정교화 = 쪼개다

Databases – Relational Database Design (Chapter 7)

Jaeyong Choi
Dept. of Software, Gachon University

Features of Good Relational Designs

- Design alternatives: Combine schemas?
  - Combine *instructor* and *department* into *inst_dept*
  - This seems like a good idea because some queries can be expressed using fewer joins

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

instructor

department

Features of Good Relational Designs

테이블만 combine 하는게 좋지 않을 수도 있다.

- Design alternatives: Combine schemas?
  - But, result is possible repetition of information
    - Some user update the budget in one tuple
    - We cannot record information about newly created department

한쪽 튜플에서만 바뀌면 정보의 중복성때문에 문제가 생길수 있다.

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

instructor

department

Features of Good Relational Designs

- A combined schema without repetition
  - Consider combining relations
    - *sec_class*(*sec_id*, *building*, *room_number*) and
    - *section*(*course_id*, *sec_id*, *semester*, *year*)
  - Into one relation
    - *section*(*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*)
  - No repetition in this case

중복

Features of Good Relational Designs

작은 스키마로 나누자

- What about smaller schemas?
  - Supposing we had started with *inst_dept*, **how would we know to *decompose* it into *instructor* and *department***?
  - **Repetition of information** resulting from having to list the building and budget for each instructor associated with a department
  - Then, in order to make a good relation, can we find the part with repetition and decompose it into smaller schema?
  - Nope! A real-word DB has a large number of schemas and data. It is unreliable process.

중복성을 체크하기 위해 작은 스키마로 나누는건 쉽지 않다

Features of Good Relational Designs

- Rules for decomposition: *functional dependency*
  - Write a rule "if there were a schema (*dept_name*, *building*, *budget*), then *dept_name* would be a candidate key"
  - *dept_name* → *building*, *budget*
  - In *inst_dept*, because *dept_name* is not a candidate key, building and budget of a department may have to be repeated

    inst_dept과 합칠때 candidate key가 아니게 됨
  - ➔ Indicates the need to decompose *inst_dept*

$F : X \to Y$

dept_name에 의해서 building과 budget이 정해짐 : functional dependency

Features of Good Relational Designs

작게만드는것

- Not all decompositions are good
  - Suppose we decompose *employee*(*ID*, *name*, *street*, *city*, *salary*) into: *employee1* (*ID*, *name*) & *employee2* (*name*, *street*, *city*, *salary*)
  - Flaw arises when two employees may have the same name

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
⋮				

ID가 같은데 street 다르고 city 다르고 ...
어떻게 분해 하느냐에 따라 원래 갖고 있던 정보가 손실

Features of Good Relational Designs

정보 손실x

- Example of *lossless-join decomposition*
  - There is no loss of information by replacing r(R) with two relation schemas $r_1(R_1)$ and $r_2(R_2)$
  - Decomposition of $R = (A, B, C)$ into $R1 = (A, B)$ & $R2 = (B, C)$

A	*B*	*C*
α	1	A
β	2	B

r

A	*B*
α	1
β	2

$r_1 = \Pi_{A,B}(r)$

B	*C*
1	A
2	B

$r_2 = \Pi_{B,C}(r)$

Select * from (select R1 from r) natural join (select R2 from r)
←→
Select * from r

A	*B*	*C*
α	1	A
β	2	B

$r_1 \bowtie r_2$

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 *R* is not in "good" form, *decompose* it into a set of relation scheme {R_1, R_2, … , R_n} so that
  - Each relation scheme is in good form
  - The decomposition is a lossless-join decomposition and, preferably, dependency preserving
- *관계형 데이터베이스의 설계에서 중복을 최소화하게 데이터를 구조화하는 프로세스*

Atomic Domains and First Normal Form

- First normal form (1NF)

  개별적인 = 독립적인

  - Domain is *atomic* if its elements are indivisible units
    - Non-atomic domains: set of names, composite attributes

      street
      city
      zip
  - 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
    - E.g., set of accounts stored with each customer, set of owners stored with each account

- First normal form *cont'd*
  - 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 "CS0011" or "EE1127"
    - Department of a student can be found by writing code that breaks up the structure of an identification number
    - Doing so requires extra programming, and information gets encoded in the application program rather than in the database

Examples of 1NF

- Composite attribute, multivalued attribute

Customer

Customer ID	First Name	Surname	Telephone Number
123	Pooja	Singh	555-861-2025, 192-122-1111
456	San	Zhang	(555) 403-1659 Ext. 53; 182-929-2929
789	John	Doe	555-808-9633

잘 나눠진 table이라고 볼 수 없다

Customer

Customer ID	First Name	Surname	Telephone Number1	Telephone Number2
123	Pooja	Singh	555-861-2025	192-122-1111
456	San	Zhang	(555) 403-1659 Ext. 53	182-929-2929
789	John	Doe	555-808-9633	

The two telephone number columns still form a "repeating group": they repeat what is conceptually the same attribute, namely a telephone number.

Examples of 1NF

- Designs that comply with 1NF

Customer

Customer ID	First Name	Surname	Telephone Number
123	Pooja	Singh	555-861-2025
123	Pooja	Singh	192-122-1111
456	San	Zhang	182-929-2929
456	San	Zhang	(555) 403-1659 Ext. 53
789			

Note that the "ID" is no longer unique in this solution with duplicated customers. To uniquely identify a row, we need to use a combination of (ID, Telephone Number).

Customer Name

Customer ID	First Name	Surname
123	Pooja	Singh
456	San	Zhang
789	John	Doe

primary key : cumtomer id

Customer Telephone Number

Id	Customer ID	Telephone Number
1	123	555-861-2025
2	123	192-122-1111
3	456	(555) 403-1659 Ext. 53
4	456	182-929-2929
5	789	555-808-9633

primary key : id

The normal forms

- UNF: Unnormalized form
- 1NF: First normal form
- 2NF: Second normal form
- 3NF: Third normal form
- EKNF: Elementary key normal form
- BCNF: Boyce–Codd normal form
- 4NF: Fourth normal form
- ETNF: Essential tuple normal form
- 5NF: Fifth normal form
- DKNF: Domain-key normal form
- 6NF: Sixth normal form

3.5 normal form

The normal forms

	UNF (1970)	1NF (1970)	2NF (1971)	3NF (1971)	EKNF (1982)	BCNF (1974)	4NF (1977)	ETNF (2012)	5NF (1979)	DKNF (1981)	6NF (2003)
Primary key (no duplicate tuples)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
No repeating groups	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Atomic columns (cells have single value)	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
No partial dependencies (values depend on the whole of every Candidate key)	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
No transitive dependencies (values depend only on Candidate keys)	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency involves either a superkey or an elementary key's subkey	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	N/A
No redundancy from any functional dependency	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	N/A
Every non-trivial, multi-value dependency has a superkey	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	N/A
A component of every explicit join dependency is a superkey[8]	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	N/A

Decomposition Using Functional Dependencies

- Goal: Devise a theory for the following
  - Decide whether a particular relation *R* is in "good" form
  - In the case *R* is not in "good" form, *decompose* it into a set of relations {R_1, R_2, ... , R_n} so that
    - Each relation is in good form
    - The decomposition is a *lossless-join decomposition*
  - Our theory is based on:
    - Functional dependencies
    - Multivalued dependencies

f : x->y
x값에 따라 y값이 정해지는 것

Functional dependencies (FD)

- A *functional dependency* (FD) is a relationship between two attributes, typically between the PK and other non-key attributes within a table.
- For any relation R, attribute Y is functionally dependent on attribute X (usually the PK), if for every valid instance of X, that **value of X uniquely determines the value of Y**.

A → B

{A , B} → C

key -> super key -
> candidate -> primary

key에 저장된 속성에 따라서
value들이 바뀐다.
FD와 비슷. key의 컨셉을
일반화하는거라 보면 됨.

$$X \rightarrow Y$$

Determinant *Dependent*

ex) depart_name에 따라
building, budget이 나눠진다

X : dept_name -> Y: building, budge

- Functional dependencies (FD) examples
  - ISBN → Title
  - StudentID → Semester
  - Employee ID → Employee Name
  - Employee ID → Department ID
  - Department ID → Department Name

Student ID	Semester	Lecture	TA
1234	6	Numerical Methods	John
1221	4	Numerical Methods	Smith
1234	6	Visual Computing	Bob
1201	2	Numerical Methods	Peter
1201	2	Physics II	Simon

Employee ID	Employee name	Department ID	Department name
0001	John Doe	1	Human Resources
0002	Jane Doe	2	Marketing
0003	John Smith	1	Human Resources
0004	Jane Goodall	3	Sales

- Functional dependencies *cont'd*
  - Let R be a relation schema and α, β (⊆R) be attributes in r
  - The *functional dependency* $\alpha \rightarrow \beta$ *holds* on R
    - 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., $t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$ **알파의 값이 같으면 베타의 값이 같다**

알파 -> 베타로 functional dependency가 성립

  - Example:
    - $A \rightarrow C$ is satisfied; $C \rightarrow A$ is not

알파 -> 베타

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

t_1
t_2

t1[A] = t2[A] => t1[C] = 52[C]

- Functional dependencies *cont'd*
  - *K* is a *superkey* for relation schema r(R), R denoting the set of attributes, if and only if $K \rightarrow R$ K->R, R->K 즉, K=R임.
  - *For every pair of tuple t_1, t_2 from the instance,*
    - *Whenever $t_1[K] = t_2[K]$, it is also the case that $t_1[R] = t_2[R]$*
  - Consider the schema:
    - *inst_dept* (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*) r(R)
  - We expect these functional dependencies to hold:
    - *dept_name* → *building*, *ID* → *building*
    - *ID*, *dept_name* → *name*, *salary*, *building*, *budget*
  - But not the following:
    - *dept_name* → *salary*

f1 = α1 -> β1
f2 = α2 -> β2 ...
{f1, f2, .., fn} = F

- 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
    - 제약조건
    - We say that *F* holds on *R* if all legal relations on *R* satisfy the set of functional dependencies *F*
    - 고정한다
  - 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.
    - E.g., an instance of *instructor* may, by chance, satisfy *name* → *ID*

1+1 = 2은 trivial 하지 않는다
trivial : 자명하다. 추가적인 증명/생각을 하지 않아도 된다
ex) 1=1

Trivial functional dependency

- A functional dependency is *trivial* if it is satisfied by all instances of a relation, e.g.,
  - *ID, name* → *ID* ID가 이미 있으니
  - *name* → *name*
- If one "side" is a subset of the other, it's considered trivial.
- In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

ID ⊆ {ID, name}

α = β -> β ⊂ α

닫혀 있다.

- 한 집합 A가 있을 때, 이 A의 모든 원소 다른 원소와 함께 어떤 변환을 겪을 때, 그 변환 결과 역시 A의 원소, 혹은 부분집합일 경우 해당 변환은 집합 A에 대해 닫혀 있는 것이 된다.
- 예를 들어 모든 양수의 집합은 덧셈에 대하여 닫혀 있으며, 모든 뺄셈의 집합은 실수에 대하여 닫혀 있다.
- 어떤 범위에 있어서 와 같이 양 끝을 포함하는 구간을 닫힌 구간, 폐구간이라한다.
- 전자전기공학에선 항상 전류가 흐르도록 폐루프가 형성되어, 개방된 구간이 없는 회로를 닫힌 회로라고 한다.

(0, 1) [0, 1]

Closure of a set of functional dependencies

- Given a set *F* of functional dependencies, there are other functional dependencies that are *logically implied* by *F*
- Example
  - Assume that schema r(A,B,C)
  - if $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
  - *For the value A, there are only on corresponding value for B, and for that value of B, there can only be one corresponding value C*

F = {A->B, B->C}
{A->B, B->C, A->C}

- The set of *all* functional dependencies logically implied by *F* is the *closure* of *F*, denoted by F^+
  - F^+ is a superset of *F*

imply 할 수 있는건 모두
끌어와서 넣어주면 closure

Exercise

- It is important to realize that an instance of a relation may satisfy some functional dependencies that are not required to hold on the relation's schema.
- Let's find out functional dependency in the following example

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure 8.5 An instance of the *classroom* relation.

Exercise

- Let's find out functional dependency in the following example
  - Room_number → capacity
  - Building, room_number → capacity

room_number -> capacity가
 보존되는 함수 집합에 포함되지 않을
수 있다.
데이터가 바뀌거나 없어질 수도 있다
그래서 room_number -> capacity가
아니라
(Building, room_number)->capacity

building	*room_number*	*capacity*
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

Figure 8.5 An instance of the *classroom* relation.

Second Normal Form (2NF)

- A relation is in the second normal form if it fulfills the following two requirements:
  - It is in first normal form.
  - It does not have any non-prime attribute (후보 키에 속하지 않은 속성) that is functionally dependent on any proper subset of any candidate key of the relation(후보키의 하위 집합에 기능적으로 의존하는).
    - **A non-prime attribute of a relation** is an attribute that is not a part of any candidate key of the relation.
- *제1 정규형에 속하면서, 기본키가 아닌 모든 속성이 기본키에 완전 함수 종속되면* ***제2 정규형****이다.*

Second Normal Form (2NF)

- Example

{학번, 과목코드} -> 성적
{학번, 과목코드} -> 학부
{학번, 과목코드} -> 등록금
학번 -> 학부
학번 -> 등록금
학부 -> 등록금

학번->학부, 학번->등록금 두개의 부분 함수 종속성을 가지고 있다. 이를 제거해 주는 것을 제2정규화라고 한다.

학번	과목코드	성적	학부	등록금
20800399	CSE011101	A+	컴퓨터공학부	350
20800399	CSE022202	A	컴퓨터공학부	350
20800399	CSE033303	B+	컴퓨터공학부	350
21300758	MEC011101	F	경영학부	300
21400001	POD032939	C+	기계공학부	400
21500399	CSE011101	D	컴퓨터공학부	350

Second Normal Form (2NF)

- Example
  - 학번->학부 함수종속성으로 볼 때, 학번만으로 학부에 대한 결정을 지을 수 있음
  - 그러나 현재 기본키가 학번, 과목코드로 이루어져 있기 때문에 학번만으로 학부에 대한 결정을 지을 수 있다는 게 의미가 없어짐
  - 그래서 부분 함수 종속성을 제거하는 제2정규화 과정 수행 필요.

  - 학번, 학부, 등록금 속성을 가지는 학생 릴레이션과 학번, 과목코드, 성적 속성을 가지는 성적릴레이션 으로 나눔.

학생 릴레이션

학번	학부	등록금
20800399	컴퓨터공학부	350
21300758	경영학부	300
21400001	기계공학부	400
21500399	컴퓨터공학부	350

성적 릴레이션

학번	과목코드	성적
20800399	CSE011101	A+
20800399	CSE022202	A
20800399	CSE033303	B+
21300758	MEC011101	F
21400001	POD032939	C+
21500399	CSE011101	D

Second Normal Form (2NF)

- Example
  - {Model, Manufacturer} -> Model Full Name
  - {~~Model,~~ Manufacturer} -> Manufacturer Country

full-key dependencies

part-key dependencies

Electric Toothbrush Models

Manufacturer	Model	Model Full Name	Manufacturer Country
Forte	X-Prime	Forte X-Prime	Italy
Forte	Ultraclean	Forte Ultraclean	Italy
Dent-o-Fresh	EZbrush	Dent-o-Fresh EZbrush	USA
Kobayashi	ST-60	Kobayashi ST-60	Japan
Hoch	Toothmaster	Hoch Toothmaster	Germany
Hoch	X-Prime	Hoch X-Prime	Germany

Second Normal Form (2NF)

- Even if the designer has specified the primary key as {Model full name}, the relation is not in 2NF because of the other candidate keys.
- {Manufacturer, Model} is also a candidate key, and Manufacturer country is dependent on a proper subset of it

Second Normal Form (2NF)

- To make the design conform to 2NF, it is necessary to have two relations:

Electric toothbrush manufacturers

<u>Manufacturer</u>	Manufacturer country
Forte	Italy
Dent-o-Fresh	USA
Brushmaster	USA
Kobayashi	Japan
Hoch	Germany

Electric toothbrush models

<u>Manufacturer</u>	<u>Model</u>	Model full name
Forte	X-Prime	Forte X-Prime
Forte	Ultraclean	Forte Ultraclean
Dent-o-Fresh	EZbrush	Dent-o-Fresh EZbrush
Brushmaster	SuperBrush	Brushmaster SuperBrush
Kobayashi	ST-60	Kobayashi ST-60
Hoch	Toothmaster	Hoch Toothmaster
Hoch	X-Prime	Hoch X-Prime

Third normal form (3NF)

- Normalizing a database design to reduce the duplication of data and ensure referential integrity by ensuring that:
  - The entity is in second normal form.
  - **All the non-prime attributes** must depend only on **the candidate keys**.
  - 기본키 이외의 다른 컬럼이 그외 다른 컬럼을 결정할 수 없음.

Third normal form (3NF)

Prime-attribute

- {Tournament, Year}→ Winner

Non-prime attribute

- Winner → Winner date of birth
- {Tournament, Year}→ Winner date of birth

Tournament Winners

Tournament	Year	Winner	Winner Date of Birth
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 September 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975
Indiana Invitational	1999	Chip Masterson	14 March 1977

Tournament Winners

Tournament	Year	Winner	Winner Date of Birth
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 September 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975
Indiana Invitational	1999	Chip Masterson	14 March 1977

Tournament Winners

Tournament	Year	Winner
Indiana Invitational	1998	Al Fredrickson
Cleveland Open	1999	Bob Albertson
Des Moines Masters	1999	Al Fredrickson
Indiana Invitational	1999	Chip Masterson

Winner Dates of Birth

Winner	Date of Birth
Chip Masterson	14 March 1977
Al Fredrickson	21 July 1975
Bob Albertson	28 September 1968

BCNF -> 3NF

Third normal form (3NF)

3NF : a->b, b->r이면 a->r인 것을 만족(이행종속)하지 못하게 만듦

= functional dependencies

- A relation schema R is in 3NF if for all $\alpha \rightarrow \beta$ in F^+ at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for R (i.e., $\alpha \rightarrow R$)
  - Each attribute A in $(\beta - \alpha)$ is contained in a **candidate key for R** (NOTE: each attribute may be in a different candidate key)

BCNF

F+에 있는 모든 functional dependencies에 대해서 1개라도 만족되면 R이 3NF에 포함된다.

조건이 많다고 해서 BCNF의 업그레이드 버전이 아니라 이 중 1개만 만족해도됨

- 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)

Boyce-Codd normal form (BCNF or 3.5NF)

- It is a slightly stronger version of the third normal form (3NF)
  - BCNF was developed in 1975 by Raymond F. Boyce and Edgar F. Codd to address certain types of anomalies not dealt with by 3NF as originally defined.
  - 후보키를 여러개 가지고 있는 릴레이션에서는 제3정규형을 만족하더라도 이상현상이 생길수 있음.
- 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 $\alpha \rightarrow \beta$ (α, $\beta \subseteq R$), at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for *R* (i.e., $\alpha \rightarrow R$)

Boyce-Codd normal form (BCNF or 3.5NF)

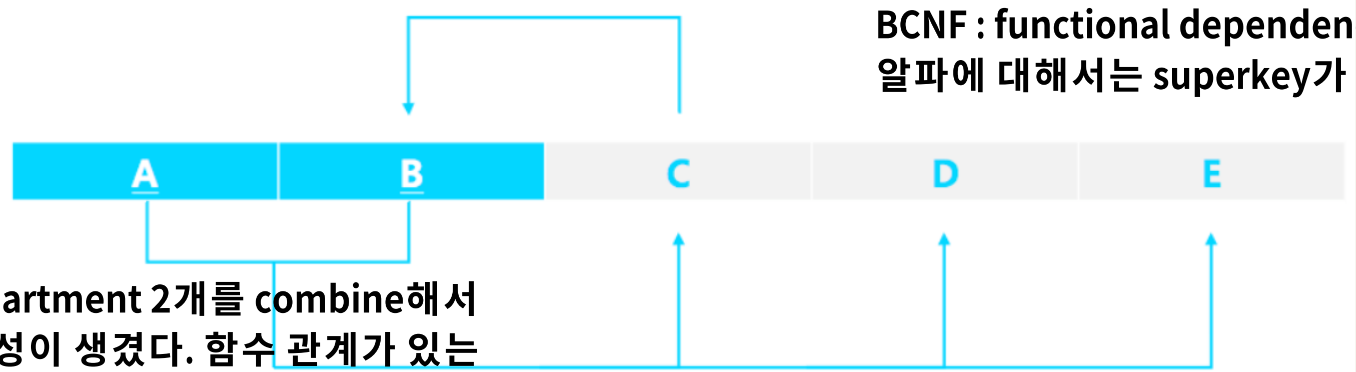
- Example schema *not* in BCNF:
  - *instr_dept* (*ID*, *name*, *salary*, *dept_name*, *building*, *budget*)
    - An FD *dept_name* → *building*, *budget* holds, but *dept_name* is not a superkey
  - AB→C,D,E, C→B

f1 = {ID, dept_name} -> {name, salary, bulding}

알파1 베타1

f2 = {dept_name} -> {building, budget}

BCNF : functional dependency가 있는 모든 알파에 대해서는 superkey가 되도록 하자



BCNF를 위반하는 경우

instructor와 department 2개를 combine해서 만들었는데 중복성이 생겼다. 함수 관계가 있는 게 어떤 사람을 추가할때마다 중복적으로 써야 되는 문제가 있었다 super key였으면 중복적으로 바꿔야 되는 문제가 안생길 수 있었다. super key에 좌지우지 되기 때문에 1:27:28

1. α1 -> β1 is trivial하지 않다.
2. dept_name이 instr_dept의 슈퍼키가 아니였다

Boyce-Codd normal form (BCNF or 3.5NF)

α, β ⊆ R

α -> β

- Decomposing a schema into BCNF
  - Suppose we have a schema *R* and a **non-trivial dependency α→β** causes a violation of BCNF
  - We **decompose** *R* into:
    - (α U β)
    - (*R* – (β – α))

이렇게 나누면
BCNF 만족

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

  - In previous example,
    - α = *dept_name*
    - β = *building*, *budget*
  - *inst_dept* is decomposed into:

super key

r1 (α U β) = (*dept_name*, *building*, *budget*)

r2 (*R* – (β – α)) = (*ID*, *name*, *salary*, *dept_name*)

Boyce-Codd normal form (BCNF or 3.5NF)

- Example

STUDENT	COURSE	INSTRUCTOR
야붕	Java Programming	James Gosling
야붕	Machine Learning	Andrew Ng
모찌	Computer Architecture	Alan Turing
양갱	Java Programming	James Gosling

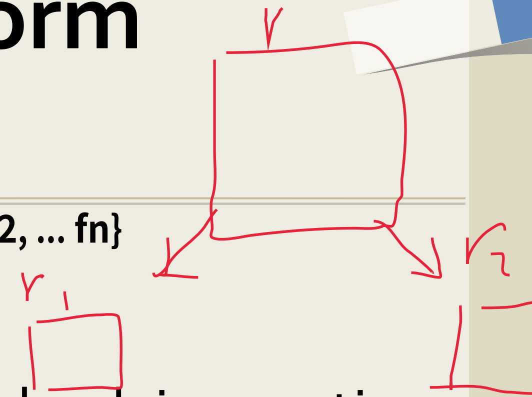
  - Candidate keys : {student, course} or {course, instructor}
  - BCNF를 위반하는 nontrivial FD 를 찾기
    - {student, course}→ instructor
    - **Instructor → course**
  - (α U β)와 (R – (β – α))로 분해하기
    - R1(instructor, course), R2(student, instructor)

INSTRUCTOR	COURSE
James Gosling	Java Programming
Andrew Ng	Machine Learning
Alan Turing	Computer Architecture

STUDENT	INSTRUCTOR
야붕	James Gosling
야붕	Andrew Ng
모찌	Alan Turing
양갱	James Gosling

Boyce-Codd normal form (BCNF or 3.5NF)

F = {f1, f2, … fn}



- BCNF and dependency preservation
  - Constraints, including FDs, 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* (3NF)

쪼갤때 functional dependency가 보존 안되는 경우도 있다. BCNF가 항상 functional dependency가 보존되는건 아니다. decomposition 하는데 잘 보존되도록 하자

1NF FD BCNF 3NF

3.5 NF?

FD가 보장되지 않을 수 있어서 고안한게 3NF

3NF vs BCNF

- Example schema in 3NF but *not* in BCNF:
  - *Advise*(<u>ID, advisor</u>, location, time)
    - <u>ID, Advisor</u> → location, time
    - <u>Advisor</u> → location
    - If the meeting location is one of the advisor's labs, the lab alone will determine the professor.
      - location→ advisor

이 예제는 2NF를 만족하지 않았으므로 3NF도 만족하지 않음. 3NF와 BCNF를 비교하기 위해서 임의로 만든 테이블 임.

ID	Advisor	Location	Time
20161	Kim	501	2019-10-10
20162	Park	504	2019-10-11
20163	Kim	501	2019-10-12

ID	Advisor	Location	Time
20161	Kim	501	2019-10-10
20162	Park	504	2019-10-11
20163	Kim	502	2019-10-12

Higher normal forms = 4NF

1번을 만족함

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a relation:
  - *inst_info* (*ID, child_name, phone*)
  - An instructor may have more than one phone and can have multiple children

99999 ID를 가진 아버지

아버지의 전화번호

No non-trivial FDs and therefore in BCNF

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

Trivial FD

AB→ A, AB→ B

핸드폰 2개

Non-trivial FD

AB→C, AB→ AC

Higher normal forms

문제점

삽입 이상

- *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)

1번만 삽입하면 되는지 2번이나 해야돼서
BCNF가 완벽하지는 않다

- Therefore, it is better to decompose *inst_info* into:
  - *inst_child* (*ID*, *child_name*)
  - *inst_phone* (*ID*, *phone*)
- This suggests the need for higher normal forms, such as *fourth normal form* (4NF)

Abnormalities

- Example

STUDENT	COURSE	INSTRUCTOR
야붕	Java Programming	James Gosling
야붕	Machine Learning	Andrew Ng
모찌	Computer Architecture	Alan Turing
양갱	Java Programming	James Gosling

- **삽입이상**
  Algorithms 라는 수업이 Dijkstra 에 의해 열렸다고 하자. 하지만 수강생이 아무도 없는 경우 삽입할 수 없다.
- **갱신이상**
  James Gosling 이 담당하는 강의가 바뀌게 될 경우 수강생의 수만큼 갱신해 줘야 하므로 하나라도 빠뜨리면 데이터 불일치 문제가 발생할 여지가 있다.
- **삭제이상**
  모찌가 자퇴해서 Computer Architecture 수업의 수강생이 없어지면 Alan Turing 이라는 강사도 사라진다.

Functional-Dependency Theory

- Closure of a set of functional dependencies
  - Given a set *F* of functional dependencies, there are other functional dependencies that are logically implied by *F*
    - E.g., if $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
  - The set of *all* functional dependencies logically implied by *F* is the *closure* of *F* denoted by F^+

F = {f1, f2, … }

찾아서 다 넣어주면 F+

Closure of a set of functional dependencies *cont'd*

- We can find F^+, the closure of F, by repeatedly applying *Armstrong's Axioms*: 공리
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ (*reflexivity, 반사*)
  - If $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$ (*augmentation, 확대*) 감마가 추가돼도
  - If $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ (*transitivity, 이행성*)
- These rules are:
  - *sound* (generate only FDs that actually hold), and
  - *complete* (generate all FDs that hold)

•건전하다(sound) : 이 규칙들을 잘못된 함수 종속을 생성하지 않는다.
•완전하다(complete) : 이 규칙은 주어진 함수 종속의 집합 F에 대해서 모든 F+를 생성할 수 있다.

위 규칙을 이용하면 더이상 찾을 필요가 없다

- 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 $CG \rightarrow I$ **양쪽에 G**
    - $CG \rightarrow HI$
      - By augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity **양쪽에 CG인데 같은거라 생략**

- Procedure for computing F^+
  - To compute the closure of a set of FDs F
    - We shall see an alternative procedure for this task later

$F^+ = F$
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
      add the resulting functional dependency to F^+
until F^+ does not change any further

$\beta \subseteq \alpha$ $\alpha \to \beta$

$\gamma\alpha \to \gamma\beta$

$\alpha \to \beta$ $\beta \to \gamma$

$\alpha \to \gamma$

- Additional rules:
  - Union rule:
    - If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds
    - (by reflexivity) $\alpha \rightarrow \beta\alpha$, (by augmentation) $\alpha\beta \rightarrow \gamma\beta$, (by transitivity) $\alpha \rightarrow \beta\gamma$
  - Decomposition rule:
    - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds
    - (by reflexivity) $\beta\gamma \rightarrow \beta$, (by transitivity) $\alpha \rightarrow \beta$
  - Pseudo-transitivity rule:
    - If $\alpha \rightarrow \beta$ holds and $\beta\gamma \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds
    - (by augmentation) $\alpha\gamma \rightarrow \beta\gamma$, (by transitivity) $\alpha\gamma \rightarrow \beta\gamma$, $\beta\gamma \rightarrow \delta$
  - The above rules can be inferred from Armstrong's axioms

Proof?

- Closure of attribute sets for finding candidate key
  - We say that an attribute *B* is *functionally determined* by α if $\alpha \rightarrow B$
  - 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*

R(A,B,C,D)

A→ B
B→D
C→B

A+ → A, B, D
Cl(A) = {A,B,D}

result := α;
repeat
  for each functional dependency $\beta \rightarrow \gamma$ **in** *F* **do**
    begin
      if $\beta \subseteq result$ **then** $result := result \cup \gamma$;
    end
until (*result* does not change)

- Example
  - $R = (\boxed{A}, B, C, \boxed{G}, H, I)$
    $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
  - $(AG)^+$
    - 1. result = AG
    - 2. result = $ABCG$ $(A \rightarrow C$ and $A \rightarrow B)$
    - 3. result = $ABCGH$ $(CG \rightarrow H$ and $CG \rightarrow AGBC)$
    - 4. result = $ABCGHI$ $(CG \rightarrow I$ and $CG \rightarrow AGBCH)$
  - Is AG a candidate key?
    - Is AG a super key?
      - 영향을 받느냐 $AG \rightarrow R? == (AG)^+ \supseteq R$
    - Is any subset of AG a superkey?
      - $A \rightarrow R? == (A)^+ \supseteq R$
      - $G \rightarrow R? == (G)^+ \supseteq R$

A determines B
A determines C
…
RHS Attribute which can not be determined

영향을 받는것들

$\alpha = AG$

$(AG)^+ \supseteq ABCGHI$

AG에 영향을 받는 모든것을 포함해뒀느냐

알파+가 R의 모든 어트리뷰트를 가지고 있으면
R이 알파에 의해 좌지우지 되어 알파가 super key다.

Uses of attribute closure

- Testing for superkey: $\alpha^+ \supset R$
  - To test if α is a superkey, we compute α^+, and check if α^+ contains all attributes of *R*
- Testing functional dependencies
  in F^+
  - 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 by using attribute closure, and then check if it contains β.
- Computing closure of *F*
  - It gives us an alternative way to compute *F*+
  - For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$

Exercise

- Let's find possible candidate key set
  - R(A,B,C,D,E,F,G,H)
  - CH→G, A→BC, B→CFH, E→A, F→EG
- Solution) [AD, BD, DE, DF]
  - D+ → D
  - DA+→ADBCFHEG
  - DB+→ DBCFHEGA
  - ~~DC+→DC~~
  - DE+→ DEA (DA is already candidate key)
  - DF+→DFEG (DE is already candidate key)
  - ~~DCH+→ DCH~~

- Cover
  - A set E of functional dependencies, if E ⊂ F+
  - all functional dependencies of E are inferred from F
  - F covers E (meaning that E has everything in F)
- Canonical cover: Fc

정규의

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

중복된

    - E.g., $A \rightarrow C$ is redundant in: $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$

불필요한

  - Parts of a functional dependency may be redundant

A->C

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

- Extraneous(관계없는) attributes
  - An attribute of a functional dependency is said to be extraneous if **we can remove it** without changing the closure of the set of functional dependencies.
  - Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F
    - Attribute A is *extraneous* in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$
    - 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

- Extraneous attributes
  - E.g., $F = \{A \to C, AB \to C\}$
    - B is extraneous in $AB \to C$ because $\{A \to C, AB \to C\}$ logically implies $A \to C$ (i.e., the result of dropping B from $AB \to C$)
    - A alone can determine C
  - E.g., $F = \{A \to C, AB \to CD\}$
    - C is extraneous in $AB \to CD$ since $AB \to C$ can be inferred even after deleting C
  - E.g., F = { A → BC, B → C, AB → D }
    - C is extraneous in the RHS (Right Hand Side) of A → BC
      - A can determine B (A → BC), B can determine C (B → C). Hence, A can determine C also (Transitivity rule)
    - B is extraneous in the LHS of AB → D
      - from A → BC, it is clear that A determines B. it would indirectly mean that if you know A and B then you know D also.

- Testing if an attribute is extraneous (formal definition)
  - Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F
  - To test if attribute $A \in \alpha$ is extraneous 왼쪽이 복잡한 경우
    - **Compute $(\alpha - \{A\})^+$ using the dependencies in F**
    - **Check that $(\alpha - \{A\})^+$ contains β; if it does, A is extraneous in α**
    - E.g, $F = \{A \rightarrow C, AB \rightarrow C\}$
      - B is extraneous in AB→ C?
      - A+→ C, we don't need to B to explain C. so we infer that B is extraneous.
  - To test if attribute $A \in \beta$ is extraneous 오른쪽이 복잡한 경우
    - **Compute α^+ using only the dependencies in $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$**
    - **Check that α^+ contains A; if it does, A is extraneous in β**
    - E.g, F ={ AB → CD, A → E, E → C}
      - **C** is extraneous in **AB → CD?**
      - We compute the attribute **closure of AB** under F' = {AB → D, A → E, and E → C}.
      - AB+ → ABCDE, which includes **CD**, so we infer that C is extraneous.
      - AB+ → ABCDE, which includes **C**, so we infer that C is extraneous.

- Canonical cover
  - A *canonical cover* for *F* is a set of dependencies F_c s.t.
    - *F* logically implies all dependencies in F_c
    - F_c logically implies all dependencies in *F*
    - No functional dependency in F_c contains an extraneous attribute
    - Each left side of functional dependency in F_c is unique
  - To compute a canonical cover for *F*:

$F_c = F$
repeat
  Use the union rule to replace any dependencies in F_c of the form
    $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$.
  Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an extraneous
    attribute either in α or in β.
    /* Note: the test for extraneous attributes is done using F_c, not F */
  If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$ in F_c.
until (F_c does not change)

A	B	C	D	E
a1	b1	c1	d1	e1
a2	b1	C2	d2	e1
a3	b2	C1	d1	e1
a4	b2	C2	d2	e1
a5	b3	C3	d1	e1

Table R

- Decompose the functional dependencies
- Check RHS, LHS is extraneous or not

- 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 $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
  - 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 $B \rightarrow C$ ➔ $F' = \{A \rightarrow B, B \rightarrow C\}$**
      - Can use attribute closure of A in more complex cases
      - A+➔ BC 설명하는지 확인, A+=ABC, thus C is extraneous
    - The canonical cover is: $\{A \rightarrow B, B \rightarrow C\}$

Check the example in slide 7!

- Lossless-join decomposition (무손실 조인 분해)
  - Removing redundancy safely from databases while preserving the original data
  - If you decompose a relation R into relations R1, R2, you will have a Lossless-Join if a natural join of the two smaller relations yields back the original relation
  - For the case of $R = (R1, R2)$, we require that for all possible relations r on schema R:
    - $r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$
  - A decomposition of R into $R1$ and $R2$ is *lossless-join* if at least one of the following dependencies is in F^+:
    - $R1 \cap R2 \rightarrow R1$
    - $R1 \cap R2 \rightarrow R2$
    - If R1 ∩ R2 is the key of R1 or R2, it is lossless decomposition.

- Example
  - $R = (\underline{A}, \underline{B}, C)$
    $F = \{\underline{A} \rightarrow B, \underline{B} \rightarrow C\}$
- Decomposition in two different ways:
  - $R1 = (\underline{A}, B)$, $R2 = (\underline{B}, C)$
    - Lossless-join decomposition: $R1 \cap R2 = \{B\}$ and $B \rightarrow BC$ — $R1 \cap R2 \rightarrow R2$
    - Dependency preserving
  - $R1 = (\underline{A}, B)$, $R2 = (\underline{A}, C)$
    - Lossless-join decomposition: $R1 \cap R2 = \{A\}$ and $A \rightarrow AB$ — $R1 \cap R2 \rightarrow R1$
    - Not dependency preserving: cannot check $B \rightarrow C$ without computing $R1 \bowtie R2$

- 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

- Testing for dependency preservation
  - To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of R into $R_1, \ldots, R_n$
  - We apply the test on all dependencies in F to check if a decomposition is dependency preserving

```
compute F+;
for each schema Ri in D do
    begin
        Fi : = the restriction of F+ to Ri;
    end
F' := Ø
for each restriction Fi do
    begin
        F' = F' ∪ Fi
    end
compute F'+;
if (F'+ = F+) then return (true)
               else return (false);
```

Let a relation R(A,B,C,D) and
set a FDs F = { A -> B , A -> C , C -> D} are given.

A relation R is decomposed into
R1 = (A, B, C) with FDs F1 = {A -> B, A -> C}
R2 = (C, D) with FDs F2 = {C -> D}.

F' = F1 ∪ F2 = {A -> B, A -> C, C -> D}
so, F' = F.
And so, F'+ = F+.

- Example
  - $R = (\underline{A}, B, C)$
    $F = \{A \rightarrow B, B \rightarrow C\}$
  - R is *not* in BCNF ⓘ
    - Decompose into $R_1 = (A, B)$, $R_2 = (B, C)$
  - R_1 and R_2 in BCNF
    - Lossless-join decomposition
    - Dependency preserving

Algorithms for Decomposition

- 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 it includes all attributes of *R*, i.e., it is a superkey of *R*
  - *Simplified test*
    - 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 either

- Testing for BCNF *cont'd*
  - 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 $F = \{\underline{A} \rightarrow B, B\underline{C} \rightarrow D\}$
    - Decompose R into $R1 = (\underline{A}, B)$ and $R2 = (A, \underline{C}, D, E)$
    - Neither of the dependencies in F *(F의 dependency 중 어떤 것도)* contain only attributes from (A, C, D, E) so we might be mislead into thinking $R2$ satisfies BCNF
    - In fact, dependency $A\underline{C} \rightarrow D$ in F^+ shows $R2$ is not in BCNF

- Testing decomposition for BCNF
  - Checking if a relation R_i in a decomposition of R is in BCNF
    - Relation R_i 가 BCNF를 만족하는지 확인하는 방법
  - Either **test R_i** for BCNF with respect to the restriction of F to R_i, i.e., all FDs in F^+ that contain only attributes from R_i
  - Or **use the original set of dependencies F** that hold on R with the following test:
    - For every set of attributes $\alpha \subseteq R_i$, check that α^+ <u>either includes no attribute of $R_i - \alpha$</u> or <u>includes all attributes of R_i</u>
      - α +가 Ri-α의 속성을 포함하지 않거나 Ri의 모든 속성을 포함하는 경우
    - If the condition is violated by some $\alpha \rightarrow \beta$ in F, the dependency <u>$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$</u> can be shown to hold on R_i, and R_i violates BCNF
      - $\alpha \rightarrow (\alpha^+ - \alpha) \cap$ Ri는 Ri를 유지하는 경우

Example

R = (A, B, C) F = {A→B, B→C}, Key = {A}

(1) B+= C ≠ (ABC-B) (2) B→(C)∩*(BC)* = *C*

BCNF decomposition algorithm

- Each R_i is in BCNF, and decomposition is lossless-join

$result := \{R\};$
$done :=$ 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 $\alpha \cap \beta = \emptyset$;
      $\boxed{result := (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta);}$
    end
  else $done :=$ true;

R이 BCNF에 속하지 않는다고 가정

- Example of BCNF decomposition
  - $R = (\underline{A}, B, C)$
    $F = \{A \rightarrow B, B \rightarrow C\}$, Key = $\{A\}$
  - R is *not* in BCNF ($B \rightarrow C$ but B is not superkey)
  - Decomposition:
    - $R_1 = (B, C)$
    - $R_2 = (A, B)$

$B \rightarrow C$, $\alpha \rightarrow \beta$, $R_1 = (B,C)$

Result R(A,B,C) = (ABC – BC) U (BC-C) U (BC)
= (A) U (B) U (BC)

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 $\alpha \cap \beta = \emptyset$;
   result := $(result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$;
  end
 else *done* := true;

$result := \{R\};$
$done :=$ 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 $\alpha \cap \beta = \emptyset$;
      $result := (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$;
    end
  else $done :=$ true;

- Example of BCNF decomposition
  - *class* (*course_id*, *title, dept_name, credits*, *sec_id*, *semester*, *year*, *building, room_number, capacity, time_slot_id*)
  - Functional dependencies *F*:
    - *F*1: *course_id*→*title*, *dept_name*, *credits*
    - *F*2: *building*, *room_number*→*capacity*
    - *F*3: *course_id*, *sec_id*, *semester*, *year*→*building*, *room_number*, *time_slot_id*
  - A candidate key: {*course_id*, *sec_id*, *semester*, *year*}
  - BCNF decomposition:
    - *F*1 is non-trivial functional dependency
      - *course* (<u>*course_id*</u>, *title*, *dept_name*, *credits*)
      - *class*-1 (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)
    - *course* is in BCNF

*F*1: *course_id*→*title*, *dept_name*, *credits*, *a* → *β*
Result class = (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*) U (*course_id*, *title*, *dept_name*, *credits*)

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

Final result

course (course_id, title, dept_name, credits)

classroom (building, room_number, capacity)

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

Example of BCNF decomposition *cont'd*

- BCNF decomposition *cont'd*:
  - *F2* is non-trivial functional dependency
  - F2 holds on *class*-1 but {*building*, *room_number*} is not a superkey; we replace *class*-1 by:
    - *classroom* (<u>*building*, *room_number*</u>, *capacity*)
    - *section* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
  - *classroom* and *section* are in BCNF

F2: building, room_number→capacity, a → β

Result class = (course_id, sec_id, semester, year, building, room_number, time_slot_id) U (building, room_number, capacity,)

- BCNF and dependency preservation
  - It is not always possible to get a BCNF decomposition that is dependency preserving
  - $R = (J, K, L)$
    $F = \{JK \rightarrow L, L \rightarrow K\}$
    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

- 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)
    - 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
  - *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*}
  - *dept_advisor* 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

- Testing for 3NF
  - if α is a superkey**,** Use attribute closure to check for each dependency α→β
  - If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of *R*
    - 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 can be done in a 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
    $i := i + 1$;
    $\boxed{R_i := \alpha\,\beta;}$
if none of the schemas R_j, $j = 1, 2, \ldots, i$ contains a candidate key for R
  then
    $i := i + 1$;
    $R_i :=$ any candidate key for R;
/* Optionally, remove redundant relations */
repeat
    if any schema R_j is contained in another schema R_k
      then
        /* Delete R_j */
        $R_j := R_i$;
        $i := i - 1$;
until no more R_js can be deleted
return $(R_1, R_2, \ldots, R_i)$

- Example of 3NF decomposition
  - *cust_banker_branch* = (*customer_id*, *employee_id*, *branch_name*, *type*)
  - Functional dependencies *F*:
    - *F*1: *customer_id*, *employee_id* → *branch_name*, *type*
    - *F*2: *employee_id* → *branch_name*
    - *F*3: *customer_id*, *branch_name* → *employee_id*
  - We first compute a canonical cover
    - *branch_name* is extraneous in the r.h.s. of *F*1
    - No other attribute is extraneous, so we get F_c = {*F*1′, *F*2, *F*3}, where *F*1′ = *customer_id*, *employee_id* → *type*

F1: *customer_id*, *employee_id* → *branch_name*, *type*
F2: *employee_id* → *branch_name*
F3: *customer_id*, *branch_name* → *employee_id*

- Example of 3NF decomposition *cont'd*
  - *for each* loop generates following 3NF schema:
    - *R1* = (*customer_id*, *employee_id*, *type*)
    - *R2* = (*employee_id*, *branch_name*)
    - *R3* = (*customer_id*, *branch_name*, *employee_id*)
  - Observe that *R1* contains a candidate key of the original schema, so no further relation schema needs be added
  - At end of for each loop, detect and delete schemas, such as *R2*, that are subsets of other schemas
    - Result will not depend on the order in which FDs are considered
  - The resultant simplified 3NF schema is:
    - *R1* = (*customer_id*, *employee_id*, *type*)
    - *R3* = (*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

Decomposition Using Multivalued Dependencies

- 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:
      - (99999, David, 512-555-1234)
      - (99999, William, 512-555-4321)
      - (99999, David, 512-555-4321)
      - (99999, William, 512-555-1234)
    - This relation is in BCNF. Why?

Check the example in slide 38!

- Functional dependencies (FD, 함수 종속)
  - If A → B, then we cannot have two tuples with the same A value but different B values.
- Multivalued dependencies (MVD, 다치 종속)
  - 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:
    - $t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$
    - $t_3[\beta] = t_1[\beta]$
    - $t_3[R - \beta] = t_2[R - \beta]$
    - $t_4[\beta] = t_2[\beta]$
    - $t_4[R - \beta] = t_1[R - \beta]$

(99999, David, 512-555-1234)
(99999, William, 512-555-4321)
(99999, David, 512-555-4321)
(99999, William, 512-555-1234)

{과목} ↠ {교재}일 때, 과목 어트리뷰트가 교재 어트리뷰트의 값 하나를 결정하는 것이 아니라, 여러 개의 값, 즉 값의 집합을 결정한다는 뜻

Multivalued dependencies *cont'd*

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

Teaching database

Class	**Book**	**Instructor**
AHA	Silberschatz	John D
AHA	Nederpelt	John D
AHA	Silberschatz	William M
AHA	Nederpelt	William M
AHA	Silberschatz	Christian G
AHA	Nederpelt	Christian G
OSO	Silberschatz	John D
OSO	Silberschatz	William M

Class $\twoheadrightarrow$ book
Class $\twoheadrightarrow$ instructor

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

$\boxed{t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]}$

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

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

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

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

- Example
  - Let *R* be a relation schema with a set of attributes that are partitioned into **3 nonempty subsets**: *Y*, *Z*, and *W*
  - We say that $Y \twoheadrightarrow Z$ (*Y* *multidetermines* *Z*) if and only if for all possible relations *r*(*R*)
    - $\langle y1, z1, w1 \rangle \in r$ and $\langle y1, z2, w2 \rangle \in r$, then
    - $\langle y1, z1, w2 \rangle \in r$ and $\langle y1, z2, w1 \rangle \in r$
  - Note that since the behavior of *Z* and *W* are identical (독립적), it follows that $Y \twoheadrightarrow Z$ if $Y \twoheadrightarrow W$
    - *다치 종속은 항상 짝을 지어 발생함*

- Example
  - 학생과 수강과목은 1:N의 관계이고, 학생과 동호회도 1:N의 관계
  - 정규화된 테이블은 모든 애트리뷰트로 구성되는 합성키(학생, 수강과목, 동호회)를 가지며, 다른 함수적 종속성이 존재하지 않으므로 BCNF에 해당함

학생	수강과목	동호회
윤경환	수학	스키반
윤경환	수학	서도회
윤경환	물리학	스키반
윤경환	물리학	서도회
윤경환	영어	스키반
윤경환	영어	서도회
신종철	물리학	영어회화반
신종철	물리학	유도회
신종철	화학	영어회화반
신종철	화학	유도회

학생 ↠ 수강과목
학생 ↠ 동호회

- Example *cont'd*
  - In *inst_info* relation:
    - $ID \twoheadrightarrow child\_name$
    - $ID \twoheadrightarrow phone\_number$
  - Formalization:
    - Given a particular value of Y (ID), it is associated with a set of values of Z ($child\_name$) and a set of values of W ($phone\_number$)
    - These two sets are in some sense independent of each other
  - Note:
    - If $Y \rightarrow Z$ then $Y \twoheadrightarrow Z$ (but *not* vice versa)

- Use of multivalued dependencies
  - We use multivalued dependencies in two ways:
    - To test relations to determine **whether they are legal under a given set of functional and multivalued dependencies**
    - 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

ID	*dept_name*	*street*	*city*
22222	Physics	North	Rye
22222	Math	Main	Manchester

Figure 8.15 An illegal r_2 relation.

ID →→ *street, city*
ID →→ *dept_name*

Add the tuples
(22222, Physics, Main, Manchester)
(22222, Math, North, Rye)

- Theory of MVDs
  - From the definition of multivalued dependency, we can derive the following rule:
    - If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$ (i.e., every functional dependency is also a multivalued dependency)
    - If $\alpha \twoheadrightarrow \beta$, then $\alpha \twoheadrightarrow R - \alpha - \beta$
  - 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

- *Fourth normal form* (4NF)
  - A relation schema R is in *fourth normal form* (4NF) with respect to a set D of functional and multivalued dependencies, if for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
    - $\alpha \twoheadrightarrow \beta$ 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
  - Let r (R) be a relation schema, and let $r_1(R_1)$, $r_2(R_2)$, .. , $r_n(R_n)$ be a decomposition of r(R).
  - To check if each relation schema r_i in the decomposition is in 4NF, we need to find what multivalued dependencies hold on each r_i.
  - 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^+; Given schema R_i, let D_i denote the restriction of D^+ to R_i
while (**not** *done*) **do**
  if (there is a schema R_i in *result* that is not in 4NF w.r.t. D_i)
    then begin
      let $\alpha \twoheadrightarrow \beta$ be a nontrivial multivalued dependency that holds on R_i such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \emptyset$;
      $\boxed{result := (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta);}$
    end
  else *done* := true;

- Example
  - $result := (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta);$
  - $R = (A, B, C, G, H, I)$
    $F = \{A \twoheadrightarrow B\ (F1),\ B \twoheadrightarrow HI\ (F2),\ CG \twoheadrightarrow H\ (F3)\}$
  - *R* is *not* in 4NF since $A \twoheadrightarrow B$ and *A* is not a superkey for *R*
  - Decomposition:
    - *Using F1:* $A \twoheadrightarrow B$
      - ***R1* = (*A*, *B*)** (in 4NF)
      - *R2* = (*A*, *C*, *G*, *H*, *I*) (not in 4NF; decompose into *R3* and *R4*)
    - *Using F3:* $CG \twoheadrightarrow H$
      - ***R3* = (*C*, *G*, *H*)** (in 4NF)
      - *R4* = (*A*, *C*, *G*, *I*) (not in 4NF; decompose into *R5* and *R6*)
        - $A \twoheadrightarrow B$ and $B \twoheadrightarrow HI$ implies $A \twoheadrightarrow HI$ (MVD transitivity), and hence $A \twoheadrightarrow I$ (MVD restriction to *R4*)
    - *Using F4′:* $A \twoheadrightarrow I$
      - ***R5* = (*A*, *I*)** **(in 4NF)**
      - ***R6* = (*A*, *C*, *G*)** **(in 4NF)**

More Normal Forms

- Higher normal forms
  - As we see earlier, **multivalued dependencies** help us understand and eliminate some forms of **repetition of information** that cannot be understood in terms of functional dependencies.
  - ***Join dependencies*** generalize multivalued dependencies
    - Leads to *project-join normal form* (PJNF) (or *fifth normal form*)
  - A class of even more general constraints, leading to a normal form called *domain-key normal form*
    - Problem with these generalized constraints: hard to reason with, and no sound and complete set of inference rules exists
    - Hence rarely used