Bases de données

Lecture 12

Nabil Mustafa

mustafa@lipn.univ-paris13.fr

new idea

LOSSLESS DECOMPOSITIONS

Lossless Decompositions

Given a relation: $R\ (a, b, c, d, e)$

say we decompose it into two tables:

$R1\ (a, b, c, d)$ $R2\ (a, b, c, e)$

Can we 'lose' information by doing this?
. . . what does that even mean?

Lossless Decompositions

DECOMPOSING TABLE:

R

a	b	c	d	e
a1	b1	c1	d1	e1
a1	b1	c1	d2	e2

R1

a	b	c	d
a1	b1	c1	d1
a1	b1	c1	d2

R2

a	b	c	e
a1	b1	c1	e1
a1	b1	c1	e2

RECONSTRUCTING IT BACK:

```
SELECT R1.a, R1.b, R1.c, R1.d, R2.e
FROM R1, R2
WHERE R1.a = R2.a AND R1.b = R2.b AND R1.c = R2.c
```

JOINING **R1** and **R2** around a, b, c

a	b	c	d	e
a1	b1	c1	d1	e1
a1	b1	c1	d1	e2
a1	b1	c1	d2	e1
a1	b1	c1	d2	e2

$\neq$ **R**

Problem: failed to reconstruct the original table!

Lossless Decompositions

Given a relation: R (a, b, c, d, e)

say we decompose it into two tables:

R1 (a, b, c, d) R2 (a, b, c, e)

Goal: (**R1** NATURAL JOIN **R2**) *must* give back exactly **R**

called a LOSSLESS DECOMPOSITION

Lossless Decompositions

R

a	*b*	*c*	*d*	*e*
a1	b1	c1	d1	e1
a1	b1	c1	d2	e2

R1

a	*b*	*c*	*d*
a1	b1	c1	d1
a1	b1	c1	d2

R2

a	*b*	*c*	*e*
a1	b1	c1	e1
a1	b1	c1	e2

JOINING R1 and R2 around a, b, c

```
SELECT R1.a, R1.b, R1.c, R1.d, R2.e
FROM R1, R2
WHERE R1.a = R2.a AND R1.b = R2.b AND R1.c = R2.c
```

a	*b*	*c*	*d*	*e*
a1	b1	c1	d1	e1
a1	b1	c1	d1	e2
a1	b1	c1	d2	e1
a1	b1	c1	d2	e2

Why is this happening?

Reason: Multiple values of `d` and `e` for fixed values of `a, b, c`!

Lossless Decompositions

R1

<table>
<tr><th>a</th><th>b</th><th>c</th><th>d</th><th>e</th><th>f</th><th>g</th></tr>
<tr><td>a1</td><td>b1</td><td>c1</td><td rowspan="2">d1</td><td rowspan="2">e1</td><td>f1</td><td>g1</td></tr>
<tr><td>a2</td><td>b2</td><td>c2</td><td>f2</td><td>g2</td></tr>
</table>

R2

So the join gives 4 lines instead of 2!

Lossless Decompositions

R1

<table>
<tr><th>a</th><th>b</th><th>c</th><th>d</th><th>e</th><th>f</th><th>g</th></tr>
<tr><td>a1</td><td>b1</td><td>c1</td><td rowspan="2">d1</td><td rowspan="2">e1</td><td>f1</td><td>g1</td></tr>
<tr><td>a2</td><td>b2</td><td>c2</td><td>f2</td><td>g2</td></tr>
</table>

R2

$\boxed{\texttt{d, e} \longrightarrow \texttt{f, g}}$

Lossless Decompositions

R1

a	b	c	d	e	f	g
a1	b1	c1	d1	e1	f1	g1
a2	b2	c2				

R2

`d, e ⟶ f, g`

Now, each line of **R1** extends to a unique line in NATURAL JOIN

. . . **the join gives** 2 **lines, correctly**

Lossless Decompositions

R1

a	b	c	d	e	f	g
a1	b1	c1	d1	e1	f1	g1
a2	b2	c2			f2	g2

R2

`d, e ⟶ a, b, c`

Lossless Decompositions

R1 (columns a, b, c, d, e); R2 (columns d, e, f, g)

a	b	c	d	e	f	g
a1	b1	c1			f1	g1
			d1	e1		
					f2	g2

`d, e ⟶ a, b, c`

Now, each line of **R2** extends to a unique line in NATURAL JOIN

. . . **the join gives** 2 **lines, correctly**

Lossless Decompositions

Given a relation **R**

say we decompose it into two tables:

R1 **R2**

CLAIM: If

`R1 ∩ R2 ⟶ R1` **or** `R1 ∩ R2 ⟶ R2`

then (**R1** NATURAL JOIN **R2**) is equal to **R**

each row of (**R1** NATURAL JOIN **R2**) gives *at least* one row of **R**

to prove: each row of (**R1** NATURAL JOIN **R2**) gives *exactly one* one row of **R**

Lossless Decompositions

Another way of looking at it:

assume **R1** ∩ **R2** ⟶ **R1**

consider the table **R2** line by line:

for each line of **R2**, exactly **one matching line** of **R1**

these together give a line of **R**

So we cannot get anything 'extra' in **R**

R2

a	*b*	*c*	*d*	*e*	*f*	*g*
a	b	c	d	e	f	v
a1	b1	c1	e1	f	g	h
a1	b1	c1	e1	f	g	h
a1	b1	c1	e1	f	g	h
a1	b1	c1	e1	f	g	h
a1	b1	c1	e1	f	g	h

R1

a	*b*	*c*	*d*	*h*	*i*	*j*
a1	b1	c1	e1	f	g	h
a1	b1	c1	e1	f	g	h
a1	b1	c1	e1	f	g	h
a	b	c	d	h	i	w
a1	b1	c1	e1	f	g	h
a1	b1	c1	e1	f	g	h

R

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*
a	b	c	d	e	f	v	h	i	w

Lossless Decompositions

R

a	*b*		*d*	*e*
a1	b1		d1	e1
a1	b1	c1	d2	e2

not possible with this dependency!

If we had this dependency

$\boxed{\texttt{a, b, c} \longrightarrow \texttt{d}}$

we get lossless decomposition!

R

a	*b*	*c*	*d*	*e*
a1	b1	c1	d1	e1
a1	b1	c1	d1	e2

R1

a	*b*	*c*	*d*
a1	b1	c1	d1

R2

a	*b*	*c*	*e*
a1	b1	c1	e1
a1	b1	c1	e2

JOINING R1 and R2
around a, b, c

a	*b*	*c*	*d*	*e*
a1	b1	c1	d1	e1
a1	b1	c1	d1	e2

= **R**

Lossless Decompositions

CLAIM: If

$\boxed{\texttt{R1} \cap \texttt{R2} \longrightarrow \texttt{R1}}$ **or** $\boxed{\texttt{R1} \cap \texttt{R2} \longrightarrow \texttt{R2}}$

then $\Big($ **R1** NATURAL JOIN **R2** $\Big)$ is equal to **R**

Recall BCNF decomposition

If there is $\boxed{\alpha \longrightarrow \beta}$ in F^+ with

$$\Big(\beta - \alpha\Big) \bigcap \mathbf{R} \text{ not empty} \quad \textit{and} \quad \alpha \subseteq \mathbf{R} \text{ **not** a superkey of } \mathbf{R}$$

then

R1	**R2**
$\alpha \bigcup \beta$	$\mathbf{R} - \beta$

Is this lossless? Note that $\mathbf{R1} \cap \mathbf{R2} = \alpha$ and $\alpha \longrightarrow \mathbf{R1}$. Therefore,

$\Longrightarrow$ decompositions used in **BCNF** are lossless

new idea

3rd NORMAL FORM: DEFINITION

PRESERVING DEPENDENCIES

Given the relation: R (c, b, a, d)

`a ⟶ b` `c, b ⟶ a, d`

Is this in **BCNF** ? NO

`a ⟶ b` problem for R R1 (c, a, d) R2 (a, b)

BCNF decomposition: R2 (a, b) R1 (c, a, d)

Unpleasant side-effect: Difficult to verify the dependency

`c, b ⟶ a, d` in the two tables R1 and R2 without a join

PRESERVING DEPENDENCIES

Given a relation **R** and a set F of functional dependencies

we decompose **R** into tables **R1**,. . . , **Rk**

each table **Ri** has a list Fi of dependencies for it

R (a, b, c, d, e, f) F `a, b ⟶ f` `d ⟶ e, f`

R1 (a, b, c, d) F1 R2 (d, e, f) F2 `d ⟶ e, f`

Over time, we insert more data into these k tables

each insert, we check new data satisfies Fi for table **Ri**

`d ⟶ e, f` for table **R2**

but unable easily to check dependencies *across* tables

`a, b ⟶ f`

to avoid this problem, forced to check by JOINS *each* time we insert

3rd Normal Forms

Given a relation **R** and a set F of functional dependencies

R is in BCNF if it has no redundancy with respect to F^+

R is in BCNF if for all functional dependencies $\boxed{\alpha \longrightarrow \beta}$ in F^+

either $\boxed{\alpha \longrightarrow \beta}$ is *trivial* … that is, $\left(\beta - \alpha\right) \bigcap$ **R** is empty

or α is a superkey of **R**

We now loosen the BCNF condition to allow some dependencies in R

$\boxed{\alpha \longrightarrow \beta}$ can remain, even if α is not a superkey, **as long as**

each attribute in $\beta - \alpha$ is 'important'

3rd Normal Forms

Given a relation **R** and a set F of functional dependencies

R is in 3NF if for all functional dependencies $\boxed{\alpha \longrightarrow \beta}$ in F^+
where $\alpha \subseteq$ **R** and $\beta \subseteq$ **R**

either $\boxed{\alpha \longrightarrow \beta}$ is *trivial* … that is, $\left(\beta - \alpha\right) \bigcap$ **R** is empty

or α is a superkey of **R**

or each attribute in $\beta - \alpha$ is part of some candidate key of **R**

unintuitive condition, will become clear later!

note: all attributes in $\beta - \alpha$ need not be part of the *same* candidate key
different attributes can be part of different candidate keys

3rd Normal Forms

Given the relations: $R\ (c, b, a, d)$

`a ⟶ b` `c, b ⟶ a, d`

Is this in **3NF** ? YES

`a ⟶ b` a problem for **R** for **3NF**?

Is b part of *some* candidate key for **R**? **yes**

`c, b ⟶ a, d`

3rd Normal Forms: Algorithm

Given a set of functional dependencies F

the algorithm for computing a **3NF** decomposition:

An Overview (details later)

Step 1. Simplify F to a 'minimal' list of dependencies F_c

Step 2. for each dependency $\alpha \longrightarrow \beta$ in F_c, add relation

$$\alpha \bigcup \beta$$

Theorem: The above algorithm gives **3NF**, and preserves all dependencies!

new idea

EXTRANEOUS ATTRIBUTES

Step 1. Simplify F to a minimal list of dependencies F_c

remove all attributes from all dependencies in F that are *not necessary*
$\Longrightarrow$ keep simplifying F $\rightarrow$ F' as long as `closure` (F) = `closure` (F')

F	a, c $\longrightarrow$ g	c, g $\longrightarrow$ b, d	a, c, $\cancel{d}$ $\longrightarrow$ b
		$\downarrow$?	
F'	a, c $\longrightarrow$ g	c, g $\longrightarrow$ b, d	a, c $\longrightarrow$ b

can do this if: `closure` (F) = `closure` (F')

F	a, c $\longrightarrow$ g	c, g $\longrightarrow$ $\cancel{b}$, d	a, c, d $\longrightarrow$ b
		$\downarrow$?	
F'	a, c $\longrightarrow$ g	c, g $\longrightarrow$ d	a, c $\longrightarrow$ b

can do this if: `closure` (F) = `closure` (F')

Step 1. Simplify F to a minimal list of dependencies F_c

called extraneous

$a,\ \alpha \longrightarrow \beta$

Q: when can a be removed from this dependency ?

A: when β can be derived from just α in F

a can be removed if β is contained in the attribute closure of α in F

Example:

is this extraneous?

F $\quad a,\ c \longrightarrow g \qquad c,\ g \longrightarrow b,\ d \qquad a,\ c,\ d \longrightarrow b$

Yes $\quad a,\ c \longrightarrow g \ + \ c,\ g \longrightarrow b,\ d \quad \Longrightarrow \quad a,\ c \longrightarrow b$

Step 1. Simplify F to a minimal list of dependencies F_c

$\alpha \longrightarrow b,\ \beta$

Q: when can b be removed from this dependency ?

A: when just the dependency $\alpha \longrightarrow \beta$ in F is sufficient to derive b

if the closure of α in $F - [\alpha \longrightarrow b, \beta] + [\alpha \longrightarrow \beta]$ contains b

Example:

is this extraneous?

F $\quad a \longrightarrow b,\ c \qquad b \longrightarrow c \qquad a,\ b \longrightarrow c$

Yes $\quad a \longrightarrow b \quad b \longrightarrow c \quad \Longrightarrow \quad a \longrightarrow c$

```
def isAttributeExtraneous(F, alpha, beta, a):
    if a in beta:
        beta_prime = beta.difference({a})
        _F = list(F)
        _F.remove( [alpha, beta] )
        _F.append( [ alpha, beta_prime ] )
        if a in computeAttributeClosure(_F, alpha):
            return True, [ alpha, beta_prime ]
    if a in alpha:
        alpha_prime = alpha.difference({a})
        if beta.issubset( computeAttributeClosure(F, alpha_prime) ):
            return True, [ alpha_prime, beta ]
    return False, [ {}, {} ]
```

$\alpha \longrightarrow$ `b,` β

To remove `b`, check if

the closure of α in	F - $[\alpha \longrightarrow b, \beta]$ + $[\alpha \longrightarrow \beta]$	contains b

`a,` $\alpha \longrightarrow \beta$

To remove `a`, check if

β is contained in the attribute closure of α in F

```
import itertools
def powerSet(inputset):
    ...
def computeAttributeClosure(F, X):
    ...
def isAttributeExtraneous(F, alpha, beta, A):
    if A in beta:
        beta_prime = beta.difference({A})
        _F = list(F)
        _F.remove( [alpha, beta] )
        _F.append( [ alpha, beta_prime ] )
        if A in computeAttributeClosure(_F, alpha):
            return True, [ alpha, beta_prime ]
    if A in alpha:
        alpha_prime = alpha.difference({A})
        if beta.issubset( computeAttributeClosure(F, alpha_prime) ):
            return True, [ alpha_prime, beta ]
    return False, [ {}, {} ]
d = [
          [ {'A'}, {'B', 'C'} ],
          [ {'B'}, {'C'} ],
          [ {'A', 'B'}, {'C'} ]
    ]
print( isAttributeExtraneous(d, *[{'A'}, {'B', 'C'}], 'C') )
print( isAttributeExtraneous(d, *[{'A'}, {'B', 'C'}], 'B') )
```

```
(True, [{'A'}, {'B'}])
(False, [{}, {}])
```

new idea

CANONICAL FORMS

ALGORITHM

Given a set of functional dependencies F

> F_c is a **CANONICAL FORM** of F if :
> - F_c is a minimal list of dependencies for F
> - no two dependencies in F_c have same left side

In other words: from F, we want to construct F_c such that

`closure` (F) = `closure` (F_c)

no dependency in F_c has an extraneous attribute

no two dependencies in F_c have same left side

ALGORITHM

Given a set of functional dependencies F, algorithm to compute F_c

computeCanonicalCover $\big($ F $\big)$

$F_c = F$

While F_c keeps changing :

  for each dependency $\alpha \longrightarrow \beta$ in F_c :

    for each attribute a in $\alpha \bigcup \beta$:

      if a is extraneous in $\alpha \longrightarrow \beta$:

        delete it from $\alpha \longrightarrow \beta$

  if there exist two dependencies $\alpha \longrightarrow \beta_1$ and $\alpha \longrightarrow \beta_2$ in F_c :

    combine into one dependency $\alpha \longrightarrow \beta_1, \beta_2$

```
def computeCanonicalCover(F):
    OUT,changed = list(F),True
    while changed:
        changed = False
        for alpha_i, beta_i in OUT:
            for alpha_j, beta_j in OUT:
                if not changed and alpha_i == alpha_j and beta_i != beta_j:
                    OUT.append( [alpha_i, beta_i.union(beta_j)] )
                    OUT.remove( [alpha_i, beta_i] )
                    OUT.remove( [alpha_j, beta_j] )
                    changed = True
        for alpha, beta in OUT:
            for A in (alpha|beta):
                to_update,[new_alpha,new_beta]=isAttributeExtraneous(OUT,*[alpha,beta],A)
                if not changed and to_update:
                    if new_alpha != set() and new_beta != set():
                        OUT.append( [new_alpha, new_beta] )
                    OUT.remove( [alpha, beta] )
                    changed = True
                    break
    return OUT
d = [
        [ {'A'}, {'B'} ],  #A->B
        [ {'A'}, {'C'} ],  #A->C
        [ {'C', 'G'}, {'H'} ],  #CG->H
        [ {'C', 'G'}, {'I'} ],  #CG->I
        [ {'B'}, {'H'} ] #B->H
    ]
print( computeCanonicalCover(d) )
```

```
[[{'B'}, {'H'}], [{'A'}, {'B', 'C'}], [{'G', 'C'}, {'H', 'I'}]]
```

```
def computeCanonicalCover(F):
    ...
d = [
        [ {'A', 'C'}, {'G'} ],
        [ {'D'}, {'E', 'G'} ],
        [ {'B', 'C'}, {'D'} ],
        [ {'C', 'G'}, {'B', 'D'} ],
        [ {'A', 'C', 'D'}, {'B'} ],
        [ {'C', 'E'}, {'A', 'G'} ]
    ]
print( computeCanonicalCover(d) )
```

$d \longrightarrow g,\ e$
$c,\ b \longrightarrow d$
$c,\ g \longrightarrow b$
$a,\ c \longrightarrow b$
$c,\ e \longrightarrow a$

```
[[{'D'}, {'G', 'E'}], [{'B', 'C'}, {'D'}], [{'G', 'C'}, {'B'}], [{'E', 'C'}, {'A'}], [{'C', 'A'}, {'B'}]]
```

$a,\ c \longrightarrow \cancel{g}$ since can get from: $a,\ c \longrightarrow b$ + $c,\ b \longrightarrow d$ + $d \longrightarrow g,\ e$

$d \longrightarrow g,\ e$

$c,\ b \longrightarrow d$

$c,\ g \longrightarrow b,\ \cancel{d}$ since can get from: $c,\ g \longrightarrow b$ + $c,\ b \longrightarrow d$

$a,\ c,\ \cancel{d} \longrightarrow b$ since can get from: $a,\ c \longrightarrow g$ + $c,\ g \longrightarrow b,\ d$ + $d \longrightarrow g,\ e$

$c,\ e \longrightarrow a,\ \cancel{g}$ since can get from: $c,\ e \longrightarrow a$ + $a,\ c \longrightarrow b$ + $c,\ b \longrightarrow d$


```
def computeCanonicalCover(F):
    ...
d = [
        [ {'A', 'C'}, {'G'} ],
        [ {'D'}, {'E', 'G'} ],
        [ {'B', 'C'}, {'D'} ],
        [ {'C', 'G'}, {'B', 'D'} ],
        [ {'A', 'C', 'D'}, {'B'} ],
        [ {'C', 'E'}, {'A', 'G'} ]
    ]
print( computeCanonicalCover(d) )
```

$d \longrightarrow g,\ e$
$c,\ b \longrightarrow d$
$c,\ g \longrightarrow b$
$a,\ c \longrightarrow b$
$c,\ e \longrightarrow a$

Depending on the order chosen by algorithm, can get different but also good results!

```
[[{'C', 'A'}, {'G'}], [{'D'}, {'E', 'G'}], [{'B', 'C'}, {'D'}], [{'C', 'G'}, {'D'}], [{'C', 'D'}, {'B'}], [{'E', 'C'}, {'A'}]]
```

```
[[{'D'}, {'E', 'G'}], [{'C', 'B'}, {'D'}], [{'C', 'G'}, {'D'}], [{'C', 'E'}, {'A'}], [{'C', 'A'}, {'B'}]]
```


```
[[{'C', 'A'}, {'G'}], [{'D'}, {'E', 'G'}], [{'B', 'C'}, {'D'}], [{'C', 'G'}, {'D'}], [{'C', 'D'}, {'B'}], [{'E', 'C'}, {'A'}]]
```

Another example:

a $\longrightarrow$ b, c	b $\longrightarrow$ a, c	c $\longrightarrow$ a, b

is this a valid canonical form?

a $\longrightarrow$ b	b $\longrightarrow$ a, c	c $\longrightarrow$ b	**Yes**

is this a valid canonical form?

a $\longrightarrow$ c	c $\longrightarrow$ b	b $\longrightarrow$ a	**Yes**

is this a valid canonical form?

a $\longrightarrow$ b	b $\longrightarrow$ c	c $\longrightarrow$ a	**Yes**

is this a valid canonical form?

a $\longrightarrow$ c	b $\longrightarrow$ c	c $\longrightarrow$ a, b	**Yes**

```
def isCanonicalCover(F, Fc):
    for alpha, beta in Fc:
        for A in alpha.union(beta):
            _isA_extra, X = isAttributeExtraneous(Fc, alpha, beta, A)
            if _isA_extra == True:
                return False
    R = set()
    for alpha, beta in F: R.update(alpha ▯ beta)
    for K in powerSet(R):
        if computeAttributeClosure(F,K) != computeAttributeClosure(Fc,K):
            return False
    return True
```

new idea

3rd NORMAL FORM: ALGORITHM

Functional Dependency

Given a set of functional dependencies F

the algorithm for computing a **3NF** decomposition:

compute3NFDecomposition (F)

F_c = canonical cover of F

OUT = $\emptyset$

for each dependency $\boxed{\alpha \longrightarrow \beta}$ in F_c:

if $\alpha \bigcup \beta$ not already part of some relation in OUT:

OUT += $\alpha \bigcup \beta$

Add a relation with a candidate key of all attributes (if required)

to ensure that the decomposition is **lossless**

Theorem: The following algorithm gives **3NF** with **no** lost dependencies

```
def computeCanonicalCover(F):
    ...
def compute3NFDecomposition(F):
    OUT = []
    F_c = computeCanonicalCover(F)
    for alpha, beta in F_c:
        if (  [ R for R in OUT if (alphabeta).issubset( R ) ] == list() ):
            OUT.append( alpha  beta )
r = [ {'A', 'B', 'C', 'D'}, {'A', 'B'}  ]
d = [
            [ {'A'}, {'B'} ],  #A->B
            [ {'C', 'B'}, {'A', 'D'} ]  #CB->AD
    ]
print( computeBCNFDecomposition(d, r) )
print( compute3NFDecomposition(d) )
```

```
BCNF:  [{'A', 'B'}, {'A', 'C', 'D'}]
3NF:  [{'A', 'B'}, {'A', 'C', 'D', 'B'}]
```

Theorem: The following algorithm gives **3NF** with **no** lost dependencies

Proof:

Given a set of dependencies F_c in canonical form

for each dependency $\alpha \longrightarrow \beta$ in F_c, we added the relation $\alpha \cup \beta$

no lost dependencies—added a relation for each dependency!

hard part: show that $\alpha \cup \beta$ is in 3NF

Have to show that it cannot happen that there is a table that is not in 3NF:

table $\alpha \bigcup \beta$ from $\boxed{\alpha \longrightarrow \beta}$ in F_c

with

$\gamma \longrightarrow a$ derivable from F_c

$\gamma \cup \{a\} \subseteq \alpha \cup \beta$

yet γ is not a superkey of $\alpha \cup \beta$

and a not in any candidate key of $\alpha \cup \beta$

$$\boxed{\alpha \longrightarrow \beta}$$

with

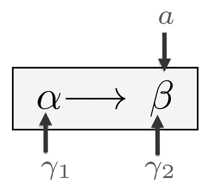
$\gamma \longrightarrow a$ derivable from F_c

$\gamma \cup \{a\} \subseteq \alpha \cup \beta$

yet γ is not a superkey of $\alpha \cup \beta$

and a not in any candidate key of $\alpha \cup \beta$

$\gamma = \gamma_1 \bigcup \gamma_2$



Key question: Did we derive $\gamma \longrightarrow a$ from F_c *using* $\alpha \longrightarrow \beta$?

yes $\Longrightarrow$ γ is a superkey in $\alpha \cup \beta$, since then all α is derivable from γ

no $\Longrightarrow$ a is redundant in $\alpha \longrightarrow \beta$, since knowing α, get γ_2 + knowing γ, get a

either way, we get a contradiction to our starting assumptions

$$\boxed{\alpha \longrightarrow \beta}$$

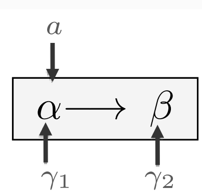
with

$\gamma \longrightarrow a$ derivable from F_c

$\gamma \cup \{a\} \subseteq \alpha \cup \beta$

yet γ is not a superkey of $\alpha \cup \beta$

and a not in any candidate key of $\alpha \cup \beta$



but then a is part of the candidate key α, so satisfies 3NF property

End of proof.

BCNF vs 3NF

In a perfect world, we want three things:

1. **No redundancy**: no non-trivial non-superkey dependency in any table
2. **Dependency preservation**: each dependency verifiable in some table
3. **Losslessness**: recover original table from JOINS of smaller tables

. . . but not always possible!

If we cannot have all three, then either:

3NF: satisfies **2.** and **3.**, but can have redundancy *or*

BCNF: satisfies **1.** and **3.**, but lose verifying some dependencies

Summary

1. Switched to a completely data dependencies point of view
2. Notion of functional dependency
3. Closure of functional dependencies
4. Closure of attributes
5. Boyce-Codd Normal Form (BCNF)

I gave you Python code for each algorithm

you should try examples by running code.

test with your own calculations and results.

6. Lossless decompositions, BCNF is lossless
7. Dependency preservation when splitting tables

9. Canonical covers

8. Extraneous attributes

10. 3NF: definition and algorithm


The End