



# Chapter 8: Relational Database Design

**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use

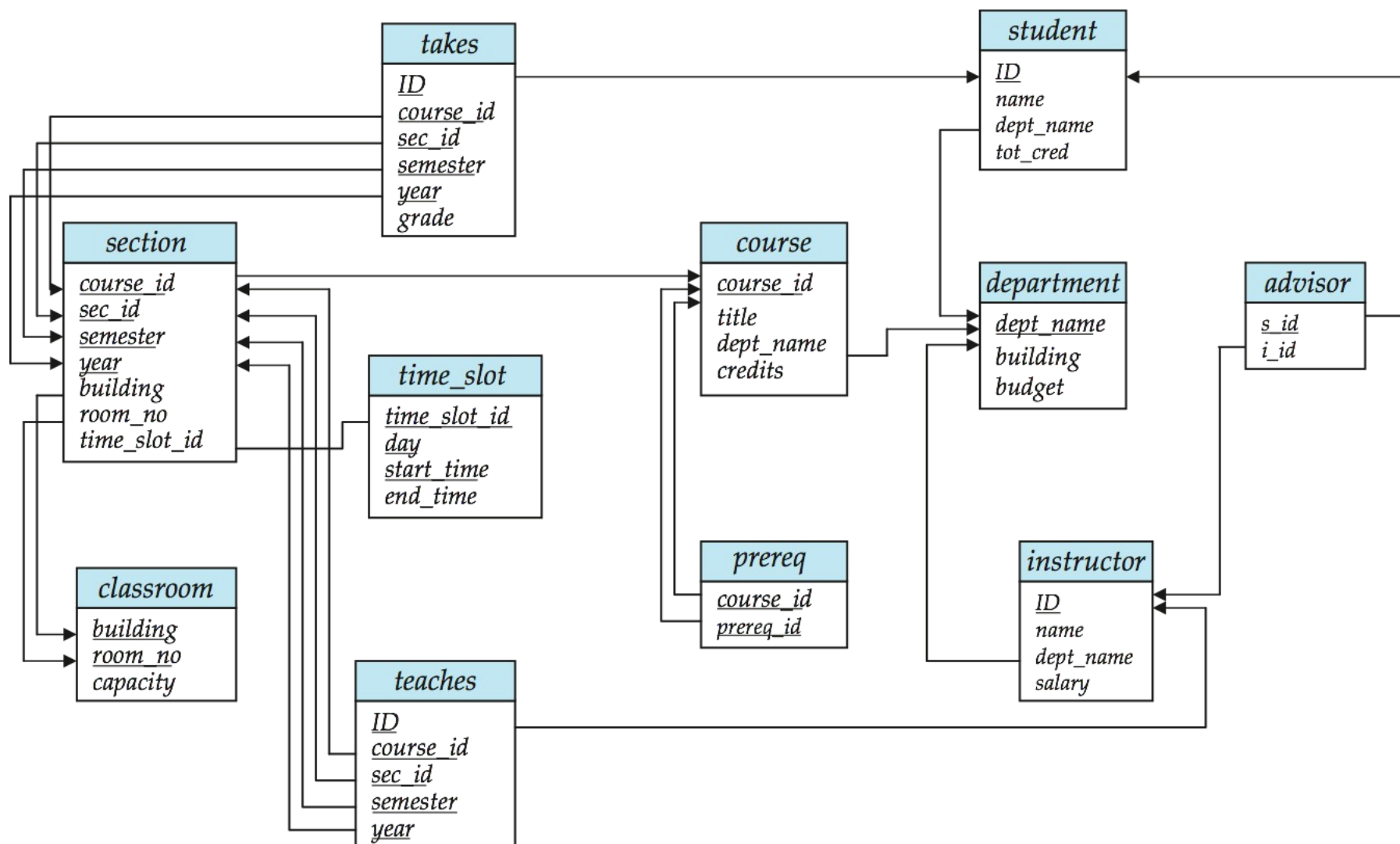


# Chapter 8: Relational Database Design

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
- Functional Dependency Theory
- Algorithms for Functional Dependencies
- Database-Design Process



# Schema Diagram for University Database





# Combine Schemas?

- Suppose we combine *instructor* and *department* into *inst\_dept*
- Result is possible repetition of information

<i>ID</i>	<i>name</i>	<i>salary</i>	<i>dept_name</i>	<i>building</i>	<i>budget</i>
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000



# What About Smaller Schemas?

- Suppose we had started with *inst\_dept*. How would we know to split up (**decompose**) it into *instructor* and *department*?
- Write a rule “if there were a schema (*dept\_name*, *building*, *budget*), then *dept\_name* would be a candidate key”
- Denote as a **functional dependency**:  
$$dept\_name \rightarrow building, budget$$
- In *inst\_dept*, because *dept\_name* is not a candidate key, the building and budget of a department may have to be repeated.
  - This indicates the need to decompose *inst\_dept*

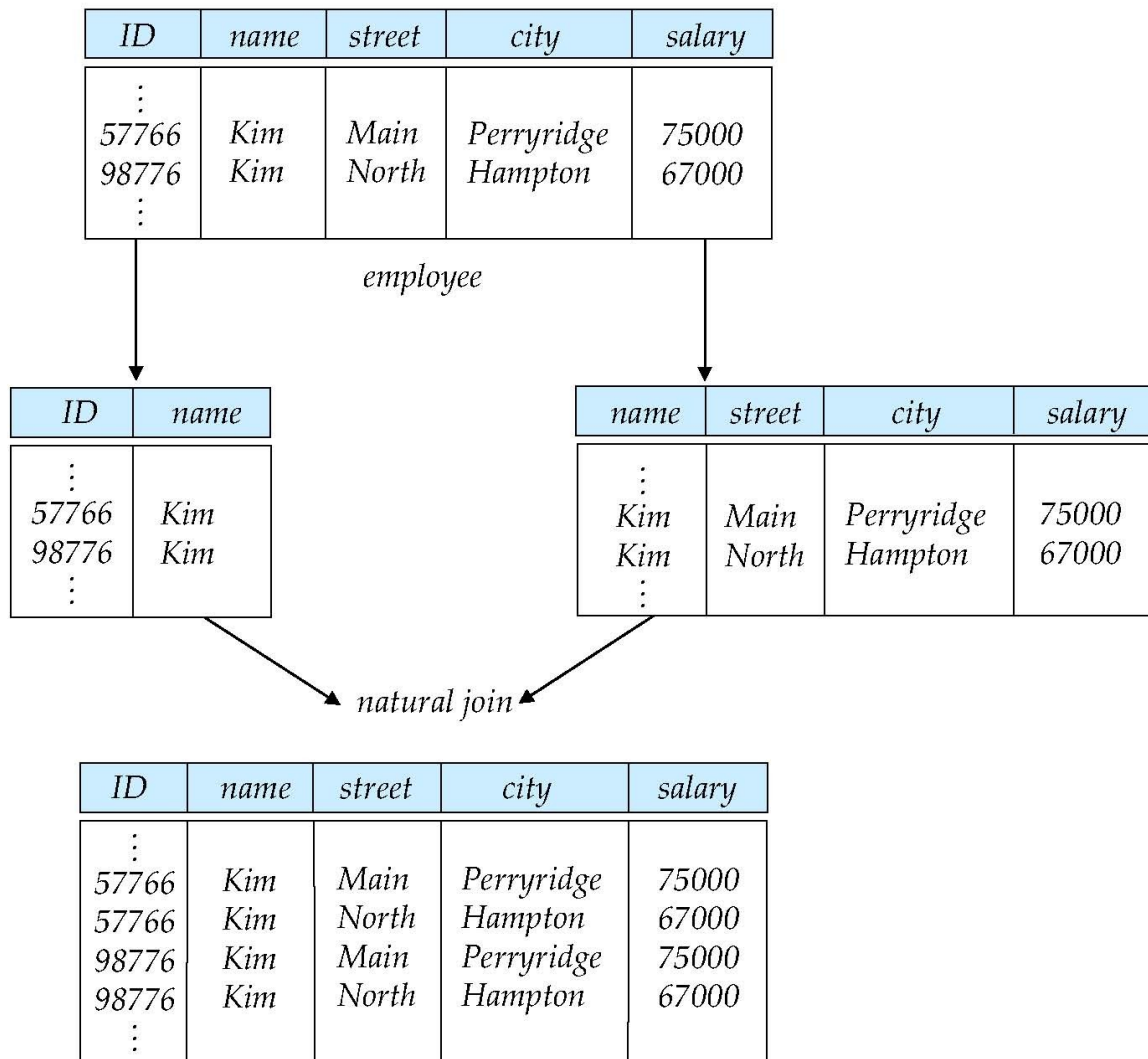


# Not all decompositions are good

- Suppose we decompose *employee*(*ID*, *name*, *street*, *city*, *salary*) into  
*employee1* (*ID*, *name*)  
*employee2* (*name*, *street*, *city*, *salary*)
- The next slide shows how we **lose information** -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.



# A Lossy Decomposition





# Example of Lossless-Join Decomposition

- **Lossless join decomposition**
- Decomposition of  $R = (A, B, C)$  into  
 $R_1 = (A, B)$  and  $R_2 = (B, C)$

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

$r$

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

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

B	C
1	A
2	B

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

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

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





# 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
  - Example: Set of accounts stored with each customer, and set of owners stored with each account
  - We assume all relations are in first normal form!



# First Normal Form (Cont'd)

- Atomicity is actually a *property of how the elements of the domain are used*.
  - Example: Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*
  - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
  - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.



# 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 (definition)

- Let  $R$  be a relation schema

$\alpha \subseteq R$  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:

*inst\_dept* ( $ID$ ,  $name$ ,  $salary$ ,  $dept\_name$ ,  $building$ ,  $budget$ ).

We expect these functional dependencies to hold:

$dept\_name \rightarrow building$

and  $ID \rightarrow building$

but would not expect the following to hold:

$dept\_name \rightarrow salary$



# Use of Functional Dependencies

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



# Functional Dependencies (Cont.)

- A functional dependency is **trivial** if it is satisfied by all instances of a relation
  - Example:
    - ▶  $ID, name \rightarrow ID$
    - ▶  $name \rightarrow name$
  - In general,  $\alpha \rightarrow \beta$  is trivial if  $\beta \subseteq \alpha$





# Closure of a Set of Functional Dependencies

- Given a set  $F$  of functional dependencies, there are certain other functional dependencies that are **logically implied by  $F$** .
  - For example: If  $A \rightarrow B$  and  $B \rightarrow C$ , then we can infer that  $A \rightarrow C$
- The set of **all** functional dependencies logically implied by  $F$  is the **closure** of  $F$ .
- We denote the *closure* of  $F$  by  **$F^+$** .
- $F^+$  is a superset of  $F$ .



# 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 schema *not* in BCNF:

*instr\_dept* (ID, name, salary, dept\_name, building, budget)

because *dept\_name*  $\rightarrow$  *building*, *budget*  
holds on *instr\_dept*, but *dept\_name* is not a superkey



# Decomposing a Schema into BCNF

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

We decompose  $R$  into:

- $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$
- In our example,
    - $\alpha = dept\_name$
    - $\beta = building, budget$and *inst\_dept* is replaced by
    - $(\alpha \cup \beta) = (dept\_name, building, budget)$
    - $(R - (\beta - \alpha)) = (ID, name, salary, dept\_name)$



# BCNF and Dependency Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation
- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving*.
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form*.



# 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).



# Goals of Normalization

- Let  $R$  be a relation scheme with a set  $F$  of functional dependencies.
- Decide whether a relation scheme  $R$  is in “good” form.
- In the case that a relation scheme  $R$  is not in “good” form, **decompose** it into a set of relation scheme  $\{R_1, R_2, \dots, R_n\}$  such that
  - each relation scheme is in **good** form
  - the decomposition is a **lossless-join decomposition**
  - Preferably, the decomposition should be **dependency preserving**.



# How good is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a relation

*inst\_info (ID, child\_name, phone)*

- where an instructor may have more than one phone and can have multiple children

<i>ID</i>	<i>child_name</i>	<i>phone</i>
99999	David	512-555-1234
99999	David	512-555-4321
99999	William	512-555-1234
99999	Willian	512-555-4321



# How good is BCNF? (Cont.)

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples

(99999, David, 981-992-3443)

(99999, William, 981-992-3443)





## How good is BCNF? (Cont.)

- Therefore, it is better to decompose *inst\_info* into:

<i>inst_child</i>	<i>ID</i>	<i>child_name</i>
	99999	David
	99999	David
	99999	William
	99999	Willian

<i>inst_phone</i>	<i>ID</i>	<i>phone</i>
	99999	512-555-1234
	99999	512-555-4321
	99999	512-555-1234
	99999	512-555-4321

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF) (not seen in this course).



# Functional-Dependency Theory

- We now consider the formal theory that tells us **which functional dependencies are implied logically by a given set of functional dependencies**.
- We then develop **algorithms** to generate lossless decompositions into BCNF and 3NF
- We then develop **algorithms** to test if a decomposition is dependency-preserving



# Closure of a Set of Functional Dependencies

- Given a set  $F$  set of functional dependencies, there are certain other functional dependencies that are logically implied by  $F$ .
  - For e.g.: If  $A \rightarrow B$  and  $B \rightarrow C$ , then we can infer that  $A \rightarrow C$
- The set of **all** functional dependencies logically implied by  $F$  is the **closure** of  $F$ .
- We denote the *closure* of  $F$  by  $F^+$ .



# How to obtain the closure?

- We can find  $F^+$  the closure of  $F$ , by repeatedly 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  $CG \rightarrow I$

●  $CG \rightarrow HI$

▶ by augmenting  $CG \rightarrow I$  to infer  $CG \rightarrow CGI$ ,  
and augmenting of  $CG \rightarrow H$  to infer  $CGI \rightarrow HI$ ,  
and then transitivity



# Procedure for Computing $F^+$

- To compute the closure  $F^+$  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 shall see an alternative procedure for this task later



# Closure of Functional Dependencies (Cont.)

- 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  $F$**
- 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  $\beta \subseteq \textit{result}$  then  $\textit{result} := \textit{result} \cup \gamma$   
    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?
  - Is  $AG$  a super key?  
Does  $AG \rightarrow R$ ?
  - Is any subset of  $AG$  a superkey?



# 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**. Examples:
  - $A \rightarrow C$  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

- **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  $\alpha$ 
  - compute  $(\{\alpha\} - A)^+$  using the dependencies in  $F$
  - check that  $(\{\alpha\} - A)^+$  contains  $\beta$ ; if it does,  $A$  is extraneous in  $\alpha$
- To test if attribute  $A \in \beta$  is extraneous in  $\beta$ 
  - compute  $\alpha^+$  using only the dependencies in
$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\},$$
  - check that  $\alpha^+$  contains  $A$ ; if it does,  $A$  is extraneous in  $\beta$



# 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  $\beta$   
/\* Note: test for extraneous attributes done using  $F_c$ , not  $F^*$  \*/
    - If an extraneous attribute is found, delete it from  $\alpha \rightarrow \beta$
  - until**  $F$  does not change



# Computing a Canonical Cover: example

- $R = (A, B, C)$   
 $F = \{A \rightarrow BC$   
     $B \rightarrow C$   
     $A \rightarrow B$   
     $AB \rightarrow C\}$
- 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:  $A \rightarrow B$   
     $B \rightarrow C$



# Lossless-join Decomposition

- For the case of  $R = (R_1, R_2)$ , we require that for all possible relations  $r$  on schema  $R$

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

- A decomposition of  $R$  into  $R_1$  and  $R_2$  is lossless join if at least one of the following dependencies is in  $F^+$ :
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies





# 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  $R_1 \bowtie R_2$ )



# Dependency Preservation

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

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

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

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



# Testing for Dependency Preservation

- To check if a dependency  $\alpha \rightarrow \beta$  is preserved in a decomposition of  $R$  into  $R_1, R_2, \dots, R_n$  we apply the following **test** (with attribute closure done with respect to  $F$ )
  - $result = \alpha$   
**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  $(F_1 \cup F_2 \cup \dots \cup F_n)^+$



# 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  $\alpha \rightarrow \beta$  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, **simplified test using only  $F$  is incorrect when testing a relation in a decomposition of  $R$** 
  - Consider  $R = (A, B, C, D, E)$ , with  $F = \{ A \rightarrow B, BC \rightarrow D \}$ 
    - ▶ Decompose  $R$  into  $R_1 = (A, B)$  and  $R_2 = (A, C, D, E)$
    - ▶ Neither of the dependencies in  $F$  contain only attributes from  $(A, C, D, E)$  so we might be misled into thinking  $R_2$  satisfies BCNF.
    - ▶ In fact, dependency  $AC \rightarrow D$  in  $F^+$  shows  $R_2$  is not in BCNF.



# 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
$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$
can be shown to hold on  $R_i$ , and  $R_i$  violates BCNF.
  - ▶ We use above dependency to decompose  $R_i$



# 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

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





# Example of BCNF Decomposition

- *class* (*course\_id*, *title*, *dept\_name*, *credits*, *sec\_id*, *semester*, *year*, *building*, *room\_number*, *capacity*, *time\_slot\_id*)
- Functional dependencies:
  - *course\_id* → *title*, *dept\_name*, *credits*
  - *building*, *room\_number* → *capacity*
  - *course\_id*, *sec\_id*, *semester*, *year* → *building*, *room\_number*, *time\_slot\_id*
- A candidate key {*course\_id*, *sec\_id*, *semester*, *year*}.
- BCNF Decomposition:
  - *course\_id* → *title*, *dept\_name*, *credits* holds
    - ▶ but *course\_id* is not a superkey.
  - We replace *class* by:
    - ▶ *course*(*course\_id*, *title*, *dept\_name*, *credits*)
    - ▶ *class-1* (*course\_id*, *sec\_id*, *semester*, *year*, *building*, *room\_number*, *capacity*, *time\_slot\_id*)



# BCNF Decomposition (Cont.)

- *course* is in BCNF
  - How do we know this?
- *building, room\_number* → *capacity* holds on *class-1*
  - but {*building, room\_number*} is not a superkey for *class-1*.
  - We replace *class-1* by:
    - ▶ *classroom* (*building, room\_number, capacity*)
    - ▶ *section* (*course\_id, sec\_id, semester, year, building, room\_number, time\_slot\_id*)
- *classroom* and *section* are in BCNF.



# BCNF and Dependency Preservation

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

■ *Example:*

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

Two candidate keys =  $JK$  and  $JL$

■  $R$  is not in BCNF

■ Any decomposition of  $R$  will fail to preserve

$JK \rightarrow L$

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



# Third Normal Form: Motivation

- There are some situations where
  - BCNF is not dependency preserving, and
  - efficient checking for FD violation on updates is important
- Solution: define a weaker normal form, called Third Normal Form (3NF)
  - Allows some redundancy (with resultant problems; we will see examples later)
  - But functional dependencies can be checked on individual relations without computing a join.
  - There is always a lossless-join, dependency-preserving decomposition into 3NF.



# 3NF Example

## ■ Relation *dept\_advisor*:

- *dept\_advisor* (*s\_ID*, *i\_ID*, *dept\_name*)
- $F = \{s\_ID, dept\_name \rightarrow i\_ID, i\_ID \rightarrow dept\_name\}$
- Two candidate keys: *s\_ID*, *dept\_name*, and *i\_ID*, *s\_ID*
- *R* is in 3NF
  - ▶  $s\_ID, dept\_name \rightarrow i\_ID$  *s\_ID*
    - *dept\_name* is a superkey
  - ▶  $i\_ID \rightarrow dept\_name$ 
    - *dept\_name* is contained in a candidate key



# Redundancy in 3NF

- There is some redundancy in this schema
- Example of problems due to redundancy in 3NF

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

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

- repetition of information (e.g., the relationship  $l_1, k_1$ )
  - $(i\_ID, dept\_name)$
- need to use null values (e.g., to represent the relationship  $l_2, k_2$  where there is no corresponding value for  $J$ ).
  - $(i\_ID, dept\_name)$  if there is no separate relation mapping instructors to departments



# 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  
  /* Optionally, remove redundant relations */  
  repeat  
    if any schema  $R_j$  is contained in another schema  $R_k$   
    then /* delete  $R_j$  */  
       $R_j = R_k$ ;;  
       $i = i - 1$ ;  
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 presentation ([click here](#))



# 3NF Decomposition: An Example

- Relation schema:

*cust\_banker\_branch = (customer\_id, employee\_id, branch\_name, type )*

- The functional dependencies for this relation schema are:

- *customer\_id, employee\_id → branch\_name, type*
- *employee\_id → branch\_name*
- *customer\_id, branch\_name → employee\_id*

- We first compute a canonical cover

- *branch\_name* is extraneous in the r.h.s. of the 1<sup>st</sup> dependency
- No other attribute is extraneous, so we get  $F_c =$

*customer\_id, employee\_id → type*

*employee\_id → branch\_name*

*customer\_id, branch\_name → employee\_id*



# 3NF Decomposition Example (Cont.)

- The **for** loop generates following 3NF schema:

*(customer\_id, employee\_id, type )*

*(employee\_id, branch\_name)*

*(customer\_id, branch\_name, employee\_id)*

- Observe that *(customer\_id, employee\_id, type )* contains a candidate key of the original schema, so no further relation schema needs be added
- At end of for loop, detect and delete schemas, such as *(employee\_id, branch\_name)*, which are subsets of other schemas
  - result will not depend on the order in which FDs are considered
- The resultant simplified 3NF schema is:

*(customer\_id, employee\_id, type)*

*(customer\_id, branch\_name, employee\_id)*



# Comparison of BCNF and 3NF

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



# Design Goals

- Goal for a relational database design is:
  - BCNF.
  - Lossless join.
  - Dependency preservation.
- If we cannot achieve this, we accept one of
  - Lack of dependency preservation
  - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.

Can specify FDs using assertions, but they are expensive to test, (and currently not supported by any of the widely used databases!)
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.



# 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 functional dependencies from non-key attributes of an entity to other attributes of the entity
  - Example: an *employee* entity with attributes *department\_name* and *building*, and a functional dependency  $department\_name \rightarrow building$
  - Good design would have made department an entity
- Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary



# Denormalization for Performance

- May want to use non-normalized schema for performance
- For example, displaying *prereqs* along with *course\_id*, and *title* requires join of *course* with *prereq*
- Alternative 1: Use **denormalized relation** containing attributes of *course* as well as *prereq* 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  
*course* (natural join) *prereq*
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors





# Chapter 8: Relational Database Design

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
- Functional Dependency Theory
- Algorithms for Functional Dependencies
- Database-Design Process



# End of Chapter

**Database System Concepts, 6<sup>th</sup> Ed.**

©Silberschatz, Korth and Sudarshan

See [www.db-book.com](http://www.db-book.com) for conditions on re-use