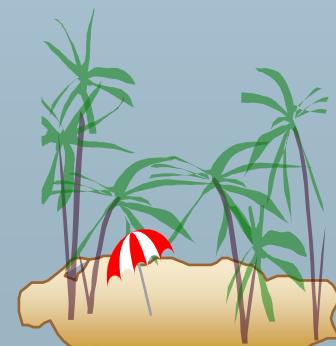


# **Chapter 7: Relational Database Design**



# Chapter 7: Relational Database Design

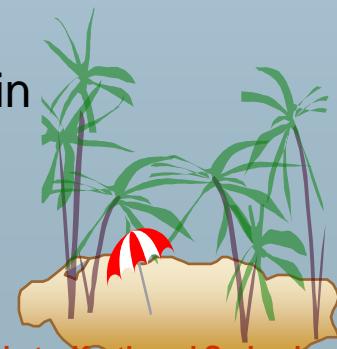
- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce-Codd Normal Form
- Third Normal Form
- Multivalued Dependencies and Fourth Normal Form
- Overall Database Design Process





# First Normal Form

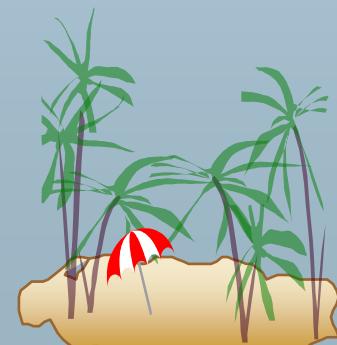
- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in **first normal form** if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - E.g. Set of accounts stored with each customer, and set of owners stored with each account
  - We assume all relations are in first normal form (revisit this in Chapter 9 on Object Relational Databases)





# First Normal Form (Contd.)

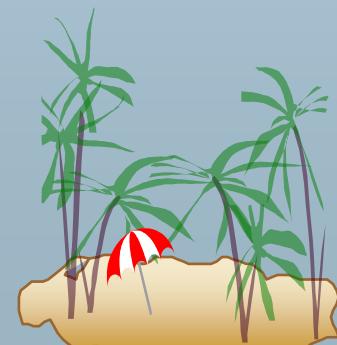
- Atomicity is actually a property of how the elements of the domain are used.
  - E.g. Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form CS0012 or EE1127
  - If the first two characters are extracted to find the department, 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.





# Pitfalls in Relational Database Design

- Relational database design requires that we find a “good” collection of relation schemas. A bad design may lead to
  - Repetition of Information.
  - Inability to represent certain information.
- Design Goals:
  - Avoid redundant data
  - Ensure that relationships among attributes are represented
  - Facilitate the checking of updates for violation of database integrity constraints.





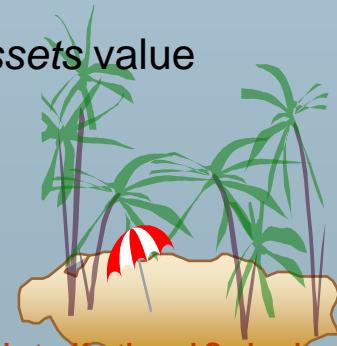
# Example

- Consider the relation schema:

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

branch-name	branch-city	assets	customer-name	loan-number	amount
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500

- Redundancy:
  - Data for *branch-name*, *branch-city*, *assets* are repeated for each loan that a branch makes
  - Wastes space
  - Complicates updating, introducing possibility of inconsistency of *assets* value
- Null values
  - Cannot store information about a branch if no loans exist
  - Can use null values, but they are difficult to handle.





# Decomposition

- Decompose the relation schema *Lending-schema* into:

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

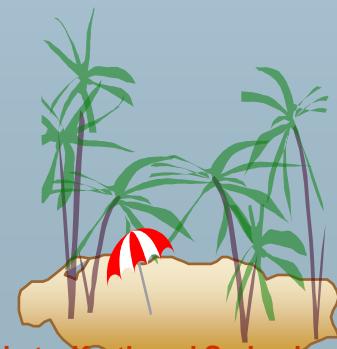
*Loan-info-schema* = (*customer-name*, *loan-number*,  
*branch-name*, *amount*)

- All attributes of an original schema ( $R$ ) must appear in the decomposition ( $R_1, R_2$ ):

$$R = R_1 \cup R_2$$

- Lossless-join decomposition.  
For all possible relations  $r$  on schema  $R$

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





# Example of Non Lossless-Join Decomposition

□ Decomposition of  $R = (A, B)$

$$R_1 = (A) \quad R_2 = (B)$$

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1

r

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

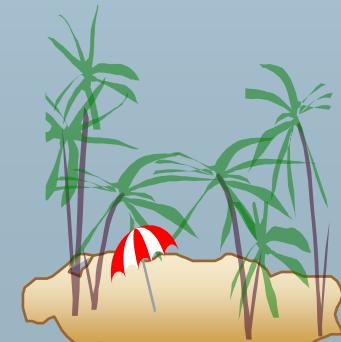
A
$\alpha$
$\beta$

$$\Pi_A(r)$$

B
1
2

$$\Pi_B(r)$$

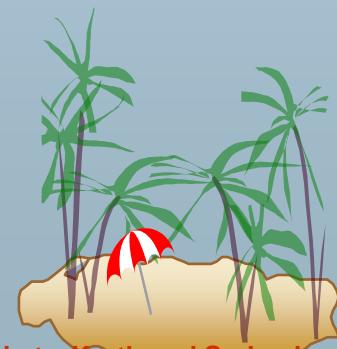
A	B
$\alpha$	1
$\alpha$	2
$\beta$	1
$\beta$	2





# Goal — Devise a Theory for the Following

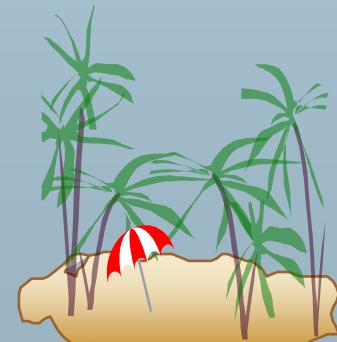
- Decide whether a particular relation  $R$  is in “good” form.
- In the case that a relation  $R$  is not in “good” form, decompose it into a set of relations  $\{R_1, R_2, \dots, R_n\}$  such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- Our theory is based on:
  - functional dependencies
  - multivalued dependencies





# Functional Dependencies

- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key*.





# Functional Dependencies (Cont.)

- Let  $R$  be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The functional dependency

$$\alpha \rightarrow \beta$$

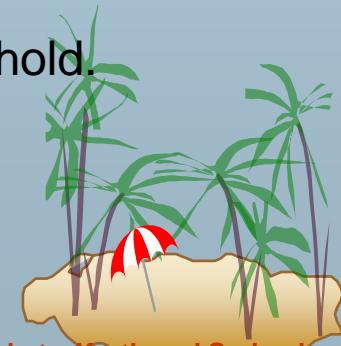
holds on  $R$  if and only if for any legal relations  $r(R)$ , whenever any two tuples  $t_1$  and  $t_2$  of  $r$  agree on the attributes  $\alpha$ , they also agree on the attributes  $\beta$ . That is,

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

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

1	4
1	5
3	7

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





# Functional Dependencies (Cont.)

- $K$  is a superkey for relation schema  $R$  if and only if  $K \rightarrow R$
- $K$  is a candidate key for  $R$  if and only if
  - $K \rightarrow R$ , and
  - for no  $\alpha \subset K$ ,  $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

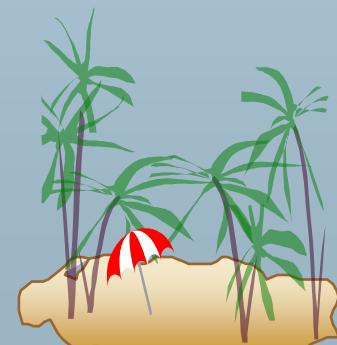
*Loan-info-schema = (customer-name, loan-number,  
branch-name, amount).*

We expect this set of functional dependencies to hold:

*loan-number  $\rightarrow$  amount  
loan-number  $\rightarrow$  branch-name*

but would not expect the following to hold:

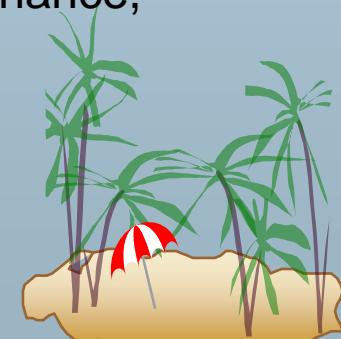
*loan-number  $\rightarrow$  customer-name*





# Use of Functional Dependencies

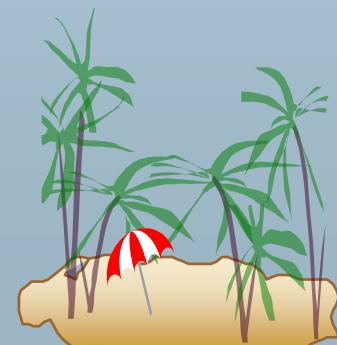
- We use functional dependencies to:
  - test relations to see if they are legal under a given set of functional dependencies.
    - If a relation  $r$  is legal under a set  $F$  of functional dependencies, we say that  $r$  **satisfies**  $F$ .
  - specify constraints on the set of legal relations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





# Functional Dependencies (Cont.)

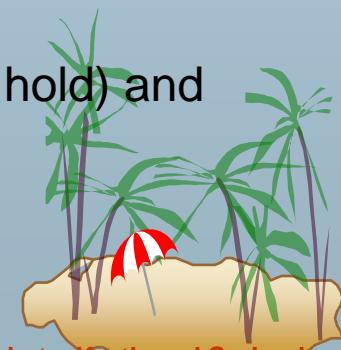
- A functional dependency is **trivial** if it is satisfied by all instances of a relation
  - *E.g.*
    - $\text{customer-name}, \text{loan-number} \rightarrow \text{customer-name}$
    - $\text{customer-name} \rightarrow \text{customer-name}$
  - In general,  $\alpha \rightarrow \beta$  is trivial if  $\beta \subseteq \alpha$





# Closure of a Set of Functional Dependencies

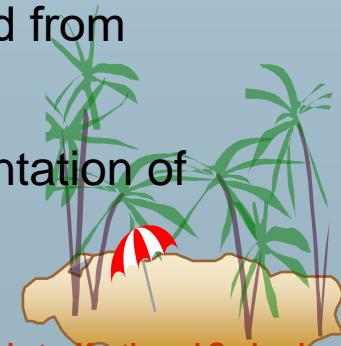
- Given a set  $F$  set of functional dependencies, there are certain other functional dependencies that are logically implied by  $F$ .
  - E.g. If  $A \rightarrow B$  and  $B \rightarrow C$ , then we can infer that  $A \rightarrow C$
- The set of all functional dependencies logically implied by  $F$  is the *closure* of  $F$ .
- We denote the *closure* of  $F$  by  $F^+$ .
- We can find all of  $F^+$  by applying Armstrong's Axioms:
  - if  $\beta \subseteq \alpha$ , then  $\alpha \rightarrow \beta$  **(reflexivity)**
  - 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** (generate only functional dependencies that actually hold) and
  - **complete** (generate all functional dependencies that hold).





# Example

- $R = (A, B, C, G, H, I)$   
 $F = \{ A \rightarrow B$   
     $A \rightarrow C$   
     $CG \rightarrow H$   
     $CG \rightarrow I$   
     $B \rightarrow H \}$
- some members of  $F^+$ 
  - $A \rightarrow H$ 
    - by transitivity from  $A \rightarrow B$  and  $B \rightarrow H$
  - $AG \rightarrow I$ 
    - by augmenting  $A \rightarrow C$  with  $G$ , to get  $AG \rightarrow CG$   
        and then transitivity with  $CG \rightarrow I$
  - $CG \rightarrow HI$ 
    - from  $CG \rightarrow H$  and  $CG \rightarrow I$ : “union rule” can be inferred from
      - definition of functional dependencies, or
      - Augmentation of  $CG \rightarrow I$  to infer  $CG \rightarrow CGI$ , augmentation of  $CG \rightarrow H$  to infer  $CGI \rightarrow HI$ , and then transitivity





# Procedure for Computing $F^+$

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

$$F^+ = F$$

**repeat**

**for each** functional dependency  $f$  in  $F^+$

    apply reflexivity and augmentation rules on  $f$

    add the resulting functional dependencies to  $F^+$

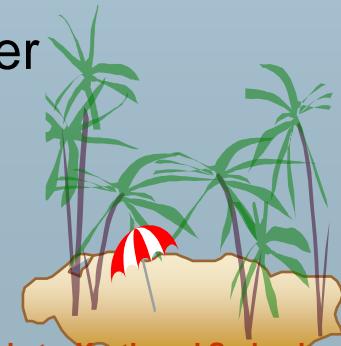
**for each** pair of functional dependencies  $f_1$  and  $f_2$  in  $F^+$

**if**  $f_1$  and  $f_2$  can be combined using transitivity

**then** add the resulting functional dependency to  $F^+$

**until**  $F^+$  does not change any further

NOTE: We will see an alternative procedure for this task later

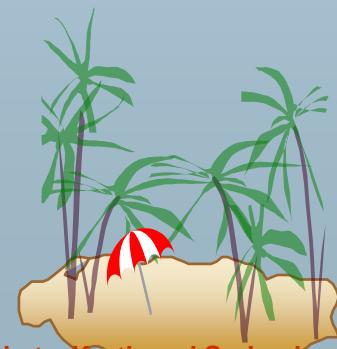




# Closure of Functional Dependencies (Cont.)

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

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





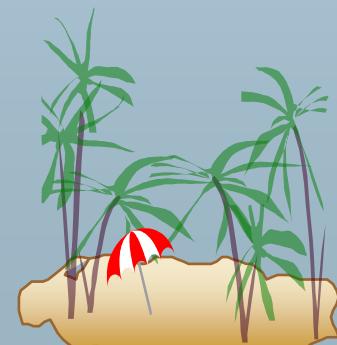
# Closure of Attribute Sets

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

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

- Algorithm to compute  $\alpha^+$ , the closure of  $\alpha$  under  $F$

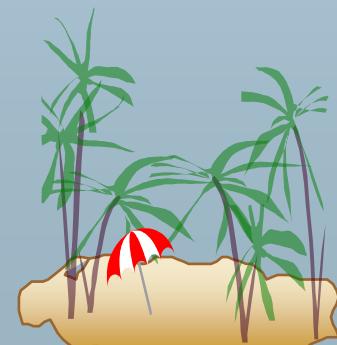
```
result :=  $\alpha$ ;  
while (changes to result) do  
  for each  $\beta \rightarrow \gamma$  in  $F$  do  
    begin  
      if  $\beta \subseteq \text{result}$  then result := result  $\cup$   $\gamma$   
    end
```





# Example of Attribute Set Closure

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

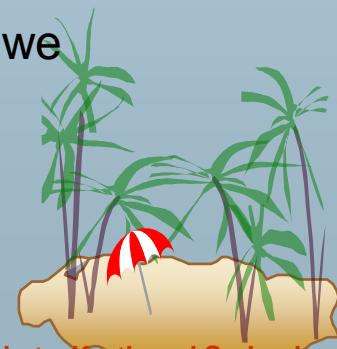




# Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

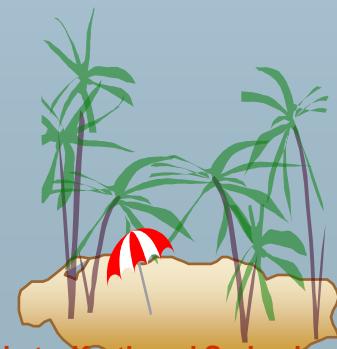
- Testing for superkey:
  - To test if  $\alpha$  is a superkey, we compute  $\alpha^+$ , and check if  $\alpha^+$  contains all attributes of  $R$ .
- Testing functional dependencies
  - To check if a functional dependency  $\alpha \rightarrow \beta$  holds (or, in other words, is in  $F^+$ ), just check if  $\beta \subseteq \alpha^+$ .
  - That is, we compute  $\alpha^+$  by using attribute closure, and then check if it contains  $\beta$ .
  - Is a simple and cheap test, and very useful
- Computing closure of  $F$ 
  - For each  $\gamma \subseteq R$ , we find the closure  $\gamma^+$ , and for each  $S \subseteq \gamma^+$ , we output a functional dependency  $\gamma \rightarrow S$ .





# Canonical Cover

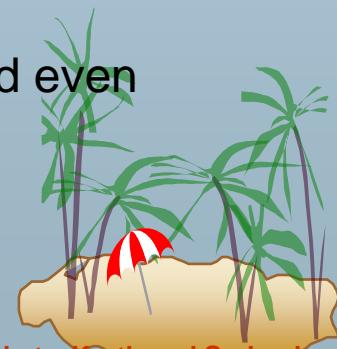
- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - Eg:  $A \rightarrow C$  is redundant in:  $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$
  - Parts of a functional dependency may be redundant
    - E.g. on RHS:  $\{A \rightarrow B, B \rightarrow C, A \rightarrow CD\}$  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 Attributes

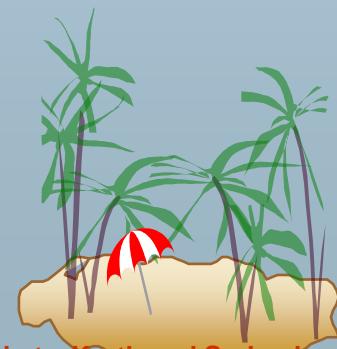
- Consider a set  $F$  of functional dependencies and the functional dependency  $\alpha \rightarrow \beta$  in  $F$ .
  - Attribute  $A$  is **extraneous** in  $\alpha$  if  $A \in \alpha$  and  $F$  logically implies  $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ .
  - Attribute  $A$  is **extraneous** in  $\beta$  if  $A \in \beta$  and the set of functional dependencies  $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$  logically implies  $F$ .
- Note: implication in the opposite direction is trivial in each of the cases above, since a “stronger” functional dependency always implies a weaker one
- Example: Given  $F = \{A \rightarrow C, AB \rightarrow C\}$ 
  - $B$  is extraneous in  $AB \rightarrow C$  because  $\{A \rightarrow C, AB \rightarrow C\}$  logically implies  $A \rightarrow C$  (I.e. the result of dropping  $B$  from  $AB \rightarrow C$ ).
- Example: Given  $F = \{A \rightarrow C, AB \rightarrow CD\}$ 
  - $C$  is extraneous in  $AB \rightarrow CD$  since  $AB \rightarrow C$  can be inferred even after deleting  $C$





# Testing if an Attribute is Extraneous

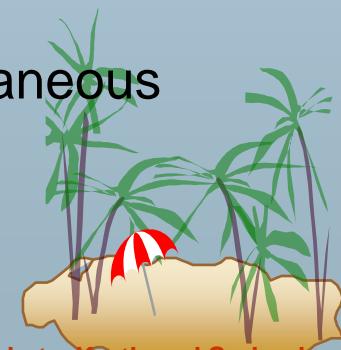
- Consider a set  $F$  of functional dependencies and the functional dependency  $\alpha \rightarrow \beta$  in  $F$ .
- To test if attribute  $A \in \alpha$  is extraneous in  $\alpha$ 
  1. compute  $(\{\alpha\} - A)^+$  using the dependencies in  $F$
  2. check that  $(\{\alpha\} - A)^+$  contains  $A$ ; if it does,  $A$  is extraneous
- To test if attribute  $A \in \beta$  is extraneous in  $\beta$ 
  1. compute  $\alpha^+$  using only the dependencies in
$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\},$$
  2. check that  $\alpha^+$  contains  $A$ ; if it does,  $A$  is extraneous





# Canonical Cover

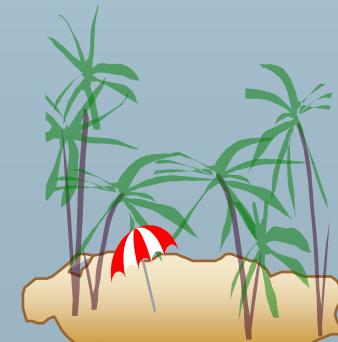
- A *canonical cover* for  $F$  is a set of dependencies  $F_c$  such that
  - $F$  logically implies all dependencies in  $F_c$ , and
  - $F_c$  logically implies all dependencies in  $F$ , and
  - No functional dependency in  $F_c$  contains an extraneous attribute, and
  - Each left side of functional dependency in  $F_c$  is unique.
- To compute a canonical cover for  $F$ :  
**repeat**
  - Use the union rule to replace any dependencies in  $F$   
 $\alpha_1 \rightarrow \beta_1$  and  $\alpha_1 \rightarrow \beta_2$  with  $\alpha_1 \rightarrow \beta_1 \beta_2$
  - Find a functional dependency  $\alpha \rightarrow \beta$  with an  
        extraneous attribute either in  $\alpha$  or in  $\beta$
  - If an extraneous attribute is found, delete it from  $\alpha \rightarrow \beta$**until**  $F$  does not change
- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied





# Example of Computing a Canonical Cover

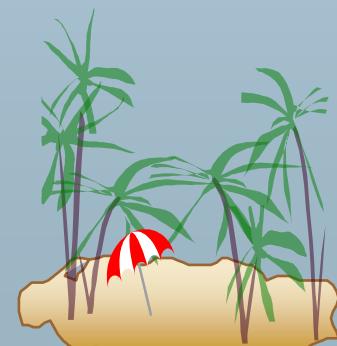
- $R = (A, B, C)$   
 $F = \{A \rightarrow BC$   
     $B \rightarrow C$   
     $A \rightarrow B$   
     $AB \rightarrow C\}$
- Combine  $A \rightarrow BC$  and  $A \rightarrow B$  into  $A \rightarrow BC$ 
  - Set is now  $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous 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  $B \rightarrow C$ .
      - Can use attribute closure of A in more complex cases
- The canonical cover is:  $A \rightarrow B$   
                         $B \rightarrow C$





# Goals of Normalization

- Decide whether a particular relation  $R$  is in “good” form.
- In the case that a relation  $R$  is not in “good” form, decompose it into a set of relations  $\{R_1, R_2, \dots, R_n\}$  such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- Our theory is based on:
  - functional dependencies
  - multivalued dependencies





# Decomposition

- Decompose the relation schema *Lending-schema* into:

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

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

- All attributes of an original schema ( $R$ ) must appear in the decomposition ( $R_1, R_2$ ):

$$R = R_1 \cup R_2$$

- Lossless-join decomposition.

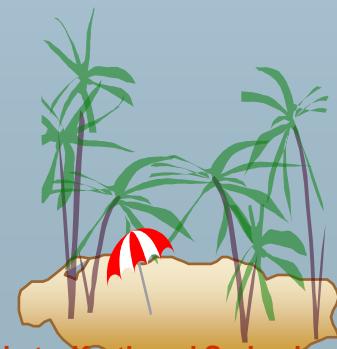
For all possible relations  $r$  on schema  $R$

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

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

- $R_1 \cap R_2 \rightarrow R_1$

- $R_1 \cap R_2 \rightarrow R_2$





# Example of Lossy-Join Decomposition

- Lossy-join decompositions result in information loss.
- Example: Decomposition of  $R = (A, B)$   
 $R_1 = (A)$      $R_2 = (B)$

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1

r

A
$\alpha$
$\beta$

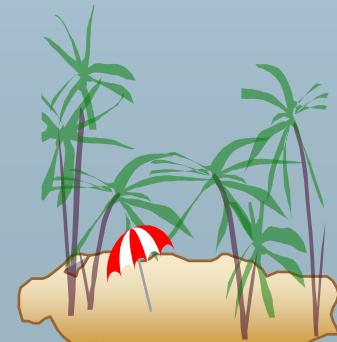
$\Pi_A(r)$

B
1
2

$\Pi_{B(r)}$

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

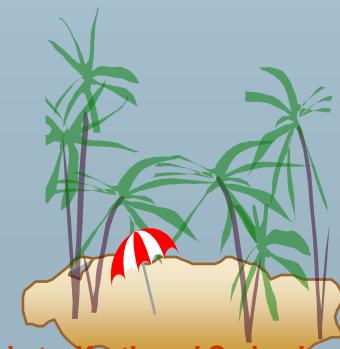
A	B
$\alpha$	1
$\alpha$	2
$\beta$	1
$\beta$	2





# Normalization Using Functional Dependencies

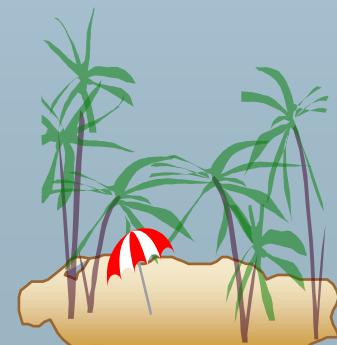
- When we decompose a relation schema  $R$  with a set of functional dependencies  $F$  into  $R_1, R_2, \dots, R_n$  we want
  - **Lossless-join decomposition:** Otherwise decomposition would result in information loss.
  - **No redundancy:** The relations  $R_i$  preferably should be in either Boyce-Codd Normal Form or Third Normal Form.
  - **Dependency preservation:** Let  $F_i$  be the set of dependencies  $F^+$  that include only attributes in  $R_i$ .
    - Preferably the decomposition should be **dependency preserving**, that is,  $(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$
    - Otherwise, checking updates for violation of functional dependencies may require computing joins, which is expensive.





# Example

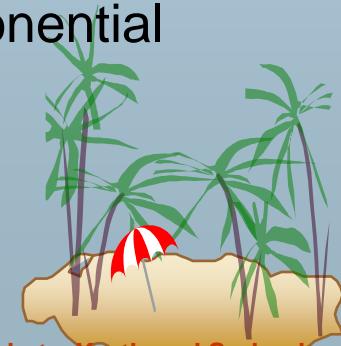
- $R = (A, B, C)$   
 $F = \{A \rightarrow B, B \rightarrow C\}$ 
  - Can be decomposed in two different ways
- $R_1 = (A, B), R_2 = (B, C)$ 
  - Lossless-join decomposition:  
$$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$
  - Dependency preserving
- $R_1 = (A, B), R_2 = (A, C)$ 
  - Lossless-join decomposition:  
$$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$
  - Not dependency preserving  
(cannot check  $B \rightarrow C$  without computing  $R_1 \bowtie R_2$ )





# Testing for Dependency Preservation

- To check if a dependency  $\alpha \rightarrow \beta$  is preserved in a decomposition of  $R$  into  $R_1, R_2, \dots, R_n$  we apply the following simplified test (with attribute closure done w.r.t.  $F$ )
  - $result = \alpha$   
**while** (changes to  $result$ ) do  
    **for each**  $R_i$  in the decomposition  
         $t = (result \cap R_i)^+ \cap R_i$   
         $result = result \cup t$
  - If  $result$  contains all attributes in  $\beta$ , then the functional dependency  $\alpha \rightarrow \beta$  is preserved.
- We apply the test on all dependencies in  $F$  to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute  $F^+$  and  $(F_1 \cup F_2 \cup \dots \cup F_n)^+$

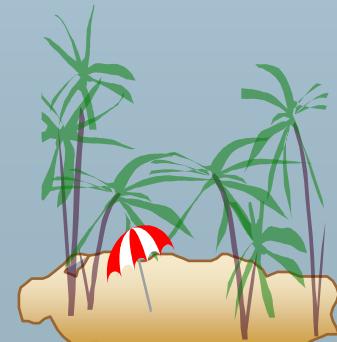




# Boyce-Codd Normal Form

A relation schema  $R$  is in BCNF with respect to a set  $F$  of functional dependencies if for all functional dependencies in  $F^+$  of the form  $\alpha \rightarrow \beta$ , where  $\alpha \subseteq R$  and  $\beta \subseteq R$ , at least one of the following holds:

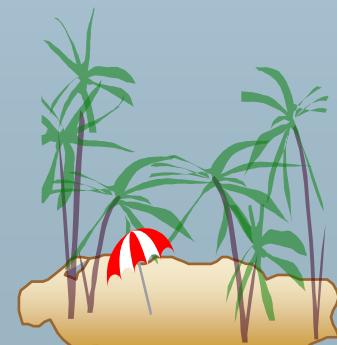
- $\alpha \rightarrow \beta$  is trivial (i.e.,  $\beta \subseteq \alpha$ )
- $\alpha$  is a superkey for  $R$





# Example

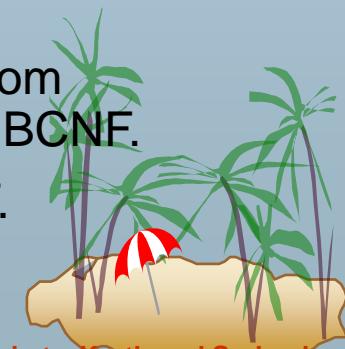
- $R = (A, B, C)$   
 $F = \{A \rightarrow B$   
 $\quad B \rightarrow C\}$   
Key = {A}
- $R$  is not in BCNF
- Decomposition  $R_1 = (A, B)$ ,  $R_2 = (B, C)$ 
  - $R_1$  and  $R_2$  in BCNF
  - Lossless-join decomposition
  - Dependency preserving





# Testing for BCNF

- To check if a non-trivial dependency  $\alpha \not\rightarrow \beta$  causes a violation of BCNF
  1. compute  $\alpha^+$  (the attribute closure of  $\alpha$ ), and
  2. verify that it includes all attributes of  $R$ , that is, it is a superkey of  $R$ .
- **Simplified test:** To check if a relation schema  $R$  is in BCNF, it suffices to check only the dependencies in the given set  $F$  for violation of BCNF, rather than checking all dependencies in  $F^+$ .
  - If none of the dependencies in  $F$  causes a violation of BCNF, then none of the dependencies in  $F^+$  will cause a violation of BCNF either.
- However, using only  $F$  is **incorrect** when testing a relation in a decomposition of  $R$ 
  - E.g. Consider  $R(A, B, C, D)$ , with  $F = \{ A \rightarrow B, B \rightarrow C \}$ 
    - Decompose  $R$  into  $R_1(A, B)$  and  $R_2(A, C, D)$
    - Neither of the dependencies in  $F$  contain only attributes from  $(A, C, D)$  so we might be misled into thinking  $R_2$  satisfies BCNF.
    - In fact, dependency  $A \rightarrow C$  in  $F^+$  shows  $R_2$  is not in BCNF.

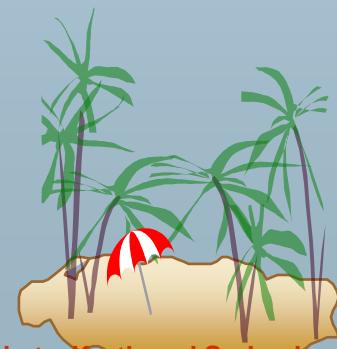




# BCNF Decomposition Algorithm

```
result := {R};  
done := false;  
compute  $F^+$ ;  
while (not done) do  
  if (there is a schema  $R_i$  in result that is not in BCNF)  
    then begin  
      let  $\alpha \rightarrow \beta$  be a nontrivial functional  
          dependency that holds on  $R_i$   
          such that  $\alpha \rightarrow R_i$  is not in  $F^+$ ,  
          and  $\alpha \cap \beta = \emptyset$ ;  
      result := (result -  $R_i$ )  $\cup$  ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ );  
    end  
  else done := true;
```

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

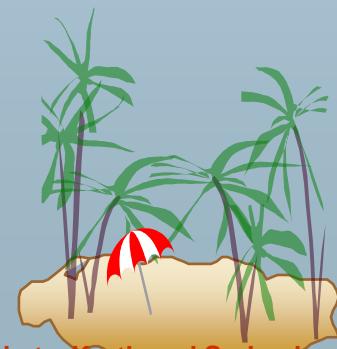




# Example of BCNF Decomposition

- $R = (\text{branch-name}, \text{branch-city}, \text{assets}, \text{customer-name}, \text{loan-number}, \text{amount})$   
 $F = \{\text{branch-name} \rightarrow \text{assets } \text{branch-city}$   
 $\text{loan-number} \rightarrow \text{amount } \text{branch-name}\}$   
Key = {loan-number, customer-name}
- Decomposition
  - $R_1 = (\text{branch-name}, \text{branch-city}, \text{assets})$
  - $R_2 = (\text{branch-name}, \text{customer-name}, \text{loan-number}, \text{amount})$
  - $R_3 = (\text{branch-name}, \text{loan-number}, \text{amount})$
  - $R_4 = (\text{customer-name}, \text{loan-number})$
- Final decomposition

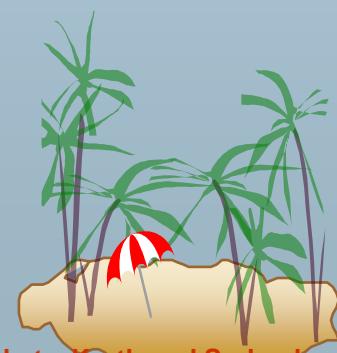
$R_1, R_3, R_4$





# Testing Decomposition for BCNF

- To check if a relation  $R_i$  in a decomposition of  $R$  is in BCNF,
  - Either test  $R_i$  for BCNF with respect to the **restriction** of  $F$  to  $R_i$  (that is, all FDs in  $F^+$  that contain only attributes from  $R_i$ )
  - or use the original set of dependencies  $F$  that hold on  $R$ , but with the following test:
    - for every set of attributes  $\alpha \subseteq R_i$ , check that  $\alpha^+$  (the attribute closure of  $\alpha$ ) either includes no attribute of  $R_i - \alpha$ , or includes all attributes of  $R_i$ .
- If the condition is violated by some  $\alpha \rightarrow \beta$  in  $F$ , the dependency  $\alpha \rightarrow (\alpha^+ - \alpha)$  can be shown to hold on  $R_i$ , and  $R_i$  violates BCNF.
- We use above dependency to decompose  $R_i$





# BCNF and Dependency Preservation

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

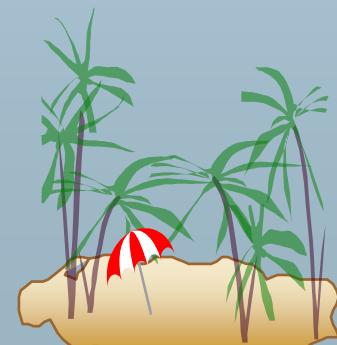
- $R = (J, K, L)$

$$F = \{JK \rightarrow L \\ L \rightarrow K\}$$

Two candidate keys =  $JK$  and  $JL$

- $R$  is not in BCNF
- Any decomposition of  $R$  will fail to preserve

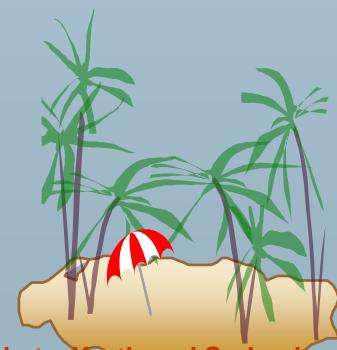
$$JK \rightarrow L$$





# 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.
  - Allows some redundancy (with resultant problems; we will see examples later)
  - But FDs can be checked on individual relations without computing a join.
  - There is always a lossless-join, dependency-preserving decomposition into 3NF.





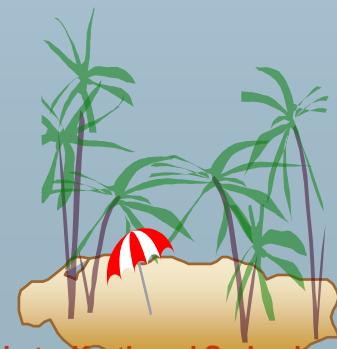
# Third Normal Form

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

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

at least one of the following holds:

- $\alpha \rightarrow \beta$  is trivial (i.e.,  $\beta \in \alpha$ )
- $\alpha$  is a superkey for  $R$
- Each attribute  $A$  in  $\beta - \alpha$  is contained in a candidate key for  $R$ .  
*(NOTE: each attribute may be in a different candidate key)*
- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).





# 3NF (Cont.)

- Example

- $R = (J, K, L)$   
 $F = \{JK \rightarrow L, L \rightarrow K\}$
- Two candidate keys:  $JK$  and  $JL$
- $R$  is in 3NF

$JK \rightarrow L$

$JK$  is a superkey

$L \rightarrow K$

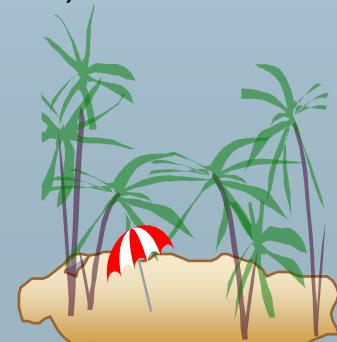
$K$  is contained in a candidate key

- BCNF decomposition has  $(JL)$  and  $(LK)$ 
  - Testing for  $JK \rightarrow L$  requires a join
- There is some redundancy in this schema
- Equivalent to example in book:

Banker-schema = (branch-name, customer-name, banker-name)

banker-name  $\rightarrow$  branch name

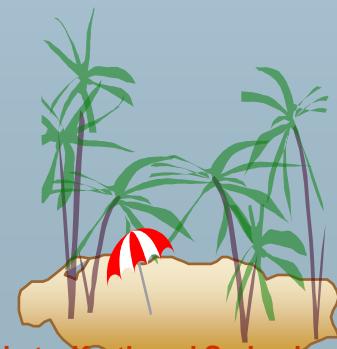
branch name customer-name  $\rightarrow$  banker-name





# Testing for 3NF

- Optimization: Need to check only FDs in  $F$ , need not check all FDs in  $F^+$ .
- Use attribute closure to check for each dependency  $\alpha \rightarrow \beta$ , if  $\alpha$  is a superkey.
- If  $\alpha$  is not a superkey, we have to verify if each attribute in  $\beta$  is contained in a candidate key of  $R$ 
  - this test is rather more expensive, since it involve finding candidate keys
  - testing for 3NF has been shown to be NP-hard
  - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time





# 3NF Decomposition Algorithm

Let  $F_c$  be a canonical cover for  $F$ ;

$i := 0$ ;

**for each** functional dependency  $\alpha \rightarrow \beta$  in  $F_c$  **do**

**if** none of the schemas  $R_j$ ,  $1 \leq j \leq i$  contains  $\alpha \beta$

**then begin**

$i := i + 1$ ;

$R_i := \alpha \beta$

**end**

**if** none of the schemas  $R_j$ ,  $1 \leq j \leq i$  contains a candidate key for  $R$

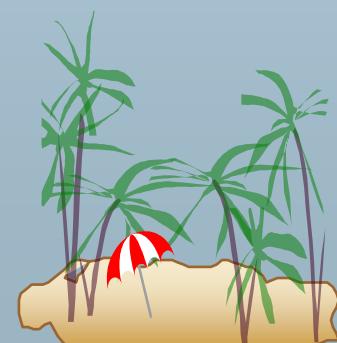
**then begin**

$i := i + 1$ ;

$R_i :=$  any candidate key for  $R$ ;

**end**

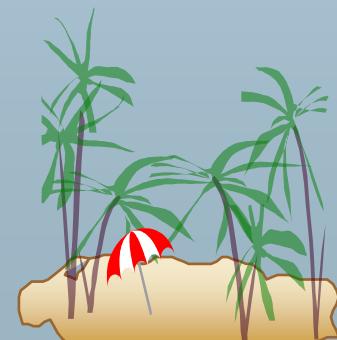
**return**  $(R_1, R_2, \dots, R_i)$





# 3NF Decomposition Algorithm (Cont.)

- Above algorithm ensures:
  - each relation schema  $R_i$  is in 3NF
  - decomposition is dependency preserving and lossless-join
  - Proof of correctness is at end of this file ([click here](#))





# Example

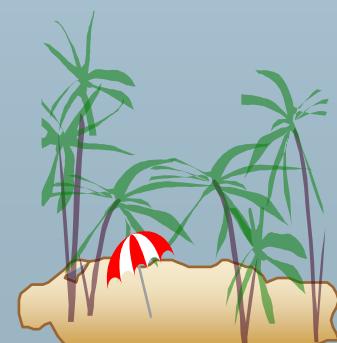
- Relation schema:

*Banker-info-schema = (branch-name, customer-name,  
banker-name, office-number)*

- The functional dependencies for this relation schema are:  
 $\text{banker-name} \rightarrow \text{branch-name}$   $\text{office-number}$   
 $\text{customer-name } \text{branch-name} \rightarrow \text{banker-name}$

- The key is:

$\{\text{customer-name}, \text{branch-name}\}$





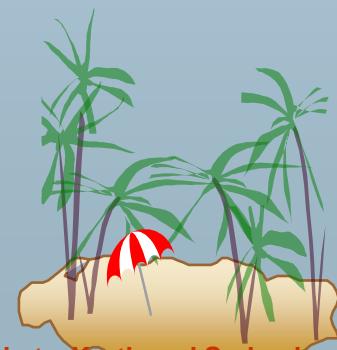
# Applying 3NF to *Banker-info-schema*

- The **for** loop in the algorithm causes us to include the following schemas in our decomposition:

*Banker-office-schema* = (*banker-name*, *branch-name*,  
*office-number*)

*Banker-schema* = (*customer-name*, *branch-name*,  
*banker-name*)

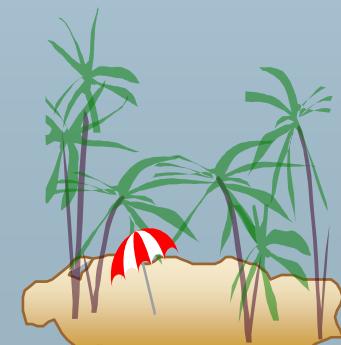
- Since *Banker-schema* contains a candidate key for *Banker-info-schema*, we are done with the decomposition process.





# Comparison of BCNF and 3NF

- It is always possible to decompose a relation into relations in 3NF and
  - the decomposition is lossless
  - the dependencies are preserved
- It is always possible to decompose a relation into relations in BCNF and
  - the decomposition is lossless
  - it may not be possible to preserve dependencies.





# Comparison of BCNF and 3NF (Cont.)

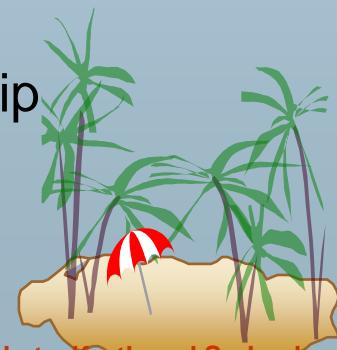
- Example of problems due to redundancy in 3NF

- $R = (J, K, L)$   
 $F = \{JK \rightarrow L, L \rightarrow K\}$

$J$	$L$	$K$
$j_1$	$l_1$	$k_1$
$j_2$	$l_1$	$k_1$
$j_3$	$l_1$	$k_1$
null	$l_2$	$k_2$

A schema that is in 3NF but not in BCNF has the problems of

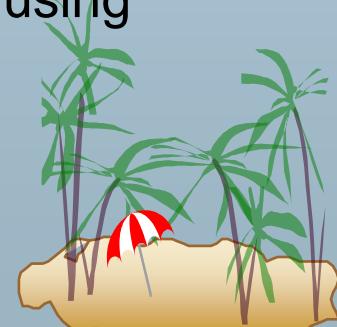
- repetition of information (e.g., the relationship  $l_1, k_1$ )
- need to use null values (e.g., to represent the relationship  $l_2, k_2$  where there is no corresponding value for  $J$ ).





# Design Goals

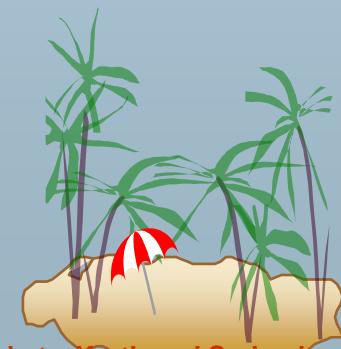
- Goal for a relational database design is:
  - BCNF.
  - Lossless join.
  - Dependency preservation.
- If we cannot achieve this, we accept one of
  - Lack of dependency preservation
  - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.  
Can specify FDs using assertions, but they are expensive to test
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.





# Testing for FDs Across Relations

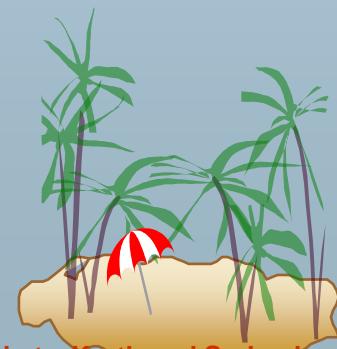
- If decomposition is not dependency preserving, we can have an extra **materialized view** for each dependency  $\alpha \rightarrow \beta$  in  $F_c$  that is not preserved in the decomposition
- The materialized view is defined as a projection on  $\alpha \beta$  of the join of the relations in the decomposition
- Many newer database systems support materialized views and database system maintains the view when the relations are updated.
  - No extra coding effort for programmer.
- The functional dependency  $\alpha \rightarrow \beta$  is expressed by declaring  $\alpha$  as a candidate key on the materialized view.
- Checking for candidate key cheaper than checking  $\alpha \rightarrow \beta$
- BUT:
  - Space overhead: for storing the materialized view
  - Time overhead: Need to keep materialized view up to date when relations are updated
  - Database system may not support key declarations on materialized views





# Multivalued Dependencies

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a database
  - $\text{classes}(\text{course}, \text{teacher}, \text{book})$   
such that  $(c,t,b) \in \text{classes}$  means that  $t$  is qualified to teach  $c$ , and  $b$  is a required textbook for  $c$
- The database is supposed to list for each course the set of teachers any one of which can be the course's instructor, and the set of books, all of which are required for the course (no matter who teaches it).





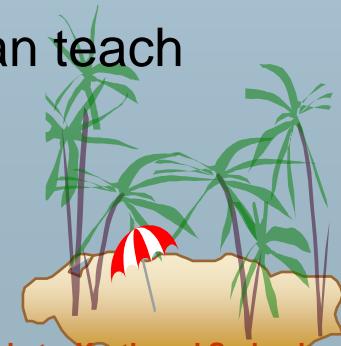
# Multivalued Dependencies (Cont.)

<i>course</i>	<i>teacher</i>	<i>book</i>
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Shaw
operating systems	Jim	OS Concepts
operating systems	Jim	Shaw

*classes*

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if Sara is a new teacher that can teach database, two tuples need to be inserted

(database, Sara, DB Concepts)  
(database, Sara, Ullman)





# Multivalued Dependencies (Cont.)

- Therefore, it is better to decompose *classes* into:

course	teacher
database	Avi
database	Hank
database	Sudarshan
operating systems	Avi
operating systems	Jim

*teaches*

course	book
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

*text*

We shall see that these two relations are in Fourth Normal Form (4NF)





# Multivalued Dependencies (MVDs)

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

$$\alpha \rightarrow\!\!\!\rightarrow \beta$$

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:

$$\begin{aligned}t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\t_3[\beta] &= t_1[\beta] \\t_3[R - \beta] &= t_2[R - \beta] \\t_4[\beta] &= t_2[\beta] \\t_4[R - \beta] &= t_1[R - \beta]\end{aligned}$$

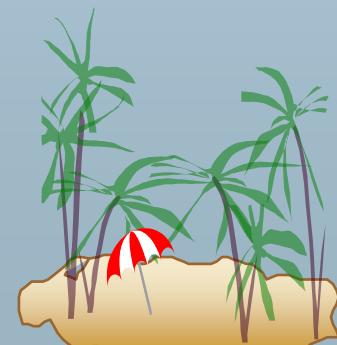




# MVD (Cont.)

- Tabular representation of  $\alpha \rightarrow\!\!\!\rightarrow \beta$

	$\alpha$	$\beta$	$R - \alpha - \beta$
$t_1$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
$t_2$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
$t_3$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
$t_4$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$





# Example

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

$Y, Z, W$

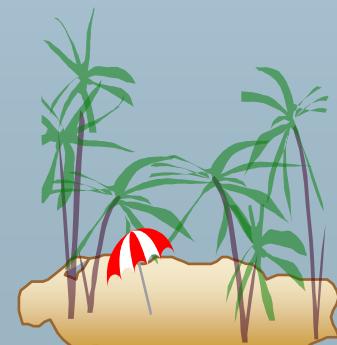
- We say that  $Y \twoheadrightarrow Z$  ( $Y$  multidetermines  $Z$ ) if and only if for all possible relations  $r(R)$

$\langle y_1, z_1, w_1 \rangle \in r$  and  $\langle y_2, z_2, w_2 \rangle \in r$

then

$\langle y_1, z_1, w_2 \rangle \in r$  and  $\langle y_2, z_2, w_1 \rangle \in r$

- Note that since the behavior of  $Z$  and  $W$  are identical it follows that  $Y \twoheadrightarrow Z$  if  $Y \twoheadrightarrow W$





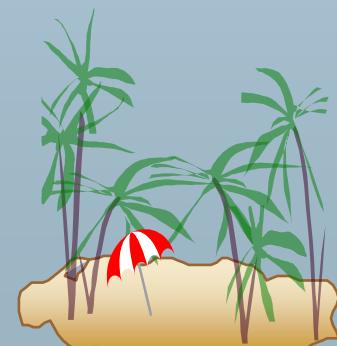
# Example (Cont.)

- In our example:

$\text{course} \rightarrow\rightarrow \text{teacher}$

$\text{course} \rightarrow\rightarrow \text{book}$

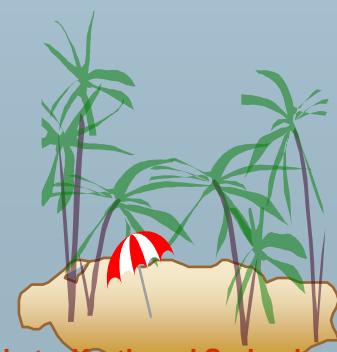
- The above formal definition is supposed to formalize the notion that given a particular value of  $Y$  (*course*) it has associated with it a set of values of  $Z$  (*teacher*) and a set of values of  $W$  (*book*), and these two sets are in some sense independent of each other.
- Note:
  - If  $Y \rightarrow Z$  then  $Y \rightarrow\rightarrow Z$
  - Indeed we have (in above notation)  $Z_1 = Z_2$   
The claim follows.





# Use of Multivalued Dependencies

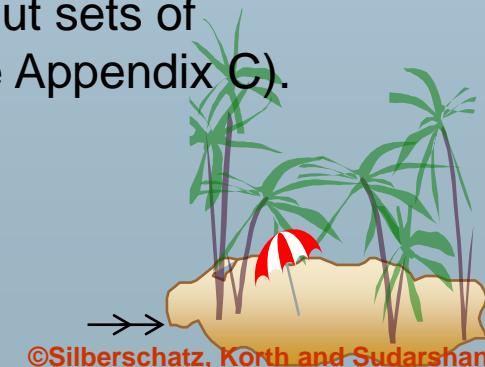
- We use multivalued dependencies in two ways:
  1. To test relations to **determine** whether they are legal under a given set of functional and multivalued dependencies
  2. To specify **constraints** on the set of legal relations. We shall thus concern ourselves *only* with relations that satisfy a given set of functional and multivalued dependencies.
- If a relation  $r$  fails to satisfy a given multivalued dependency, we can construct a relations  $r'$  that does satisfy the multivalued dependency by adding tuples to  $r$ .





# Theory of MVDs

- From the definition of multivalued dependency, we can derive the following rule:
  - If  $\alpha \rightarrow \beta$ , then  $\alpha \twoheadrightarrow \beta$
- That is, every functional dependency is also a multivalued dependency
- The **closure**  $D^+$  of  $D$  is the set of all functional and multivalued dependencies logically implied by  $D$ .
  - We can compute  $D^+$  from  $D$ , using the formal definitions of functional dependencies and multivalued dependencies.
  - We can manage with such reasoning for very simple multivalued dependencies, which seem to be most common in practice
  - For complex dependencies, it is better to reason about sets of dependencies using a system of inference rules (see Appendix C).





# Fourth Normal Form

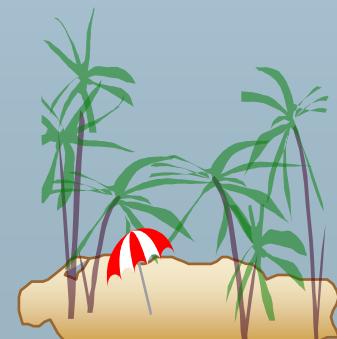
- A relation schema  $R$  is in 4NF with respect to a set  $D$  of functional and multivalued dependencies if for all multivalued dependencies in  $D^+$  of the form  $\alpha \twoheadrightarrow \beta$ , where  $\alpha \subseteq R$  and  $\beta \subseteq R$ , at least one of the following hold:
  - $\alpha \rightarrow \beta$  is trivial (i.e.,  $\beta \subseteq \alpha$  or  $\alpha \cup \beta = R$ )
  - $\alpha$  is a superkey for schema  $R$
- If a relation is in 4NF it is in BCNF





# Restriction of Multivalued Dependencies

- The restriction of  $D$  to  $R_i$  is the set  $D_i$  consisting of
  - All functional dependencies in  $D^+$  that include only attributes of  $R_i$
  - All multivalued dependencies of the form
$$\alpha \rightarrow\rightarrow (\beta \cap R_i)$$
where  $\alpha \subseteq R_i$  and  $\alpha \rightarrow\rightarrow \beta$  is in  $D^+$





# 4NF Decomposition Algorithm

*result* := { $R$ };

*done* := false;

*compute*  $D^+$ ;

Let  $D_i$  denote the restriction of  $D^+$  to  $R_i$

**while** (*not done*)

**if** (there is a schema  $R_i$  in *result* that is not in 4NF) **then**

**begin**

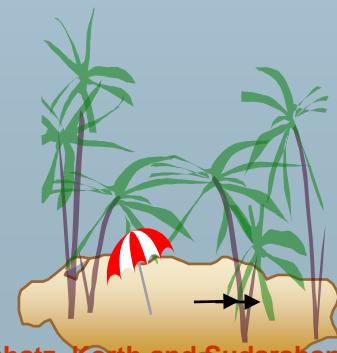
let  $\alpha \rightarrow\!\!\!\rightarrow \beta$  be a nontrivial multivalued dependency that holds  
on  $R_i$  such that  $\alpha \rightarrow R_i$  is not in  $D_i$ , and  $\alpha \cap \beta = \emptyset$ ;

*result* := (*result* -  $R_i$ )  $\cup$  ( $R_i$  -  $\beta$ )  $\cup$  ( $\alpha, \beta$ );

**end**

**else** *done* := true;

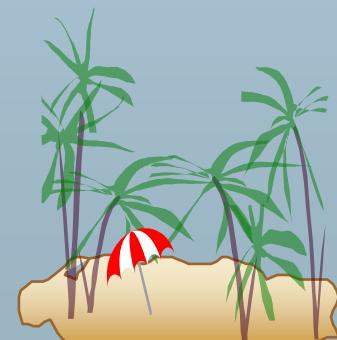
Note: each  $R_i$  is in 4NF, and decomposition is lossless-join





# Example

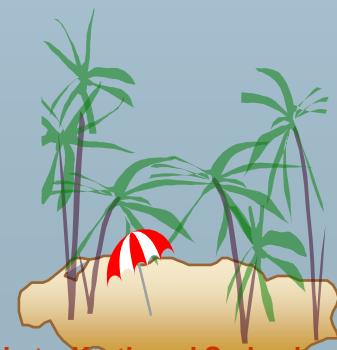
- $R = (A, B, C, G, H, I)$
- $F = \{ A \rightarrow\!\!\!\rightarrow B$   
         $B \rightarrow\!\!\!\rightarrow HI$   
         $CG \rightarrow\!\!\!\rightarrow H \}$
- $R$  is not in 4NF since  $A \rightarrow\!\!\!\rightarrow B$  and  $A$  is not a superkey for  $R$
- Decomposition
  - a)  $R_1 = (A, B)$   $(R_1$  is in 4NF)
  - b)  $R_2 = (A, C, G, H, I)$   $(R_2$  is not in 4NF)
  - c)  $R_3 = (C, G, H)$   $(R_3$  is in 4NF)
  - d)  $R_4 = (A, C, G, I)$   $(R_4$  is not in 4NF)
- Since  $A \rightarrow\!\!\!\rightarrow B$  and  $B \rightarrow\!\!\!\rightarrow HI$ ,  $A \rightarrow\!\!\!\rightarrow HI$ ,  $A \rightarrow\!\!\!\rightarrow I$ 
  - e)  $R_5 = (A, I)$   $(R_5$  is in 4NF)
  - f)  $R_6 = (A, C, G)$   $(R_6$  is in 4NF)





# Further Normal Forms

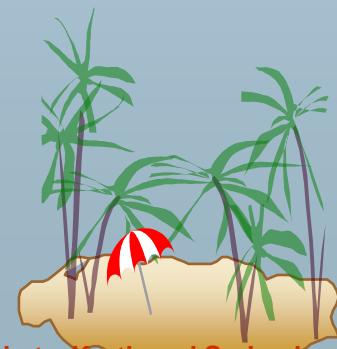
- **Join dependencies** generalize multivalued dependencies
  - lead to **project-join normal form (PJNF)** (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used





# Overall Database Design Process

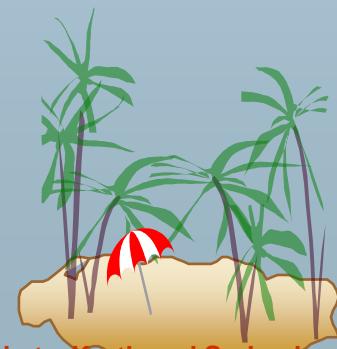
- We have assumed schema  $R$  is given
  - $R$  could have been generated when converting E-R diagram to a set of tables.
  - $R$  could have been a single relation containing *all* attributes that are of interest (called **universal relation**).
  - Normalization breaks  $R$  into smaller relations.
  - $R$  could have been the result of some ad hoc design of relations, which we then test/convert to normal form.





# ER Model and Normalization

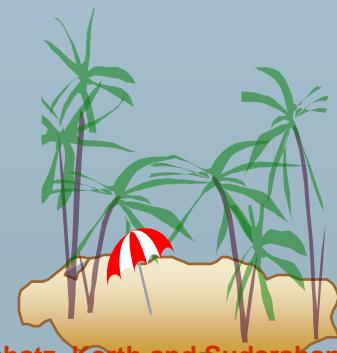
- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design there can be FDs from non-key attributes of an entity to other attributes of the entity
- E.g. *employee* entity with attributes *department-number* and *department-address*, and an FD  $\text{department-number} \rightarrow \text{department-address}$ 
  - Good design would have made department an entity
- FDs from non-key attributes of a relationship set possible, but rare --- most relationships are binary





# Universal Relation Approach

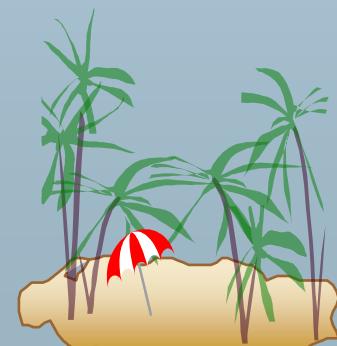
- **Dangling tuples** – Tuples that “disappear” in computing a join.
  - Let  $r_1 (R_1)$ ,  $r_2 (R_2)$ , ...,  $r_n (R_n)$  be a set of relations
  - A tuple  $r$  of the relation  $r_i$  is a dangling tuple if  $r$  is not in the relation:
$$\Pi_{R_i} (r_1 \bowtie r_2 \bowtie \dots \bowtie r_n)$$
- The relation  $r_1 \bowtie r_2 \bowtie \dots \bowtie r_n$  is called a *universal relation* since it involves all the attributes in the “universe” defined by
$$R_1 \cup R_2 \cup \dots \cup R_n$$
- If dangling tuples are allowed in the database, instead of decomposing a universal relation, we may prefer to synthesize a collection of normal form schemas from a given set of attributes.





# Universal Relation Approach

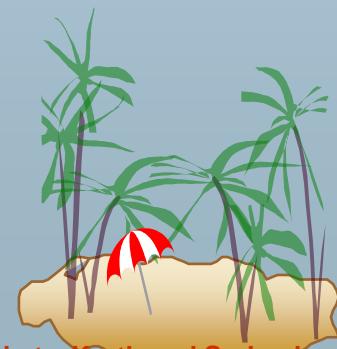
- Dangling tuples may occur in practical database applications.
- They represent incomplete information
- E.g. may want to break up information about loans into:
  - (branch-name, loan-number)
  - (loan-number, amount)
  - (loan-number, customer-name)
- Universal relation would require null values, and have dangling tuples





# Universal Relation Approach (Contd.)

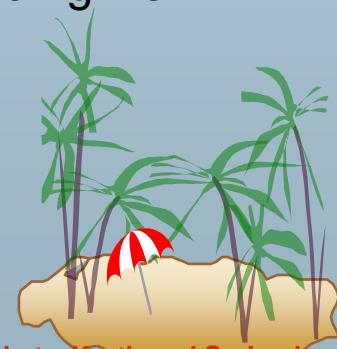
- A particular decomposition defines a restricted form of incomplete information that is acceptable in our database.
  - Above decomposition requires at least one of customer-name, branch-name or amount in order to enter a loan number without using null values
  - Rules out storing of customer-name, amount without an appropriate loan-number (since it is a key, it can't be null either!)
- Universal relation requires unique attribute names **unique role assumption**
  - e.g. *customer-name, branch-name*
- Reuse of attribute names is natural in SQL since relation names can be prefixed to disambiguate names





# Denormalization for Performance

- May want to use non-normalized schema for performance
- E.g. displaying *customer-name* along with *account-number* and *balance* requires join of *account* with *depositor*
- Alternative 1: Use denormalized 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 materialized 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



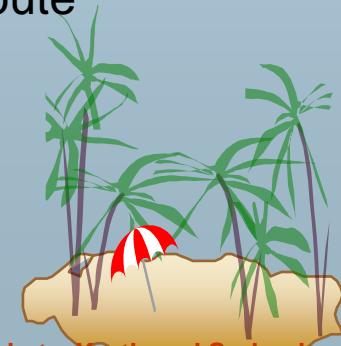


# Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:

Instead of *earnings(company-id, year, amount)*, use

- *earnings-2000, earnings-2001, earnings-2002*, etc., all on the schema (*company-id, earnings*).
  - Above are in BCNF, but make querying across years difficult and needs new table each year
- *company-year(company-id, earnings-2000, earnings-2001, earnings-2002)*
  - Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
  - Is an example of a **crosstab**, where values for one attribute become column names
  - Used in spreadsheets, and in data analysis tools

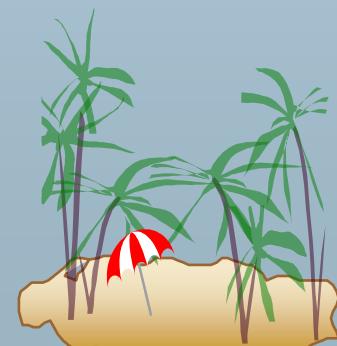


# **Proof of Correctness of 3NF Decomposition Algorithm**



# Correctness of 3NF Decomposition Algorithm

- 3NF decomposition algorithm is dependency preserving (since there is a relation for every FD in  $F_c$ )
- Decomposition is lossless join
  - A candidate key ( $C$ ) is in one of the relations  $R_i$  in decomposition
  - Closure of candidate key under  $F_c$  must contain all attributes in  $R$ .
  - Follow the steps of attribute closure algorithm to show there is only one tuple in the join result for each tuple in  $R_i$

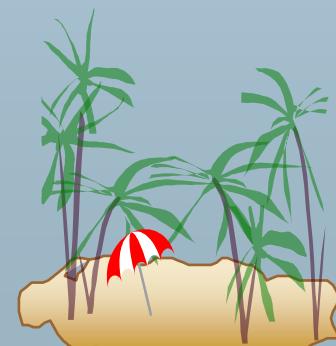




# Correctness of 3NF Decomposition Algorithm (Contd.)

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

- Let  $R_i$  be generated from the dependency  $\alpha \rightarrow \beta$
- Let  $\gamma \rightarrow B$  be any non-trivial functional dependency on  $R_i$ . (We need only consider FDs whose right-hand side is a single attribute.)
- Now,  $B$  can be in either  $\beta$  or  $\alpha$  but not in both. Consider each case separately.

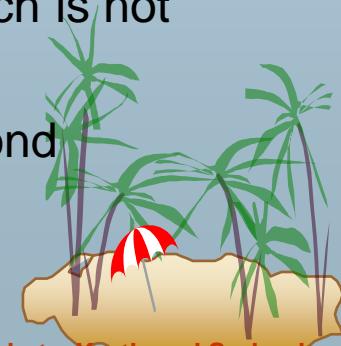




# Correctness of 3NF Decomposition (Contd.)

## □ Case 1: If $B$ in $\beta$ :

- If  $\gamma$  is a superkey, the 2nd condition of 3NF is satisfied
- Otherwise  $\alpha$  must contain some attribute not in  $\gamma$
- Since  $\gamma \rightarrow B$  is in  $F^+$  it must be derivable from  $F_c$ , by using attribute closure on  $\gamma$ .
- Attribute closure not have used  $\alpha \rightarrow \beta$  - if it had been used,  $\alpha$  must be contained in the attribute closure of  $\gamma$ , which is not possible, since we assumed  $\gamma$  is not a superkey.
- Now, using  $\alpha \rightarrow (\beta - \{B\})$  and  $\gamma \rightarrow B$ , we can derive  $\alpha \rightarrow B$   
(since  $\gamma \subseteq \alpha \beta$ , and  $B \notin \gamma$  since  $\gamma \rightarrow B$  is non-trivial)
- Then,  $B$  is extraneous in the right-hand side of  $\alpha \rightarrow \beta$ ; which is not possible since  $\alpha \rightarrow \beta$  is in  $F_c$ .
- Thus, if  $B$  is in  $\beta$  then  $\gamma$  must be a superkey, and the second condition of 3NF must be satisfied.

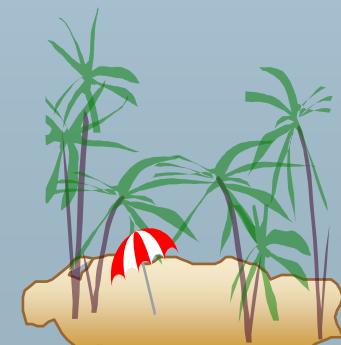


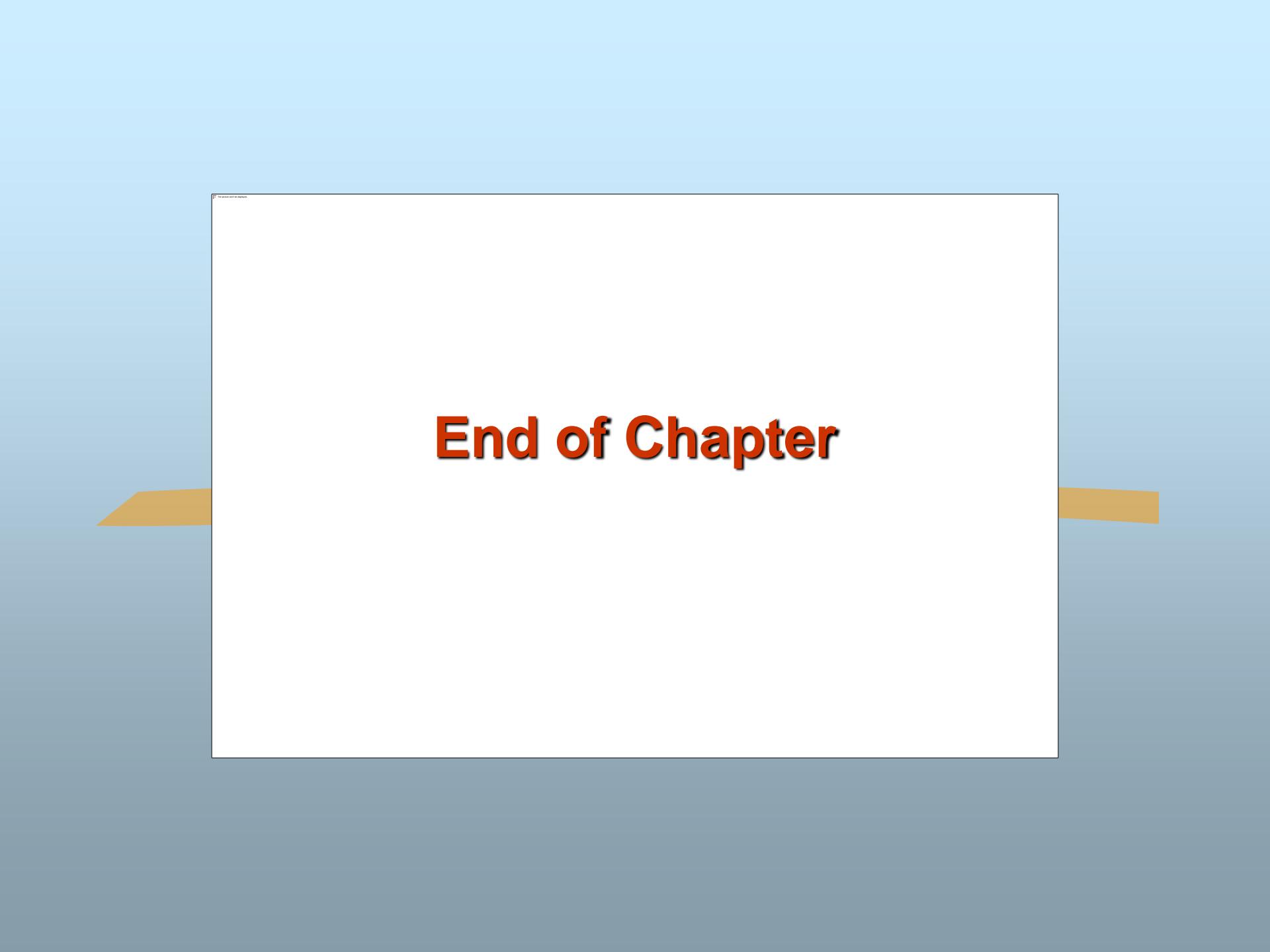


# Correctness of 3NF Decomposition (Contd.)

- Case 2:  $B$  is in  $\alpha$ .
  - Since  $\alpha$  is a candidate key, the third alternative in the definition of 3NF is trivially satisfied.
  - In fact, we cannot show that  $\gamma$  is a superkey.
  - This shows exactly why the third alternative is present in the definition of 3NF.

Q.E.D.





The slide features a central white rectangular area with a thin black border. The text "End of Chapter" is centered within this area in a large, bold, red font. The background of the slide is a solid light blue color. In the corners of the white area, there are two yellow triangular shapes: a larger one on the left and a smaller one on the right, both pointing towards the center.

**End of Chapter**



# Sample *lending* Relation

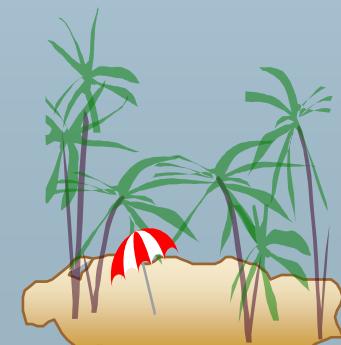
<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>	<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200





# Sample Relation $r$

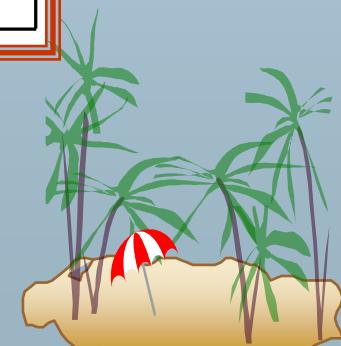
A	B	C	D
$a_1$	$b_1$	$c_1$	$d_1$
$a_1$	$b_2$	$c_1$	$d_2$
$a_2$	$b_2$	$c_2$	$d_2$
$a_2$	$b_2$	$c_2$	$d_3$
$a_3$	$b_3$	$c_2$	$d_4$





# The *customer* Relation

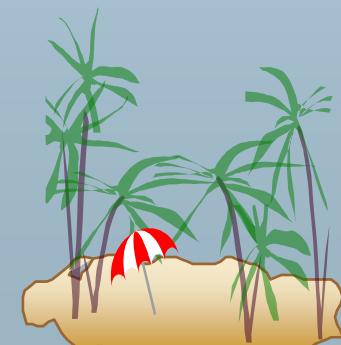
<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford





# The *loan* Relation

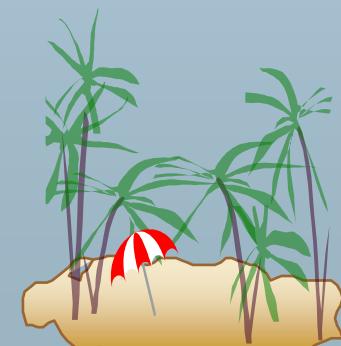
<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-17	Downtown	1000
L-23	Redwood	2000
L-15	Perryridge	1500
L-14	Downtown	1500
L-93	Mianus	500
L-11	Round Hill	900
L-29	Pownal	1200
L-16	North Town	1300
L-18	Downtown	2000
L-25	Perryridge	2500
L-10	Brighton	2200





# The *branch* Relation

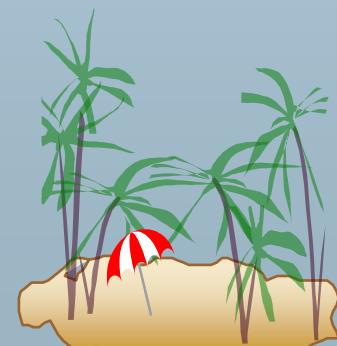
<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
Mianus	Horseneck	400000
Round Hill	Horseneck	8000000
Pownal	Bennington	300000
North Town	Rye	3700000
Brighton	Brooklyn	7100000





# The Relation *branch-customer*

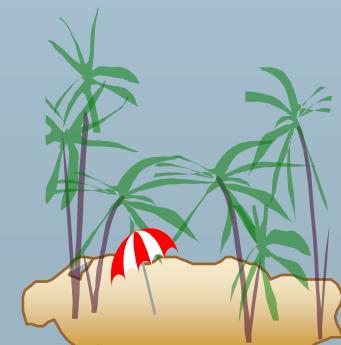
<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>	<i>customer-name</i>
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
Pownal	Bennington	300000	Williams
North Town	Rye	3700000	Hayes
Downtown	Brooklyn	9000000	Johnson
Perryridge	Horseneck	1700000	Glenn
Brighton	Brooklyn	7100000	Brooks





# The Relation *customer-loan*

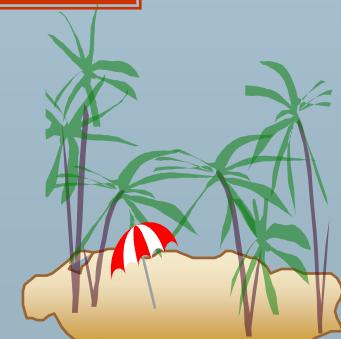
<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-93	500
Turner	L-11	900
Williams	L-29	1200
Hayes	L-16	1300
Johnson	L-18	2000
Glenn	L-25	2500
Brooks	L-10	2200





# The Relation *branch-customer* $\bowtie$ *customer-loan*

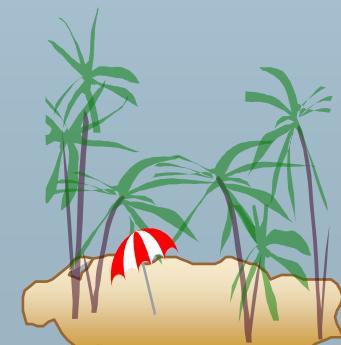
<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>	<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200





# An Instance of *Banker-schema*

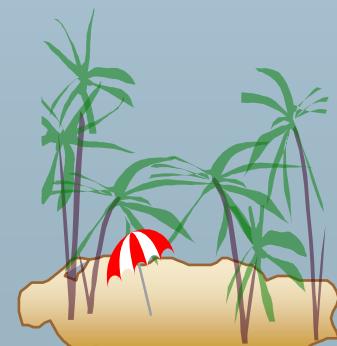
<i>customer-name</i>	<i>banker-name</i>	<i>branch-name</i>
Jones	Johnson	Perryridge
Smith	Johnson	Perryridge
Hayes	Johnson	Perryridge
Jackson	Johnson	Perryridge
Curry	Johnson	Perryridge
Turner	Johnson	Perryridge





# Tabular Representation of $\alpha \rightarrow\!\!\rightarrow \beta$

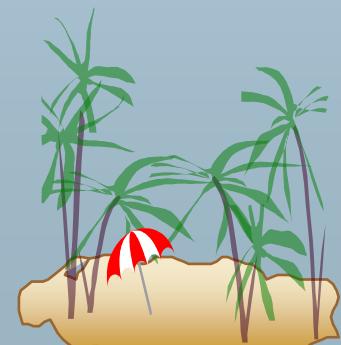
	$\alpha$	$\beta$	$R - \alpha - \beta$
$t_1$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
$t_2$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
$t_3$	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
$t_4$	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$





## Relation bc: An Example of Reduncy in a BCNF Relation

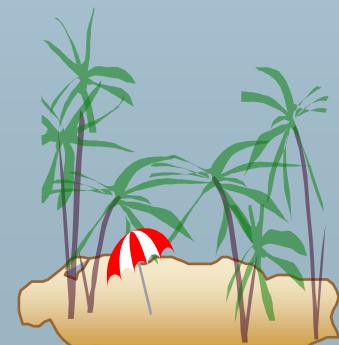
<i>loan-number</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
L-23	Smith	North	Rye
L-23	Smith	Main	Manchester
L-93	Curry	Lake	Horseneck





# An Illegal *bc* Relation

<i>loan-number</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
L-23	Smith	North	Rye
L-27	Smith	Main	Manchester



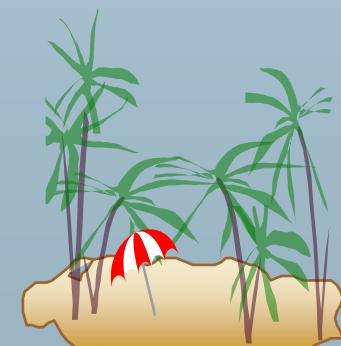


# Decomposition of *loan-info*

<i>branch-name</i>	<i>loan-number</i>
Round Hill	L-58

<i>loan-number</i>	<i>amount</i>

<i>loan-number</i>	<i>customer-name</i>
L-58	Johnson





# Relation of Exercise 7.4

A	B	C
$a_1$	$b_1$	$c_1$
$a_1$	$b_1$	$c_2$
$a_2$	$b_1$	$c_1$
$a_2$	$b_1$	$c_3$

