Chapter 7: Normalization

Database System Concepts, 7th Ed.

Outline

- Features of Good Relational Design

- Functional Dependencies

- Decomposition Using Functional Dependencies

- Normal Forms

- Functional Dependency Theory

- Algorithms for Decomposition using Functional Dependencies

- Decomposition Using Multivalued Dependencies

- More Normal Form

- Atomic Domains and First Normal Form

- Database-Design Process

- Modeling Temporal Data ***

Overview of Normalization

Features of Good Relational Designs

- Suppose we combine *instructor* and *department* into *in_dep,* which represents the natural join on the relations *instructor* and *department*

ID	name	salary	dept_name	building	budget
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

- There is repetition of information

- Need to use null values (if we add a new department with no instructors)

Decomposition

- The only way to avoid the repetition-of-information problem in the *in_dep* schema is to decompose it into two schemas – *instructor* and *department* schemas.

- Not all decompositions are good. Suppose we decompose

    employee(ID, name, street, city, salary)

  into
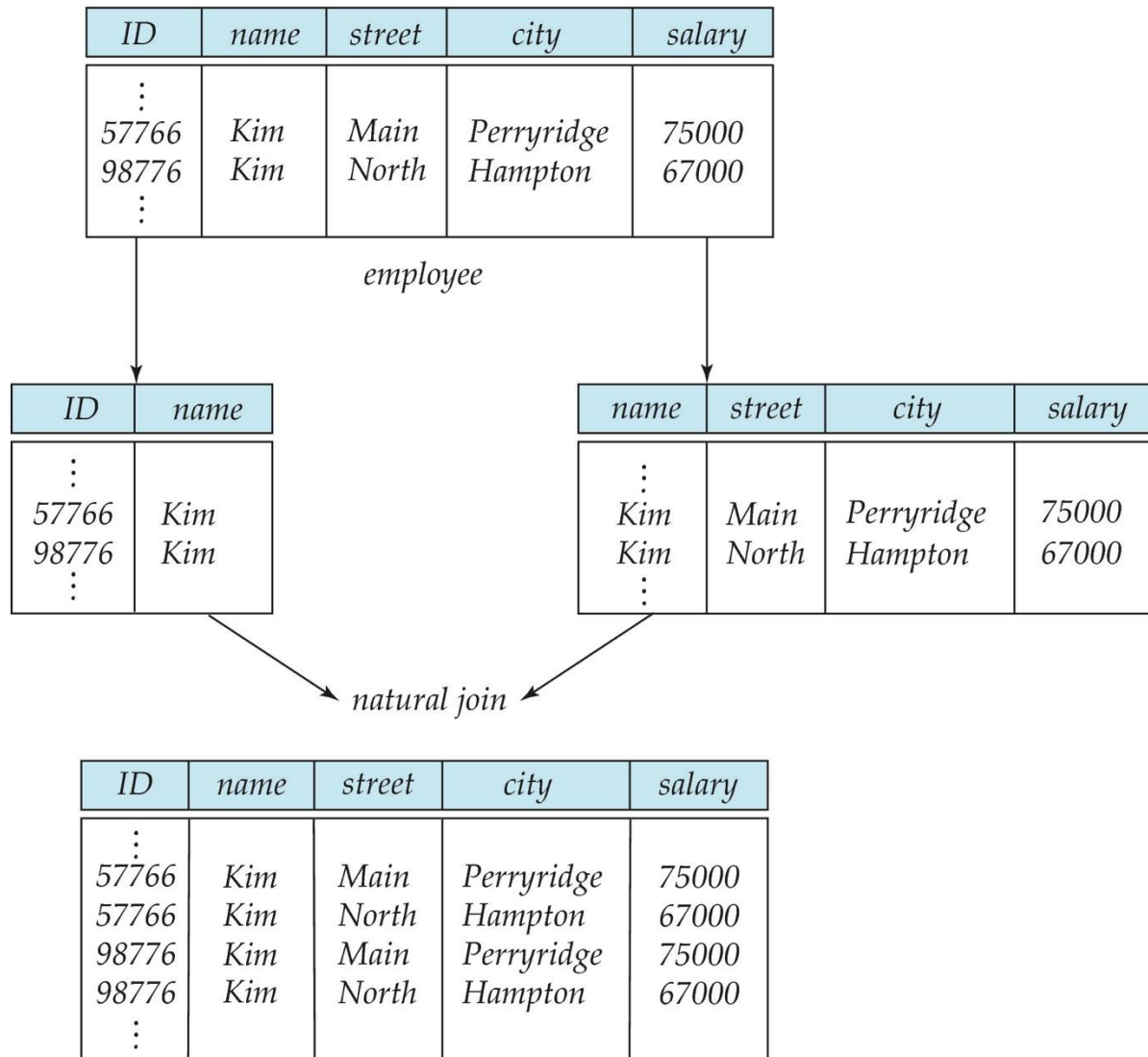
    employee1 (*ID*, *name*)

    employee2 (*name*, *street*, *city*, *salary*)

  The problem arises when we have two employees with the same name

- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

A Lossy Decomposition

Lossless Decomposition

- Let R be a relation schema and let R_1 and R_2 form a decomposition of R . That is R = R_1 U R_2

- We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing R with the two relation schemas R_1 U R_2

- Formally,

$$\Pi_{R_1}(r) \Join \Pi_{R_2}(r) = r$$

- And, conversely a decomposition is lossy if

$$r \subset \Pi_{R_1}(r) \Join \Pi_{R_2}(r)$$

Example of Lossless Decomposition

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

$$R_1 = (A, B) \qquad R_2 = (B, C)$$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

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

A	B	C
α	1	A
β	2	B

Normalization Theory

- Decide whether a particular relation R is in "good" form.

- In the case that a relation R is not in "good" form, decompose it into set of relations $\{R_1, R_2, ..., R_n\}$ such that

  - Each relation is in good form

  - The decomposition is a lossless decomposition

- Our theory is based on:

  - Functional dependencies

  - Multivalued dependencies

Functional Dependencies

- There are usually a variety of constraints (rules) on the data in the real world.

- For example, some of the constraints that are expected to hold in a university database are:

  - Students and instructors are uniquely identified by their ID.

  - Each student and instructor has only one name.

  - Each instructor and student is (primarily) associated with only one department.

  - Each department has only one value for its budget, and only one associated building.

Functional Dependencies (Cont.)

- An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation;

- A legal instance of a database is one where all the relation instances are legal instances

- Constraints on the set of legal relations.

- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.

- A functional dependency is a generalization of the notion of a *key.*

Functional Dependencies Definition

- Let R be a relation schema

$$\alpha \subseteq R \ and \ \beta \subseteq R$$

- 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. That is,

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

- Example: Consider $r(A,B)$ with the following instance of r.

A	B
1	4
1	5
3	7

The definition is similar to Function

- On this instance, $B \rightarrow A$ hold; $A \rightarrow B$ does **NOT** hold,

"Implies"

- " ⟹" (Implies, 蕴含/逻辑蕴含) is the connective in propositional calculus which has the meaning:

  - "if A is true, then B is also true."

- $A \Longrightarrow B$ is an abbreviation for $\neg A \lor B$

A	B	$A \Rightarrow B$
T	T	T
T	F	F
F	T	T
F	F	T

Closure of a Set of Functional Dependencies

- Given a set *F* set of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.

  - If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$

  - etc.

- The set of **all** functional dependencies logically implied by *F* is the **closure** of *F*.

- We denote the *closure* of *F* by ***F⁺***.

Keys and Functional Dependencies

- K is a superkey for relation schema R if and only if $K \rightarrow R$

- K is a candidate key for R if and only if

    - $K \rightarrow R$, and

    - for no $\alpha \subset K$, $\alpha \rightarrow R$

- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

    in_dep (*ID, name, salary, dept_name, building, budget*)*.*

    We expect these functional dependencies to hold:

    $$dept\_name \rightarrow building$$

    $$ID \rightarrow building$$

    but would not expect the following to hold:

    $$dept\_name \rightarrow salary$$

Use of Functional Dependencies

- We use functional dependencies to:
  - To test relations to see if they are legal under a given set of functional dependencies.
    - If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* **satisfies** *F*.
  - To specify constraints on the set of legal relations
    - We say that *F* **holds on** *R* if all legal relations on *R* satisfy the set of functional dependencies *F*.
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
  - For example, a specific instance of *instructor* may, by chance, satisfy *name* $\rightarrow$ *ID.*

Trivial Functional Dependencies

- *A* functional dependency is **trivial** if it is satisfied by all instances of a relation

- Example*:*

  - *ID, name* $\rightarrow$ *ID*

  - *name* $\rightarrow$ *name*

- In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Lossless Decomposition

- We can use functional dependencies to show when certain decomposition are lossless.

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

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

- A decomposition of R into R_1 and R_2 is lossless decomposition if at least one of the following dependencies is in F^+:

  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$

- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

  - We shall see other types of constraints (e.g. multivalued dependencies) later that can also ensure that a decomposition is lossless

Example

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

- $R_1 = (A, B), \quad R_2 = (B, C)$

  - Lossless decomposition:

    $$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$

- $R_1 = (A, B), \quad R_2 = (A, C)$

  - Lossless decomposition:

    $$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

- *Note:*

  - $B \rightarrow BC$

    is a shorthand notation for

  - $B \rightarrow \{B, C\}$

Dependency Preservation

- Testing functional dependency constraints each time the database is updated can be costly

- It is useful to design the database in a way that constraints can be tested efficiently.

- If testing a functional dependency can be done by considering just one relation, then the cost of testing this constraint is low

- When decomposing a relation it is possible that it is no longer possible to do the testing without having to perform a Cartesian Product.

- A decomposition that makes it computationally hard to enforce functional dependency is said to be NOT **dependency preserving**.

Dependency Preservation Example

- Consider a schema:

  dept_advisor(s_ID, i_ID, department_name)

- With function dependencies:

  $i\_ID \rightarrow dept\_name$

  $s\_ID, dept\_name \rightarrow i\_ID$

- In the above design we are forced to repeat the department name once for each time an instructor participates in a *dept_advisor* relationship.

- To fix this, we need to decompose *dept_advisor*

- Any decomposition will not include all the attributes in

  $s\_ID, dept\_name \rightarrow i\_ID$

  1. *(i_ID, dept_name), (s_ID, i_ID)*?
  2. *(i_ID, dept_name), (s_ID, dept_name)*?

- Thus, the composition NOT be dependency preserving

Normal Forms

- 什么是数据依赖

1. 完整性约束的表现形式

– 属性内取值范围：例如学生成绩必须在**0-100**之间

– 属性间的相互关连（主要体现于值的相等与否）： 这就是数据依赖，它是数据库模式设计的关键，是语义的体现

2. 数据依赖的类型

– 函数依赖（**Functional Dependency**，简记为**FD**）

– 多值依赖（**Multivalued Dependency**，简记为**MVD**）

– 其他（*传递依赖、部分依赖*）

- 数据依赖对关系模式的影响

例：描述学校的数据库：

学生的学号（**Sno**）、所在系（**Sdept**）

系主任姓名（**Mname**）、课程名（**Cname**）

成绩（**Grade**）

单一的关系模式： **Student <U、F>**

U ＝ ｛ Sno, Sdept, Mname, Cname, Grade ｝

- ## 数据依赖对关系模式的影响

学校数据库的语义：
1. 一个系有若干学生，一个学生只属于一个系；
2. 一个系只有一名主任；
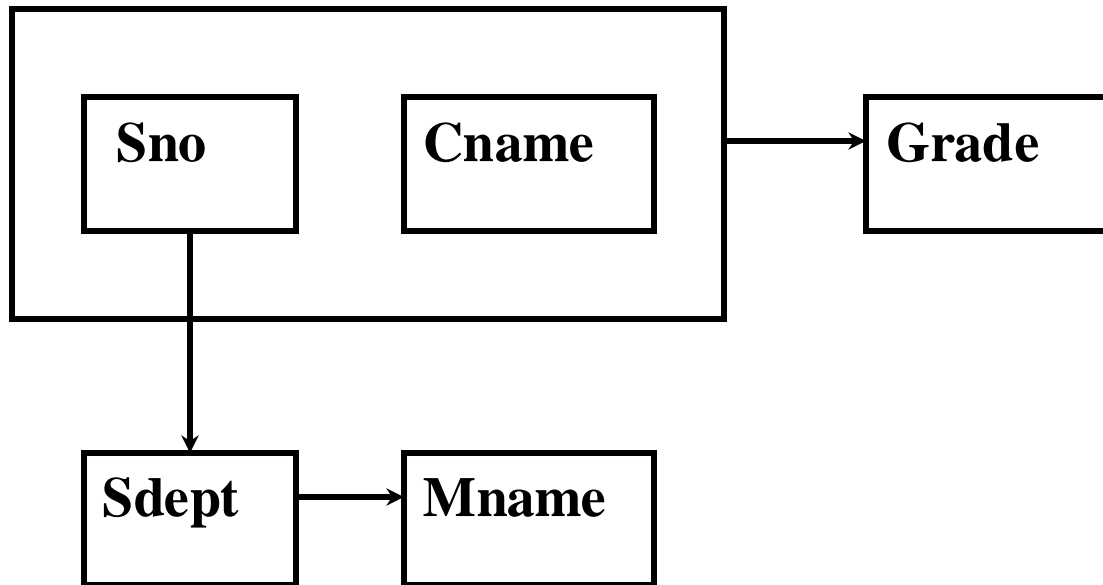3. 一个学生可以选修多门课程，每门课程有若干学生选修；
4. 每个学生所学的每门课程都有一个成绩。

U = ｛ Sno, Sdept, Mname, Cname, Grade ｝

属性组**U**上的一组函数依赖**F**：
F = *｛ Sno → Sdept, Sdept → Mname,*
(Sno, Cname) → Grade ｝

- 数据依赖对关系模式的影响

数据依赖

- **关系模式Student<U, F>中存在的问题**

学 号 Sno	所 在 系 Sdept	系主任 Mname	课程名 Cname	成绩 Grade
95001	IS	李勇	高数	80
95002	IS	李勇	高数	73
95003	MA	王敏	高数	91
95004	IS	李勇	外语	67

- **关系模式Student<U, F>中存在的问题**

1. 数据冗余太大

　　浪费大量的存储空间

　　例：每一个系主任的姓名重复出现

2. 更新异常（**Update Anomalies**）

　　数据冗余，更新数据时，维护数据完整性代价大。

　　　　例：某系更换系主任后，系统必须修改与该系学生有关的每一个元组

3. 插入异常（**Insertion Anomalies**）

　　该插的数据插不进去

　　例，如果一个系刚成立，尚无学生，我们就无法把这个系及其系主任的信息存入数据库。

4. 删除异常（**Deletion Anomalies**）

　　不该删除的数据不得不删

　　　　例，如果某个系的学生全部毕业了，我们在删除该系学生信息的同时，把这个系及其系主任的信息也丢掉了。

- **数据依赖对关系模式的影响**

结论：

- **Student**关系模式不是一个好的模式。
- "好"的模式：

不会发生插入异常、删除异常、更新异常，

数据冗余应尽可能少。

原因：

由存在于模式中的某些数据依赖引起的

解决方法：

通过分解关系模式来消除其中不合适的数据依赖。

规范化理论正是用来改造关系模式，通过分解关系模式来消除其中不合适的数据依赖，以解决插入异常、删除异常、更新异常和数据冗余问题。

范式

- 范式是符合某一种级别的关系模式的集合。
- 关系数据库中的关系必须满足一定的要求。满足不同程度要求的为不同范式。
- 范式的种类：

第一范式(1NF)***
第二范式(2NF)***
第三范式(3NF)
BC范式(BCNF)
第四范式(4NF)
第五范式(5NF)***

- 各种范式之间存在联系：

$$1NF \supset 2NF \supset 3NF \supset BCNF \supset 4NF \supset 5NF$$

- 某一关系模式R为第n范式，可简记为R∈nNF。

Boyce-Codd Normal Form

- 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

$$\alpha \to \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \to \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- α is a superkey for R

Boyce-Codd Normal Form (Cont.)

- Example schema that is *not* in BCNF:

   *in_dep* (*ID, name, salary, dept_name, building, budget*)

    because :

    - *dept_name*$\rightarrow$ *building, budget*
        - holds on *in_dep*
        - but
    - *dept_name* is not a superkey

- When decompose *in_dept* into *instructor* and *department*
    - *instructor* is in BCNF
    - *department* is in BCNF

Decomposing a Schema into BCNF

- Let R be a schema R that is not in BCNF. Let $\alpha \rightarrow \beta$ be the FD that causes a violation of BCNF.

- We decompose R into:
  - $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$

- In our example of *in_dep*,
  - α = *dept_name*
  - β = *building, budget*

  and *in_dep* is replaced by
  - $(\alpha \cup \beta)$ = (*dept_name, building, budget*)
  - $(R - (\beta - \alpha))$ = (*ID, name, dept_name, salary*)
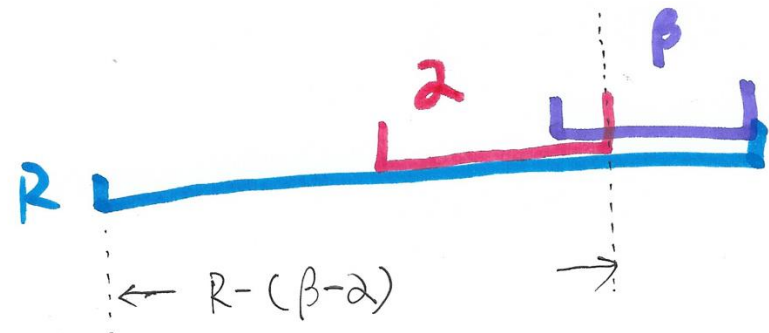
Example

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

- $R_1 = (A, B), \quad R_2 = (B, C)$

  - Lossless-join decomposition:

  $$R_1 \cap R_2 = \{B\} \quad \text{and } B \rightarrow BC$$

  - Dependency preserving

- $R_1 = (A, B), \quad R_2 = (A, C)$

  - Lossless-join decomposition:

  $$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

BCNF and Dependency Preservation

- It is not always possible to achieve both BCNF and dependency preservation

- Consider a schema:

   dept_advisor(s_ID, i_ID, department_name)

- With function dependencies:

   i_ID $\rightarrow$ dept_name

   s_ID, dept_name $\rightarrow$ i_ID

- *dept_advisor* is not in BCNF

  - *i_ID* is not a superkey.

- Any decomposition of *dept_advisor* will not include all the attributes in

   s_ID, dept_name $\rightarrow$ i_ID

- Thus, the composition is NOT be dependency preserving

Third Normal Form

- A relation schema R is in **third normal form (3NF)** if for all:

$$\alpha \rightarrow \beta \text{ in } F^+$$

at least one of the following holds:

  - $\alpha \rightarrow \beta$ 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.

    (**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 (will see why later).

3NF Example

- Consider a schema:

    dept_advisor(s_ID, i_ID, dept_name)

- With function dependencies:

    $i\_ID \rightarrow dept\_name$

    $s\_ID, dept\_name \rightarrow i\_ID$

- Two candidate keys = {*s_ID, dept_name*}, {*s_ID, i_ID* }
- We have seen before that *dept_advisor* is not in BCNF
- *R,* however, is in 3NF

    - *s_ID, dept_name* is a superkey
    - $i\_ID \rightarrow dept\_name$ *and* *i_ID* is NOT a superkey, but:
        - { *dept_name*} – {*i_ID* } = {*dept_name* } and
        - *dept_name* is contained in a candidate key

Redundancy in 3NF

- Consider the schema R below, which is in 3NF

  - $R = (J, K, L)$
  - $F = \{JK \rightarrow L, L \rightarrow K\}$
  - And an instance table:

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

- What is wrong with the table?

  - Repetition of information
  - Need to use null values (e.g., to represent the relationship l_2, k_2 where there is no corresponding value for J)

Comparison of BCNF and 3NF

- Advantages to 3NF over BCNF. It is always possible to obtain a 3NF design without sacrificing losslessness or dependency preservation.

- Disadvantages to 3NF.

  - We may have to use null values to represent some of the possible meaningful relationships among data items.

  - There is the problem of repetition of information.

Goals of Normalization

- Let R be a relation scheme with a set F of functional dependencies.

- Decide whether a relation scheme R is in "good" form.

- In the case that a relation scheme R is not in "good" form, need to decompose it into a set of relation scheme $\{R_1, R_2, ..., R_n\}$ such that:

    I.   Each relation scheme is in good form

    II. The decomposition is a lossless decomposition

    III. Preferably, the decomposition should be dependency preserving.

How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized

- Consider a relation

  inst_info (ID, child_name, phone)

  - where an instructor may have more than one phone and can have multiple children

  - Instance of *inst_info*

ID	child_name	phone
99999	David	512-555-1234
99999	David	512-555-4321
99999	William	512-555-1234
99999	William	512-555-4321

How good is BCNF? (Cont.)

- There are no non-trivial functional dependencies and therefore the relation is in BCNF

- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples

        (99999, David,   981-992-3443)
        (99999, William, 981-992-3443)

Higher Normal Forms

- It is better to decompose *inst_info* into:
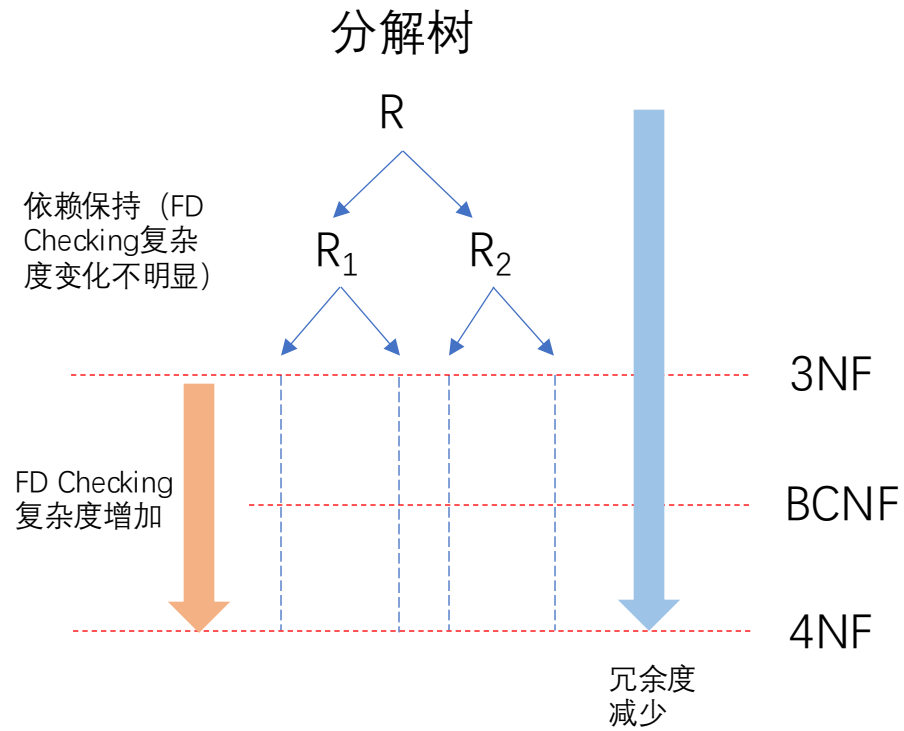
  - *inst_child*:

ID	child_name
99999	David
99999	William

  - *inst_phone:*

ID	phone
99999	512-555-1234
99999	512-555-4321

- This suggests the need for higher normal forms, such as Fourth Normal Form (4NF), which we shall see later

Comparisons

分解树

R

R₁　R₂

依赖保持（FD Checking复杂度变化不明显）

FD Checking 复杂度增加

3NF

BCNF

4NF

冗余度减少

Functional-Dependency Theory

Functional-Dependency Theory Roadmap

- We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies.

- We then develop algorithms to generate lossless decompositions into BCNF and 3NF

- We then develop algorithms to test if a decomposition is dependency-preserving

Closure of a Set of Functional Dependencies

- Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F.

  - If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$

  - etc.

- The set of **all** functional dependencies logically implied by F is the **closure** (闭包) of F.

- We denote the *closure* of F by F^+.

Closure of a Set of Functional Dependencies

- We can compute F$^+$, the closure of F, by repeatedly applying **Armstrong's Axioms:**

  - **Reflexive rule:** if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$

  - **Augmentation rule:** if $\alpha \rightarrow \beta$, then $\gamma\,\alpha \rightarrow \gamma\,\beta$

  - **Transitivity rule:** if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$

- These rules are

  - **Sound** -- generate only functional dependencies that actually hold, and

  - **Complete** -- generate all functional dependencies that hold.

Example of F^+

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $\quad A \rightarrow C$
  $\quad CG \rightarrow H$
  $\quad CG \rightarrow I$
  $\quad B \rightarrow H\}$

- Some members of F^+

  - $A \rightarrow H$

    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$

  - $AG \rightarrow I$

    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$
      and then transitivity with $CG \rightarrow I$

  - $CG \rightarrow HI$

    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,

      and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$,

      and then transitivity

Closure of Functional Dependencies (Cont.)

- Additional rules:

    - **Union rule**: If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.

    - **Decomposition rule**: If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.

    - **Pseudotransitivity rule**: If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

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

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**: We shall see an alternative procedure for this task later

Closure of Attribute Sets

- Given a set of attributes α, define the ***closure*** of α **under** F (denoted by α^+) as the set of attributes that are functionally determined by α under F

- Algorithm to compute α^+, the closure of α under F

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

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B$
    $A \rightarrow C$
    $CG \rightarrow H$
    $CG \rightarrow I$
    $B \rightarrow H\}$
- $(AG)^+$
  1. $result = AG$
  2. $result = ABCG$ $(A \rightarrow C$ and $A \rightarrow B)$
  3. $result = ABCGH$ $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. $result = ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq AGBCH)$
- Is *AG* a candidate key?
  1. Is AG a super key?
     1. Does $AG \rightarrow R?$ == Is R $\supseteq (AG)^+$
  2. Is any subset of AG a superkey?
     1. Does $A \rightarrow R?$ == Is R $\supseteq (A)^+$
     2. Does $G \rightarrow R?$ == Is R $\supseteq (G)^+$
     3. In general: check for each subset of size *n-1*

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
    - To test if α is a superkey, we compute α^+, and check if α^+ contains all attributes of R.

- Testing functional dependencies
    - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$.
    - That is, we compute α^+ by using attribute closure, and then check if it contains β.
    - Is a simple and cheap test, and very useful

- Computing closure of F
    - For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover

- Suppose that we have a set of functional dependencies F on a relation schema. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies; that is, all the functional dependencies in F are satisfied in the new database state.

- If an update violates any functional dependencies in the set F, the system must roll back the update.

- We can reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set.

- This simplified set is termed the **canonical cover** (正则覆盖)

- To define canonical cover we must first define **extraneous attributes** (无关属性)**.**

  - An attribute of a functional dependency in F is **extraneous** if we can remove it without changing F^+

Extraneous Attributes

- Removing an attribute from the left side of a functional dependency could make it a stronger constraint.

  - For example, if we have AB $\rightarrow$ C and remove B, we get the possibly stronger result A $\rightarrow$ C. It may be stronger because A $\rightarrow$ C logically implies AB $\rightarrow$ C, but AB $\rightarrow$ C does not, on its own, logically imply A $\rightarrow$ C

- But, depending on what our set F of functional dependencies happens to be, we may be able to remove B from AB $\rightarrow$ C safely.

  - For example, suppose that

  - F = {AB $\rightarrow$ C, A $\rightarrow$ D, D $\rightarrow$ C}

  - Then we can show that F logically implies A $\rightarrow$ C, making B extraneous in AB $\rightarrow$ C.

Extraneous Attributes (Cont.)

- Removing an attribute from the right side of a functional dependency could make it a weaker constraint.

  - For example, if we have $AB \rightarrow CD$ and remove C, we get the possibly weaker result $AB \rightarrow D$. It may be weaker because using just $AB \rightarrow D$, we can no longer infer $AB \rightarrow C$.

- But, depending on what our set F of functional dependencies happens to be, we may be able to remove C from $AB \rightarrow CD$ safely.

  - For example, suppose that

    $$F = \{ AB \rightarrow CD, A \rightarrow C \} .$$

  - Then we can show that even after replacing $AB \rightarrow CD$ by $AB \rightarrow D$, we can still infer $AB \rightarrow C$ and thus $AB \rightarrow CD$.

  - So C is extraneous in $AB \rightarrow CD$ here.

Extraneous Attributes

- An attribute of a functional dependency in F is **extraneous** if we can remove it without changing F^+

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.

  - **Remove from the left side**: Attribute A is **extraneous** in α if

    - $A \in \alpha$ and

    - F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.

  - **Remove from the right side**: Attribute A is **extraneous** in β if

    - $A \in \beta$ and

    - The set of functional dependencies

      $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.

- *Note:* implication in the opposite direction is trivial in each of the cases above, since a "stronger" functional dependency always implies a weaker one

  $$F \Leftrightarrow (F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$$

  $$(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\} \Leftrightarrow F$$

Testing if an Attribute is Extraneous

- Let R be a relation schema and let F be a set of functional dependencies that hold on R. Consider an attribute in the functional dependency $\alpha \rightarrow \beta$.

- To test if attribute $A \in \alpha$ is extraneous in α
  - Let $\gamma = \alpha - \{A\}$. Check if $\gamma \rightarrow \beta$ can be inferred from F.
    - Compute γ^+ using the dependencies in F
    - If γ^+ includes all attributes in β then , A is extraneous in α

- To test if attribute $A \in \beta$ is extraneous in β
  - Consider the set:
    $$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\},$$
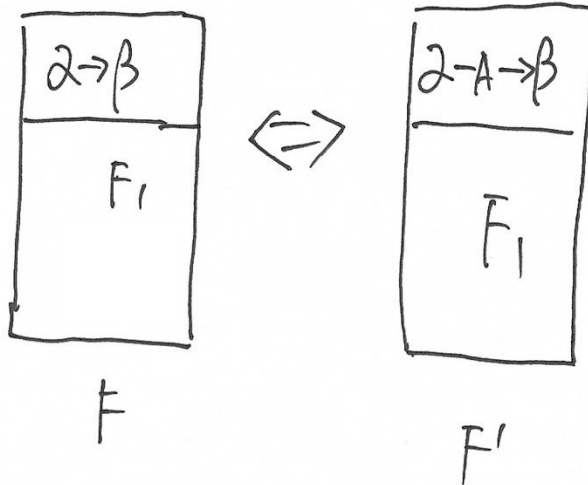 - check that α^+ contains A; if it does, A is extraneous in β

① 左边删除 A

$$\alpha - A \to \beta \implies \alpha \to \beta$$

例如

$$\left.\begin{array}{c} \alpha \to \beta \\ (\alpha - A) \to A \end{array}\right\} \implies \alpha - A \to \beta$$

② 右边删除 A

$$\alpha \to \beta \iff \begin{cases} \alpha \to A \\ \alpha \to \beta - A \end{cases}$$
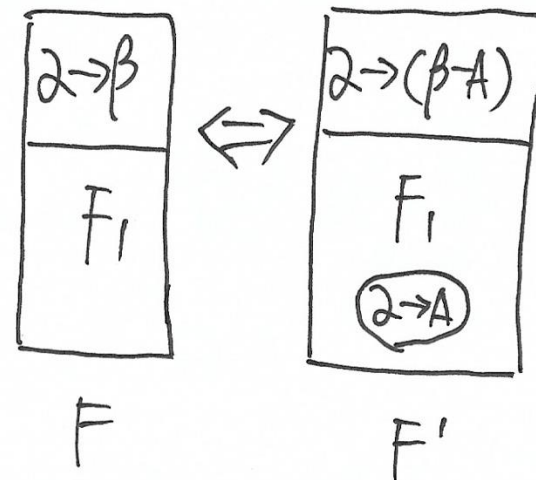
Examples of Extraneous Attributes

- Let $F = \{AB \rightarrow CD, A \rightarrow E, E \rightarrow C\}$

- To check if C is extraneous in $AB \rightarrow CD$, we:

  - Compute the attribute closure of AB under $F = \{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$

  - The closure is $ABCDE$, which includes CD

  - This implies that C is extraneous

Canonical Cover

A **canonical cover** for F is a set of dependencies F_c such that

- F logically implies all dependencies in F_c , and

- F_c logically implies all dependencies in F, and

- No functional dependency in F_c contains an extraneous attribute, and

- Each left side of functional dependency in F_c is unique. That is, there are no two dependencies in F_c

  - $\alpha_1 \to \beta_1$ and $\alpha_2 \to \beta_2$ such that

  - $\alpha_1 = \alpha_2$

Canonical Cover

- To compute a canonical cover for F:

repeat

Use the union rule to replace any dependencies in F of the form

$$\alpha_1 \rightarrow \beta_1 \text{ and } \alpha_1 \rightarrow \beta_2 \text{ with } \alpha_1 \rightarrow \beta_1 \beta_2$$

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

/* Note: test for extraneous attributes done using F_c, not F*/

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

until (F_c not change)

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

Example: Computing a Canonical Cover

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

- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$

  - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$

- A is extraneous in $AB \rightarrow C$

  - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies

    - Yes: in fact, $B \rightarrow C$ is already present!

  - Set is now $\{A \rightarrow BC, B \rightarrow C\}$

- C is extraneous in $A \rightarrow BC$

  - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies

    - Yes: using transitivity on $A \rightarrow B$ and $B \rightarrow C$.

      - Can use attribute closure of A in more complex cases

- The canonical cover is: $\quad A \rightarrow B$
  $\quad\quad\quad\quad\quad\quad\quad\quad\quad B \rightarrow C$

Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i.

  - A decomposition is **dependency preserving**, if

    $$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$$

- Using the above definition, testing for dependency preservation take exponential time.

- Not that if a decomposition is NOT dependency preserving then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Dependency Preservation (Cont.)

- Let F be the set of dependencies on schema R and let $R_1, R_2, .., R_n$ be a decomposition of R.

- The **restriction (限定)** of F to R_i is the set F_i of all functional dependencies in F^+ that include **only** attributes of R_i.

  - Since all functional dependencies in a restriction involve attributes of only one relation schema, it is possible to test such a dependency for satisfaction by checking only one relation.

  - Note that the definition of restriction uses all dependencies in F^+, not just those in F.

- The set of restrictions $F_1, F_2, .. , F_n$ is the set of functional dependencies that can be checked efficiently.

Testing for Dependency Preservation

- To check if a dependency $\alpha \to \beta$ is preserved in a decomposition of R into $R_1, R_2, \ldots, R_n$, we apply the following test (with attribute closure done with respect to F)

  - *result* $= \alpha$
    repeat
      for each R_i in the decomposition
          $t = (result \cap R_i)^+ \cap R_i$
          $result = result \cup t$

      until (*result* does not change)

       Using F, not F_i

  - If *result* contains all attributes in β, then the functional dependency $\alpha \to \beta$ is preserved.

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

- This procedure takes polynomial time, instead of the exponential time required to compute F^+ and $(F_1 \cup F_2 \cup \ldots \cup F_n)^+$

Example

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

- R is not in BCNF

- Decomposition $R_1 = (A, B),\ R_2 = (B, C)$

  - R_1 and R_2 in BCNF

  - Lossless-join decomposition

  - Dependency preserving

Algorithm for Decomposition Using Functional Dependencies

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
  1. compute α^+ (the attribute closure of α), and
  2. verify that it includes all attributes of R, that is, it 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.
- However, **simplified test** using only F is incorrect when testing a relation in a decomposition of R
  - Consider $R = (A, B, C, D, E)$, with $F = \{ A \rightarrow B, BC \rightarrow D\}$
    - Decompose R into $R_1 = (A,B)$ and $R_2 = (A,C,D, E)$
    - Neither of the dependencies in F contain only attributes from (A,C,D,E) so we might be mislead into thinking R_2 satisfies BCNF.
    - In fact, dependency $AC \rightarrow D$ in F^+ shows R_2 is not in BCNF.

Testing Decomposition for BCNF

To check if a relation R_i in a decomposition of R is in BCNF

- Either test R_i for BCNF with respect to the **restriction** of F^+ to R_i (that is, all FDs in F^+ that contain only attributes from R_i)

- Or use the original set of dependencies F that hold on R, but with the following test:

  - for every set of attributes $\alpha \subseteq R_i$, check that α^+ (the attribute closure of α) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i.

  - If the condition is violated by some $\alpha \rightarrow \beta$ in F^+, the dependency
    $$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$
    can be shown to hold on R_i, and R_i violates BCNF.

  - We use above dependency to decompose R_i

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 := (*result* – R_i) $\cup$ (R_i – β) $\cup$ (α, β);
        end
    else *done* := **true;**

Call the modules in p73&p74

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

Example of BCNF Decomposition

- *class* (*course_id*, *title*, *dept_name*, *credits*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)

- Functional dependencies:

  - *course_id*→ *title*, *dept_name*, *credits*

  - *building*, *room_number*→*capacity*

  - *course_id*, *sec_id*, *semester*, *year*→*building*, *room_number*, *time_slot_id*

- A candidate key {*course_id*, *sec_id*, *semester*, *year*}.

- BCNF Decomposition:

  - *course_id*→ *title*, *dept_name*, *credits* holds

    - but *course_id* is not a superkey.

  - We replace *class* by:

    - *course*(*course_id*, *title*, *dept_name*, *credits*)

    - *class-1* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)

BCNF Decomposition (Cont.)

- *course* is in BCNF

  - How do we know this?

- *building*, *room_number*→*capacity* holds on *class-1*

  - but {*building*, *room_number*} is not a superkey for *class-1*.

  - We replace *class-1* by:

    - *classroom* (*building*, *room_number*, *capacity*)

    - *section* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)

- *classroom* and *section* are in BCNF.

Third Normal Form

- There are some situations where

  - BCNF is not dependency preserving, and

  - 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; we will see examples later)

  - But functional dependencies can be checked on individual relations without computing a join.

  - There is always a lossless-join, dependency-preserving decomposition into 3NF.

3NF Example -- Relation *dept_advisor*

- *dept_advisor* (*s_ID, i_ID, dept_name*)
  $F = \{s\_ID, dept\_name \rightarrow i\_ID, \ i\_ID \rightarrow dept\_name\}$

- Two candidate keys: *s_ID, dept_name,* and *i_ID, s_ID*

- *R* is in 3NF

  - *s_ID, dept_name* $\rightarrow$ *i_ID*

    - *(s_ID, dept_name)* is a superkey

  - *i_ID* $\rightarrow$ *dept_name*

    - *dept_name* is contained in a candidate key

- *If we make a BCNF decomposition: (i_ID, dept_name) , (s_ID, i_ID)*

  - *This is not dependancy-preserving.*

Testing for 3NF

- Need to check only FDs in F, need not check all FDs in F^+.

- Use attribute closure to check for each dependency $\alpha \to \beta$, 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 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

1、一般不需使用3NF的判定算法(p80)，而是直接使用3NF的定义来判定（因为一般的例子里属性不多，候选码容易找。候选码不好找的模式比较少见）。
2、一旦不符合定义，直接使用3NF的分解算法(p81)。该分解算法使用一次即可，使用过程中不需再调用判定算法（与BCNF的分解算法不同）。
3、判定算法:top-down
4、分解算法:一步到位

3NF Decomposition Algorithm

Let F_c be a canonical cover for F;

$i := 0$; /*Decomposition set is Φ*/

/*step 1: ensure the dependency preservation*/

for each functional dependency $\alpha \rightarrow \beta$ in F_c **do**
 if none of the schemas R_j, $1 \leq j \leq i$ contains $\alpha\ \beta$
  then begin
    $i := i + 1$;
    $R_i := \alpha\ \beta$ **end**

/*step 2: ensure there is a candidate key for R, and the decomp is lossless*/

if none of the schemas R_j, $1 \leq j \leq i$ contains a candidate key for R
 then begin
   $i := i + 1$;
   $R_i :=$ any candidate key for R **end**

/*step 3: Optionally, remove redundant relations */

repeat
 if any schema R_j is contained in another schema R_k
  then /* delete R_j */
   $R_j = R_i$;
   $i=i-1$;

return $(R_1, R_2, ..., R_i)$

No need to call the 3NF testing algorithm

3NF Decomposition Algorithm (Cont.)

Above algorithm ensures

- Each relation schema R_i is in 3NF

  - Proof is at end of this presentation ([click here](#))

- Decomposition is dependency preserving

  - Obvious in Step 1

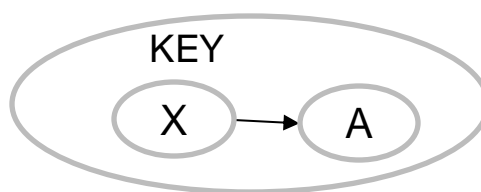- Decomposition is lossless-join

  - Exercise 7.16

In which case $X \to A$ violates 3NF?

- A relation schema R is in **third normal form** (**3NF**) if for all:
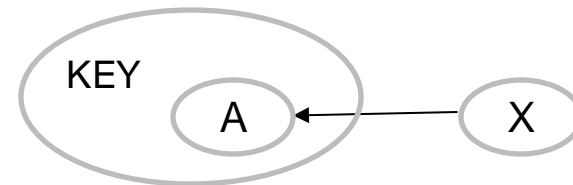
  $\alpha \to \beta$ in F^+
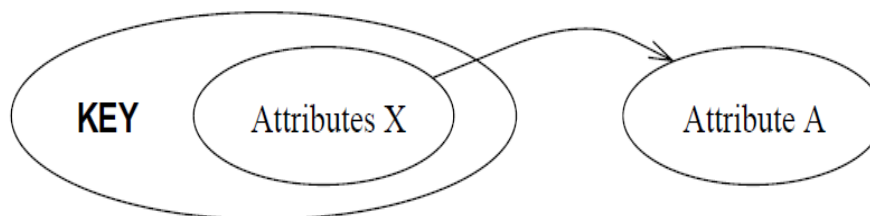 at least one of the following holds:

  - $\alpha \to \beta$ is trivial
  - α is a superkey for R
  - Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.
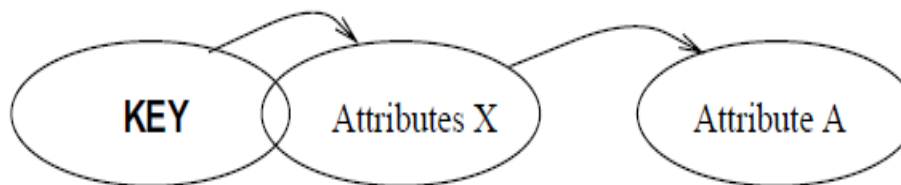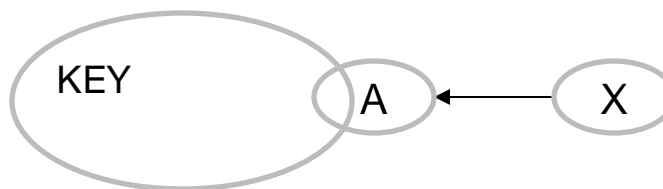

Case 1

Case 2

Case 3

Case 4

Case 5

Case 1-5: N N Y Y Y

3NF Decomposition: An Example

- Relation schema:

  cust_banker_branch = (*customer_id, employee_id, branch_name, type*)

- The functional dependencies for this relation schema are:

  - *customer_id, employee_id* $\rightarrow$ *branch_name, type*

  - *employee_id* $\rightarrow$ *branch_name*

  - *customer_id, branch_name* $\rightarrow$ *employee_id*

- We first compute a canonical cover

  - *branch_name* is extraneous in the r.h.s. of the 1st dependency

  - No other attribute is extraneous, so we get F_C =

    customer_id, employee_id $\rightarrow$ *type*
    employee_id $\rightarrow$ *branch_name*
    customer_id, branch_name $\rightarrow$ *employee_id*

3NF Decompsition Example (Cont.)

- The **for** loop generates following 3NF schema:

  (*customer_id, employee_id, type*)

  (<u>*employee_id*</u>, *branch_name*)

  (*customer_id, branch_name, employee_id)*

  - Observe that (*customer_id, employee_id, type*) contains a candidate key of the original schema, so no further relation schema needs be added

- At end of for loop, detect and delete schemas, such as (<u>*employee_id, branch_name*</u>), which are subsets of other schemas

  - result will not depend on the order in which FDs are considered

- The resultant simplified 3NF schema is:

  (*customer_id, employee_id, type*)

  (*customer_id, branch_name, employee_id)*

Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
  - The decomposition is lossless
  - The dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
  - The decomposition is lossless
  - It may not be possible to preserve dependencies.

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, (and currently not supported by any of the widely used databases!)

- 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.

Multivalued Dependencies

Multivalued Dependencies (MVDs)

- Suppose we record names of children, and phone numbers for instructors:

  - *inst_child*(*ID*, *child_name*)

  - *inst_phone*(*ID*, *phone_number*)

- If we were to combine these schemas to get

  - *inst_info*(*ID*, *child_name*, *phone_number*)

  - Example data:
    (99999, David, 512-555-1234)
    (99999, David, 512-555-4321)
    (99999, William, 512-555-1234)
    (99999, William, 512-555-4321)

- This relation is in BCNF

  - Why?

Multivalued Dependencies

- Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency**

$$\alpha \twoheadrightarrow \beta$$

holds on R if in any legal relation $r(R)$, for all pairs for 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:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
$$t_3[\beta] \quad\quad = \quad t_1[\beta]$$
$$t_3[R - \beta] = \quad t_2[R - \beta]$$
$$t_4[\beta] \quad\quad = \quad t_2[\beta]$$
$$t_4[R - \beta] = \quad t_1[R - \beta]$$

MVD -- Tabular representation

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

	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	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

MVD (Cont.)

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

$$Y, Z, W$$

- We say that $Y \twoheadrightarrow Z$ (Y **multidetermines** 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

  $Y \twoheadrightarrow Z$ if $Y \twoheadrightarrow W$

Example

- In our example:

$$ID \twoheadrightarrow child\_name$$
$$ID \twoheadrightarrow phone\_number$$

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

- Note:

  - If $Y \rightarrow Z$ then $Y \twoheadrightarrow Z$

    - Indeed we have (in above notation) $Z_1 = Z_2$
      The claim follows.

  - $Y \rightarrow Z \not\Rightarrow Y \rightarrow W$

  - $Y \twoheadrightarrow Z \not\Rightarrow Y \rightarrow Z$

$$
\begin{array}{c c c}
Y & Z & W \\
\hline
y_1 & z_1 & w_1 \\
 & z_2 & w_2 \\
 & \vdots & \vdots \\
 & z_m & w_n \\
\end{array}
$$

$$
y_1
$$

$$
\left.\vphantom{\begin{array}{c}a\\b\\c\\d\\e\\f\\g\\h\end{array}}\right\} m \times n
$$

Use of Multivalued Dependencies

- We use multivalued dependencies in two ways:

    1. To test relations to **determine** whether they are legal under a given set of functional and multivalued dependencies

    2. To specify **constraints** on the set of legal relations. We shall concern ourselves *only* with relations that satisfy a given set of functional and multivalued dependencies.

- If a relation *r* fails to satisfy a given multivalued dependency, we can construct a relations *r′* that does satisfy the multivalued dependency by adding tuples to *r.*

Theory of MVDs

- From the definition of multivalued dependency, we can derive the following rule:

    - If $\alpha \rightarrow \beta$, then $\alpha \rightarrow\rightarrow \beta$

    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 (Chapter 28: https://www.db-book.com/db7/online-chapters-dir/28.pdf).

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

Restriction of Multivalued Dependencies

- 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

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

    where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ 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 $\alpha \cap \beta = \phi$;
       $result := (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$;
     end
   else done:= true;

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

Example

- $R = (A, B, C, G, H, I)$

  $F = \{\; A \twoheadrightarrow B$

  $\qquad B \twoheadrightarrow HI$

  $\qquad CG \twoheadrightarrow H \}$

- R is not in 4NF since $A \twoheadrightarrow B$ and <u>A is not a superkey for R</u> *(why?)*

- Decomposition

  a) $R_1 = (A, B)$ (R_1 is in 4NF) *(why?)*

  b) $R_2 = (A, C, G, H, I)$ (R_2 is not in 4NF, decompose into R_3 and R_4)

  c) $R_3 = (C, G, H)$ (R_3 is in 4NF)

  d) $R_4 = (A, C, G, I)$ (R_4 is not in 4NF, decompose into R_5 and R_6)

  - $A \twoheadrightarrow B$ and $B \twoheadrightarrow HI$ ➔ $A \twoheadrightarrow HI$, (MVD transitivity), and

  - and hence $A \twoheadrightarrow I$ *(MVD restriction to R_4)*

  e) $R_5 = (A, I)$ (R_5 is in 4NF)

  f) $R_6 = (A, C, G)$ (R_6 is in 4NF)

Additional issues

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

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 functional dependencies from non-key attributes of an entity to other attributes of the entity

  - Example: an *employee* entity with

    - attributes
      department_name and *building*,

    - functional dependency
      department_name $\rightarrow$ *building*

    - Good design would have made department an entity

- Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary

Denormalization for Performance

- May want to use non-normalized schema for performance

- For example, displaying *prereqs* along with *course_id,* and *title* requires join of *course* with *prereq*

- Alternative 1: Use denormalized relation containing attributes of *course* as well as *prereq* 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 a *course* ⋈ *prereq*

  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

Other Design Issues

- Some aspects of database design are not caught by normalization

- Examples of bad database design, to be avoided:

  Instead of *earnings* (*company_id, year, amount*), use

  - *earnings_2004, earnings_2005, earnings_2006*, etc., all on the schema (*company_id, earnings*).

    - Above are in BCNF, but make querying across years difficult and needs new table each year

  - *company_year* (*company_id, earnings_2004, earnings_2005, earnings_2006*)

    - 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

Modeling Temporal Data

- **Temporal data** have an association time interval during which the data are *valid.*

- A **snapshot** is the value of the data at a particular point in time

- Several proposals to extend ER model by adding valid time to

  - attributes, e.g., address of an instructor at different points in time

  - entities, e.g., time duration when a student entity exists

  - relationships, e.g., time during which an instructor was associated with a student as an advisor.

- But no accepted standard

- Adding a temporal component results in functional dependencies like

    $ID \rightarrow street, city$

not holding, because the address varies over time

- A **temporal functional dependency** $X \rightarrow Y$ holds on schema R if the functional dependency $X \rightarrow Y$ holds on all snapshots for all legal instances $r(R)$.

Modeling Temporal Data (Cont.)

- In practice, database designers may add start and end time attributes to relations

  - E.g., *course*(*course_id, course_title*) is replaced by

    course(*course_id, course_title, start, end*)

  - Constraint: no two tuples can have overlapping valid times

    - Hard to enforce efficiently

- Foreign key references may be to current version of data, or to data at a point in time

  - E.g., student transcript should refer to course information at the time the course was taken

End of Chapter 7

Assignments

- 7.2, 7.3, 7.9
  - Submission deadline: Oct 27, 2024
- 7.30, 7.40
  - Submission deadline: Nov 3, 2024

Redundancy Ranking

- Redundancy: 4NF<BCNF(3.5NF)<3NF<1NF

- 以下关于数据库设计中范式的叙述, 不正确的是(B)。
  A. 范式级别越高, 数据冗余程度越小
  B. 随着范式级别的提高, 在需求变化时数据的稳定性越强
  C. 范式级别越高, 存储 同样的数据就需要分解成更多张表
  D. 范式级别提高, 数据库性能(速度)将下降

Proof of Correctness of 3NF Decomposition Algorithm

Correctness of 3NF Decomposition Algorithm

- 3NF decomposition algorithm is dependency preserving (since there is a relation for every FD in F_c)

- Decomposition is lossless

  - A candidate key (C) is in one of the relations R_i in decomposition

  - Closure of candidate key under F_c must contain all attributes in R.

  - Follow the steps of attribute closure algorithm to show there is only one tuple in the join result for each tuple in R_i

Correctness of 3NF Decomposition Algorithm (Cont.)

- Claim: if a relation R_i is in the decomposition generated by the above algorithm, then R_i satisfies 3NF.

- Proof:

  - Let R_i be generated from the dependency $\alpha \rightarrow \beta$

  - Let $\gamma \rightarrow B$ be any non-trivial functional dependency on R_i. (We need only consider FDs whose right-hand side is a single attribute.)

  - Now, B can be in either β or α but not in both. Consider each case separately.

Correctness of 3NF Decomposition (Cont.)

- Case 1: If B in β:

  - If γ is a superkey, the 2nd condition of 3NF is satisfied

  - Otherwise α must contain some attribute not in γ

  - Since $\gamma \rightarrow B$ is in F^+ it must be derivable from F_c, by using attribute closure on γ.

  - Attribute closure not have used $\alpha \rightarrow \beta$. If it had been used, α must be contained in the attribute closure of γ, which is not possible, since we assumed γ is not a superkey.

  - Now, using $\alpha \rightarrow (\beta - \{B\})$ and $\gamma \rightarrow B$, we can derive $\alpha \rightarrow B$

    (since $\gamma \subseteq \alpha \beta$, and $B \notin \gamma$ since $\gamma \rightarrow B$ is non-trivial)

  - Then, B is extraneous in the right-hand side of $\alpha \rightarrow \beta$; which is not possible since $\alpha \rightarrow \beta$ is in F_c.

  - Thus, if B is in β then γ must be a superkey, and the second condition of 3NF must be satisfied.

Correctness of 3NF Decomposition (Cont.)

- Case 2: *B* is in α.

  - Since α is a candidate key, the third alternative in the definition of 3NF is trivially satisfied.

  - In fact, we cannot show that γ is a superkey.

  - This shows exactly why the third alternative is present in the definition of 3NF.

Q.E.D.

First Normal Form

- Domain is **atomic** if its elements are considered to be indivisible units

  - Examples of non-atomic domains:

    - Set of names, composite attributes

    - Identification numbers like CS101 that can be broken up into parts

- A relational schema R is in **first normal form** if the domains of all attributes of R are atomic

- Non-atomic values complicate storage and encourage redundant (repeated) storage of data

  - Example: Set of accounts stored with each customer, and set of owners stored with each account

  - We assume all relations are in first normal form (and revisit this in Chapter 22: Object Based Databases)

First Normal Form (Cont.)

- Atomicity is actually a property of how the elements of the domain are used.

  - Example: 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.