# Principles of Database Systems (CS307)

## Lecture 11: Normalization - Part 2

**Zhong-Qiu Wang**

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.
- The slides are largely based on the slides provided by Dr. Yuxin Ma

# Announcements

- Deadline of Project I is further extended to 18th Nov., Tuesday, 10pm

## Functional Dependency Theory

# Closures, Attribute Closures, Canonical Cover, and Dependency Preservation

It is strongly recommended to read Section 7.4 of the reference textbook "Database System Concepts, 7th Edition" for more details about the 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

Computing Closure $F^+$

- We then develop algorithms to generate lossless decompositions into BCNF and 3NF

Decomposition into BCNF and 3NF

- Furthermore, we develop algorithms to test if a decomposition is dependency-preserving

Testing whether decomposition is dependency-preserving

# (Recall) 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 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*

# Computing Closure with Armstrong's Axioms

- 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 $\alpha \rightarrow \gamma$

> - Greek letters ($\alpha$, $\beta$, $\gamma$) represent *sets of attributes*
> - "$\gamma\alpha$" means "$\gamma \cup \alpha$"

- These rules are sound and complete
  - *Sound* （可靠）: Generate only functional dependencies that actually hold
  - *Complete* （完备）: Generate all functional dependencies that hold

# Computing Closure with Armstrong's Axioms

- 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 $\alpha \rightarrow \gamma$

- Greek letters ($\alpha$, $\beta$, $\gamma$) represent *sets of attributes*
- "$\gamma\alpha$" means "$\gamma \cup \alpha$"

- These rules are sound and complete
  - *Sound* （可靠） : Generate only functional dependencies that actually hold
  - *Complete* （完备） : Generate all functional dependencies that hold

However, it is difficult and tiresome to use them for deriving $F^+$

# Computing Closure with Armstrong's Axioms

- 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
  - Pseudotransitivity rule: If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds
    - From $\alpha \rightarrow \beta$, we have $\gamma\alpha \rightarrow \gamma\beta$, together with $\gamma\beta \rightarrow \delta$, we have $\gamma\alpha \rightarrow \delta$

The above rules can be inferred from Armstrong's axioms

# Procedure for Computing $F^+$

$F^+ = F$
apply the reflexivity rule /* Generates all trivial dependencies */
**repeat**
    **for each** functional dependency $f$ in $F^+$
        apply the augmentation rule 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
            add the resulting functional dependency to $F^+$
**until** $F^+$ does not change any further

- Problem: The target $F^+$ can be very lengthy
  - For $\alpha \rightarrow \beta$, there may be $2^n$ possible values for $\alpha$ and $2^n$ for $\beta$
  - We will introduce other ways of computing $F^+$ later

# Closure of Attribute Sets （属性集闭包）

- We say that an attribute $B$ is functionally determined by $\alpha$ if $\alpha \to B$

- 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 := α;        // by reflexibility rule, α → result
repeat
      for each functional dependency β → γ in F do
         begin
            if β ⊆ result then result := result ∪ γ;
         end
until (result does not change)
```

// based on $\alpha \to$ *result*, $\beta \to \gamma$ and $\beta \subseteq$ *result*
// *result* $\cup\, \beta \to$ *result* $\cup\, \gamma$, by augmentation rule
// *result* $\to$ *result* $\cup\, \gamma$
// $\alpha \to$ *result* $\cup\, \gamma$

# Example of Attribute Set Closure

- ***Given***:

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

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

- What is $(AG)^+$ ?
  1. $result = AG$
  2. $result = ABG$ $\quad (A \rightarrow B)$
  3. $result = ABCG$ $\quad (A \rightarrow C)$
  4. $result = ABCGH$ $\quad (CG \rightarrow H \text{ and } CG \subseteq ABCG)$
  5. $result = ABCGHI$ $\quad (CG \rightarrow I \text{ and } CG \subseteq ABCGH)$

# Example of Attribute Set Closure

- ***Given***:

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

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

- What is $(AG)^+$ ?
  1. $result = AG$
  2. $result = ABG$ $\quad (A \rightarrow B)$
  3. $result = ABCG$ $\quad (A \rightarrow C)$
  4. $result = ABCGH$ $\quad (CG \rightarrow H$ and $CG \subseteq ABCG)$
  5. $result = ABCGHI$ $\quad (CG \rightarrow I$ and $CG \subseteq ABCGH)$

## Further Questions:

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*

# Use of Attribute Closures

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 $\beta$
  - It 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（正则/最小覆盖）

- 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

# Canonical Cover（正则/最小覆盖）

- 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
  - Any database satisfying the simplified set of functional dependencies also satisfies the original set, and vice versa, since the two sets have the same closure
  - 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 (Stronger Constraint vs. Weaker Constraint)

- Removing an attribute from the left side of a functional dependency could make it a stronger constraint

$AB \rightarrow C$ *(Remove B on the left side)* → $A \rightarrow C$

weaker — stronger

$A \rightarrow C$ is stronger than $AB \rightarrow C$ because $A \rightarrow C$ logically implies $AB \rightarrow C$

- However, depending on what our set $F$ of functional dependencies happens to be, we may be able to remove $B$ from $AB \rightarrow C$ safely
  - E.g., $F = \{AB \rightarrow C, A \rightarrow D, D \rightarrow C\}$

$A \rightarrow D, D \rightarrow C$ → logically imply $A \rightarrow C$ → *$AB \rightarrow C$ can be removed*

# Extraneous Attributes (Stronger Constraint vs. Weaker Constraint)

- In reverse, removing an attribute from the <u>right side</u> of a functional dependency could make it a weaker constraint

$AB \rightarrow CD$ $\xrightarrow{\textit{Remove C on the right side}}$ $AB \rightarrow D$

stronger weaker

$AB \rightarrow CD$ is stronger than $AB \rightarrow D$ because $AB \rightarrow D$ cannot logically imply $AB \rightarrow C$

- However, depending on what our set $F$ of functional dependencies happens to be, we may be able to remove $C$ from $AB \rightarrow CD$ safely
  - E.g., $F = \{AB \rightarrow CD, A \rightarrow C\}$

logically imply $AB \rightarrow BC$ → logically imply $AB \rightarrow C$

*C can be removed*

# 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

$$\alpha \to \beta \text{ in } F$$

*Remove from the Left Side*

- Attribute A is **extraneous** in $\alpha$ if
  - $A \in \alpha$, and
  - $F$ logically implies

$$(F - \{\alpha \to \beta\}) \cup \{(\alpha - A) \to \beta\}$$

*Remove from the Right Side*

- Attribute A is **extraneous** in $\beta$ if
  - $A \in \beta$, and
  - The set of functional dependency $(F - \{\alpha \to \beta\}) \cup \{(\alpha - A) \to \beta\}$ logically implies $F$

# Testing if an Attribute is Extraneous

- Let $R$ be a relation schema; $F$ be a set of functional dependencies held on $R$
- Consider an attribute in the functional dependency $\alpha \rightarrow \beta$

*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 $\beta$, $A$ is extraneous in $\alpha$

*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 $\beta$

# Examples of Extraneous Attributes

- Let $F = \{AB \rightarrow CD, A \rightarrow E, E \rightarrow C\}$
- To check if $C$ is extraneous in $AB \rightarrow CD$
  - 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

*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 $\beta$

# 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 <u>left side</u> of functional dependency in $F_c$ is <u>unique</u>.

    ... that is, there are no two dependencies in $F_c$
    - $\alpha_1 \to \beta_1$ and $\alpha_2 \to \beta_2$ such that
    - $\alpha_1 = \alpha_2$

# Canonical Cover

- To compute a canonical cover for *F*:

$F_c = F$
**repeat**
  Use the union rule to replace any dependencies in $F_c$ 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: the test for extraneous attributes is done using $F_c$, not $F$ */
  If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$ in $F_c$.
**until** ($F_c$ does not change)

# Example: Computing a Canonical Cover

*Given*:

$R = (A, B, C)$

$F = \{A \rightarrow BC$
$B \rightarrow C$
$A \rightarrow B$
$AB \rightarrow C\}$

- 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* $B \rightarrow C$

The resulting canonical cover:

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

# Dependency Preservation

- Let $F_i$ be the set of dependencies in $F^+$ 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
    - A faster algorithm is introduced in Section 7.4.4 of the reference book
- If a decomposition is NOT dependency-preserving, checking updates for violation of functional dependencies may require computing joins, which is expensive

Functional Dependency Theory

# Algorithms for BCNF and 3NF Decompositions Using Functional Dependencies

# Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
  - compute $\alpha^+$ (the attribute closure of $\alpha$), and
  - 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

# BCNF Decomposition Algorithm

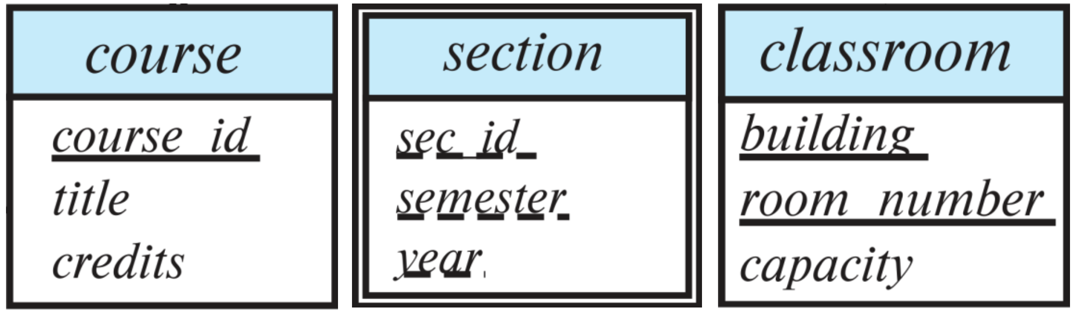
$result := \{R\};$
$done :=$ false;
**while** (**not** $done$) **do**
  **if** (there is a schema $R_i$ in $result$ that is not in BCNF)
   **then begin**
      let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds on $R_i$ such that $\alpha^+$ does not contain $R_i$ and $\alpha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

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

# Example of BCNF Decomposition

- **Schema**
  - *class (course_id, title, dept_name, credits, sec_id, semester, year, building, room_number, capacity, time_slot_id)*

* A better design




- **Candidate key**
  - *{course_id, sec_id, semester, year}*
- **Functional dependencies**
  - *course_id → {title, dept_name, credits}*
  - *{building, room_number} → capacity*
  - *{course_id, sec_id, semester, year} → {building, room_number, time_slot_id}*

# Example of BCNF Decomposition

- BCNF Decomposition (round 1)
  - *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*)

*Here*, *course* is in BCNF

# Example of BCNF Decomposition

- BCNF Decomposition (round 2)
  - *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 (3NF)

- 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, 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 Decomposition Algorithm

- The algorithm ensures
  - Each relation schema $R_i$ is in 3NF
  - Decomposition is dependency preserving and lossless-join

let $F_c$ be a canonical cover for $F$;
$i := 0$;
**for each** functional dependency $\alpha \rightarrow \beta$ in $F_c$
    $i := i + 1$;
    $R_i := \alpha\,\beta$;
**if** none of the schemas $R_j, j = 1, 2, \ldots, i$ contains a candidate key for $R$
  **then**
    $i := i + 1$;
    $R_i :=$ any candidate key for $R$;
/* Optionally, remove redundant relations */
**repeat**
    **if** any schema $R_j$ is contained in another schema $R_k$
      **then**
        /* Delete $R_j$ */
        $R_j := R_i$;
        $i := i - 1$;
**until** no more $R_j$s can be deleted
**return** $(R_1, R_2, \ldots, R_i)$

# 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

An alternative method of generating a BCNF design: First use the 3NF algorithm. Then, for any schema in the 3NF design that is not in BCNF, decompose using the BCNF algorithm. If the result is not dependency-preserving, revert to the 3NF design.

# Other Normal Forms

# 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
    - Actually, we would better use two relations: (*ID*, *child_name*) and (*ID*, *phone*)

An instance of *inst_info*:

(99999, David, 512-555-1234)
(99999, David, 512-555-4321)
(99999, William, 512-555-1234)
(99999, William, 512-555-4321)

# How good is BCNF?

- inst_info is in BCNF
  - *Primary key is {ID, child_name, phone}*
  - *ID $\rightarrow$ child_name ?No*
  - *ID $\rightarrow$ phone? No*
  - No non-trivial functional dependencies exist, except trivial ones

- However,
  - Insertion anomalies
    - If we add a phone number 981-992-3443 to the instructor 99999, we need to add two tuples:

      (99999, David, 981-992-3443)
      (99999, William, 981-992-3443)
    - If we only add one of the two tuples above, it will imply that only David or William corresponds to 981-992-3443, which is not the functional dependency we need to keep

# Fourth Normal Form (4NF)

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

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

| *ID* | *phone* |
|---|---|
| 99999 | 512-555-1234 |
| 99999 | 512-555-4321 |

  - This suggests a need for higher normal forms, such as Fourth Normal Form (4NF) that resolves such kind of multivalued dependencies

# Wait, Where are 1NF and 2NF?

- 1NF is about attribute domains but not decompositions
  - … and hence not quite related to dependencies we have learned in this section

# Wait, Where are 1NF and 2NF?

- 2NF: Partial dependency
  - A functional dependency $\alpha \to \beta$ is called a partial dependency if there is a proper subset $\gamma$ of $\alpha$ such that $\gamma \to \beta$
    - We say that $\beta$ is partially dependent on $\alpha$

  - A relation schema *R* is in second normal form (2NF) if each attribute *A* in *R* meets one of the following criteria:
    - It appears in a candidate key
    - It is not partially dependent on a candidate key

# Wait, Where are 1NF and 2NF?

- 2NF: Partial dependency
  - You can try to prove that a relation meeting 3NF also satisfies 2NF
    - Exercise 7.19 in “Database System Concepts, 7th Edition”

  - In practice, we usually choose to satisfy 3NF or BCNF

# Summary for Database Design

# 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

# E-R 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*
    - … but with functional dependency: $department\_name \rightarrow building$
    - Good design would have made department an entity

# Denormalization for Performance

- We may want to use non-normalized schemas for better 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 on *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
      - It is an example of a **crosstab**, where values for one attribute become column names
      - Such crosstabs are widely used in spreadsheets and data analysis tools