Chapter 8: Relational Database Design

Revision by Gun-Woo Kim
Dept. of Computer Science and Engineering
Hanyang University

Database System Concepts, 6th Ed.

Contents

8.1 Features of Good Relational Designs – Larger Schemas?

Suppose we combine *instructor* and *department* into *inst_dept*

(No connection to relationship set inst_dept)

Result is possible repetition of information

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

Smaller Schemas?

Suppose we had started with *inst_dept.* How would we know to split up (**decompose**) it into *instructor* and *department*?

Write a rule "if there were a schema (*dept_name, building, budget*), then *dept_name* would be a candidate key"

Denote as a **functional dependency**: (함수적 종속)

dept_name → *building, budget*

결정자라고함 종속자

결정자를 통해서 종속자를 알수있다는 뜻

잘 쪼개진것이 잘 설계된 DB

In *inst_dept*, because *dept_name* is not a candidate key, the building and budget of a department may have to be repeated.

This indicates the need to decompose *inst_dept*

Not all decompositions are good. Suppose we decompose *employee(ID, name, street, city, salary)* into

employee1 (*ID*, *name*)

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

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

(손실 분해)

함수적 종속: 주어지는 것인가? 찾아야 하는 것인가?

A Lossy Decomposition

손실 분해

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

employee

ID	name
⋮	
57766	Kim
98776	Kim
⋮	

name	street	city	salary
⋮			
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000
⋮			

natural join

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

- 오히려 새로운 투플이 추가됨!
- 더 많은 정보를 가지게 되었지만, 누가 누군인지 모르게 되는 결과를 초래함
- 즉, Perryridge, Main에 사는 kim의 ID는 57766인가 98776인가?

→ 오히려 정보손실

Example of Lossless-Join Decomposition

Lossless join decomposition

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

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

항상 무손실이어야 성립함

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

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

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

같은 키값이 없으면 아예 카테이션 곱으로 작동함
(위는 손실분해)

A	B	C
α	1	A
β	2	B

8.2 Atomic Domains and First Normal Form

더이상 분해되지 않는 경우

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

CS101: computer science의 1학년 과목의 1번째 과목이라는 의미이므로 분해가능

→ (CS101에서 CS와 101을 분리하려는 시도를 하지 않으면 atomic)

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

제1 정규형

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 → 결국 composite 속성이 됨

We assume all relations are in first normal form

이때 CS101을 듣는 학생이 EE로 전학가면 나눈 코스의 접두어는 CS에서 EE로 바뀌기 때문엥 일반적으로 나누지 않을때가 존재함

Atomic Domains and 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.

→ 즉, DB 차원이 아니라 application차원에서 roll number를 분해하여 소속학과를 알아내야 한다. 만약 그렇게 되면 학생이 소속학과를 변경하게 되면 application 프로그램을 다시 작성해 한다. (바람직하지 않다)

→ 그러면, course 스키마에서 course_id(e.g. CS-101, BIO-301 등)는 atomic한가?

→ DB application 프로그램에서 course_id를 나누려는 시도를 하지 않거나, 이를 학과의 축약형으로 해석하지 않는 한 atomic하다고 볼 수 있다.

Goal — Devise a Theory for the Following

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 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
- multivalued dependencies

} 어떤 스키마를 분해할지 말지를 평가하는 방법론을 제시!

8.3 Decomposition Using Functional Dependencies

Functional dependency

- 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.*

후보키나 주키에 일반화된 경우

Functional Dependencies

Let R be a relation schema

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

The **functional dependency**

R을 보존함

$$\alpha \rightarrow \beta$$

α : 결정자 (determinant), β : 종속자 (dependent)

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,

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

t1, t2에 대해서 속성 α의 값이 서로 같다면, 속성 β의 값도 서로 같다.

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

t1 과 t2는 임의의 row

A	B
1	4
1	5
3	7

$B \rightarrow A$

$t_1[\alpha] \neq t_2[\alpha] \vee t_1[\beta] = t_2[\beta]$

(항상 True) ∨ (true or false) = T

On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

$$(t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]) \equiv (t_1[\alpha] \neq t_2[\alpha] \vee t_1[\beta] = t_2[\beta])$$

Key and Functional Dependencies

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

Functional dependencies allow us to express constraints that cannot be expressed using superkeys.

Consider the schema:

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

dn은 슈퍼키는 아님

We expect these functional dependencies to hold:

$$dept\_name \rightarrow building$$

and $ID \rightarrow building$

ID -> inst_dept 이면 super key인
ID가지고 모든 속성을 구분질 수 있다는 말

but would not expect the following to hold:

$$dept\_name \rightarrow salary$$

만약 {ID,dept_name} 이면 참

즉, dept_name이 inst_dept의 superkey는 아님 → 함수적 종속은 superkey를 표현할 수 없는 제한조건까지 표현할 수 있다.

Use of Functional Dependencies

We use functional dependencies 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.*

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.*

동명이인이 없는 경우 우연히 *name* $\rightarrow$ *ID 만족될 수도 있음.*

Use of Functional Dependencies (Cont.)

A functional dependency is **trivial** if it is satisfied by all instances of a relation
(자명하다)

Example*:*

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

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

Closure of a Set of Functional Dependencies

(폐포) 닫혀있다.

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

For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$

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

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

F^+ is a superset of *F*.

subset의 반대

["닫혀있다"의 개념]

constraint: "A⊙B=C", A, B, C는 모두 정수이다"
"정수 A, B, C에 대해, A⊙B=C 이다."

→ 이 constraint는 +, -, x 에 닫혀있다. / 에는 닫혀있지 않다.

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

$$\alpha \rightarrow \beta,$$

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

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

Example schema *not* in BCNF:

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

because *dept_name* $\rightarrow$ *building, budget* holds on *instr_dept,* but *dept_name* is not a superkey

dept_name $\rightarrow$ *building*

trivial 한가?
Superkey 인가?

Decomposing a Schema into BCNF

Suppose we have a schema *R* and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.

We decompose *R* into:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

In our example,

α = *dept_name*

β = *building, budget*

inst_dept is replaced by

- $(\alpha \cup \beta)$ = (*dept_name, building, budget*)
- $(R - (\beta - \alpha))$ = (*ID, name, salary, dept_name*)

BCNF and Dependency Preservation

Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation

If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving.*

Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form.*

뭔 말인가?
원래 스키마 R을 R1과 R2로 분해한 후에도 R상에 정의된 FD가 R1 및 R2에 대해서 그대로 유지가 된다면 dependency preservation(종속성 보존)이라 한다.

이때, FD에 존재하는 모든 함수종속들이 R1과 R2에 대해서 만족할 필요가 없고, FD중에서 R1에 존재하는 속성들만으로 구성된 함수 종속들에 대해서만 테스트하면 된다. 마찬가지로 R2에 존재하는 속성들만으로 구성된 함수 종속들만 테스트하면 된다.

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).

- $\alpha \rightarrow \beta$ 라는 함수의 종속성을 보전하기 위한 조건
- 이 조건의 의미는 뒷 부분에서 다시 설명 (강의노트 44~45 페이지 참고)

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

inst_info

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

존재하는 FD 찾아보기? ID → child_name or phone? 존재하지 않음

ID, child_name → phone? 존재하지 않음

결국, 존재하는 모든 FD는 (ID, child_name, phone) → ID or child_name or phone 형태임.
So, inst_info에 존재하는 함수적 종속은 모두 trivial 하다. (BCNF임)

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)

How good is BCNF? (Cont.)

Therefore, it is better to decompose *inst_info* into:

inst_child

ID	*child_name*
99999	David
99999	David
99999	William
99999	Willian

inst_phone

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

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, decompose it into a set of relation scheme $\{R_1, R_2, ..., R_n\}$ such that

- each relation scheme is in good form
- the decomposition is a lossless-join decomposition
- Preferably, the decomposition should be dependency preserving.

8.4 Functional Dependency Theory

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* of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.

For e.g.: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$

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

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

Closure of a Set of Functional Dependencies (Cont.)

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

- if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)** 재귀법칙
- if $\alpha \rightarrow \beta$, then $\gamma\ \alpha \rightarrow \gamma\ \beta$ **(augmentation)** 증가법칙
- if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)** 이행법칙

These rules are

- **sound** (generate only functional dependencies that actually hold), and
- **complete** (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* $B \rightarrow H$

$AG \rightarrow I$

- 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 a Set of Functional Dependencies (Cont.)

Additional rules:

- If $\alpha \rightarrow \beta$ holds *a*nd $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds **(union)** 결합
- If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds **(decomposition)** 분해
- If $\alpha \rightarrow \beta$ holds *a*nd $\gamma\ \beta \rightarrow \delta$ holds, then $\alpha\ \gamma \rightarrow \delta$ holds **(pseudotransitivity)** 가이행

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

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*

(α^+ :는 결국 α 에 의해 결정되는 속성 집합을 의미 즉, $\alpha \rightarrow \alpha^+$)

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 $(AG)^+ \supseteq R$
2. Is any subset of AG a superkey?
   1. Does $A \rightarrow R$? == Is $(A)^+ \supseteq R$
   2. Does $G \rightarrow R$? == Is $(G)^+ \supseteq R$

(A)+
1. Result = A
2. Result = ABC
3. Result = ABCH

A → R ? No

(G)+
1. Result = G

G → R ? No

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 (규준커버)

Sets of functional dependencies may have redundant dependencies that can be inferred from the others

For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \quad B \rightarrow C, A \rightarrow C\}$

Parts of a functional dependency may be redundant

- E.g.: on RHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow CD\}$ can be simplified to $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$
- E.g.: on LHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad AC \rightarrow D\}$ can be simplified to $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$

Intuitively, a canonical cover of F is a "minimal" set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

A→CD

A→C, A→D (분해법칙)

A→C (A→B, B→C 로 얻을 수 있음)
A→D

A→D 만 있으면 AC→D 표현가능?

AC→CD (증가 법칙-양쪽에 C를 더해줌)

AC→C, AC→D (분해법칙)

Lossless Decomposition

For the case of $R = (R_1, R_2)$, we require that 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 at least one of the following dependencies is in F^+:

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

} 결국 $R_1 \cap R_2$ 가 R_1 혹은 R_2 의 superkey를 구성한다.

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

employee1(ID, name)
employee2(name, street, city, salary)

} name은 분해된 두 릴레이션에서 어느쪽에서도 superkey가 되지 못함

inst_dept(ID, name, salary, dept_name, building, budget)

instructor(ID, name, dept_name, salary)
department(dept_name, building, budget)

} dept_name은 department의 superkey를 형성함

↓

loseless decomposition

Example

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

Can be decomposed in two different ways

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

Lossless-join decomposition:

$R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$ (B는 R_2의 superkey)

Dependency preserving (본래 A→B, B→C는 분해 이후에도 보존됨)

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

Lossless-join decomposition:

$R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$ (A는 R_1의 superkey)

Not dependency preserving (B→C는 분해된 이후 보존되지 않음)
(cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

결국, B→C라는 FD가 분해 이후에도 지켜지고 있는지를 확인하기 위해선 $R_1 \bowtie R_2$ 한 후에 check해 봐야 한다.

단순한 경우는 종속보존 여부를 쉽게 살펴볼 수 있지만, FD가 많고 복잡하면 종속보존을 Check하기 매우 어렵다.

Dependency Preservation

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

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

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

- If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

- R_i 에 있는 속성들로만 구성된 FD를 F_i라고 하자
- $F' = F_1 \cup F_2 \cup \ldots \cup F_n$ 하자
- 일반적으로 $F' \neq F$ 일 수 있지만, 결국 $F'^+ = F^+$이다
- 다시 말해, F의 모든 종속은 F′를 논리적으로 내포하고 있다.

R 상에 정의된 FD의 집합 F : 총 10개

R을 R1과 R2로 분해했을 때,
F중에서 R1에 속한 속성만으로 구성된 FD의 집합 F_1 : 5개
F중에서 R1에 속한 속성만으로 구성된 FD의 집합 F_2 : 2개

→ 비록 $F \neq F_1 \cup F_2$ 이지만 만약 $F^+ = (F_1 \cup F_2)^+$ 이라면, 이 분해는 종속보존분해 !

Example

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

R is not in BCNF

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

R_1 and R_2 in BCNF

(why? → A와 B가 각각 R_1과 R_2의 superkey 이므로…)

Lossless-join decomposition

(why? → $R_1 \cap R_2$ = B and B→BC 이므로…)

Dependency preserving

(why? → F_1: A→B, F_2:B→C, $(F_1 \cup F_2)^+ = F^+$ 이므로…)

그럼 Decomposition $R_1 = (A, B)$, $R_2 = (A, C)$ 어떻게 될까?

8.5 Algorithms for Decomposition

BCNF decomposition

- The definition of BCNF can be used directly to test if a relation is in BCNF
- However, computation of ***F+*** can be a tedious task
- First, describe the simplified tests for verifying if a relation is in BCNF
- If a relation is not in BCNF, it can be decomposed to create relations that are in BCNF

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.

→ 즉, (A,C,D,E) 상에서 정의된 FD는 없음. 결국 non-trivial dependency가 하나도 없음으로 BCNF 이다.!

- In fact, dependency $AC \rightarrow D$ in F^+ shows R_2 is not in BCNF.

→ 즉, AC→D 는 non-trivial 하며, AC가 R_2의 superkey도 아님.

Testing a 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.

  1) trivial한지 아닌지를 check! 2) superkey인지 check !

1) $R_i - \alpha$: 결정자에 속하지 않는 속성
2) $\alpha \rightarrow \alpha^+$ 즉, $\alpha^+ = \beta$
3) α^+(즉, β)가 결정자에 속하지 않는 속성($R_i - \alpha$)을 하나도 가지고 있지 않다면, trivial하다.

- If a functional dependency shows that R_i violates BCNF, we use the above dependency to decompose R_i

Example of BCNF Decomposition

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

R is not in BCNF ($B \rightarrow C$ but B is not superkey)

(두 FD 모두 non-trivial함, A→B는 A가 superkey임)

Decomposition

1. 먼저 BCNF를 위반하는 FD를 이용해 분해함

   $R_1 = (B, C)$

2. 먼저 분해된 R_1의 superkey와 그 이외의 속성들 새로운 릴레이션 구성

   $R_2 = (A,B)$

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*)

Example of 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.

BCNF and Dependency Preservation

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

$R = (J, K, L)$
$F = \{JK \rightarrow L$ => non-trivial, but JK is superkey
$L \rightarrow K\}$ => non-trivial, and L is not superkey
Two candidate keys = JK and JL

R is not in BCNF

Any decomposition of R will fail to preserve

$$JK \rightarrow L$$

This implies that testing for $JK \rightarrow L$ requires a join

Decomposition 해보자!

- 먼저 R_1은 (L,K)로 분해하고, 나머지 R_2는 (J,L)로 분해된다.
- 그러면, 본래 있던 FD인 JK → L 은 분해된 릴레이션에서 보존되지 않는다.
- 보존하려면 ?

→ R_2에 K (candidate key에 포함되지만 R_2에는 없는 속성)을 추가한다.
→ 또 다른 문제! 중복 발생

Third Normal Form: Motivation

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.

$\alpha \rightarrow \beta$를 non-trivial하게 만드는 속성들

[3NF조건] 다음 3조건 중에 하나라도 만족

1. $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
2. α is a superkey for *R*
3. Each attribute *A* in $\beta - \alpha$ is contained in a candidate key for *R*.
   (**NOTE***:* each attribute may be in a different candidate key)
   This is a condition for dependency preserving

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*
  - *non-trivial*
  - *s_ID*, *dept_name* is a superkey
- *i_ID* $\rightarrow$ *dept_name*
  - *non-trivial*
  - *i_ID is not superkey*
  - *dept_name* is contained in a candidate key (*dept_name* – *i_ID* = *dept_name*)

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.

8.6 Decomposition Using Multivalued Dependencies

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 ? → non-trivial dependency가 하나도 없음
즉, (ID, child_name, phone_number) → any attribute

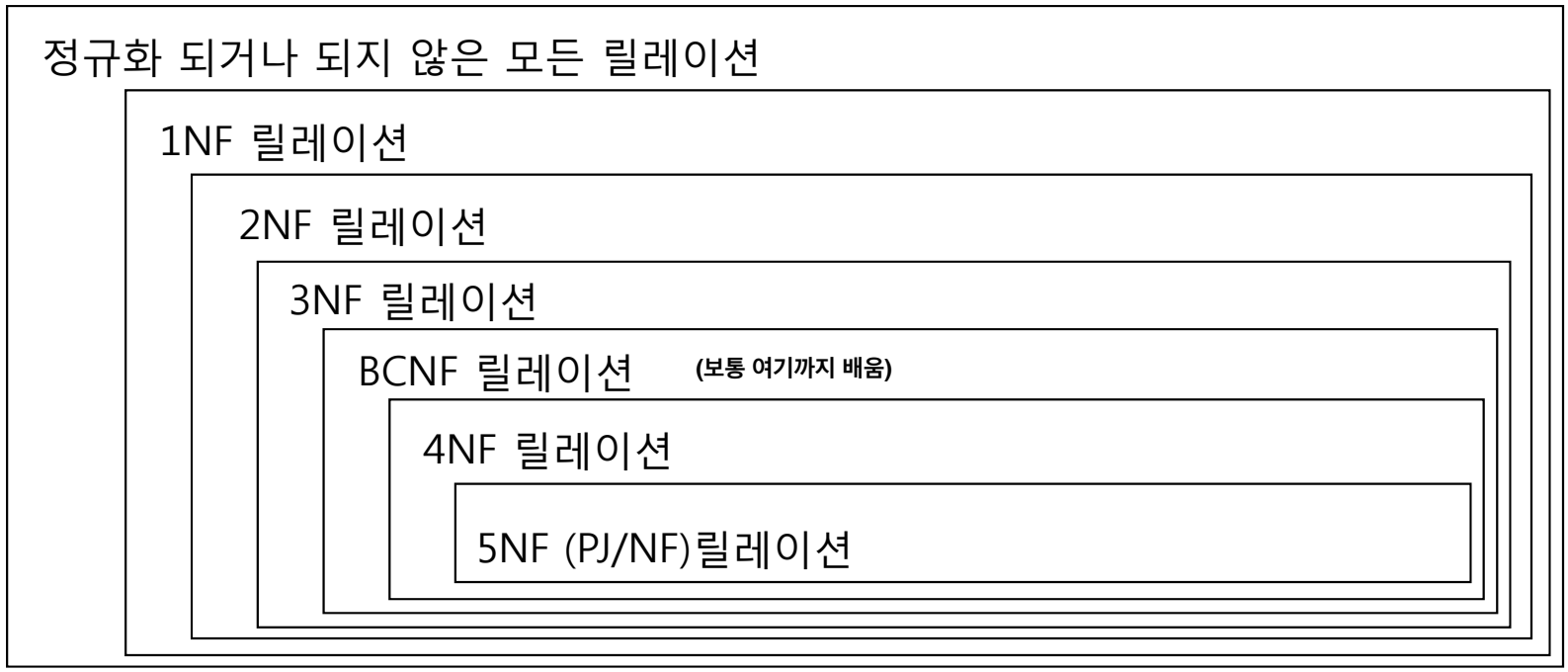
But, is it a good design ? → If your answer is NO, why ?

참고: Normalization

Database System Concepts, 6th Ed.

정규화 정리

대부분의 릴레이션은 BCNF까지 정규화하면 실제적인 이상현상이 없어지기 때문에 보통 BCNF까지 정규화를 진행함.



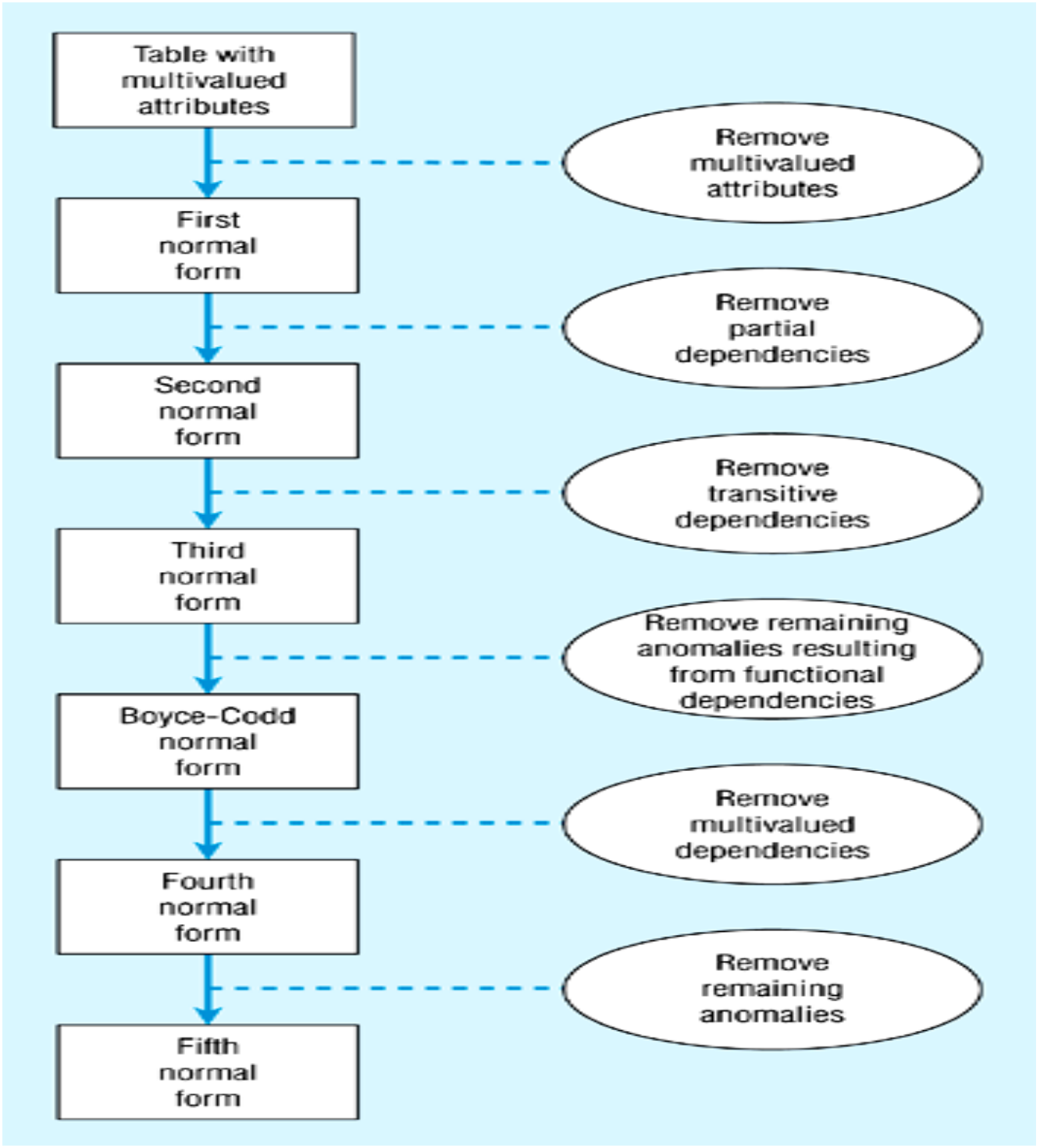


Table with multivalued attributes
First normal form
Second normal form
Third normal form
Boyce-Codd normal form
Fourth normal form
Fifth normal form
Remove multivalued attributes
Remove partial dependencies
Remove transitive dependencies
Remove remaining anomalies resulting from functional dependencies
Remove multivalued dependencies
Remove remaining anomalies