CS2855 - Databases

8. Database Design: Normalisation Theory

Argyrios Deligkas
Department Of Computer Science

Normalisation Theory Contents

1. **Goals** of normalisation
2. Illustrative Examples
3. General **scheme** for normalising tables
4. **Basic notions** for normalization
   - Functional Dependencies
   - Closure of a set of FDs
   - Redundant attributes
   - Canonical Covers
5. Decomposition
6. From Functional Dependencies to **normal forms**:
   - 1^{st} Normal Form
   - BCNF
   - 3^{rd} Normal Form
7. Some **general** considerations

(**1**) Goals of Normalisation

- **Goal of normalisation of relational database design**: generate relation schemas that:
    - Store information <u>without unnecessary redundancy</u>;
    - Allows us to <u>retrieve information easily</u>.

- We introduce a **formal approach** to relational database design based on the notion of ***functional dependencies.***
- We then define ***normal forms*** in terms of functional dependencies.

(2) Illustrative Examples

Recall the Banking Schema

branch = (*branch_name*, *branch_city*, *assets*)

customer = (*customer_id*, *customer_name*, *customer_street*, *customer_city*)

loan = (*loan_number*, *amount*)

account = (*account_number*, *balance*)

employee = (*employee_id*. *employee_name*, *telephone_number*, *start_date*)

dependent_name = (*employee_id, dname*)

account_branch = (*account_number*, *branch_name*)

loan_branch = (*loan_number*, *branch_name*)

borrower = (*customer_id, loan_number*)

depositor = (*customer_id, account_number*)

cust_banker = (*customer_id, employee_id*, *type*)

works_for = (*worker_employee_id*, *manager_employee_id*)

payment = (*loan_number, payment_number*, *payment_date*, *payment_amount*)

savings_account = (*account_number*, *interest_rate*)

checking_account = (*account_number*, *overdraft_amount*)

Example 1: Combining Two Schemas (bad)

- Combine ***borrower*** and ***loan*** as follows:

  bor_loan = (customer_id, loan_number, amount)

- Result: **repetition** of information

  (L-100, 10000 in example below)

loan_number	amount
⋮	⋮
L-100	10000
⋮	⋮

loan

customer_id	loan_number
⋮	⋮
23-652	L-100
15-202	L-100
23-521	L-100
⋮	⋮

borrower

customer_id	loan_number	amount
⋮	⋮	⋮
23-652	L-100	10000
15-202	L-100	10000
23-521	L-100	10000
⋮	⋮	⋮

bor_loan

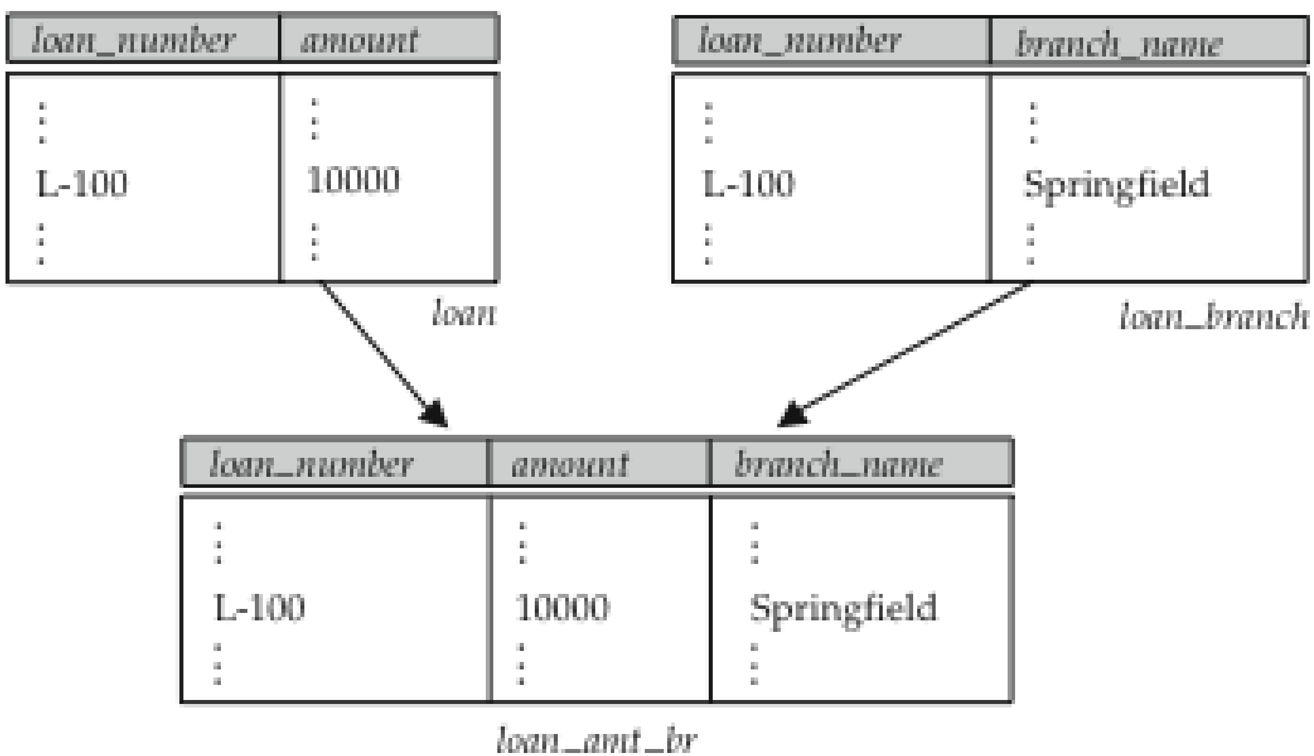
There are redundancies here: repeating "10000"!

Example 2: Combining Two Schemas (good)

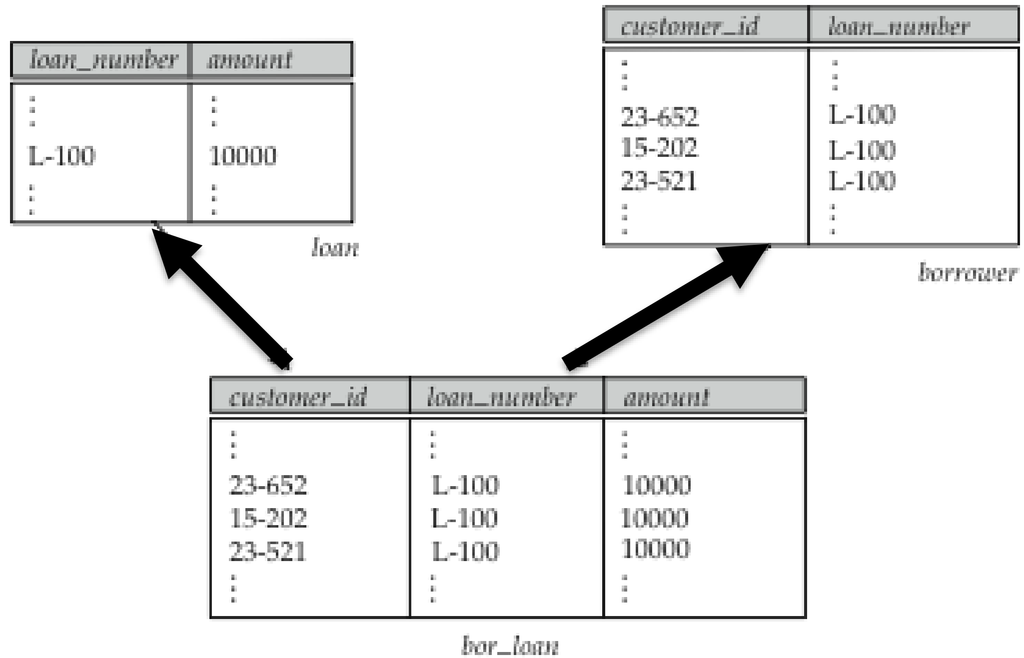
- Combining ***loan*** and ***loan_branch*** as follows:

  loan_amt_br = (*loan_number*, *amount*, *branch_name*)

- **No** repetitions here!



Example 3: Decomposing a Schema



- Suppose we had started with ***bor_loan.*** How would we know that we need to split it up (decompose), and to which two tables?
  - In *bor_loan*, because *loan_number* **is not a candidate key**, the amount of a loan may have to be repeated. This will indicate the need to decompose *bor_loan*.

Example 4: Lossy Decomposition (very bad)

- Suppose we decompose employee into:

employee1 = (employee_id, employee_name)

employee2 = (employee_name, telephone_number, start_date)

- Then we **lose** information: **cannot** reconstruct the original employee relation.
- We call it a **lossy decomposition**.

Because we didn't put the **key** *employee_id* in 2nd table we can't identify "Kim" anymore.

employee_id	*employee_name*	*telephone_number*	*start_date*
⋮			
123-45-6789	Kim	882-0000	1984-03-29
987-65-4321	Kim	869-9999	1981-01-16
⋮			

employee

employee_id	*employee_name*
⋮	
123-45-6789	Kim
987-65-4321	Kim
⋮	

employee_name	*telephone_number*	*start_date*
⋮		
Kim	882-0000	1984-03-29
Kim	869-9999	1981-01-16
⋮		

$\bowtie$

employee_id	*employee_name*	*telephone_number*	*start_date*
⋮			
123-45-6789	Kim	882-0000	1984-03-29
123-45-6789	Kim	869-9999	1981-01-16
987-65-4321	Kim	882-0000	1984-03-29
987-65-4321	Kim	869-9999	1981-01-16
⋮			

Example 4: Lossy Decomposition (very bad)

Is there a way to a **general method or theory** that will show us when and how to decompose a schema (to save on repetitions) without losing information?

(3) General Scheme for Normalising Tables

Goal: Devise a Method for the Following

- Decide whether a particular relation *R* is in a "**good**" form.
- In the case *R* is **not** in a "good" form, **decompose** it into a set of relations

  $\{R_1, R_2, ..., R_n\}$ such that

  - each relation is in a "**good"** form;
  - the decomposition is **not** a lossy decomposition.
- We will see such a method based on the notion of **functional dependencies**

Overall Database Design Process

We have assumed schema *R* **is given**

- *R* could have been generated when **converting** E-R **diagram** to a set of tables.
- *R* could have been a single relation containing ***all*** **attributes** that are of interest (called **universal relation**). Then normalisation could be used to break *R* into smaller relations.
- *R* could have been the result of some **ad hoc design** of relations, which we then test/convert to normal form.

(4) Basic Notions in Normalisation

Functional Dependencies

Functional Dependencies

- A **functional dependency** (**FD** for short) is a **generalisation** of the notion of a **key**.
- 1st view: **Constraints** on the set of all possible relation-instances.
- 2nd view: A constraint that you know hold in the **real world**.

Functional Dependencies

- A **func**… **ncy** (**FD** f… short) … **ion** of th… notion of a **key**.

I know some constraint holds in the real world; and I describe this constraint using FDs

I may want my table to comply with some constraint

- 1st view: **Constraints** on the set of all possible relation-instances.
- 2nd view: A constraint that you know hold in the **real world**.
- It requires that the value for a certain set of attributes **determines uniquely** the values for another set of attributes (hence a "functional" dependency).

Example (Functional Dependency)

customer_id	loan_number	amount
101	L-10	10000
104	L-11	10000
203	L-11	10000
97	L-05	13301

So **loan_number $\rightarrow$ amount**
(many to one)

But **amount $\not\rightarrow$ loan_number**
(not many to one)

Definition: Let R be a relation schema, and let $\alpha \subseteq R$ *and* $\beta \subseteq R$ be subsets of attributes in R. The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relations r(R), whenever any two tuples t_1 and t_2 of r **agree** on the attributes α, they also agree on the attributes β.

That is: $t_1[\alpha] = t_2[\alpha] \;\Rightarrow\; t_1[\beta] = t_2[\beta]$

Example: Consider r (A,B) with the instance of r as in figure. On this instance, $A \rightarrow B$ does **NOT** hold but $B \rightarrow A$ does hold.

A	B
1	4
1	5
3	7

- K is a **superkey** for relation schema R precisely when $K \rightarrow R$

E.g., cust_id, first_name $\rightarrow$ cust_id, first_name, last_name, date_of_birth

- K is a **candidate key** for R precisely when both $K \rightarrow R$ and for no $\alpha \subset K$, $\alpha \rightarrow R$

E.g., cust_id $\rightarrow$ cust_id, first_name, last_name, date_of_birth

R

K

bor_loan

customer_id	first_name	last_name	DOB
97	John	Phillips	1.1.1990
101	Alice	Hayes	1.1.1990
104	Bob	Gray	2.4.1985
203	John	Pink	3.6.1973
297	Mary	Green	23.6.1969

- But **FD**s allow us to express constraints on the values of attributes (that **cannot** be expressed using superkeys).
- In fact, it is a **generalisation** of superkeys.

Example:

loan_number $\rightarrow$ amount,

But loan_number is **not** a key!

customer_id	loan_number	amount
101	L-10	10000
104	L-11	10000
203	L-11	10000
97	L-05	13301

Don't confuse: saying a table-**schema** satisfies a FD is different from saying a table-**instance** satisfies a FD

- Given a set *F* of functional dependencies, a table-instance can either **satisfy** *F* or **not satisfy** *F*.

Example: F = {Street_name → first_name}

Question: Does this table-instance satisfy F ?

First_Name	Street_name	Phone
John	Liverpool-street	87094829
Alice	Regent-Street	94379439
Bob	Kennington-lane	11324242
Sarah	Church-road	9992828

- We say that a table-schema *R* **satisfies** a set of FDs *F* if all **potential real-world instances** relations on *R* satisfy *F.*

A functional dependency is called **trivial** if it is satisfied by **all possible table-instances** (not just the real-world instances in the enterprise, but *all* possible values over the domains of attributes);

Example:

customer_name, loan_number $\rightarrow$ *customer_name*

customer_name $\rightarrow$ *customer_name*

Formally: **$\alpha \rightarrow \beta$ is trivial if and only if $\beta \subseteq \alpha$**

Closure of a set of FDs

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.

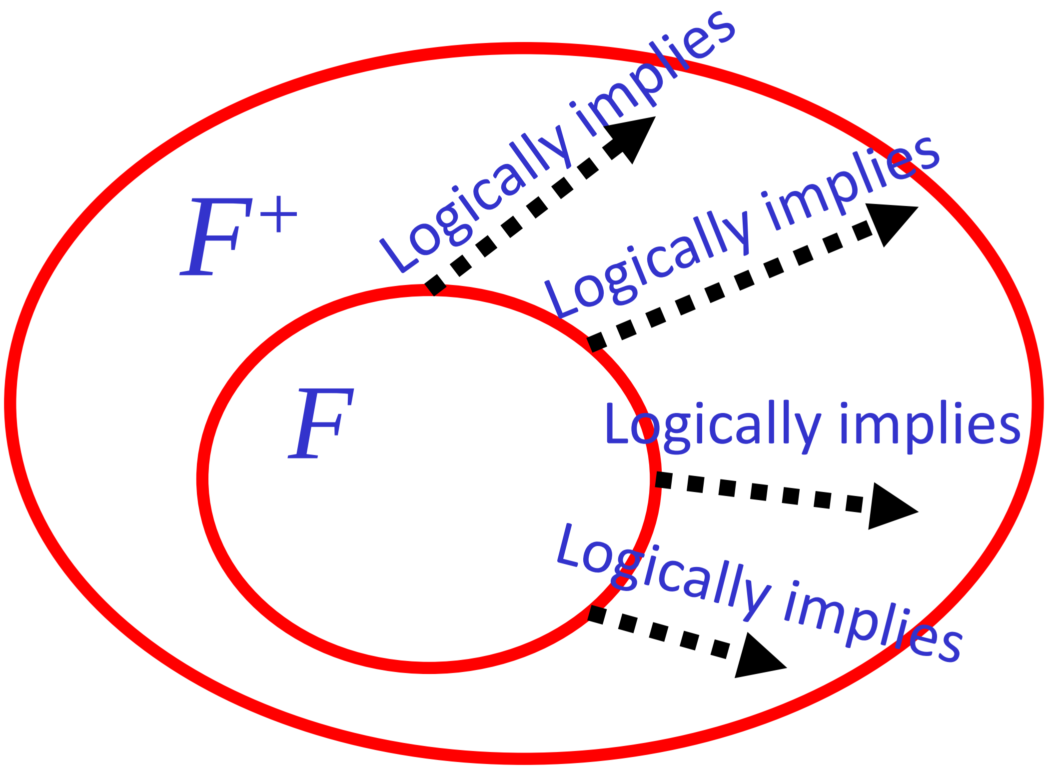
Example: If $A \rightarrow B$ and $B \rightarrow C$, then it is also true that $A \rightarrow C$

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

- We denote the closure of F by F^+.
- F^+ is a superset of F.

- We now consider a general way to tell which functional dependencies are **<u>logically implied</u>** by a given set of functional dependencies.
- Namely, find **the closure of** F.

Closure of a set of FDs Diagram



Closure of a Set of FDs

Given a set of F of FDs we can find all of F^+ by applying

Armstrong's Axioms:

if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)** i.e. trivial FDs

if $\alpha \rightarrow \beta$, then $\gamma\, \alpha \rightarrow \gamma\, \beta$ **(augmentation)**

if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**

These rules are:

- **Sound**: they generate **only** functional dependencies that actually hold
- **Complete**: they generate **all** functional dependencies that are logically implied by *F*

Example:

R

Student_id	Academic_year	Course	Degree
101	201617	Algo	BSc
204	201617	Databases	MSci
101	201516	Algo	BSc

Assume the following FD holds:

Student_id, Academic_year → Degree

By **augmentation rule** we conclude the following FD also holds:

Student_id, Academic_year, Course → Degree, Course

We can find this way the **superkey** of R:

Since

Student_id, Academic_year, Course → Degree, Course

by applying the augmentation rule:

$$\text{Student\_id, Academic\_year, Course} \rightarrow \underbrace{\text{Student\_id, Academic\_year, Degree, Course}}_{R}$$

Example

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

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

Armstrong's Axioms:

if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(refl.)**
if $\alpha \rightarrow \beta$, then $\gamma\, \alpha \rightarrow \gamma\, \beta$ **(augm.)**
if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(trans.)**

- 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

Algorithm for Computing F^+

if $\alpha \rightarrow \beta$, then add $\gamma\alpha \rightarrow \gamma\beta$ for each possible γ

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

If $\alpha \rightarrow \beta$ then add $(\alpha \cup \beta) \rightarrow \beta$

Let $F^+ := F$

repeat

for each functional dependency f in F^+

- apply augmentation and reflexivity rules on f
- add the resulting functional dependencies to F^+

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

if f_1 and f_2 can be combined using transitivity

then add the resulting functional dependency to F^+

until F^+ does not change any further

Each iteration of the repeat loop of the procedure, except the last iteration, **adds at least one functional** dependency to F^+. Thus, the procedure is guaranteed to **terminate**.
(**Note**: the amount of FDs on a finite set of attributes is finite.)

Compu

if $\alpha \rightarrow \beta$, then add $\gamma\alpha \rightarrow \gamma\beta$ for each possible γ

NB. This is called a "greedy algorithm" in Algorithms.

If $\alpha \rightarrow \beta$ then add $(\alpha \cup \beta) \rightarrow \beta$

Let $F^+ := F$

repeat

for each functional dependency f in F^+

- apply augmentation and reflexivity rules on f
- add the resulting functional dependencies to F^+

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

if f_1 and f_2 can be combined using transitivity

then add the resulting functional dependency to F^+

until F^+ does not change any further

We can further simplify manual computation of F^+ by using the following additional rules.

If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, **then** $\alpha \rightarrow \beta\gamma$ holds (**union**)

If $\alpha \rightarrow \beta\gamma$ holds, **then** $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ hold (**decomposition**)

If $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$ hold, **then** $\alpha\gamma \rightarrow \delta$ holds (**pseudotransitivity**)

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

Redundant Attributes

An attribute of a functional dependency is **redundant** if it can be removed **without changing the closure** of the set of functional dependencies.

Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$

- B is **redundant** in $AB \rightarrow C$ because dropping it will **not** change the closure of F :

  $\{A \rightarrow C, AB \rightarrow C\}^+ = \{A \rightarrow C, A \rightarrow C\}^+ = \{A \rightarrow C\}^+$

- **Proof:**
  - $\{A \rightarrow C, AB \rightarrow C\}$ **logically implies** $\{A \rightarrow C\}$ (i.e. the result of dropping B from $AB \rightarrow C$). This is trivial.
  - $\{A \rightarrow C\}$ **logically implies** $\{A \rightarrow C, AB \rightarrow C\}$ because $A \rightarrow C$, logically implies $AB \rightarrow C$ (why?).

Example: Consider $F = \{A \rightarrow C, AB \rightarrow CD\}$

- C is redundant in CD in the FD $AB \rightarrow CD$ since:
  - $\{A \rightarrow C, AB \rightarrow D\}$ logically implies $\{A \rightarrow C, AB \rightarrow CD\}$

Redundant Attributes

Formally (this is equivalent to the definition in the example in previous slide):

Consider a set $F = \{F_1, \ldots, F_m, A\alpha \rightarrow \beta\}$ of FDs.

- Case Left: Attribute A is **redundant** in $A\alpha \rightarrow \beta$ **with respect to** F if F logically implies $\alpha \rightarrow \beta$

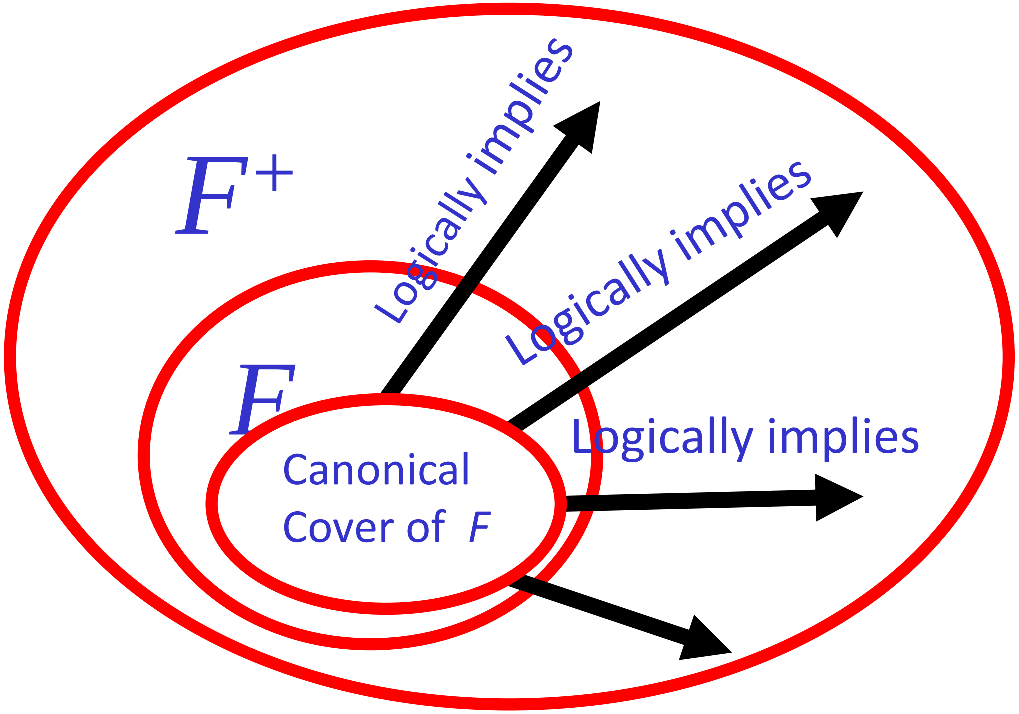
Consider a set $F = \{F_1, \ldots, F_m, \alpha \rightarrow B\beta\}$

- Case Right: Attribute B is **redundant in** β **with respect to** F, if the set $\{F_1, \ldots, F_m, \alpha \rightarrow \beta\}$ logically implies $\alpha \rightarrow B\beta$ (and hence it implies F).

Recall: You can check that h is **logically implied** by F, by checking that h is contained in F^+. E.g., simply show a derivation of h by **Armstrong's axioms** from F.

Canonical Cover

Canonical Cover Diagram



- **Canonical cover** of a set *F of* FD*s* is a "minimal" set of functional dependencies equivalent to F
- Formally, canonical cover of F is a set of FDs that
  - has no unnecessary **FDs**; and
  - has no redundant **attributes** in FDs; and
  - each left side of functional dependency is unique.

Canonical Cover

1. **Unnecessary** FDs: some functional dependencies may have that can be inferred from the others

- For example: a **FD** $A \rightarrow C$ is redundant in:

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

2. **Attributes** in a FD may be redundant

E.g.: on right hand side: $\{A \rightarrow B,\ \ B \rightarrow C,\ \ A \rightarrow CD\}$ can be simplified to

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

E.g.: on left hand side: $\{\mathrm{A} \rightarrow B,\ \ B \rightarrow C,\ \ AC \rightarrow D\}$ can be simplified to

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

Repeating the definition: A **canonical cover** for F is a set of dependencies F_c such that

1) *F logically implies all dependencies in F_c, and*
2) *F_c logically implies all dependencies in F, and*
3) No functional dependency in F_c contains a redundant attribute, and
4) Each left side of functional dependency in F_c is unique.

F and F_c are logically equivalent

To compute a canonical cover for F:

$F_c = F$;
repeat
  Use the union rule to replace any dependencies in F_c
    $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$ // cond 4
  Find a functional dependency $\alpha \rightarrow \beta$ in F_c with a
    redundant attribute either in α or in β
  If a redundant attribute is found, delete it from $\alpha \rightarrow \beta$
until F_c does not change

Note that the union rule may become applicable after some redundant attributes have been deleted, so it has to be re-applied.

Repeating the definition: A **canon**
dependencies F_c such that

1) *F* logically implies all dependencies in F_c, and
2) F_c logically implies all dependencies in *F*, and

F and F_c are logically equivalent

3) No functional dependency in F_c contains a redundant attribute, and
4) 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
    $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$ // cond 4
  Find a functional dependency $\alpha \rightarrow \beta$ in F_c with a
    redundant attribute either in α or in β
  If a redundant attribute is found, delete it from $\alpha \rightarrow \beta$
until F_c does not change

Note that redundant attributes can be computed by e.g. comparing the closures of a set of FDs (closures can be computed by the algorithm we've seen before)

Here we didn't delete redundant FDs (this can be done greedily: delete them until there's no redundant FD left).

Example

$R = (A, B, C)$
$F = \{A \to BC$
$B \to C$
$A \to B$
$AB \to C\}$

$F_c = F;$
repeat
Use the union rule to replace any dependencies in F_c
$\alpha_1 \to \beta_1$ and $\alpha_1 \to \beta_2$ with $\alpha_1 \to \beta_1 \beta_2$
Find a functional dependency $\alpha \to \beta$ in F_c with a redundant attribute either in α or in β
If a redundant attribute is found, delete it from $\alpha \to \beta$
until F_c does not change

Combine $A \to BC$ and $A \to B$ into $A \to BC$ (union rule)

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

A is redundant in $AB \to C$

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

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

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

C is redundant in $A \to BC$

Check if $\{A \to B, B \to C\}$ logically implies $\{A \to BC, B \to C\}$

Yes: using transitivity and then union on $A \to B$ *and* $B \to$ C.

The canonical cover is: $\{A \to B,$
$B \to C\}$

Now that we understand better functional dependencies we shall **apply** them to design good relations!

(5) Decomposition

Decomposition

- R is a relation-scheme R=(A,B,C,D,E)
- **If** $R_1 \subseteq R$ and $R_2 \subseteq R$ and $R_1 \cup R_2 = R$,

Then R_1,R_2 is a **decomposition** of R.

(Can be generalized to $R_1,\ldots,R_n$, decomposing further R_1 or R_2)

Goal

- We want to **avoid bad designs**, like repetition of information in a table (as we’ve seen before)
- We can do this by **decomposing** the relations into smaller ones
- But we **don’t** want to **loose information** in this process.
- Recall…

Recall: A Lossy Decomposition

Here, in the decomposition, we **lost information** (we cannot recover the original employee relation) and so, this is a **lossy decomposition**.

employee_id	*employee_name*	*telephone_number*	*start_date*
⋮			
123-45-6789	Kim	882-0000	1984-03-29
987-65-4321	Kim	869-9999	1981-01-16
⋮			

employee

employee_id	*employee_name*
⋮	
123-45-6789	Kim
987-65-4321	Kim
⋮	

employee_name	*telephone_number*	*start_date*
⋮		
Kim	882-0000	1984-03-29
Kim	869-9999	1981-01-16
⋮		

$\bowtie$

employee_id	*employee_name*	*telephone_number*	*start_date*
⋮			
123-45-6789	Kim	882-0000	1984-03-29
123-45-6789	Kim	869-9999	1981-01-16
987-65-4321	Kim	882-0000	1984-03-29
987-65-4321	Kim	869-9999	1981-01-16
⋮			

We must avoid a lossy decomposition.
I.e., we must have only **lossless decompositions**.

More formally: Lossless Decomposition

R: relation schema

F: set of **FDs** on R

R_1 and R_2: a **decomposition** of R.

Recall that r(R) denotes a relation with schema R.

Definition: The decomposition is a **lossless decomposition** if for all possible instances of r(R) that satisfy all the functional dependencies in *F*:

$$\prod_{R_1}(r) \bowtie \prod_{R_2}(r) = r$$

- In other words, if we project r onto R_1 and R_2 and then compute the natural join of the projection results, we get back exactly r.

Example: Lossless Decomposition

loan_number	amount
⋮	⋮
L-100	10000
⋮	⋮

loan

customer_id	loan_number
⋮	⋮
23-652	L-100
15-202	L-100
23-521	L-100
⋮	⋮

borrower

r

customer_id	loan_number	amount
⋮	⋮	⋮
23-652	L-100	10000
15-202	L-100	10000
23-521	L-100	10000
⋮	⋮	⋮

bor_loan

Loan=(loan_number, amount)

Borrower= (customer_id, loan_number)

$$\prod_{loan\_number, amount}(r) \bowtie \prod_{customer\_id, loan\_number}(r) = r$$

- Note: Loan ∩ Borrower = loan_number, which is a (super) **key** of loan.

How to check if a decomposition is lossless

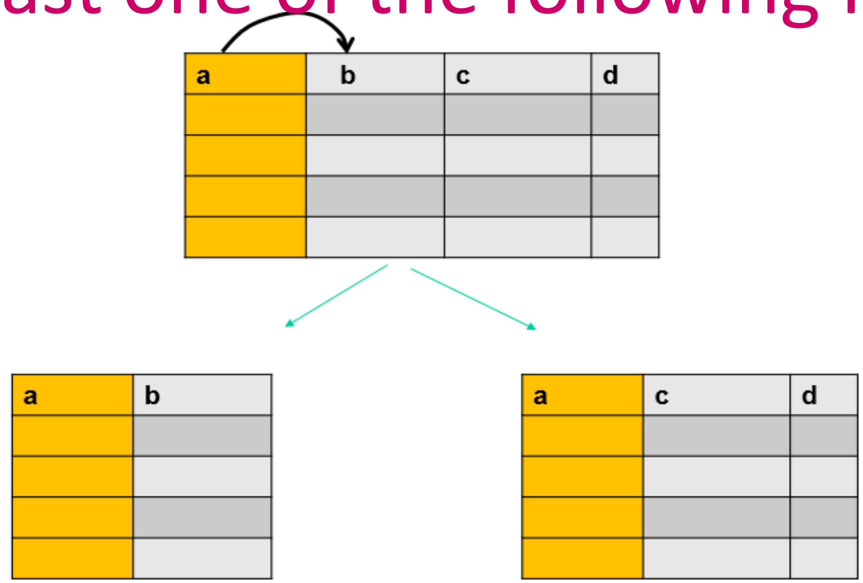
R: relational schema

F: set of **FDs** on R

R_1 and R_2: a **decomposition** of R.

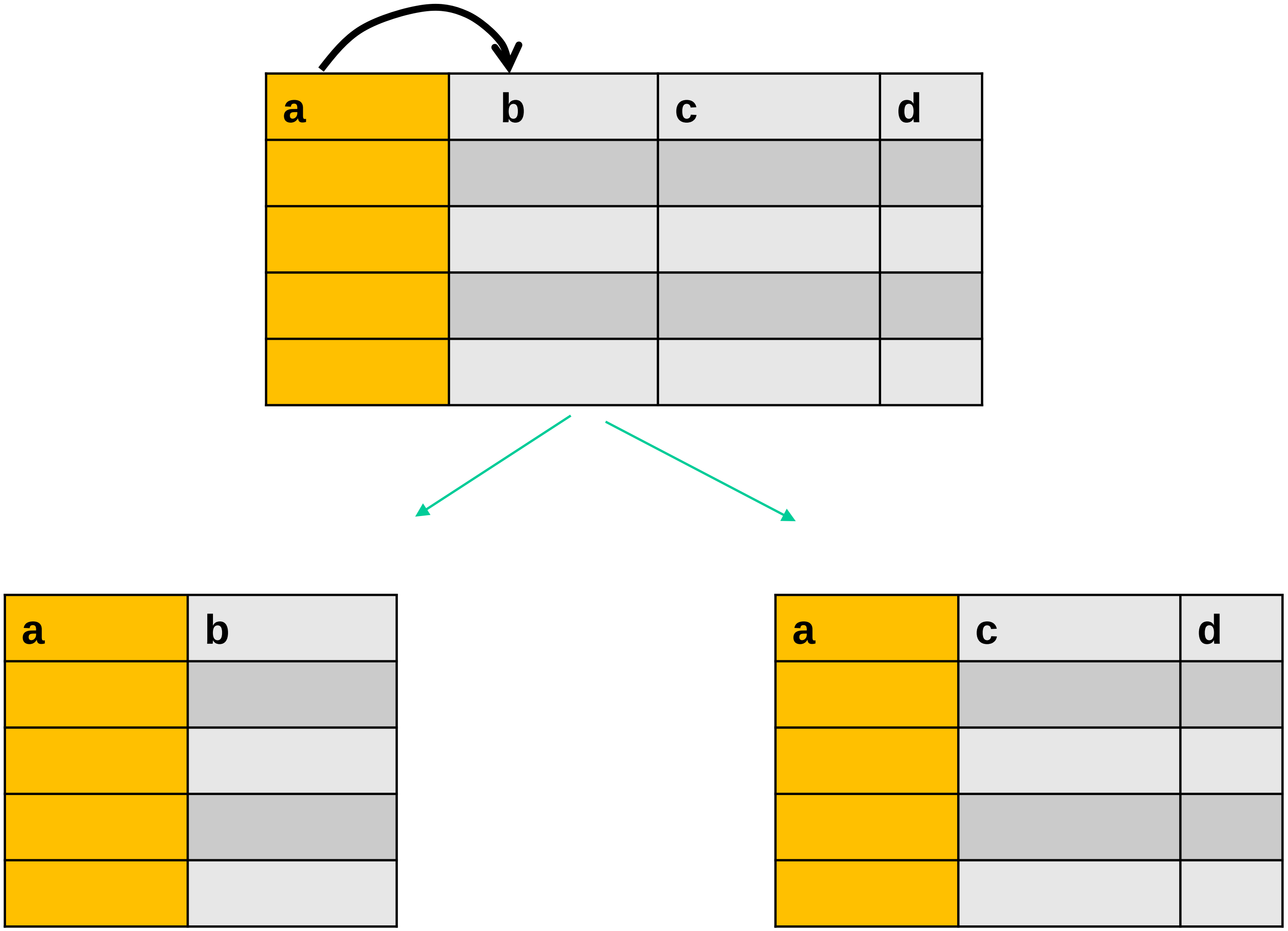
<u>Criterion</u>: The decomposition R_1, R_2 is a **lossless decomposition** if at least one of the following FDs is in $\boldsymbol{F^+}$:

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



In other words, if the intersection of R_1 and R_2 is a **superkey** of either R_1 or R_2.

Illustration of lossless decomposition



Example

F= {*branch-name* → *branch-city assets*,
loan-number → *amount branch-name*}

Goal: Decompose

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

- Decompose *Lending-schema* into two schemas:

Branch-schema = (branch-name, branch-city, assets)

Loan-info-schema = (branch-name, customer-name, loan-number, amount)

- By augmentation rule *branch-name* → *branch-city assets* implies *branch-name* → *branch-name branch-city assets*
- Since *branch-schema* ∩ *Loan-info-schema = {branch-name}*, it follows that our initial decomposition is a **lossless decomposition**!

Example

F= {*branch-name* → *branch-city assets*,
loan-number → *amount branch-name*}

Goal: Decompose

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

- Decompose *Lending-schema* into two schemas:

Branch-schema = (branch-name, branch-city, assets)

Loan-info-schema = (branch-name, customer-name, loan-number, amount)

- Next, we decompose *Loan-info-schema* into
  Loan-schema = (loan-number, branch-name, amount)
  Borrower-schema = (customer-name, loan-number)
- This step results in a lossless decomposition, since *loan-number* is a common attribute
  and *loan-number* → *amount branch-name*.

(6) Normal Forms

1. First normal form
2. Boyce-Codd normal form
3. Third normal form

Main goal of normalisation: Simply **avoid** tables with functional dependencies from ***non-key*** attributes to other attributes. (This helps to avoid accumulating redundant information!)

loan_number	*amount*
⋮	⋮
L-100	10000
⋮	⋮

loan

customer_id	*loan_number*
⋮	⋮
23-652	L-100
15-202	L-100
23-521	L-100
⋮	⋮

borrower

customer_id	*loan_number*	*amount*
⋮	⋮	⋮
23-652	L-100	10000
15-202	L-100	10000
23-521	L-100	10000
⋮	⋮	⋮

bor_loan

❶ First (simple) Normal Form

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

Examples of **non-atomic** domains:
set of names, **composite** attributes

Non-atomic values **complicate** storage and encourage **redundant** storage of data

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

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

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

Example: Strings would normally be considered indivisible.

But 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.

❷ Boyce-Codd Normal Form (**BCNF**)

i.e. different from $\alpha \rightarrow \beta$ where $\beta \subseteq \alpha$

A relation schema *R* is in **BCNF** with respect to a set *F* of functional dependencies if:

- For every **non-trivial** FD $\alpha \rightarrow \beta$ in F^+ with $\alpha \subseteq R$ and $\beta \subseteq R$, the following holds:

  α is a **superkey** of *R* (*i.e.,* $\alpha \rightarrow R$)

Example: schema *not* in BCNF:

bor_loan = (*customer_id, loan_number, amount*)

because *loan_number* → *amount* holds on *bor_loan* but *loan_number* is not a superkey

- Note indeed it leads to redundant repetitions: amount is determined by L-100, but because L-100 doesn't determine the whole row we have different rows with the same [L-100, 10000].

customer_id	*loan_number*	*amount*
⋮	⋮	⋮
23-652	L-100	10000
15-202	L-100	10000
23-521	L-100	10000
⋮	⋮	⋮

bor_loan

Decompose a Schema into BCNF

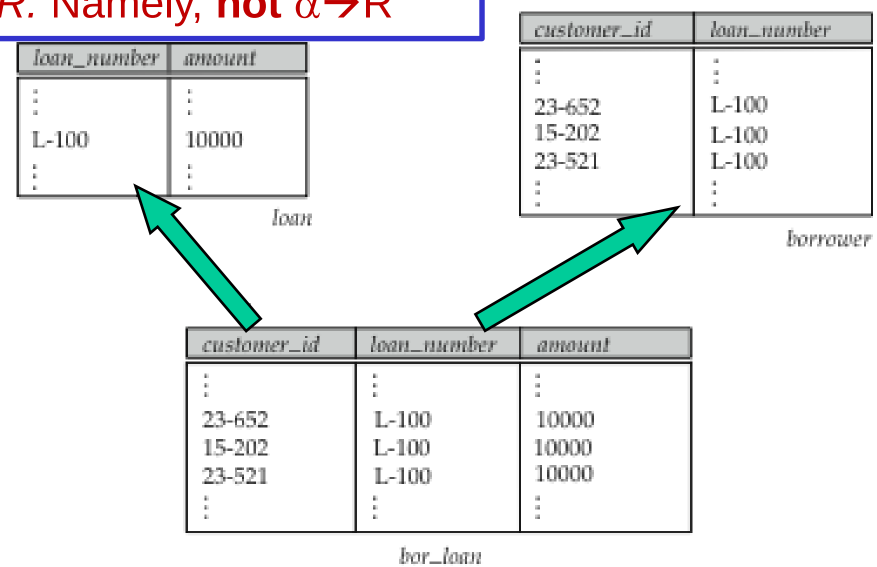
- Suppose we have a schema R with $\alpha \subseteq R$ and $\beta \subseteq R$. Assume a **non-trivial** dependency $\alpha \rightarrow \beta$ in F^+ causes a **violation of BCNF.** Then we replace R with two schemas:

i.e., α is not a superkey of R. Namely, **not** $\alpha \rightarrow R$

$$\alpha \cup \beta$$

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

- We **continue** decomposing the tables until we **don't** have such violations (on **other** FDs).



Example: α = *loan_number* , β = *amount, and we have* $\alpha \rightarrow \beta$.

Then *bor_loan* is decomposed to

- $\alpha \cup \beta$ = (*loan_number, amount*); and
- $\alpha \cup (R - \beta)$ = (*customer_id, loan_number*)

Decompose a Schema into BCNF

Important note:
When asked to decompose a relation into BCNF you must use this algorithm!

Reason:
- You may find many possible "decompositions" that are in BCNF on the same set of attributes.
- But if you don't follow this algorithm, them may not be lossless-join decompositions.
- I.e., you may lose information in this decomposition.
- Losing information must be avoided in all cost!

Dependency Preservation

Bank schemas in **BCNF** :

(customer_id, employee_id, type)

(employee_id, branch_name)

The constraint: "a customer may have at most one personal banker at a given branch" can be expressed as:

(*) customer_id, branch_name → employee_id

But in our BCNF design there is no schema that includes all the attributes appearing in this functional dependency.

➔ **not <u>dependency preserving</u>** (i.e., checking the dependencies on each relation, would not be enough to ensure (*) holds).

BCNF and Dependency Preservation

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

Namely, for each relation R in decomposition, you check only the FDs $\alpha \rightarrow \beta$ in F^+ with $\alpha \subseteq R$ and $\beta \subseteq R$.

- A decomposition is ***dependency preserving*** with respect to set of FDs F if: it is sufficient to test only dependencies on **each individual relation,** in order to ensure that *all* functional dependencies hold.

❸ Third Normal Form (**3NF**)

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

Third Normal Form (**3NF**)

A relation schema *R* **is in 3NF with respect to a set *F*** of functional dependencies if, for every **non-trivial** FD $\alpha \rightarrow \beta$ with $\alpha \subseteq R$ and $\beta \subseteq R$, in F^+, **<u>at least one</u>** of the following holds:

- α is a **superkey** for *R*; or
- Each attribute *A* in $\beta - \alpha$ is **contained in a candidate key** for *R*.

Note: If a relation is in BCNF it is in 3NF (since in BCNF the first condition of 3NF above must hold).

> Second condition is a "minimal" relaxation of BCNF to make it possible to **ensure dependency preservation.**

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; and
  - all 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; and
  - it may not be possible to preserve dependencies.

(7) General Discussion

Design Goals in Relational DBs

Goal for a relational database design is:

- BCNF
- Lossless join
- Dependency preservation

If we cannot achieve this, we accept one of

1. Lack of dependency preservation
2. Redundancy due to use of 3NF

- *SQL does not provide a direct way of specifying functional dependencies other than superkeys.*
- *Can specify functional dependencies using assertions, but they are expensive to test*

ER Model and Normalisation

When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalisation.

However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity

Denormalisation for Performance

May want to use non-normalised schemas for **performance** sake.

For example, displaying *customer_name* along with *account_number* and *balance* requires join of *account* with *depositor*

Alternative 1: Use denormalised relation containing attributes of *account* as well as *depositor* with all above attributes

- faster lookup
- extra space and extra execution time for updates
- extra coding work for programmer and possibility of error in extra code

Alternative 2: use a **view** defined as

$$\text{account} \bowtie \text{depositor}$$

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

What you need to know

- Definition of FDs
- Know the difference between a FD holding on an instance and an FD holding on a relation schema
- Armstrong's rules (axioms); how to apply them to get the closure F^+ of a set of FDs.
- Canonical cover: definition (and definition of redundant attributes, and how to find them).
- How to decompose a relation into a lossless-join decomposition (with respect to a set F of FDs).
- How to decompose a table into a BCNF
- How to decompose a table into a 3NF
- The definitions of lossless-join decomposition and BCNF, 3NF and 1NF.

End