



Chapter 7: Normalization

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Outline

- Features of Good Relational Design
- Functional Dependencies
- Decomposition Using Functional Dependencies
- Normal Forms
- Functional Dependency Theory
- Algorithms for Decomposition using Functional Dependencies
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Atomic Domains and First Normal Form
- Database-Design Process
- Modeling Temporal Data



Features of Good Relational Designs

- Suppose we combine *instructor* and *department* into *in_dep*, which represents the natural join on the relations *instructor* and *department*

<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

- There is repetition of information
- Need to use null values (if we add a new department with no instructors)



Decomposition

- The only way to avoid the repetition-of-information problem in the *in_dep* schema is to decompose it into two schemas – *instructor* and *department* schemas.
- Not all decompositions are good. Suppose we decompose

employee(*ID*, *name*, *street*, *city*, *salary*)

into

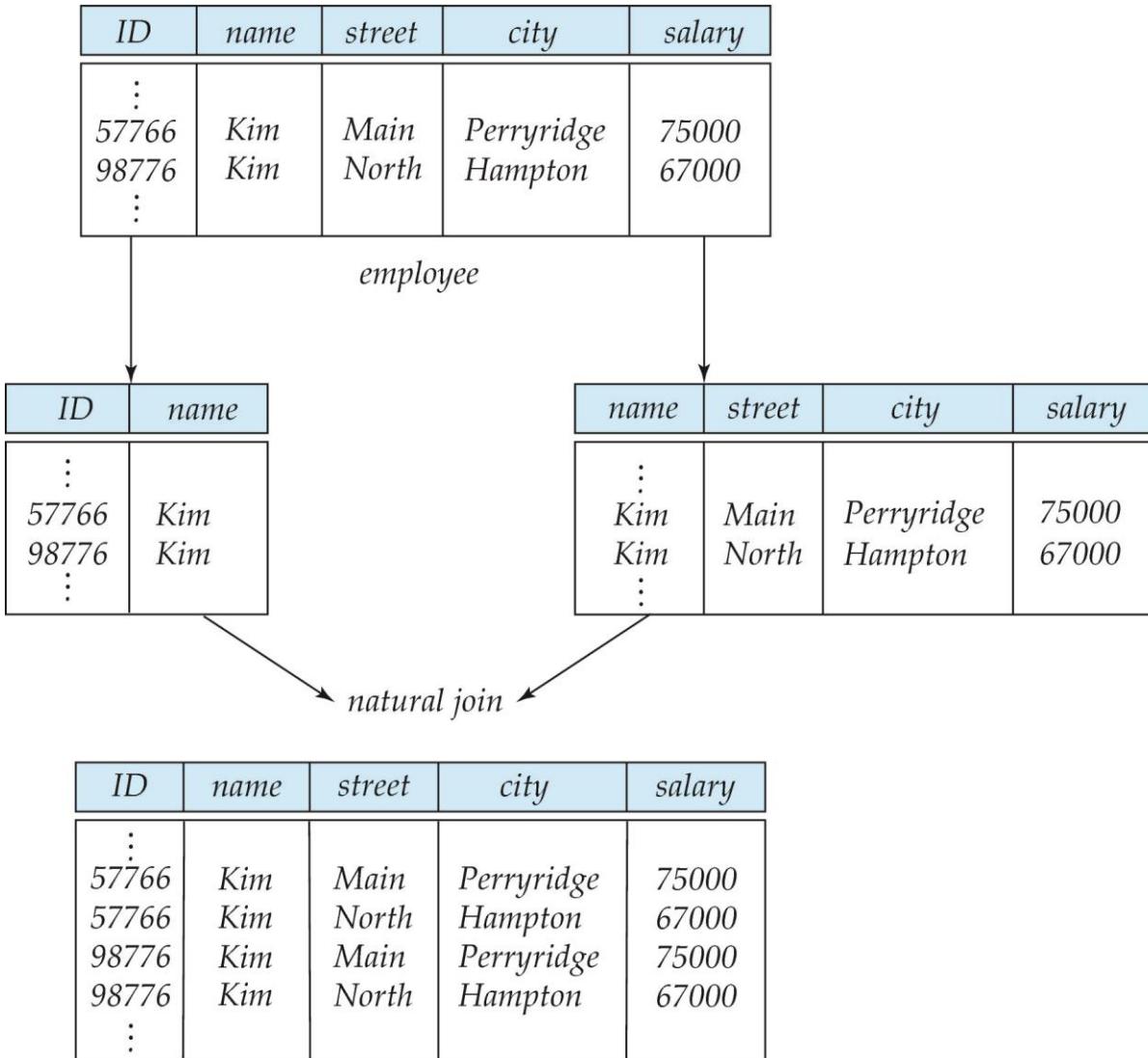
employee1 (*ID*, *name*)

employee2 (*name*, *street*, *city*, *salary*)

- The problem arises when we have two employees with the same name
- 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





Lossless Decomposition

- Let R be a relation schema and let R_1 and R_2 form a decomposition of R . That is $R = R_1 \cup R_2$
- We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing R with the two relation schemas $R_1 \cup R_2$
- Formally,

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

- And, conversely a decomposition is lossy if

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



Example of Lossless Decomposition

- Decomposition of $R = (A, B, C)$

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

A	B	C
α	1	A

β

1	B
---	---

r

A	B
α	1

β

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
α	1	A

β

2	B
---	---



Normalization Theory

- 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 set of relations $\{R_1, R_2, \dots, R_n\}$ such that
 - Each relation is in good form
 - The decomposition is a lossless decomposition
- Our theory is based on:
 - Functional dependencies
 - Multivalued dependencies



Functional Dependencies

- There are usually a variety of constraints (rules) on the data in the real world.
- For example, some of the constraints that are expected to hold in a university database are:
 - Students and instructors are uniquely identified by their ID.
 - Each student and instructor has only one name.
 - Each instructor and student is (primarily) associated with only one department.
 - Each department has only one value for its budget, and only one associated building.



Functional Dependencies (Cont.)

- An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation;
- A legal instance of a database is one where all the relation instances are legal instances
- 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

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

- 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 α , they also agree on the attributes β . That is,

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

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

1	4
1	5
3	7

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



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 .
 - If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
 - etc.
- The set of **all** functional dependencies logically implied by F is the **closure** of F .
- We denote the *closure* of F by F^* .



Keys and Functional Dependencies

- 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:

in_dep (ID, name, salary, dept_name, building, budget).

We expect these functional dependencies to hold:

$dept_name \rightarrow building$

$ID \rightarrow building$

but would not expect the following to hold:

$dept_name \rightarrow salary$



Use of Functional Dependencies

- We use functional dependencies to:
 - 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 .
 - To 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 $name \rightarrow ID$.



Trivial Functional Dependencies

- 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 $\beta \subseteq \alpha$



Lossless Decomposition

- We can use functional dependencies to show when certain decompositions are lossless.
- 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 decomposition if at least one of the following dependencies is in F^\dagger :
 - $R_1 \cap R_2 \rightarrow R_1$
 - $R_1 \cap R_2 \rightarrow R_2$
- 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

- $R = (A, B, C)$
 $F = \{A \rightarrow B, B \rightarrow C\}$
- $R_1 = (A, B), R_2 = (B, C)$
 - Lossless decomposition:

$$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$

- $R_1 = (A, B), R_2 = (A, C)$
 - Lossless decomposition:

$$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

- Note:
 - $B \rightarrow BC$
is a shorthand notation for
 - $B \rightarrow \{B, C\}$



Dependency Preservation

- Testing functional dependency constraints each time the database is updated can be costly
- It is useful to design the database in a way that constraints can be tested efficiently.
- If testing a functional dependency can be done by considering just one relation, then the cost of testing this constraint is low
- When decomposing a relation it is possible that it is no longer possible to do the testing without having to perform a Cartesian Product.
- A decomposition that makes it computationally hard to enforce functional dependency is said to be NOT **dependency preserving**.



Dependency Preservation Example

- Consider a schema:
 $\text{dept_advisor}(s_ID, i_ID, \text{department_name})$
- With function dependencies:
 $i_ID \rightarrow \text{dept_name}$
 $s_ID, \text{dept_name} \rightarrow i_ID$
- In the above design we are forced to repeat the department name once for each time an instructor participates in a *dept_advisor* relationship.
- To fix this, we need to decompose *dept_advisor*
- Any decomposition will not include all the attributes in
 $s_ID, \text{dept_name} \rightarrow i_ID$
- Thus, the composition NOT be dependency preserving



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., $\beta \subseteq \alpha$)
- α is a superkey for R



Boyce-Codd Normal Form (Cont.)

- Example schema that is *not* in BCNF:

in_dep (ID, name, salary, dept_name, building, budget)

because :

- - $\text{dept_name} \rightarrow \text{building}, \text{budget}$
 - holds on *in_dep*
 - but
 - *dept_name* is not a superkey
- When decompose *in_dept* into *instructor* and *department*
 - *instructor* is in BCNF
 - *department* is in BCNF



Decomposing a Schema into BCNF

- Let R be a schema R that is not in BCNF. Let $\alpha \rightarrow \beta$ be the FD that causes a violation of BCNF.
- We decompose R into:
 - $(\alpha \cup \beta)$
 - $(R - (\beta - \alpha))$
- In our example of in_dep ,
 - $\alpha = dept_name$
 - $\beta = building, budget$and in_dep is replaced by
 - $(\alpha \cup \beta) = (dept_name, building, budget)$
 - $(R - (\beta - \alpha)) = (ID, name, dept_name, salary)$



Example

- $R = (A, B, C)$
 $F = \{A \rightarrow B, B \rightarrow C\}$
- $R_1 = (A, B), R_2 = (B, C)$
 - 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 $R_1 \bowtie R_2$)



BCNF and Dependency Preservation

- It is not always possible to achieve both BCNF and dependency preservation
- Consider a schema:
$$\text{dept_advisor}(s_ID, i_ID, \text{department_name})$$
- With function dependencies:
$$i_ID \rightarrow \text{dept_name}$$

$$s_ID, \text{dept_name} \rightarrow i_ID$$
- dept_advisor is not in BCNF
 - i_ID is not a superkey.
- Any decomposition of dept_advisor will not include all the attributes in
$$s_ID, \text{dept_name} \rightarrow i_ID$$
- Thus, the composition is NOT be dependency preserving



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., $\beta \in \alpha$)
- α 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 Example

- Consider a schema:
 $dept_advisor(s_ID, i_ID, dept_name)$
- With function dependencies:
 $i_ID \rightarrow dept_name$
 $s_ID, dept_name \rightarrow i_ID$
- Two candidate keys = $\{s_ID, dept_name\}$, $\{s_ID, i_ID\}$
- We have seen before that $dept_advisor$ is not in BCNF
- R , however, is in 3NF
 - $s_ID, dept_name$ is a superkey
 - $i_ID \rightarrow dept_name$ and i_ID is NOT a superkey, but:
 - $\{ dept_name \} - \{ i_ID \} = \{ dept_name \}$ and
 - $dept_name$ is contained in a candidate key



Redundancy in 3NF

- Consider the schema R below, which is in 3NF

- $R = (J, K, L)$
- $F = \{JK \rightarrow L, L \rightarrow K\}$
- And an instance table:

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

- What is wrong with the table?
 - Repetition of information
 - Need to use null values (e.g., to represent the relationship l_2, k_2 where there is no corresponding value for J)



Comparison of BCNF and 3NF

- Advantages to 3NF over BCNF. It is always possible to obtain a 3NF design without sacrificing losslessness or dependency preservation.
- Disadvantages to 3NF.
 - We may have to use null values to represent some of the possible meaningful relationships among data items.
 - There is the problem of repetition of information.



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, need to 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 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
- Instance of *inst_info*

<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	William	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)



Higher Normal Forms

- It is better to decompose *inst_info* into:
 - *inst_child*:

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

- *inst_phone*:

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

- This suggests the need for higher normal forms, such as Fourth Normal Form (4NF), which we shall see later



Functional-Dependency Theory Roadmap

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



Closure of a Set of Functional Dependencies

- We can compute F^+ , the closure of F , by repeatedly applying **Armstrong's Axioms**:
 - **Reflexive rule:** if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$
 - **Augmentation rule:** if $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$
 - **Transitivity rule:** if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$
- These rules are
 - **Sound** -- generate only functional dependencies that actually hold, and
 - **Complete** -- generate all functional dependencies that hold.



Example of F^\dagger

- $R = (A, B, C, G, H, I)$
 $F = \{ A \rightarrow B$
 $\quad A \rightarrow C$
 $\quad CG \rightarrow H$
 $\quad CG \rightarrow I$
 $\quad B \rightarrow H \}$
- Some members of F^\dagger
 - $A \rightarrow H$
 - 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



Closure of Functional Dependencies (Cont.)

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



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 shall see an alternative procedure for this task later



Closure of Attribute Sets

- Given a set of attributes α , define the ***closure*** of α **under** F (denoted by α^+) as the set of attributes that are functionally determined by α under F
- Algorithm to compute α^+ , the closure of α under F

```
result :=  $\alpha$ ;  
while (changes to result) do  
  for each  $\beta \rightarrow \gamma$  in  $F$  do  
    begin  
      if  $\beta \subseteq result$  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 $AG \rightarrow R?$ == Is $R \supseteq (AG)^+$
 2. Is any subset of AG a superkey?
 1. Does $A \rightarrow R?$ == Is $R \supseteq (A)^+$
 2. Does $G \rightarrow R?$ == Is $R \supseteq (G)^+$
 3. In general: check for each subset of size $n-1$



Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
 - To test if α is a superkey, we compute α^+ , and check if α^+ 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 α^+ by using attribute closure, and then check if it contains β .
 - Is a simple and cheap test, and very useful
- Computing closure of F
 - For each $\gamma \subseteq R$, we find the closure γ^+ , and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.



Canonical Cover

- Suppose that we have a set of functional dependencies F on a relation schema. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies; that is, all the functional dependencies in F are satisfied in the new database state.
- If an update violates any functional dependencies in the set F , the system must roll back the update.
- We can reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set.
- This simplified set is termed the **canonical cover**
- To define canonical cover we must first define **extraneous attributes**.
 - An attribute of a functional dependency in F is **extraneous** if we can remove it without changing F^+



Extraneous Attributes

- Removing an attribute from the left side of a functional dependency could make it a stronger constraint.
 - For example, if we have $AB \rightarrow C$ and remove B, we get the possibly **stronger** result $A \rightarrow C$. It may be stronger because $A \rightarrow C$ logically implies $AB \rightarrow C$, but $AB \rightarrow C$ does not, on its own, logically imply $A \rightarrow C$
- But, depending on what our set F of functional dependencies happens to be, we may be able to remove B from $AB \rightarrow C$ safely.
 - For example, suppose that
 - $F = \{AB \rightarrow C, A \rightarrow D, D \rightarrow C\}$
 - Then we can show that F logically implies $A \rightarrow C$, making extraneous in $AB \rightarrow C$.



Extraneous Attributes (Cont.)

- Removing an attribute from the right side of a functional dependency could make it a weaker constraint.
 - For example, if we have $AB \rightarrow CD$ and remove C, we get the possibly **weaker** result $AB \rightarrow D$. It may be weaker because using just $AB \rightarrow D$, we can no longer infer $AB \rightarrow C$.
- But, depending on what our set F of functional dependencies happens to be, we may be able to remove C from $AB \rightarrow CD$ safely.
 - For example, suppose that
$$F = \{ AB \rightarrow CD, A \rightarrow C \}.$$
 - Then we can show that even after replacing $AB \rightarrow CD$ by $AB \rightarrow D$, we can still infer $\$AB \rightarrow C$ and thus $AB \rightarrow CD$.



Extraneous Attributes

- An attribute of a functional dependency in F is **extraneous** if we can remove it without changing F^+
- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F .
 - **Remove from the left side:** Attribute A is **extraneous** in α if
 - $A \in \alpha$ and
 - F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
 - **Remove from the right side:** Attribute A is **extraneous** in β if
 - $A \in \beta$ and
 - The set of functional dependencies
$$(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$
 logically implies F .
- *Note:* implication in the opposite direction is trivial in each of the cases above, since a “stronger” functional dependency always implies a weaker one



Testing if an Attribute is Extraneous

- Let R be a relation schema and let F be a set of functional dependencies that hold on R . Consider an attribute in the functional dependency $\alpha \rightarrow \beta$.
- To test if attribute $A \in \beta$ is extraneous in β
 - Consider the set:
$$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\},$$
 - check that α^+ contains A ; if it does, A is extraneous in β
- To test if attribute $A \in \alpha$ is extraneous in α
 - Let $\gamma = \alpha - \{A\}$. Check if $\gamma \rightarrow \beta$ can be inferred from F .
 - Compute γ^+ using the dependencies in F
 - If γ^+ includes all attributes in β then , A is extraneous in α



Examples of Extraneous Attributes

- Let $F = \{AB \rightarrow CD, A \rightarrow E, E \rightarrow C\}$
- To check if C is extraneous in $AB \rightarrow CD$, we:
 - Compute the attribute closure of AB under $F = \{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$
 - The closure is $ABCDE$, which includes CD
 - This implies that C 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. That is, there are no two dependencies in F_c
 - $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ such that
 - $\alpha_1 = \alpha_2$



Canonical Cover

- To compute a canonical cover for F :

repeat

 Use the **union** rule to replace any dependencies in F of the form

$$\alpha_1 \rightarrow \beta_1 \text{ and } \alpha_1 \rightarrow \beta_2 \text{ with } \alpha_1 \rightarrow \beta_1 \beta_2$$

 Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an **extraneous** attribute either in α or in β

 /* Note: test for extraneous attributes done using F_c , not F^* */

 If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$

until (F_c not change)

- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied



Example: Computing a Canonical Cover

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



Dependency Preservation

- Let F_i be the set of dependencies F^+ that include **only attributes in R_i** .
 - A decomposition is **dependency preserving**, if
$$(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$$
- Using the above definition, testing for dependency preservation take exponential time.
- Note that if a decomposition is NOT dependency preserving then checking updates for violation of functional dependencies may require computing joins, which is expensive.



Dependency Preservation (Cont.)

- Let F be the set of dependencies on schema R and let R_1, R_2, \dots, R_n be a decomposition of R .
- The restriction of F to R_i is the set F_i of all functional dependencies in F^+ that include **only** attributes of R_i .
- Since all functional dependencies in a restriction involve attributes of only one relation schema, it is possible to test such a dependency for satisfaction by checking only one relation.
- Note that the definition of restriction uses all dependencies in F^+ , not just those in F .
- The set of restrictions F_1, F_2, \dots, F_n is the set of functional dependencies that can be checked efficiently.



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)
 - $result = \alpha$
 - repeat
 - for each R_i in the decomposition
 - $t = (result \cap R_i)^+ \cap R_i$
 - $result = result \cup t$
 - until ($result$ does not change)
 - If $result$ contains all attributes in β , 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 α^+ (the attribute closure of α), 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 α^+ (the attribute closure of α) 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

- $\text{class}(\text{course_id}, \text{title}, \text{dept_name}, \text{credits}, \text{sec_id}, \text{semester}, \text{year}, \text{building}, \text{room_number}, \text{capacity}, \text{time_slot_id})$
- Functional dependencies:
 - $\text{course_id} \rightarrow \text{title}, \text{dept_name}, \text{credits}$
 - $\text{building}, \text{room_number} \rightarrow \text{capacity}$
 - $\text{course_id}, \text{sec_id}, \text{semester}, \text{year} \rightarrow \text{building}, \text{room_number}, \text{time_slot_id}$
- A candidate key $\{\text{course_id}, \text{sec_id}, \text{semester}, \text{year}\}$.
- BCNF Decomposition:
 - $\text{course_id} \rightarrow \text{title}, \text{dept_name}, \text{credits}$ holds
 - but course_id is **not** a superkey.
 - We replace class by:
 - $\text{course}(\text{course_id}, \text{title}, \text{dept_name}, \text{credits})$
 - $\text{class-1}(\text{course_id}, \text{sec_id}, \text{semester}, \text{year}, \text{building}, \text{room_number}, \text{capacity}, \text{time_slot_id})$



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.



Third Normal Form

- 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*

- $\text{dept_advisor} (s_ID, i_ID, \text{dept_name})$
 $F = \{s_ID, \text{dept_name} \rightarrow i_ID, i_ID \rightarrow \text{dept_name}\}$
- Two candidate keys: $s_ID, \text{dept_name}$, and i_ID, s_ID
- R is in 3NF
 - $s_ID, \text{dept_name} \rightarrow i_ID$
 - $s_ID, \text{dept_name}$ is a superkey
 - $i_ID \rightarrow \text{dept_name}$
 - dept_name is contained in a candidate key



Testing for 3NF

- 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 α is a superkey.
- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of R
 - This test is rather more expensive, since it involves 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 */

$R_j = R_p$;

$i = i - 1$;

return (R_1, R_2, \dots, R_i)

Back to the example:

$f_1: i_ID \rightarrow dept_name$

$f_2: s_ID, dept_name \rightarrow i_ID$

There are no extraneous attributes in any of the functional dependencies in F , so F_c contains f_1 and f_2 . The algorithm then generates as R_1 the schema, $(i_ID \text{ dept_name})$, and as R_2 the schema $(s_ID, \text{dept_name}, i_ID)$. The algorithm then finds that R_2 contains a candidate key, so no further relation schema is created.



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:

$\text{cust_banker_branch} = (\underline{\text{customer_id}}, \underline{\text{employee_id}}, \text{branch_name}, \text{type})$

- The functional dependencies for this relation schema are:

- $\text{customer_id}, \text{employee_id} \rightarrow \text{branch_name}, \text{type}$
- $\text{employee_id} \rightarrow \text{branch_name}$
- $\text{customer_id}, \text{branch_name} \rightarrow \text{employee_id}$

- We first compute a canonical cover

- branch_name is extraneous in the r.h.s. of the 1st dependency
- No other attribute is extraneous, so we get $F_C =$

$\text{customer_id}, \text{employee_id} \rightarrow \text{type}$
 $\text{employee_id} \rightarrow \text{branch_name}$
 $\text{customer_id}, \text{branch_name} \rightarrow \text{employee_id}$



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.