# Normalization

## 1 2 3 Normal Forms

# Schema Refinement

- Redundancy
- Schema Refinement
  - Minimizing Redundancy
  - Functional Dependencies (FDs)
  - Normalization using FDs
    - First Normal Form (1NF)
    - Second Normal Form (2NF)
    - Third Normal Form (3NF)
    - Boyce-Codd Normal Form (BCNF)
  - Multivalued Dependencies (MVDs), Join Dependencies JDs)
  - Normalization using MVDs and JDs
    - Higher Normal Forms ( 4NF, 5NF)

# Normal Forms

# 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 (and revisit this in Chapter 22: Object Based Databases)

# First Normal Form (Cont.)

- 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.

# 1NF Summarized

- Each attribute must be atomic (single value)
  - No repeating columns within a row (composite attributes)
  - No multi-valued columns.

  All key attributes defined

  All attributes dependent on primary key

- 1NF simplifies attributes
  - Queries become easier.

# Dependencies and Normal Form

1 2 3

- The schema R is in 1NF if domain of all attributes of R is atomic
- Desirable
  - PK dependencies
- Undesirable
  - Partial Key : based on part of composite PK
  - Transitive: one nonprime attribute depends on another nonprime attribute

# Normal Forms

- Scalar Values
  - 1 NF
- Normal Forms based on PK
  - 2 NF
  - 3 NF
- Normal Forms based on CKs
  - Boyce-Codd Normal Form (BCNF)
- Other Normal Forms
  - 4 NF (Multivalued Dependencies)
  - 5 NF (Join Dependencies)
  - Deal with very rare practical situations

# Normal Forms

- 1NF : A relation is in a first normal form, if every tuple contains exactly one value for each attribute
- Example: Suppliers1 ( S# , p#, status, city, qty)
- Primary key ( S#, P#)
- City→ status (FD)
- FD Diagram (FDD) can be drawn
- Observe
  - Status depends on the city (FD)
  - City depends only on the supplier number

- Primary Key: (s#,p#) -> qty
- Partial: s#-> city
- Transitive: S#->city, city->status implies s#-> status

# Redundancies in Suppliers1

Suppliers1 ( **<u>s#</u>** , <u>p#</u>, status, city, qty)

FD city → status

- **<u>Insert</u>:** We cannot insert the fact that a particular supplier is located in the city unless he/she actually supplies some part
- The Suppliers1 may not show a supplier is located in Chennai
- **Delete:** If we delete a sole Suppliers1 tuple for a particular supplier, we also delete that he/she is located in a particular city
- **<u>Update:</u>** to update the city for a particular supplier

FD

N

1NF

# Schema Refinement for Suppliers1

- Refined Schema Suppliers1
  - Suppliers2 ( <u>S#</u>, status, city)
    - Primary : s#-> city
    - Transitive: s#->city, city->status implies s#->status
  - Suppliers_Parts ( <u>S#, P#,</u> qty)
    - Primary: (s#,p#) -> qty
- Insert: We can insert the info that s-5 is located in Pune although s-5 does not currently supply any part
- Delete: We do not lose the info that s-18 is located in Surat even if all the s-18 tuples from the suppliers_parts are deleted
- Update: city for a supplier can be updated easily through suppliers2

# Second Normal Form 2NF

- A relation is in 2NF only if it is in 1NF and every nonkey attribute is irreducibly dependent on the primary key.

  (here, we are assuming only single candidate ( hence primary ) key case)
- The original relation ***Suppliers1*** can be converted to the 2NF form by taking projection of it to a set of 2 relations ***suppliers2*** and ***suppliers_parts***

- All relations with single attribute PK are in 2 NF!!
- 2NF applies to relations with composite keys

# 2 NF

- A relation that is in 1NF & every non-PK attribute is fully dependent on the PK, is said to be in 2 NF

1 NF $\xrightarrow{\textit{Remove all Partial Dependencies}}$ 2 NF

# 2NF Problems: Suppliers2

Suppliers2 ( <u>S#</u>, status, city)

- Primary : s#-> city
- Transitive: s#->city, city->status implies s#->status

- Insert: we cannot insert the fact that a particular city has a particular status unless we have some supplier located in that city
- Delete: if we delete a sole tuple for a particular city we delete the information for the supplier and also the information that a city has a particular status
- Update: the status for a city appears many a times in the suppliers2. The change in the Mumbai status from 80 to 90 may need changes in 100 tuples

# Schema Refinement for Suppliers2

FD

- Replace Suppliers2 (s#,city, status) by
  - Suppliers3 ( S#, city )
  - City_Info ( city, status)
- **3NF :** A relation is in a 3NF if it is in 2NF and every nonkey attribute is notransitively dependent on the primary key
- A relation is in a 3NF if nonkey attributes are:
  - Mutually independent
  - Irreducibly/nontransitively dependent on the primary key
- A nonkey attribute is any attribute that does not participate in the primary key of that relation

# 3 NF

- A relation that is in 1NF & 2 NF & no non-PK attribute is transitively dependent on the PK, is said to be in 3 NF

$$\textbf{2 NF} \xrightarrow{\textit{Remove all Transitive Dependencies}} \textbf{3 NF}$$

# Heath's Theorem

⋈

- R(A,B,C) where A B C are sets of attributes
- If R has FD A --> B, then R equals join of {A B} and {A C}

{B C} ⋈ {A C}

- Example:

**S ( <u>s#,</u> status, city) ;**

PK: s# implies s# --> city and s# --> status

FD: City --> status is troublesome ( out of non-candidate key)

Then decomposition

(s#,city), (status, city) is **Dependency Preserving**

A B

# Heath's Theorem

- it says that a relation *R* over an attribute set *U* and satisfying a functional dependency $X \rightarrow Y$ can be safely split in two relations having the lossless-join decomposition property, namely into

$$\pi_{XY}(R) \bowtie \pi_{XZ}(R) = R$$

- where $Z = U - XY$ are the rest of the attributes.
- functional dependency provides a simple way to construct a lossless-join decomposition of *R* in two smaller relations

# Normal Forms

- Returning to the issue of schema refinement, the first question to be asked is whether any refinement is needed!
- If a relation is in a certain *normal form* (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized. This can be used to help us decide whether decomposing the relation will help.
- Role of FDs in detecting redundancy:
  - Consider a relation R with 3 attributes, ABC.
    - No FDs hold: There is no redundancy here.
    - Given $A \rightarrow B$: Several tuples could have the same A value, and if so, they'll all have the same B value!

# Boyce-Codd Normal Form BCNF (1974)

- 2 or more candidate keys
- Composite candidate keys
- Overlapped keys

- A relation is in BCNF if and only if only determinants are candidate keys

# BCNF

- Based on FDs that take into account all candidate keys of a relation
- For a relation with only 1 CK (PK), 3NF & BCNF are equivalent
- A relation is said to be in BCNF if every determinant is a CK

# BCNF

- Relation Suppliers1 ( **s#,** status, city, **p#,** qty) is not in BCNF. It has 3 determinants s#, city and (s#,p#)
- Primary Key: (s#,p#) -> qty
- Partial: s#-> city
- Transitive: S#->city, city->status implies s#-> status

- Relation Suppliers2( s#, status, city) is not in BCNF. It has s# and city as determinants where city is nonPK attribute

- But SP (s#,p#,qty), SC (s# ,city) and CS ( city,status) are in BCNF

# Nonoverlapping candidate keys

S( s#, sname,city ,status)

–s# and sname are candidate keys and city->status no longer holds

CK:

– s#-> sname, city, status

–sname-> s#, city, status

–s# <-> sname

–S is in BCNF

# Overlapping candidate keys

SSP( s#, sname, p#, qty) is not in BCNF

CK:

– (s#,p#)->qty

– (sname,p#)-> qty

– s# <-> sname

X

– Redudndancies: same (supplier number,name) pair repeated for various parts that the supplier is supplying

1 – SS( s#, sname) , SP( s#,p#,qty)

2 SS, SP(Sname, p#, qty)

# 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 $R$

# Boyce-Codd Normal Form (Cont.)

- Example schema that is ***not*** in BCNF:

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

  because :
  - *dept_name* $\rightarrow$ *building, 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 $\alpha \rightarrow \beta$ *(nontrivial where* $\alpha$ is not a superkey *)* 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

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C)$
- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Lossless-join decomposition:
    $$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$
  - Dependency preserving
- $R_1 = (A, B), \quad R_2 = (A, C)$
  - 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

- database consistency constraints: primary-key constraints, functional dependencies, check constraints, assertions, and triggers.

- It is not always possible to achieve both BCNF and dependency preservation
- Consider a schema:

  *dept_advisor(s_ID, i_ID, department_name)*
- With function dependencies:

  $i\_ID \rightarrow dept\_name$

  $s\_ID, dept\_name \rightarrow i\_ID$
- *dept_advisor* is not in BCNF
  - *i_ID* is not a superkey.
- Any decomposition of *dept_advisor* will not include all the attributes in

  $s\_ID, dept\_name \rightarrow i\_ID$
- Thus, the composition is NOT be dependency preserving

# Dependency Preservation

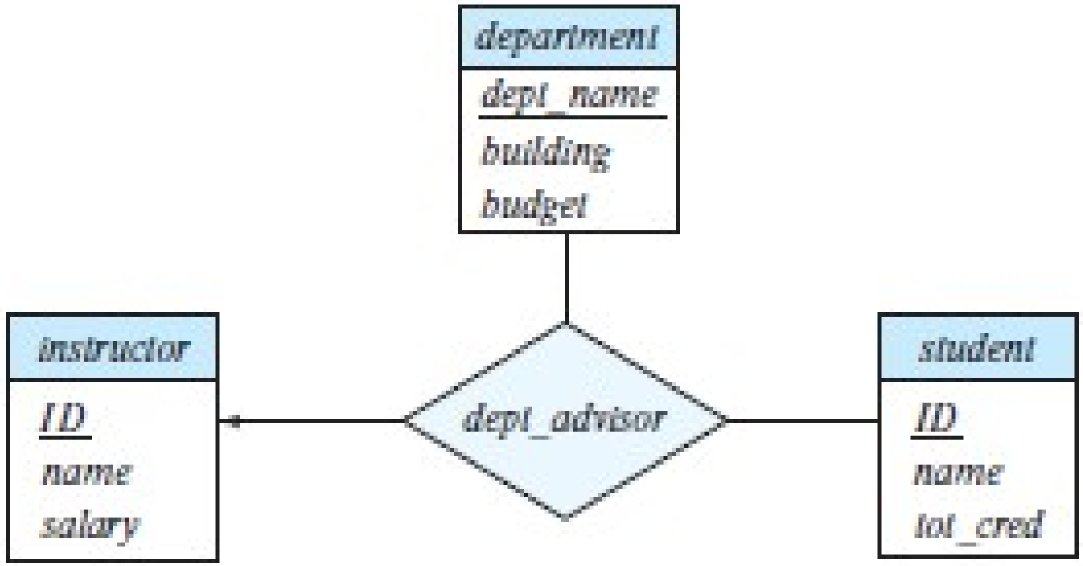


Figure 7.6 The *dept_advisor* relationship set.

# 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 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 $\{R_1, R_2, ..., R_n\}$ 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*

| *ID* | *child_name* | *phone* |
|---|---|---|
| 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*:

| *ID* | *child_name* |
|---|---|
| 99999 | David |
| 99999 | William |

  - *inst_phone:*

| *ID* | *phone* |
|---|---|
| 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

# 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 $F$ set 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 $A \rightarrow C$

- The set of **all** functional dependencies logically implied by $F$ is the **closure** of $F$.
- We denote the *closure* of $F$ by $\boldsymbol{F^+}$.

# 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 $F^+$

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- 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

# Closure of Functional Dependencies (Cont.)

- Additional rules:
  - **Union rule**: If $\alpha \rightarrow \beta$ holds *a*nd $\alpha \rightarrow \gamma$ 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.
  - **Pseudotransitivity rule**:If $\alpha \rightarrow \beta$ holds *a*nd $\gamma \beta \rightarrow \delta$ holds, then $\alpha \gamma \rightarrow \delta$ 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 $\alpha$, 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** $\beta \subseteq$ *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 $\alpha$ 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

- 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 $\alpha$ 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 $\beta$ 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

# 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 $\beta$
  - Consider the set:

    $$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\},$$
  - check that $\alpha^+$ contains *A;* if it does*, A* is extraneous in $\beta$
- To test if attribute $A \in \alpha$ is extraneous in $\alpha$
  - Let $\gamma = \alpha - \{A\}$. Check if $\gamma \rightarrow \beta$ can be inferred from *F.*
    - Compute $\gamma^+$ using the dependencies in *F*
    - If $\gamma^+$ includes all attributes in $\beta$ then , *A* is extraneous in $\alpha$

# 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

  $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$

  Find a functional dependency $\alpha \rightarrow \beta$ in $F_c$ 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** ($F_c$ 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

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

# Dependency Preservation

- Let $F_i$ be the set of dependencies $F^+$ that include only attributes in $R_i$.
  - A decomposition is **dependency preserving**, if

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

- 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 $R_1, R_2$, .., $R_n$ 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 in $F^+$, not just those in $F$.
- The set of restrictions $F_1, F_2$, .. , $F_n$ 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 $R_1, R_2, \ldots, R_n$ , 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 $\beta$, 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 $(F_1 \cup F_2 \cup \ldots \cup F_n)^+$

# 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

# Algorithm for Decomposition Using Functional Dependencies

# 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 mislead 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

- *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.

# 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*

- *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

# 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 $\alpha$ 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 $R_j$ ***/**
    $R_j = R_i$;
    *i=i-1;*
**return** $(R_1, R_2, ..., R_i)$

# 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* = (<u>*customer_id, employee_id*</u>, *branch_name, type* )

- The functional dependencies for this relation schema are:
  - *customer_id, employee_id* $\rightarrow$ *branch_name, type*
  - *employee_id* $\rightarrow$ *branch_name*
  - *customer_id, branch_name* $\rightarrow$ *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 $F_C$ =

    *customer_id, employee_id* $\rightarrow$ *type*
    *employee_id* $\rightarrow$ *branch_name*
    *customer_id, branch_name* $\rightarrow$ *employee_id*

# 3NF Decompsition Example (Cont.)

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

  (*customer_id, employee_id, type* )

  (<u>*employee_id*</u>, *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 (<u>*employee_id*</u>*, 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.

# Multivalued Dependencies

# Multivalued Dependencies (MVDs)

- Suppose we record names of children, and phone numbers for instructors:
  - *inst_child*(*ID*, *child_name*)
  - *inst_phone*(*ID*, *phone_number*)
- If we were to combine these schemas to get
  - *inst_info*(*ID*, *child_name*, *phone_number*)
  - Example data:
    (99999, David, 512-555-1234)
    (99999, David, 512-555-4321)
    (99999, William, 512-555-1234)
    (99999, William, 512-555-4321)
- This relation is in BCNF
  - Why?

# Multivalued Dependencies

- 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 -- Tabular representation

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

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

# MVD (Cont.)

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

$$Y, Z, W$$

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

$$< y_1, z_1, w_1 > \in r \text{ and } < y_1, z_2, w_2 > \in r$$

  then

$$< y_1, z_1, w_2 > \in r \text{ and } < y_1, z_2, w_1 > \in r$$

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

# Example

- In our example:

$$ID \rightarrow\rightarrow child\_name$$
$$ID \rightarrow\rightarrow phone\_number$$

- The above formal definition is supposed to formalize the notion that given a particular value of $Y$ ($ID$) it has associated with it a set of values of $Z$ *(child_name)* and a set of values of $W$ *(phone_number)*, 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 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 $\alpha \rightarrow\rightarrow \beta$

  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 (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 $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
  - $\alpha \rightarrow\rightarrow \beta$ 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 $\mathbf{R}_i$ 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 $\alpha \cap \beta = \phi$;
      *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, decompose into $R_3$ and $R_4$)

  c) $R_3 = (C, G, H)$ ($R_3$ is in 4NF)

  d) $R_4 = (A, C, G, I)$ ($R_4$ is not in 4NF, decompose into $R_5$ and $R_6$)

  - $A \rightarrow\rightarrow B$ and $B \rightarrow\rightarrow HI$ ➔ $A \rightarrow\rightarrow HI$, (MVD transitivity), and
  - and hence $A \rightarrow\rightarrow I$ *(MVD restriction to $R_4$)*

  e) $R_5 = (A, I)$ ($R_5$ is in 4NF)

  f) $R_6 = (A, C, G)$ ($R_6$ is in 4NF)

# Additional issues

# 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 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*,
    - 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 a *course* $\bowtie$ *prereq*
  - 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_2004, earnings_2005, earnings_2006*, 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_2004, earnings_2005, earnings_2006*)
    - 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

# Modeling Temporal Data

- **Temporal data** have an association time interval during which the data are *valid.*
- A **snapshot** is the value of the data at a particular point in time
- Several proposals to extend ER model by adding valid time to
  - attributes, e.g., address of an instructor at different points in time
  - entities, e.g., time duration when a student entity exists
  - relationships, e.g., time during which an instructor was associated with a student as an advisor.
- But no accepted standard
- Adding a temporal component results in functional dependencies like

  $$ID \rightarrow street, city$$

  not holding, because the address varies over time
- A **temporal functional dependency** $X \rightarrow Y$ holds on schema $R$ if the functional dependency $X \rightarrow Y$ holds on all snapshots for all legal instances r ($R$).

# Modeling Temporal Data (Cont.)

- In practice, database designers may add start and end time attributes to relations
  - E.g., *course*(*course_id, course_title*) is replaced by

    *course*(*course_id, course_title, start, end*)
  - Constraint: no two tuples can have overlapping valid times
    - Hard to enforce efficiently
- Foreign key references may be to current version of data, or to data at a point in time
  - E.g., student transcript should refer to course information at the time the course was taken

# End of Chapter 7

# 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
  - 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 (Cont.)

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

- 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 (Cont.)

- 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.

# Example: Constraints on Entity Set

- Consider relation obtained from Hourly_Emps:
  - Hourly_Emps (*ssn*, *name, lot, rating, hrly_wages*, *hrs_worked*)
- *Notation*: We will denote this relation schema by listing the attributes: SNLRWH
  - This is really the *set* of attributes {S,N,L,R,W,H}.
  - Sometimes, we will refer to all attributes of a relation by using the relation name. (e.g., Hourly_Emps for SNLRWH)
- Some FDs on Hourly_Emps:
  - *ssn* is the key: $S \rightarrow SNLRWH$
  - *rating* determines *hrly_wages*: $R \rightarrow W$

# Example (Contd.)

Wages

| R | W |
|---|---|
| 8 | 10 |
| 5 | 7 |

Hourly_Emps2

| S | N | L | R | H |
|---|---|---|---|---|
| 123-22-3666 | Attishoo | 48 | 8 | 40 |
| 231-31-5368 | Smiley | 22 | 8 | 30 |
| 131-24-3650 | Smethurst | 35 | 5 | 30 |
| 434-26-3751 | Guldu | 35 | 5 | 32 |
| 612-67-4134 | Madayan | 35 | 8 | 40 |

- Problems due to R $\rightarrow$ W :
  - *Update anomaly*: Can we change W in just the 1st tuple of SNLRWH?
  - *Insertion anomaly*: What if we want to insert an employee and don't know the hourly wage for his rating?
  - *Deletion anomaly*: If we delete all employees with rating 5, we lose the information about the wage for rating 5!

| S | N | L | R | W | H |
|---|---|---|---|---|---|
| 123-22-3666 | Attishoo | 48 | 8 | 10 | 40 |
| 231-31-5368 | Smiley | 22 | 8 | 10 | 30 |
| 131-24-3650 | Smethurst | 35 | 5 | 7 | 30 |
| 434-26-3751 | Guldu | 35 | 5 | 7 | 32 |
| 612-67-4134 | Madayan | 35 | 8 | 10 | 40 |

# Solution

Decompose the relation:

- Hourly_Emps (<u>*ssn*</u>*, name, lot, rating, hrly_wages*, *hrs_worked*)
- Into set of relations:
- Hourly_Emps(ssn,name,lot,rating, hours_worked)
- Rating_Wages( rating,hrly_wages)

- What happened to update anomalies?
- We need to find out the basis for decomposing a relation to get rid of update anomalies

# Functional Dependency

- FD is a many-to-one relationship from one set attributes to another
- Example: there is a FD from the set of attributes {S#,P#} to the set of attributes {QTY}
- For any given value for pair of attributes S# and P#, there is just one corresponding value of attribute QTY, but, many distinct values of the pair of attributes S# and P# can have the same corresponding value for attribute QTY

# 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*

# Reasoning About FDs

- Given some FDs, we can usually infer additional FDs:
  - *ssn* → *did*, *did* → *lot* implies *ssn* → *lot*
- An FD *f* is *implied by* a set of FDs *F* if *f* holds whenever all FDs in *F* hold.
  - *closure of F* is the set of all FDs that are implied by *F*

  - It is constraint in the real world and hence be obeyed
  - Declare FD and make sure that it is followed (integrity constraint)