)$
  - $F = \{A \rightarrow B$
    $A \rightarrow C$
    $CG \rightarrow H$
    $CG \rightarrow I$
    $B \rightarrow H\}$
  - Is AG a candidate key?

$(AG)^+$

1. $result = AG$
2. $result = ABCG$ ($\because A \rightarrow B$, $A \rightarrow C$, and $A \subseteq AG$)
3. $result = ABCGH$ ($\because CG \rightarrow H$ and $CG \subseteq ABCG$)
4. $result = ABCGHI$ ($\because CG \rightarrow I$ and $CG \subseteq ABCGH$)

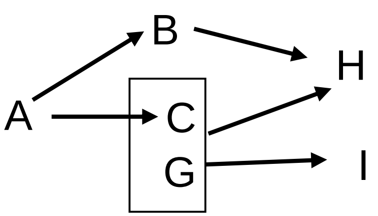
1. Is AG a super key? α^+
   1. Does $AG \rightarrow R$? == is $(AG)^+ \supseteq R$ $\quad \therefore$ AG is a superkey
2. Is AG a candidate key? --- Is any subset of AG a superkey? X
   1. Does $A \rightarrow R$? $\Rightarrow$ Is $(A)^+ \not\supseteq R$ $\because (A)^+ = ABCH$
   2. Does $G \rightarrow R$? $\Rightarrow$ Is $(G)^+ \not\supseteq R$ $\because (G)^+ = G$

$\therefore$ AG is a candidate key

- Is AB a candidate key?

Example of Attribute Set Closure (Cont.)

$\therefore$ *AB* is not a candidate key.

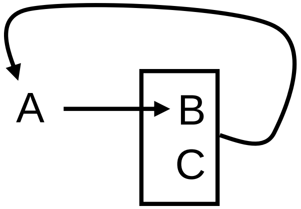


$F = \{A \rightarrow B,$
$A \rightarrow C,$
$CG \rightarrow H,$
$CG \rightarrow I,$
$B \rightarrow H\}$

- Example 2: $R = (A, B, C)$, $F = \{A \rightarrow B, BC \rightarrow A\}$, which is candidate key?

  $(BC)^+ = (BCA) \supseteq R$. $(AC)^+ = (ACB) \supseteq R$. $(AB)^+ = (AB) \not\supseteq R$

  $\therefore$ Candidate key: *BC*, *AC*



Uses of Attribute Set Closure

- There are 3 kind uses of the attribute set closure algorithm:
  - Testing for a superkey --- ($\alpha \to R$?)
    - To test if α is a superkey, we compute α^+ and then check if α^+ contains all attributes of R, i.e., check if $R \subseteq \alpha^+$
  - Testing functional dependencies --- ($\alpha \to \beta$?)
    - To check if a functional dependency $\alpha \to \beta$ holds (or, in other words, is in F^+), only check if $\beta \subseteq \alpha^+$.
    - It's a simple and cheap test, and very useful.
  - Computing the closure of F --- ($F^+ = ?$)
    - For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \to S$, and all $\gamma \to S$ form F^+.

Canonical Cover (正则覆盖)

- DBMS should always check to ensure not violate any Functional Dependency (FD) in *F*.
  - If the *F* is too big, the check is costly. Thus, we need simplify the set of FDs.

- Intuitively, a canonical cover of *F*, denoted by F_c, is a “minimal” set of FDs equivalent to *F*.
  - Having no redundant FDs and no redundant parts of FDs, i.e., no functional dependency in *Fc* contains an extraneous attribute.
  - Each left side is unique.
  - E.g., $\alpha_1 \rightarrow \beta_1$, $\alpha_1 \rightarrow \beta_2$, $\Rightarrow \alpha_1 \rightarrow \beta_1\beta_2$

$$F_c \underset{\textbf{Imply}}{\overset{\textbf{Logically}}{==========}} F$$

Canonical Cover (Cont.)

- How to get $F_c \Rightarrow$ delete extraneous attributes (多余属性)

- There are 3 cases for the extraneous attributes:

- (1) Sets of Functional Dependency (FD) may have redundant dependencies that can be inferred from the others
  - E.g.: $A \to C$ is redundant in:

$$F = \{\cancel{A \to C}, A \to B, B \to C\}$$

$$\Rightarrow \quad F_c = \{A \to B, B \to C\}$$

Canonical Cover (Cont.)

- (2) Parts of a functional dependency on left side may be redundant-- extraneous attributes
  - E.g., $F = \{A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$ can be inferred to

$$\Rightarrow \{A \rightarrow B, B \rightarrow C, \underline{AC \rightarrow D, A \rightarrow D}\}, \Rightarrow \{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$

$\therefore$ F is simplified to $F' = \{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$, i.e., Attribute C is extraneous (多余的).

由F: $B \rightarrow C$, => $\underline{AB \rightarrow AC}$, 又∵ $\underline{AC \rightarrow D}$, ∴ $\underline{AB \rightarrow D}$;

又∵ $\underline{A \rightarrow B}$, => $A \rightarrow AB$, ∴ $A \rightarrow D$, ∴ F蕴涵F'

由Armstrong's Axioms, $A \rightarrow D$ implies $AC \rightarrow D$

Canonical Cover (Cont.)

- (3) Parts of a functional dependency on right side may be redundant
  - E.g., $F = \{A \rightarrow B, B \rightarrow C, A \rightarrow CD\}$ can be inferred to

    $\Rightarrow \{A \rightarrow B, B \rightarrow C, A \rightarrow C, A \rightarrow D\}$,

    but $A \rightarrow C$ is implied by $A \rightarrow B, B \rightarrow C$,

  $\therefore F$ is simplified to: $F' = \{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$, (即F'蕴涵F), i.e.,
  Attribute C is extraneous.

Extraneous Attributes (无关属性)

- Consider the functional dependency $\alpha \rightarrow \beta$ in F.
  - Attribute A is extraneous in α, if $A \in \alpha$ and F logically implies $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
    - E.g., $\alpha = \{A\alpha'\}$, $\{A\alpha'\} \rightarrow \beta$. 若$F$蕴涵$\alpha' \rightarrow \beta$, 则$\{A\alpha'\} \rightarrow \beta$多余, 即$A$多余.
    - Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$

      Because $F = \{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$, $\therefore$ B is extraneous in $AB \rightarrow C$, $F' = \{A \rightarrow C, A \rightarrow C\} = \{A \rightarrow C\}$
  - Attribute A is extraneous in β, if $A \in \beta$ and the set of functional dependencies $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.
    - E.g., $\beta = A\beta'$, $\alpha \rightarrow \{A\beta'\}$,有$\{\alpha \rightarrow A, \alpha \rightarrow \beta'\}$. 若$F'$ 蕴涵$\alpha \rightarrow A$, 则$\alpha \rightarrow A$多余, (即可用F'代替F).
    - Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$

      Since $AB \rightarrow CD \Rightarrow \{AB \rightarrow C, AB \rightarrow D\}$, and $AB \rightarrow C$ can be inferred from $F' = \{A \rightarrow C, AB \rightarrow D\}$, $\therefore$ C is extraneous in $AB \rightarrow CD$

Testing if an Attribute is Extraneous

$(\alpha')^+$

- To test if attribute $A \in \alpha$ is extraneous in α:
  - Compute $(\alpha - A)^+$ using the dependencies in F
  - Check that $(\alpha - A)^+$ contains β; if it does, A is extraneous.

$\alpha = \{A\alpha'\}$, $\{A\alpha'\} \to \beta$. 若F蕴涵$\alpha' \to \beta$, 则A多余. 故只要证明$\beta \in (\alpha')^+$.

- To test if attribute $A \in \beta$ is extraneous in β:
  - Compute α^+ using only the dependencies in F'

    $$F' = (F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\}$$

  - Check that α^+ contains A; if it does, A is extraneous.

$\beta = \{A\beta'\}$, $\alpha \to \{A\beta'\}$. 若F'蕴涵$\alpha \to A$, 则A可删. 故只要在F'下证明$A \in (\alpha)^+$.

Canonical Cover (Cont.)

- 正则覆盖F_c是函数依赖集F的最小化。得到F_c的关键步骤是消去现有函数依赖中的extraneous（无关的、多余的）属性，从而排除相应的函数依赖，使函数依赖集最小化。

- 要消去现有函数依赖$\alpha \rightarrow \beta$中的extraneous（无关的、多余的）属性，无非有2种情况：
  - (1) Extraneous属性在左边（即α中，$\alpha = A\alpha', A\alpha' \rightarrow \beta$），
  - (2) Extraneous属性在右边（即β中，$\beta = A\beta', \alpha \rightarrow A\beta'$）。

- 对于情况(1), $A\alpha' \rightarrow \beta$: 如果$\alpha' \rightarrow \beta$已经由原来的函数依赖集$F$所蕴涵（即$F$中已经包含了$\alpha' \rightarrow \beta$，或$F$可以推出$\alpha' \rightarrow \beta$），则根据Armstrong公理，$\alpha' \rightarrow \beta$可以推出$A\alpha' \rightarrow \beta$，因此$A\alpha' \rightarrow \beta$是多余的(replace $A\alpha' \rightarrow \beta$ with $\alpha' \rightarrow \beta$)，也即$A$是多余的属性。也就是说，如果$F$蕴涵$F'$，则左属性$A$可删除，只要保留剩余部分就可以了。

Canonical Cover (Cont.)

- 对于情况（2），$\alpha \to A\beta'$，等价于$\{\alpha \to \beta', \alpha \to A\}$，如果$\alpha \to A$可以由其余的函数依赖所蕴涵，则说明$\alpha \to A$多余，即$\alpha \to A\beta'$中的$A$多余，只要保留$\alpha \to \beta'$就可以了。换句话说，如果我们把$F$中去掉$\alpha \to A$之后余下的部分叫$F'$, 即$F' = (F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\})$，则如果$F'$可以推出$\alpha \to A$，这说明$\alpha \to A$多余，只要保留$F'$就可以了。也就是说，如果$F'$蕴涵$F$，则右属性$A$可删除。

Canonical Cover (Cont.)

- To compute a canonical cover for *F*:

  repeat

  Use the union rule to replace any dependencies in *F* like $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1\beta_2$

  Find a functional dependency $\alpha \rightarrow \beta$ with an extraneous attribute either in α or in β

  If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$

  until *F* does not change

- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied.

Example of Computing a Canonical Cover

- $R = (A, B, C)$
  $F = \{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now $F' = \{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
  - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies $B \rightarrow C$
  - Set is now $F' = \{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
  - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and $B \rightarrow$ C
- The canonical cover is: $F_C = \{A \rightarrow B, B \rightarrow C\}$

Outline

- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce-Codd Normal Form
- Third Normal Form
- Multivalued Dependencies
- Fourth Normal Form

Goals of Normalization

- Judge whether a particular relation R is in a "good" form (--- no redundant, no insert/delete/update anomalies).
- In the case that a relation R is not in a "good" form, decompose it into a set of relations $\{R_1, R_2, ..., R_n\}$ such that:
  - The decomposition is a lossless-join decomposition (无损连接分解).
  - The decomposition is dependency preservation (依赖保持).
  - Each relation R_i is in a good form --- BCNF or 3NF.

Desirable properties of decomposition

- All attributes of an original schema (R) must appear in the decomposition (R_1, R_2): $R = R_1 \cup R_2$

- Lossless-join decomposition.

  For all possible relations r on schema R

  - $r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$
  - A decomposition of R into R_1 and R_2 is lossless-join if and only if at least one of the following dependencies are held in F^+ :
    - $\{R_1 \cap R_2\} \to R_1$
    - $\{R_1 \cap R_2\} \to R_2$

无损连接分解的条件： 分解后的二个子模式的共同属性必须是R_1或R_2的码（适用于一分为二的分解）。

Desirable properties of decomposition (Cont.)

- Dependency preservation（依赖保持）
  - To check updates (to ensure not violate any FD) efficiently, allow updates validation in sub-relations R_i respectively, without executing the join of them.
  - Restriction of F to R_i is: $F_i \subseteq F^+$, F_i includes only attributes of R_i
  - $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$, where F_i be the set of dependencies in F^+ that include only attributes in R_i.
- No redundancy: The relations R_i preferably should be in either Boyce-Codd Normal Form or Third Normal Form, i.e., BCNF or 3NF.

$R = (A, B, C)$, $F = \{A \rightarrow B, B \rightarrow C\}$,
$R_1 = (A, B)$, $R_2 = (A, C)$

Example of Lossless-join decomposition and Dependency preserving

- Example: $R = (A, B, C)$, $F = \{A \rightarrow B, B \rightarrow C)$, Can be decomposed in two different ways:

- One way: $R_1 = (A, B)$, $R_2 = (B, C)$
  - Lossless-join decomposition:

    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow C$, $\therefore (B)^+ = \{BC\} \supseteq R_2$
  - Dependency preserving: $F_1 = \{A \rightarrow B\}$ for R_1, $F_2 = \{B \rightarrow C\}$ for R_2, $\therefore (F_1 \cup F_2)^+ = F^+$

- 2nd way: $R_1 = (A, B)$, $R_2 = (A, C)$
  - Lossless-join decomposition:

    $R_1 \cap R_2 = \{A\}$ and $(A)^+ = \{AB\} \supseteq R_1$
  - $F_1 = \{A \rightarrow B\}$ for R_1, $F_2 = \{A \rightarrow C\}$ for R_2, $(F_1 \cup F_2)^+ = \{A \rightarrow B, A \rightarrow C\}^+ \neq F^+$, cannot check $B \rightarrow C$ in R_1, R_2 without computing $R_1 \bowtie R_2$

$\boxed{\therefore \text{ Not dependency preserving!}}$

Testing for Dependency Preservation

- To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of R into $R_1, R_2, \ldots, R_n$, we apply the following simplified test:
  - *result* = α
    while (changes to *result*) do
      for each R_i in the decomposition
        $t = ((result \cap R_i)^+) \cap R_i$ α^+
      result = *result* $\cup\ t$
  - If *result* contains all attributes in β,
    then the functional dependency $\alpha \rightarrow \beta$ is preserved.

对于F中的某个$\alpha \rightarrow \beta$, 投影到各个R_i中, 判别是否有某个R_i能保持函数依赖$\alpha \rightarrow \beta$.

- Apply the test on all dependencies in F to check if a decomposition is dependency preserving.

若对F中的每个$\alpha \rightarrow \beta$都能有一个R_i满足函数依赖, 则该分解保持依赖.

∴必然有$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
$F_i = (\alpha \rightarrow \beta)$

Outline

- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce-Codd Normal Form
- Third Normal Form
- Multivalued Dependencies
- Fourth Normal Form

Boyce-Codd Normal Form

- Definition: A relation schema R is in BCNF, with respect to a set F of functional dependencies, if for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

$$\left.\begin{array}{l} \alpha \rightarrow \beta \text{ is trivial (i.e., } \beta \subseteq \alpha) \\ \text{or} \\ \alpha \text{ is a superkey for } R \text{ (i.e., } R \subseteq \alpha^+,\ \alpha \rightarrow R) \end{array}\right\} \text{For all } \alpha \rightarrow \beta \text{ in } F^+$$

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B$
  $\quad B \rightarrow C\}$
  Key = $\{A\}$

Any relation schema with two attributes is in BCNF.

- R is not in BCNF
  - $\because F^+ = \{A \rightarrow B, B \rightarrow C, A \rightarrow C, \ldots\}$, for $B \rightarrow C$, B is not a key)
- Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$
  - R_1 and R_2 in BCNF ($\because A$ is the key for R_1, B is the key for R_2).
  - Lossless-join decomposition ($\because R_1 \cap R_2 = B$, and B is the key for R_2).
  - Dependency preserving ($\because F_1 = \{A \rightarrow B\}$ holds on R_1, $F_2 = \{B \rightarrow C\}$ holds on R_2).

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.
  - Compute α^+ (the attribute closure of α), and
  - Verify that if α^+ includes all attributes of R, i.e., α is a superkey of R.
- Simplified test: To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than checking all dependencies in F^+.
  - If none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.

∵F^+是由Armstrong的3个公理从F推出的, 而任何公理都不会使Functional Dependency (FD)左边变小(拆分), 故如果F中没有违反BCNF的FD (即左边是superkey), 则F^+中也不会.

Testing for BCNF (Cont.)

- However, using only F to test for BCNF may be incorrect when testing a relation R_i in a decomposition of R.
  - E.g., Consider R (A, B, C, D), with $F = \{A \rightarrow B, B \rightarrow C\}$
    - R is in BCNF? Which is a candidate key?
    - Decompose R into R_1 (A, B) and R_2 (A, C, D)
    - Neither of the functional dependencies in F contain only attributes from (A, C, D), Thus, no FD violates BCNF, we might be mislead into thinking R_2 satisfies BCNF.
    - In fact, dependency $A \rightarrow C$ in F^+ shows R_2 is not in BCNF.

可在F下判别R是否违反BCNF, 但必须在F^+下判别R的分解式是否违反BCNF.

BCNF Decomposition Algorithm

result := {*R*};
    done := false;
    compute F^+;
    while (not done) do
        if (there is a schema R_i in *result* that is not in BCNF)
            then begin
                let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds on R_i such that $\alpha \rightarrow R_i$ is not in F^+, and $\alpha \cap \beta = \varnothing$;
                $result := \underline{(result - R_i)} \cup \underbrace{(\alpha, \beta)}_{R_{i1}} \cup \underbrace{(R_i - \beta)}_{R_{i2}}$;
            end
        else *done* := true;

It means there is nontrivial FD $\alpha \rightarrow \beta$ in R_i, and α is not a key.

将R_i分解为二个子模式: $R_{i1} = (\alpha, \beta)$和$R_{i2} = (R_i - \beta)$, α是 R_{i1} R_{i2}的共同属性.

Note: Finally, every sub-schema is in BCNF, and the decomposition is lossless-join.

Example of BCNF Decomposition

- R = (*branch-name*, *branch-city*, *assets*, *customer-name*, *loan-number*, *amount*)

  F = {*branch-name* $\rightarrow$ *branch-city assets*

  loan-number $\rightarrow$ *amount branch-name*}

  Key = {*loan-number*, *customer-name*},

  $\because$(*loan-number*, *customer-name*)$^+$ = R

- Decomposition α β
  - R_1 = (*branch-name*, *branch-city*, *assets*), $R_1 = (\alpha, \beta)$
  - R_2 = (*branch-name*, *customer-name*, *loan-number*, *amount*), $R_2 = R - \beta$
  - R_3 = (*loan-number*, *branch-name*, *amount*)
  - R_4 = (*customer-name*, *loan-number*)

- Final decomposition:

  R_1, R_3, and R_4 are all in BCNF.

BCNF and Dependency Preservation

- It is not always possible to get a BCNF decomposition that is dependency preserving.

J: student, *K*: course, *L*: teacher (一门有多个教师, 一个教师上一门课, 一个学生选多门课, 一门课有多个学生选)

- Example:

  $R = (J, K, L)$ --- *J*学生、*K*课程、*L*教师

  $F = \{JK \rightarrow L, L \rightarrow K\}$

  - Two candidate keys = *JK* and *JL*.
  - *R* is not in BCNF ($\because$ for $L \rightarrow K$, *L* is not a key).
  - Any decomposition of *R* will fail to preserve $JK \rightarrow L$.

    E.g., $R_1 = (L, K)$, $R_2 = (J, L)$, $\subseteq$ BCNF, but not dependency preserving.

- Therefore, we cannot always satisfy all three design goals:
  - Lossless join
  - BCNF
  - Dependency preservation

Outline

- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce-Codd Normal Form
- Third Normal Form
- Multivalued Dependencies
- Fourth Normal Form

Motivation of Third Normal Form

- There are some situations where
  - Decompose into BCNF is not dependency preserving,
  - But efficient checking for FD violation on updates is important.

- Solution: define a weaker normal form, called Third Normal Form (3NF).
  - Allows some redundancy (with resultant problems).
  - But FDs can be checked on individual relations without computing a join --- dependency preserving.
  - There is always a lossless-join, dependency-preserving decomposition into 3NF.

Third Normal Form (3NF)

- Definition: A relation schema R is in third normal form (3NF) if for all $\alpha \to \beta$ in F^+, at least one of the following conditions holds:
  - $\alpha \to \beta$ is trivial (i.e., $\beta \in \alpha$).
  - α is a superkey for R.
  - Each attribute A in $\beta - \alpha$ is contained in a candidate key for R (即$A \in \beta - \alpha$是主属性, 若$\alpha \cap \beta = \varnothing$, 则$A = \beta$是主属性).
  - Note: each attribute may be in a different candidate key.
- If a relation is in BCNF, it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation.

讨论: 国内其他教材关于3NF的定义: 不存在非主属性对码的部分依赖和传递依赖. 该定义实际是说, 当β为非主属性时, α必须是码; 但当β为主属性时, 则α无限制. 国内外这二种定义本质上是一致的.

3NF (Cont.)

- Example:
  - $R = (J, K, L)$
    $F = \{JK \rightarrow L, L \rightarrow K\}$
  - Two candidate keys: JK and JL
  - R is in 3NF
    $JK \rightarrow L$, JK is a superkey.
    $L \rightarrow K$, K is contained in a candidate key.
  - But BCNF decomposition gets (JL) and (LK), and $JK \rightarrow L$ *is* not preserved.
    - Testing for $JK \rightarrow L$ requires a join.
  - There is some redundancy in this schema.

J: student, *K*: course, *L*: teacher (一门有多个教师, 一个教师上一门课, 一个学生选多门课, 一门课有多个学生选)

Redundancy of 3NF

- Example of problems due to redundancy in 3NF
  - $R = (J, K, L)$
    $F = \{JK \rightarrow L, L \rightarrow K\}$

J	L	K
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
null	l_2	k_2

J: student, *K*: course, *L*: teacher (一门有多个教师, 一个教师上一门课, 一个学生选多门课, 一门课有多个学生选)

A schema that is in 3NF but not in BCNF has the problems of repetition of information (e.g., the relationship l_1, k_1), and may need to use null values (e.g., to represent the relationship l_2, k_2, where there is no corresponding value for *J*).

Testing for 3NF

- Optimization: Need to check only FDs in F, need not check all FDs in F^+.
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, to see if α is a superkey.
- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of R.
  - This test is rather more expensive, since it involve finding all candidate keys.
  - Testing for 3NF has been shown to be NP-hard.
  - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time.

3NF Decomposition Algorithm

Let F_c be a canonical cover for F;
i := 0;
for each functional dependency $\alpha \rightarrow \beta$ in F_c do
  {if none of the schemas R_j, $1 \le j \le i$ contains $\alpha\ \beta$
    then begin
      i := i + 1;
      R_i := $(\alpha\ \beta)$
    end}
if none of the schemas R_j, $1 \le j \le i$ contains a candidate key for R then
begin
  i := i + 1;
  R_i := any candidate key for R;
end
return (R_1, R_2, ..., R_i)

将F_c中的每个$\alpha \rightarrow \beta$分解为子模式R_i := (α, β), 从而保证dependency-preserving.

保证至少在一个R_i中存在R的候选码, 从而保证lossless-join.

讨论: 对于多于二个子模式R_i (i > 2)的分解, 判别是否无损连接的方法, 其他教材中是用一张i行n列的表来表示. 如果各子模式中函数依赖的相关性使得R中所有的属性都涉及, 则是无损连接分解. 而根据候选码的含义, 候选码必与所有属性相关. 从而二者本质上一致.

Example

- Relation schema:

  Banker-info-schema = (*branch-name*, *customer-name*,
  banker-name, *office-number*)

- The functional dependencies for this relation schema are:

  F = {*banker-name* $\rightarrow$ *branch-name office-number*,
  customer-name branch-name $\rightarrow$ *banker-name*}

- The key is: {*customer-name*, *branch-name*}
- The *for* loop in the algorithm on the previous page causes us to include the following schemas in our decomposition:

  Banker-office-schema = (*banker-name*, *branch-name*, *office-number*)

  Banker-schema = (*customer-name*, *branch-name*, *banker-name*)

- Since *Banker-schema* contains a candidate key (*customer-name*, *branch-name*) for *Banker-info-schema*, we are done with the decomposition process.

Comparison of BCNF and 3NF

- It is always possible to decompose a relation into relations in 3NF and
  - The decomposition is lossless.
  - The dependencies are preserved.
- It is always possible to decompose a relation into relations in BCNF and
  - The decomposition is lossless.
  - But it may not be possible to preserve dependencies.

Example 1

- Given $F = \{AB \to E, BE \to I, E \to G, GI \to H\}$, using only Armstrong Axiom to prove $AB \to GH$.
- Answer:

  $AB \to E$, $AB \to BE$; $BE \to I$, $BE \to EI$;

  $\therefore AB \to EI$ --- (1)

  $E \to G$, $EI \to GI$; $GI \to H$, $GI \to GH$

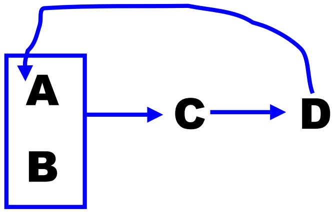
  $\therefore EI \to GH$ --- (2)

  $\therefore AB \to GH$ according to (1) and (2)

Example 2

- For a relation schema $R(A, B, C, D)$ with $F = \{AB \rightarrow C, C \rightarrow D, D \rightarrow A\}$.
  - 1) List all the candidate keys for the relation schema R.
  - 2) Decompose the relation schema R into a collection of BCNF relation schemas.
  - 3) Explain whether the decomposition of 2) is dependency preserving.

- Answer:



1): $(AB)^+ = (ABCD) \supseteq R$, $(BC)^+ = (ABCD) \supseteq R$,

$D \rightarrow A$, $BD \rightarrow AB$; $AB \rightarrow C$; $\therefore (BD)^+ = (ABCD) \supseteq R$,

$\therefore$ AB, BC, BD are candidate keys, and R is not in BCNF.

2): ⇨ $R_1(C, D)$; $R_2(A, B, C)$,

(R_1 is in BCNF, R_2 is not BCNF, $\because C \rightarrow D$, $D \rightarrow A$, $\therefore C \rightarrow A$, C is not key of R_2);

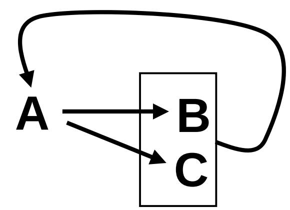
$R_{21}(A, C)$, $R_{22}(B, C)$, R_{21} is in BCNF, R_{22} is in BCNF.

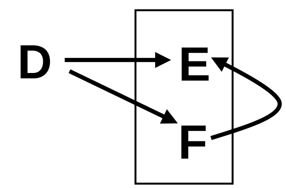
3): $D \rightarrow A$, $AB \rightarrow C$ are not preserved.

Example 3

- $R = (A, B, C, D, E, F)$, $F = \{A \to B, A \to C, BC \to A, D \to EF, F \to E\}$
  - 1) Find all candidate keys.
  - 2) Whether R is in BCNF or 3NF?
  - 3) If it is not in BCNF, decompose R into a set of BCNF relations. Explain that your decomposition is lossless-join.
  - 4) Whether the decomposition of 3) is dependency preserving or not? Why?

- Answer:





1) Candidate keys: AD, BCD

2)$\because F \to E$, F is not a key, E is not in key, $\therefore$ not BCNF, not 3NF.

3) $R_1 = (A, B, C)$, $R_2 = (A, D, E, F)$. (由$A \to BC$), $R_{21} = (D, E, F)$, $R_{22} = (A, D)$, but R_{21} is not BCNF, $\because F \to E$, (AD is key) $R_{211} = (F, E)$, $R_{212} = (D, F)$

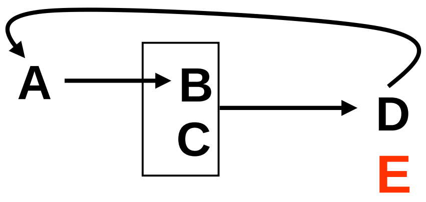
4) $D \to E$ is not preserved.

方法 2: $R_1 = (B, C, A)$, $R_2 = (B, C, D, E, F)$; $R_{21} = (F, E)$, $R_{22} = (B, C, D, F)$, $R_{221} = (D, F)$, $R_{222} = (B, C, D)$. Thus, $D \to E$ is not preserved.

Example 4

- $R = (A, B, C, D, E)$, $F = \{A \rightarrow B, BC \rightarrow D, D \rightarrow A\}$
  - 1) Find all candidate keys.
  - 2) Whether R is in BCNF or 3NF or neither?
  - 3) If it is not in BCNF, decompose R into a set of BCNF relations. Explain that your decomposition is lossless-join.
  - 4) Whether the decomposition of 3) is dependency preserving or not? Why?

- Answer:



1) Candidate keys: *ACE*, *BCE*, *CDE*

2) $\because$Every right attribute in F is in key. $\therefore$ R is 3NF.

3) $R_1 = (AB)$, $R_2 = (ACDE)$, $R_{21} = (AD)$, $R_{22} = (CDE)$

3′) $R_1 = (AB)$, $R_2 = (ACDE)$, $R_{21} = (ACD)$, $R_{22} = (ACE)$, $R_{211} = (AD)$, $R_{212} = (CD)$

3′′) … …

4) $BC \rightarrow D$ is not preserved.

Design Goals

- Goal for a relational database design is:
  - BCNF.
  - Lossless join.
  - Dependency preservation.
- If we cannot achieve this, we accept one of
  - Lack of dependency preservation.
  - Redundancy due to use of 3NF.
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.

  Can specify FDs using assertions, but they are expensive to test.
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

*Testing for FDs Across Relations: Materialized View

- If decomposition is not dependency preserving, we can have an extra materialized view for each dependency $\alpha \rightarrow \beta$ in F_c that is not preserved in the decomposition.
- The materialized view is defined as a projection on $\alpha\,\beta$ of the join of the relations in the decomposition.
- Many newer database systems support materialized views and database system maintains the view when the relations are updated.
  - No extra coding effort for programmer.
- The functional dependency $\alpha \rightarrow \beta$ is expressed by declaring α as a candidate key on the materialized view.
- Checking for candidate key cheaper than checking $\alpha \rightarrow \beta$
- BUT:
  - Space overhead: for storing the materialized view.
  - Time overhead: Need to keep materialized view up to date when relations are updated.
  - Database system may not support key declarations on materialized views.

Outline

- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce-Codd Normal Form
- Third Normal Form
- Multivalued Dependencies
- Fourth Normal Form

Multivalued Dependencies

- There are database schemas in BCNF that do not seem to be sufficiently normalized.
- Consider a database *classes*(*course*, *teacher*, *book*), we denote that (c, t, b) $\in$ *classes* means that t is to teach c, and b is a required textbook for c.
- The database is supposed to list for each course the set of teachers (any one of which can be the course's instructor), and the set of books (all of which are required for the course no matter who teaches it).

  (*Course*: *teacher* = *1*:*n*, *course*:*book* = *1*:*n.*

  teacher and book are multi-value attributes, and teacher and book are independent.)

Multivalued Dependencies (Cont.)

course	*teacher*	*book*
database	Avi	DB Concepts
database	Avi	DB system (Ullman)
database	Hank	DB Concepts
database	Hank	DB system (Ullman)
database	Sudarshan	DB Concepts
database	Sudarshan	DB system (Ullman)
operating systems	Avi	OS Concepts
operating systems	Avi	OS system (Shaw)
operating systems	Jim	OS Concepts
operating systems	Jim	OS system (Shaw)

classes

- There are only trivial functional dependencies and therefore the relation is in BCNF (key = {*course*, *teacher*, *book*}).
- Redundant and insertion anomalies – i.e., if Sara is a new teacher that can teach database, two tuples need to be inserted: (database, Sara, DB Concepts) and (database, Sara, DB system (Ullman)).

Multivalued Dependencies (Cont.)

- Therefore, it is better to decompose *classes* into:

course	***teacher***
database	**Avi**
database	**Hank**
database	**Sudarshan**
operating systems	**Avi**
operating systems	**Jim**

teaches

Key = {*course*, *teacher*}

course	***book***
database	**DB Concepts**
database	**DB system (Ullman)**
operating systems	**OS Concepts**
operating systems	**OS system (Shaw)**

text

Key = {*course*, *book*}

- We shall see that these relations are in Fourth Normal Form (4NF).

Multivalued Dependencies (Cont.)

Employee-name	*project*	*dependent*
smith	p1	tom
smith	p2	anna
smith	p1	anna
smith	p2	tom
julia	p3	tom
julia	p3	mary
julia	p1	tom
julia	p1	mary

Key = {*employee-name*, *project*, *dependent*}

Multivalued Dependencies (MVDs)

- Definition: Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$, the *multivalued dependency*

$$\alpha \rightarrow\rightarrow \beta$$

holds on R, if in any legal relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$

$$t_3[\beta] = t_1[\beta]$$

$$t_4[\beta] = t_2[\beta]$$

$$t_3[R - \alpha - \beta] = t_2[R - \alpha - \beta]$$

$$t_4[R - \alpha - \beta] = t_1[R - \alpha - \beta]$$

Let $R - \alpha - \beta = z$

$t_3[z] = t_2[z]$

$t_4[z] = t_1[z]$

Multivalued Dependencies (MVDs)

- Tabular representation of $\alpha \rightarrow\rightarrow \beta$

	α	β	$R - \alpha - \beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

t1[ ]=t2[ ]=t3[ ]=t4[ ]	t1[ ]=t3[ ]	t1[ ]=t4[ ]
	t2[ ]=t4[ ]	t2[ ]=t3[ ]

- If $\beta \subseteq \alpha$, or $\alpha \cup \beta = R$, then $\alpha \rightarrow\rightarrow \beta$ is trivial.

MVD (Cont.)

- Another definition

	α	β	$R-\alpha-\beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$

t1[ ]= t2[ ]= t3[ ] | **t1[ ]= t3[ ]** | **t2[ ]=t3[ ]**

对任意三个元组都成立

Example

- Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets.

  $$Y, Z, W$$

- We say that $Y \rightarrow\rightarrow Z$ (Y multi-determines Z) if and only if for all possible relations $r(R)$

  $$(y_1, z_1, w_1) \in r \text{ and } (y_1, z_2, w_2) \in r$$

  then

  $$(y_1, z_1, w_2) \in r \text{ and } (y_1, z_2, w_1) \in r$$

- Note that since the behavior of Z and W are identical, it follows that $Y \rightarrow\rightarrow Z$ if $Y \rightarrow\rightarrow W$.

Example (Cont.)

- In our example:

  $$course \rightarrow\rightarrow \text{teacher}$$
  $$course \rightarrow\rightarrow book$$

- The above formal definition is supposed to formalize the notion that given a particular value of *Y* (*course*) it has associated with it a set of values of *Z* (*teacher*) and a set of values of *W* (*book*), and these two sets are in some sense independent of each other.

Theory of MVDs

- From the definition of multivalued dependency, we can derive the following rule:
  - If $\alpha \rightarrow \beta$, then $\alpha \rightarrow\rightarrow \beta$, (if $R - \alpha - \beta = \emptyset$, i.e., $\alpha \cup \beta = R$)
    $\because t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$; $t_1[\beta] = t_3[\beta]$, $t_2[\beta] = t_4[\beta]$
    That is, every functional dependency is also a multivalued dependency.
- The closure D^+ of D is the set of all functional and multivalued dependencies logically implied by D.
  - We can compute D^+ from D, using the formal definitions of functional dependencies and multivalued dependencies.
  - We can manage with such reasoning for very simple multivalued dependencies, which seem to be most common in practice.
  - For complex dependencies, it is better to reason about sets of dependencies using a system of inference rules (see Appendix C).

Outline

- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce-Codd Normal Form
- Third Normal Form
- Multivalued Dependencies
- Fourth Normal Form

Fourth Normal Form

- A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
  - $\alpha \rightarrow\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
  - α is a superkey for schema R

- If a relation is in 4NF, it is in BCNF.

Requirement for decomposition --- Restriction of Multivalued Dependencies

- Assume R is decomposed into R_1, R_2, …, R_n, each R_i is required to conform to 4NF.
- The restriction of D to R_i is the set D_i consisting of
  - All functional dependencies in D^+ that include only attributes of R_i.
  - All multivalued dependencies of the form

$$\alpha \to\to (\beta \cap R_i)$$

where $\alpha \subseteq R_i$ and $\alpha \to\to \beta$ is in D^+.

4NF Decomposition Algorithm

result := $\{R\}$;
done := false;
compute D^+;
Let D_i denote the restriction of D^+ to R_i
    while (not *done*)
        if (there is a schema R_i in *result* that is not in 4NF) then
            begin
                let $\alpha \rightarrow\rightarrow \beta$ be a nontrivial multivalued dependency that holds on R_i such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \phi$;
                    $result := (result - R_i) \cup \underbrace{(\alpha, \beta)}_{R_{i1}} \cup \underbrace{(R_i - \beta)}_{R_{i2}}$;
            end
        else *done*:= true;

Note: each R_i is in 4NF, and decomposition is lossless-join.

*Further Normal Forms

- Join dependencies generalize multivalued dependencies
  - Lead to project-join normal form (PJNF) (also called fifth normal form).
- A class of even more general constraints, leads to a normal form called domain-key normal form.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used.

Example

- $R = (A, B, C, G, H, I)$
  $D = \{A \rightarrow\rightarrow B$
  $B \rightarrow\rightarrow HI$
  $CG \rightarrow\rightarrow H\}$
- R is not in 4NF since $A \rightarrow\rightarrow B$ and A is not a superkey for R
- Decomposition
  a) $R_1 = (A, B)$ (R_1 is in 4NF)
  b) $R_2 = (A, C, G, H, I)$ (R_2 is not in 4NF)
  c) $R_{21} = (C, G, H)$ (R_{21} is in 4NF)
  d) $R_{22} = (A, C, G, I)$ (R_{22} is in 4NF)

*Overall Database Design Process

- We have assumed schema *R* is given
  - *R* could have been generated when converting E-R diagram to a set of tables.
  - *R* could have been a single relation containing *all* attributes that are of interest (called universal relation).
  - Normalization breaks *R* into smaller relations.
  - *R* could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

*ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design there can be FDs from non-key attributes of an entity to other attributes of the entity.
- E.g., *employee* entity with attributes *department-number* and *department-address*, and an FD *department-number* $\rightarrow$ *department-address*.
  - Good design would have made department an entity.
- FDs from non-key attributes of a relationship set possible, but rare - most relationships are binary.

*Universal Relation Approach

- Dangling tuples (无归属)– Tuples that "disappear" in computing a join.
  - Let $r_1(R_1)$, $r_2(R_2)$, …., $r_n(R_n)$ be a set of relations.
  - A tuple r of the relation r_i is a dangling tuple if r is not in the relation: $\Pi_{R_i}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n)$
- The relation $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$ is called a *universal* relation since it involves all the attributes in the "universe" defined by $R_1 \cup R_2 \cup \ldots \cup R_n$.
- If dangling tuples are allowed in the database, instead of decomposing a universal relation, we may prefer to synthesize a collection of normal form schemas from a given set of attributes.

*Universal Relation Approach (Cont.)

- Dangling tuples may occur in practical database applications.
- They represent incomplete information.
- E.g., may want to break up information about loans into:

    (branch-name, loan-number)

    (loan-number, amount)

    (loan-number, customer-name)
- Universal relation would require null values, and have dangling tuples.

*Universal Relation Approach (Cont.)

- A particular decomposition defines a restricted form of incomplete information that is acceptable in our database.
  - Above decomposition requires at least one *of customer-name*, *branch-name* or *amount* in order to enter a loan number without using null values.
  - Rules out storing of *customer-name*, *amount* without an appropriate *loan-number* (since it is a key, it can't be null either!).
- Universal relation requires unique attribute names unique role assumption.
  - E.g., *customer-name*, *branch-name*
- Reuse of attribute names is natural in SQL since relation names can be prefixed to disambiguate names.

Denormalization for Performance

- Sometimes we may want to use non-normalized schema for performance.
- E.g., displaying *customer-name* along with *account-number* and *balance* requires join of *account* with *depositor*.
- Alternative 1: Use denormalized relation containing attributes of *account* as well as *depositor* with all above attributes.
  - Faster lookup.
  - Extra space and extra execution time for updates.
  - Extra coding work for programmer and possibility of error in extra code.
- Alternative 2: use a materialized view defined as account $\bowtie$ depositor
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors.

(*customer-name*, *account-number*, *balance*)

Other Design Issues

- Some aspects of database design are not caught by normalization.
- Examples of bad database design, to be avoided: To store information of yearly-earnings of each company.
  - Design 1: *earnings-2000*, *earnings-2001*, *earnings-2002*, etc., all on the schema: *earnings-20XX* (*company-id, earnings*)
    - Above are in BCNF, but make querying across years difficult and needs new table each year.
  - Design 2: *company-year* (*company-id*, *earnings-2000*, *earnings-2001*, *earnings-2002*, *earnings-2003*, **…**)
    - Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
    - Is an example of a crosstab, where values for one attribute become column names.
    - Used in spreadsheets, and in data analysis tools.
  - Design 3: *earnings* (*company-id*, *year*, *amount*)

Q & A

Your questions and suggestions are expected for me.

Thanks a lot!

Questions?

Exercises: 7.1, 7.13, 7.21, 7.22, and 7.29 (see Pages 353-357)