Chapter 7: Normalization

Database System Concepts, 7th Ed.

Outline

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

ER versus Normalization

- Features of ER
  - Entity sets and Relationship sets
  - Mapping to tables
- Features of Normalization
  - Sets of all attributes used in the database
  - Distribution of the attributes to various tables

Overview of Normalization

Normalization Goal

- Given a set of attributes

  $R = \{A_1 , A_2 , A_3 \ \ldots . A_n\}$

- Partition the attributes among M relations
  - With no repetition of information
- Seems to be mission impossible.

Features of Good Relational Designs

- Consider the new relation *in_dep* that combines the *instructor* and *department* tables

ID	*name*	*salary*	*dept_name*	*building*	*budget*
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 in the relation *in_dep*
- Need to use null values (if we add a new department with no instructors)
- This relation is NOT in good form

Good Form

- A relation is said to be in “good” form if:
  - There is no repetition of information
  - There is no need to use null values

- Goal:
  - Devise a scheme to make sure that all tables are in good form

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 (which we used all along).

- Not all decompositions are good. Suppose we have a schema:

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

  and we decompose it into

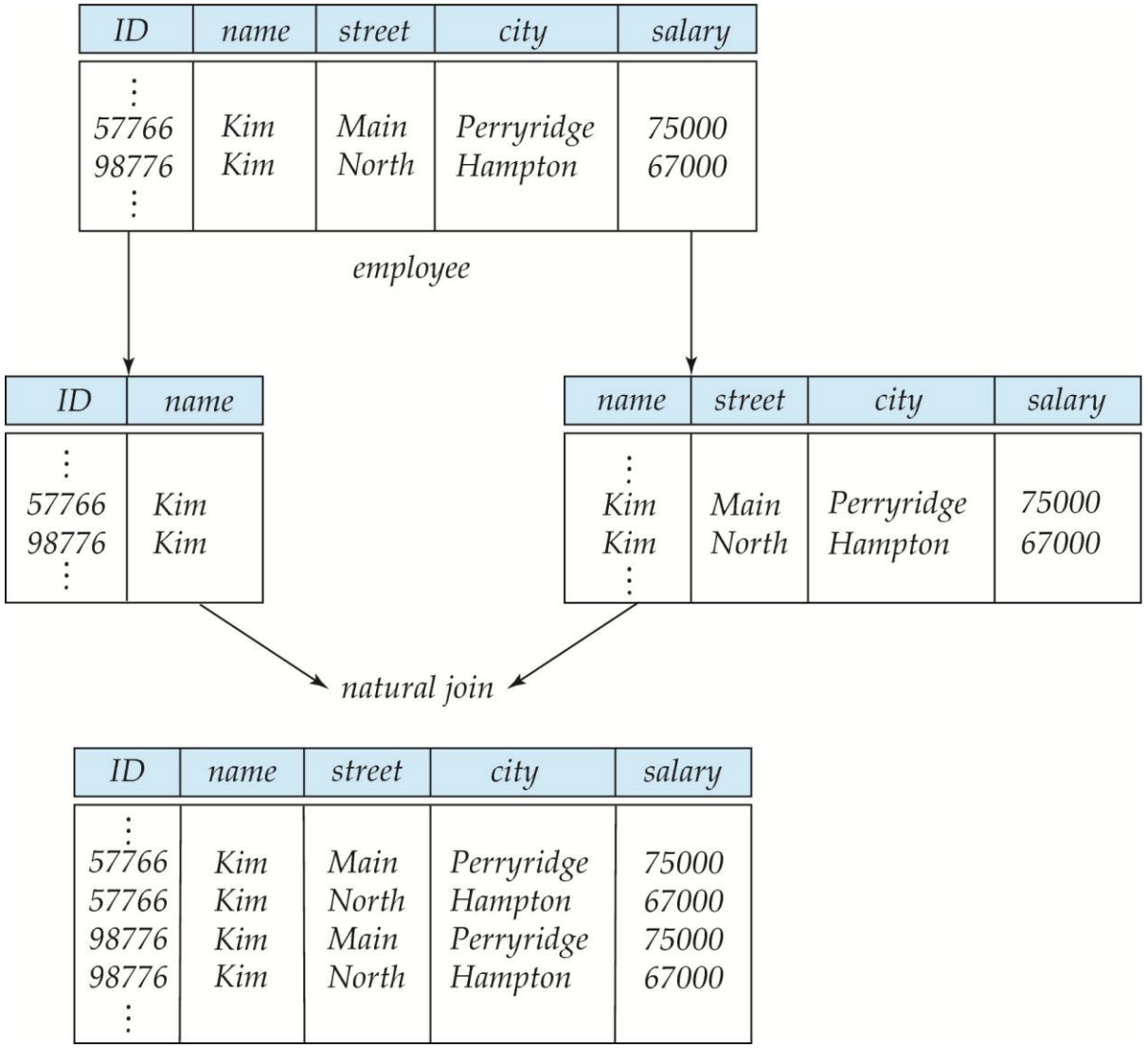
  employee1 (*ID*, *name*)

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

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

- The problem is that we can lose information when we decompose the relation *employee* -- 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 and R_2

Lossless Decomposition Example

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

  $R_1 = (A, B)$ $\quad$ $R_2 = (B, C)$
- Example of a database instance

A	B	C
α	1	A
β	2	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

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.
- Such rules help us decide:
  - which data combinations are valid
  - how to design good database schemas
- An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation;
  - One student ID → two different names, this is **illegal. But** One student ID → one name **legal instance.**
- Functional dependencies describe **which tables are legal and which are not**.

Functional Dependencies (Cont.)

- **A functional dependency exists when:**
  - **knowing the value of one set of attributes uniquely determines the value of another set of attributes**
- Example: If I know a **student ID**, I automatically know: the student's **name** and the student's **department.**
- This must hold for all valid states of the database.

- Functional dependencies are a generalization of keys and are used to guide normalization.

Functional Dependencies Definition

- Let *R* be a relation schema
- Let

$$\alpha \subseteq R \quad and \quad \beta \subseteq R$$

- 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,

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

Note: Normalization is the process of decomposing relations based on functional dependencies to improve database design.

Functional Dependencies Example

- Consider *r* (A,*B*) with the following instance of *r.*

A	B
1	4
1	6
3	7

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

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→ *building*

  ID → *building*

  but would not expect the following to hold:

  dept_name → *salary*

Use of Functional Dependencies

- We use functional dependencies to:
  - Test relations to see if they are legal under a given set of functional dependencies.
    - If a relation instance **r** (actual data at a given time) does not violate any FD in **F**, we say **r satisfies F**.
  - Specify constraints on the set of legal relations
    - **F holds on R** if **every possible legal instance** of schema **R** satisfies 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.*

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 $A \rightarrow C$
  - etc.
- Closure of F is the set of all functional dependencies that are logically implied by F, including F itself.
- We denote the *closure* of F by $\mathbf{F^+}$.
- We will show later on how compute the closure of F

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 a decomposition is lossless**.

- A decomposition is lossless if we can always reconstruct the original table without generating extra or missing tuples.
- A decomposition of R into R_1 and R_2 is lossless decomposition if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$

- The above functional dependencies are a **sufficient condition** for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

Example

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

    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
- $R_3 = (A, B)$, $R_4 = (A, C)$
  - Lossless decomposition:

    $R_3 \cap R_4 = \{A\}$ and $A \rightarrow AB$
- Notational Note:
  - $B \rightarrow BC$

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

Dependency Preservation

- Checking functional dependencies every time data is inserted, updated, or deleted can be **time-consuming**.
- Therefore, we should design the database so that **functional dependencies can be checked easily**.
- If a functional dependency can be verified by looking at **just one table**, the checking cost is **low**.

- After decomposing a table, some dependencies may involve attributes that are **spread across multiple tables**.
- In such cases, checking the dependency may require **joining tables**, which is **expensive**.

- A decomposition is called **dependency preserving** if **all functional dependencies can be checked using the decomposed tables without joining them**.

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
- $R_1 = (A, B)$, $R_2 = (B, C)$
  - Lossless-join decomposition:
  - Dependency preserving
- $R_3 = (A, B)$, $R_4 = (A, C)$
  - Lossless-join decomposition:
  - Not dependency preserving
    - Cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$

University Example

- Consider a schema:

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

  $i\_ID \rightarrow dept\_name$

  $s\_ID, dept\_name \rightarrow i\_ID$
- In the above design, dept_name is repeated every time an instructor appears in dept_advisor because one instructor can advise multiple students.

- To fix this, we need to decompose *dept_advisor*
- Any decomposition will not include all the attributes in

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

Design Goals when Decomposing a Relation

- Lossless join decomposition
- Dependency preserving decomposition

Normalization

- Normalization is the systematic process of organizing relations (transforming a relation into higher normal forms by decomposing it)—based on functional dependencies—to
  - reduce redundancy
  - avoid update anomalies
  - ensure lossless and dependency-preserving decomposition

- Next, we will study **normal forms**, which provide formal criteria that guide this normalization process.

- Normal forms provide **levels of normalization** based on functional dependencies.

Normal Forms

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 Examples

- Example of a schema that is in BCNF:

  instructor (*ID, name, salary, dept_name*)

  because :

  - *ID* $\rightarrow$ *name, salary, dept_name*

  Is the only meaningful FD and *ID* is a superkey

- 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

Decomposing a Schema into BCNF

- Let R be a schema that is not in BCNF. Let $\alpha \rightarrow \beta$ be the FD that causes a violation of BCNF.
- We decompose *R* into:
  - (α U β)
  - (*R* - (β - α))
- In our example of relation *in_dep*,

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

  We have
  - α = *dept_name*
  - β = *building, budget*

  and *in_dep* is replaced by
  - (α U β) = (*dept_name, building, budget*)
  - (*R* - (β - α)) = (*ID, name, dept_name, salary*)
  - We are done!

Decomposing a Schema into BCNF (Cont.)

- In our example of the relation *in_dep*

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

  we were done after one iteration of replacing the original relation
- In general, we may need to go through several iterations before we are done.

BCNF and Dependency Preservation

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

  dept_advisor (s_ID, i_ID, department_name)
- With functional 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
- If we insist on dependency preserving, we must consider other normal forms

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
- It is, 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.
- What should one use in practice? BCNF or 3NF?

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

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 $\alpha \rightarrow \gamma$

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

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 union Rule

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

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^+

```
result := α;
while (changes to result) do
        for each β → γ in F do
            begin
                if β ⊆ result  then  result := result  ∪ γ
            end
```

Example of Closure of Attribute Sets

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

- Computing F^+
  - For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

Algorithms for Decomposition Using FD

BCNF

BCNF Decomposition Algorithm

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

3NF

Canonical Cover

- A database relation has a set of functional dependencies **F**.
- The database must **check these dependencies** whenever data is inserted, updated, or deleted.
- Checking **all dependencies in F** every time can be **costly**.
- We can simplify **F** by removing:
  - redundant dependencies
  - unnecessary attributes
- The simplified set must still imply **exactly the same rules** as F.

- A **canonical cover (F_c)** is a **minimal set of functional dependencies** such that:
  - $F_c^+ = F^+$ (same closure)
  - No dependency in F_c is redundant
  - No attribute in any dependency is unnecessary

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**
begin
i := *i* + 1;
$R_i := \alpha\ \beta$
end
if none of the schemas R_j, $1 \le j \le i$ 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-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

3NF Decomposition: An Example

- Relation schema:

  cust_banker_branch = (*customer_id, employee_id, branch_name, type*)

- The functional dependencies for this relation schema are:
  - *customer_id, employee_id* $\rightarrow$ *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*)

  (*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

Extra

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.

End of Chapter 7