MODULE 2 – LOGICAL DATABASE DESIGN

Database System Concepts, 7th Ed.

Outline

- Need for good database design

Functional Dependency:

- Properties of functional dependency
- Types of Functional Dependencies and Keys
- Closure of Functional Dependencies set
- Closure of Attributes
- Canonical Cover
- Dependency Preservation with example

Normalization:

- Need of normalization
- First Normal Form, Second Normal Form
- Third Normal Form, Boyce Codd Normal Form
- Fourth Normal Form, Fifth Normal Form
- Join Dependencies - Types of Join Dependencies & Example of Join Dependencies

Need for good database design

Need for good database design

- To ensure efficient data storage, retrieval, and manipulation by minimizing redundancy
- To maintain data integrity
- To optimize the performance for the specific application needs, ultimately leading to a reliable and well-functioning system.

Features of Good Relational Designs

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

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**
- **Need to use null values (if we add a new department with no instructors)**

Decomposition

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

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

  into

  ***employee1* (*ID*, *name*)**

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

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

- We cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

A Lossy Decomposition

ID	name	street	city	salary
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000

employee

ID	name
57766	Kim
98776	Kim

name	street	city	salary
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000

natural join

ID	name	street	city	salary
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000

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 U 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 U R_2

- Formally,

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

- And, conversely a decomposition is lossy if

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

Example of Lossless Decomposition

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

  R_1 = (A, B) R_2 = (B, C)

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

Normalization Theory

- Decide whether a particular relation *R* is in "good" form.
- In the case that a relation *R* is not in "good" form, decompose it into set of relations $\{R_1, R_2, ..., R_n\}$ such that
  - **Each relation is in good form**
  - **The decomposition is a lossless decomposition**
- Our theory is based on:
  - **Functional dependencies**
  - **Multivalued dependencies**

Functional Dependencies

Functional Dependencies

- **If one set of attributes in a table determines another set of attributes in the table, then the second set of attributes is said to be functionally dependent on the first set of attributes.**
- There are usually a variety of constraints (rules) on the data in the real world.
- For example, some of the constraints that are expected to hold in a university database are:
  - Students and instructors are uniquely identified by their ID.
  - Each student and instructor has only one name.
  - Each instructor and student is (primarily) associated with only one department.
  - Each department has only one value for its budget, and only one associated building.

Functional Dependencies (Cont.)

- An instance of a relation that satisfies all such real-world constraints is called a **legal instance** of the relation;
- **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.
- **The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table**.

$$X \rightarrow Y$$

- The left side of FD is known as a **determinant**, the right side of the production is known as a **dependent**.

Functional Dependencies Definition

- Let *R* be a relation schema

$$\alpha \subseteq R \quad \textit{and} \quad \beta \subseteq R$$

- 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,

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

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

1	4
1	5
3	7

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

Functional Dependencies - EXAMPLE

ISBN	Title	Price
0-321-32132-1	Balloon	$34.00
0-55-123456-9	Main Street	$22.95
0-123-45678-0	Ulysses	$34.00
1-22-233700-0	Visual Basic	$25.00

Functional Dependencies: {ISBN} → {Title}
{ISBN} → {Price}

Functional Dependencies - EXAMPLE

- Social security number determines employee name
  - **SSN -> ENAME**
- Project number determines project name and location
  - **PNUMBER -> {PNAME, PLOCATION}**
- Employee ssn and project number determines the hours per week that the employee works on the project
  - **{SSN, PNUMBER} -> HOURS**

Note:

- **If K is a key of R, then K functionally determines all attributes in R**
  - (since we never have two distinct tuples with t1[K]=t2[K])

Keys and Functional Dependencies

- ***K* is a superkey** for relation schema *R* **if and only if** $K \rightarrow R$
- ***K* is a candidate key** for *R* **if and only if**
  - $K \rightarrow R$**, and**
  - **for no** $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

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

  We expect these functional dependencies to hold:

  $dept\_name \rightarrow building$

  $ID \rightarrow building$

  but would not expect the following to hold:

  $dept\_name \rightarrow salary$

Properties of Functional Dependency

Inference Rules for FDs

- Given a set of FDs F, we can infer additional FDs that hold whenever the FDs in F hold
- **Armstrong's inference rules:**

  IR1. (Reflexive): **If Y *subset-of* X, then X -> Y**

  Example: {STU_ID, NAME} → **NAME** is valid reflexive relation

  IR2. (Augmentation):

  If X -> Y, then XZ -> YZ (Notation: XZ stands for X U Z)

  Example: {STU_ID, NAME} → **{ DEPT_BUILDING}** is valid then **{STU_ID,NAME,DEPT_NAME}** → **{DEPT_BUILDING,DEPT_NAME}** is also valid.

  IR3. (Transitive): **If X -> Y and Y -> Z, then X -> Z**

  Example: If **STU_ID→CLASS, CLASS→LECTURE_HALL** holds true then according to the axiom of transitivity, **STU_ID→LECTURE_HALL** will also hold true.

Inference Rules for FDs

- **Decomposition: If X -> YZ, then X -> Y and X -> Z**

  Example: If **STU_ID → {STU_NAME, COURSE}** holds true, then **STU_ID → STU_NAME, STU_ID → COURSE** holds true.

- **Union: If X -> Y and X -> Z, then X -> YZ**

  Example: If **STU_ID → STU_NAME, STU_ID → COURSE**, then **STU_ID→ {STU_NAME, COURSE}** holds true.

- **Psuedotransitivity: If X -> Y and WY -> Z, then WX -> Z**

- **Example:** If **Customer_ID → Order_ID**, and **{Order_ID, Delivery_Address} → Shipping_Cost** holds true, then **{Customer_ID, Delivery_Address} → Cost** holds true.

Types of Functional Dependencies

Types of Functional Dependencies

- **They are four types of functional dependencies:**

1. Trivial Functional Dependency

- A functional dependency A→B is said to be trivial functional dependency, if 'B' is a subset of 'A.'
- The following dependencies are also trivial like: **A → A, B → B**

Example:

{Employee_Id, Employee_Name}→ Employee_Id is a **trivial functional dependency** as Employee_Id is a subset of {Employee_Id, Employee_Name}.

Types of Functional Dependencies

2. Non-trivial Functional Dependency

- A → B has a non-trivial functional dependency, if B is not a subset of A.
- When A intersection B is NULL, then A → B is called as complete non-trivial.

Example:

A functional dependency **STUDENT_ID → STUDENT_NAME** is a **non-trivial dependency** since STUDENT_NAME is not a subset of STUDENT_ID and the intersection of both of these will be NULL;

Types of Functional Dependencies

3. Multivalued Functional Dependency

- When two attributes in a table are independent of each other but both rely on a third attribute, this is referred to as multivalued dependency.

STU_ID	COURSE	PASSING_YEAR
1	Science	2020
2	Science	2021

- Here, **STU_ID** can determine both **COURSE and PASSING_YEAR**.
- **STU_ID →{ COURSE, PASSING_YEAR },** but there is **no functional dependency between COURSE and PASSING_YEAR**. Hence we can say that COURSE and PASSING_YEAR both are independent of each other which makes them a **multivalued dependent on STU_ID**.

Types of Functional Dependencies

4. Transitive Functional Dependency

- A functional dependency that is indirectly formed by two functional dependencies is called transitive functional dependency.
- For example, if **A → B, B → C holds true**, then according to the axiom of transitivity, **A → C will also hold true**.

STU_ID	CLASS	LECTURE_HALL
1	7	L202
2	6	B101

- Here, **STU_ID → CLASS**, and **CLASS → LECTURE_HALL** for that particular class. Therefore **STU_ID → LECTURE_HALL holds true**.

Closure of Functional Dependencies set

- The closure of a set of functional dependencies **F+** is the set of all functional dependencies that can be inferred from **F**.
- To find **F+**, we need to apply the **Armstrong's Axioms** (which consist of **Reflexivity**, **Augmentation**, and **Transitivity**) iteratively to derive new functional dependencies until no new dependencies can be inferred.

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 F^+.

Example:

- **F = {A → B, B → C}**
- **F^+= {A → B, B → C, A → C}**

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.

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^+

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- **Solution:**
  - $A \rightarrow BC$ (by **Union** of $A \rightarrow B$ *and* $A \rightarrow C$)
  - $A \rightarrow H$ (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** of $CG \rightarrow I$ and $CG \rightarrow H$)

$F^+ = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H, A \rightarrow BC, A \rightarrow H, AG \rightarrow I , CG \rightarrow HI \}$

Example of F^+

Problem:

Given the relation **R(A, B, C, D)** with the following functional dependencies:

A → B

B → C

AC → D

Find the closure of the set of functional dependencies **F+**, i.e., find all the functional dependencies that can be derived from the given set of functional dependencies.

Solution:

Step 1: Write down the initial functional dependencies.

Step 2: Use the functional dependencies to find new ones.

- From **A → B** and **B → C**, we can derive **A → C**. **(By Transitivity)**
- From **A → C** and **C → D** (which we can derive from **AC → D** since **A → C** is true), we can derive **A → D**. **(By Transitivity)**
- $F^+ = \{A \rightarrow B, B \rightarrow C, AC \rightarrow D, A \rightarrow C, A \rightarrow D\}$

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

- **Attribute Closure of an attribute set** is defined as a set of all attributes that can be functionally determined from it.
- The closure of an attribute is represented as $\mathbf{A}^{+}$.

Closure of Attribute Sets

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

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

Example of Attribute Set Closure

- ***Given:***
- ***R = (A, B, C, G, H, I)***
- ***F = {A → B***
  A → C
  CG → H
  CG → I
  B → H}
- **Find the Attribute Closure *{AG}*$^+$**

  {AG}$^+$ = *{AG}*

  {AG}$^+$ = *{ABCG}* ***(A → C ,A → B and A ⊆ AG)***

  {AG}$^+$ = *{ABCGH}* ***(CG → H and CG ⊆ ABCG)***

  {AG}$^+$ = *{ABCGHI}* ***(CG → I and CG ⊆ ABCGH)***

The closure of {AG} (i.e., {AG}$^+$) is {A, B, C, G, H, I}.

Example of Attribute Set Closure

Problem:

Given the relation **R(A, B, C, D)** with the set of functional dependencies:

$A \rightarrow B$

$B \rightarrow C$

$AC \rightarrow D$

Find the closure of the attribute set {A}, i.e., $\{A\}^+$.

Solution:

$\{A\}^+ = \{A\}$

$\{A\}^+ = \{A, B\}$ **($A \rightarrow B$, Since A is in $\{A\}^+$)**

$\{A\}^+ = \{A, B, C\}$ **($B \rightarrow C$, since B is in $\{A\}^+$)**

$\{A\}^+ = \{A, B, C, D\}$ **($AC \rightarrow D$, since both A and C are in $\{A\}^+$)**

The closure of {A} (i.e., $\{A\}^+$) is {A, B, C, D}.

To check Super key and Candidate key

StudentID	Name	Course	Instructor	Building
S001	Alice	CS101	Dr. Smith	Building A
S002	Bob	CS102	Dr. Johnson	Building B
S003	Charlie	CS101	Dr. Smith	Building A
S004	David	CS103	Dr. Lee	Building C

Step 1: Attribute Set
StudentID, Name,Course,Instructor,Building

Step 2: Attribute Closure
To find the **Super Keys** and **Candidate Keys**, we'll calculate attribute closures based on functional dependencies.

Assume the following functional dependencies (FDs):
StudentID → Name, Course, Instructor, Building
Course → Instructor
Instructor → Building

To check Super key and Candidate key

Step 3: Find Super Keys Using Attribute Closure

A set of attributes X is a **Super Key**, if the closure X^+ contains all attributes in the relation.

- **{StudentID}$^+$ = {StudentID,Name,Course,Instructor,Building}**

  (Using StudentID → Name, Course, Instructor, Building)
- **{Course}$^+$ = {Course,Instructor,Building}**

  (Using Course → Instructor & Instructor → Building)
- **{Instructor,Course}$^+$= {Instructor,Course,Building}**

  (Using Course → Instructor & Instructor → Building)
- **{StudentID,Course}$^+$ ={StudentID,Name,Course,Instructor,Building}**

  (Using StudentID → Name, Course, Instructor, Building)

{StudentID} and {StudentID, Course} are Super Keys, since the closure contains all the attributes of the relation

To check Super key and Candidate key

Step 4: Find Candidate Keys

A **Candidate Key** is a minimal Super Key, meaning it is a Super Key with no extraneous attributes (i.e., you cannot remove any attributes from the set without losing the ability to uniquely identify records).

- **Checking if {StudentID} is minimal:**

  {StudentID}$^{+}$ = {StudentID,Name,Course,Instructor,Building}

  There are no extraneous attributes in this set, so **{StudentID}** is a **Candidate Key**.

- **Checking if {Course,Instructor} is minimal:**

  {StudentID,Course}$^{+}$ ={StudentID,Name,Course,Instructor,Building}

  It does not contain StudentID and Name, so **{Course, Instructor}** is **not a Candidate Key**.

To check Super key and Candidate key

Conclusion:

Super Keys:

- Any set of attributes that includes **{StudentID}** is a Super Key. For example, **{StudentID}**, **{StudentID, Name}**, **{StudentID, Course}**, **{StudentID, Instructor, Course}**, etc.

Candidate Key

- The only **Candidate Key** is **{StudentID}** because it is the minimal set of attributes that uniquely identifies a record.

Primary Key:

- {StudentID} (chosen from the Candidate Keys)

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- **Testing for superkey:**
  - To test if α is a superkey, we compute α^+, and check if α^+ contains all attributes of R.
- **Testing functional dependencies**
  - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$.
  - That is, we compute α^+ by using attribute closure, and then check if it contains β.
  - Is a simple and cheap test, and very useful
- **Computing closure of F**
  - For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover

Canonical Cover

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

Extraneous Attributes

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

Extraneous Attributes (Cont.)

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

    $F = \{ AB \rightarrow CD, A \rightarrow C\}$
  - Then we can show that even after replacing AB $\rightarrow$ CD by AB $\rightarrow$ D, we can still infer AB $\rightarrow$ C and thus AB $\rightarrow$ CD.

Extraneous Attributes

- An attribute of a functional dependency in F is **extraneous** if we can remove it without changing F^+
- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.
  - **Remove from the left side**: Attribute A is **extraneous** in α if
    - $A \in \alpha$ and
    - F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - **Remove from the right side**: Attribute A is **extraneous** in β if
    - $A \in \beta$ and
    - The set of functional dependencies

      $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.
- *Note:* implication in the opposite direction is trivial in each of the cases above, since a "stronger" functional dependency always implies a weaker one

Testing if an Attribute is Extraneous

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

Examples of Extraneous Attributes

- Let $F = \{AB \rightarrow CD, A \rightarrow E, E \rightarrow C\}$
- **To check if C is extraneous in $AB \rightarrow CD$**, we:
  - Compute the attribute closure of AB under $F' = \{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$
  - The closure is ***ABCDE***, which includes *CD*
  - **This implies that C is extraneous**

Canonical Cover

A **canonical cover** for F is a set of dependencies F_c such that

- F logically implies all dependencies in F_c , and
- F_c logically implies all dependencies in F, and
- **No functional dependency in F_c contains an extraneous attribute, and**
- Each left side of functional dependency in F_c is unique. That is, there are no two dependencies in F_c
  - $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ such that
  - $\alpha_1 = \alpha_2$

Canonical Cover

- **To compute a canonical cover for *F*:**

  repeat

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

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

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

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

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

  until (F_c not change)

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

Example: Computing a Canonical Cover

- ***R* = (*A*, *B*, *C*)**
  F* = {*A* → *BC
  B* → *C
  A* → *B
  ***AB* → *C*}**
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
  - Set is now ***{A → BC, B → C, AB → 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 → C* is already present!**
  - Set is now ***{A → BC, B → 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 → B and B → C.***
      - Can use attribute closure of A in more complex cases
- **The canonical cover is:** ***A → B***
  B → C

Dependency Preservation

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**.
- Not 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* = α
    repeat
    for each R_i in the decomposition
    $$t = (result \cap R_i)^+ \cap R_i$$
    $$result = result \cup t$$
    until (*result* does not change)
  - If *result* contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved.
- We apply the test on all dependencies in F to check if a decomposition is dependency preserving
- This procedure takes **polynomial time**, instead of the exponential time required to compute F^+ and $(F_1 \cup F_2 \cup \ldots \cup F_n)^+$

Example

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

Lossless Decomposition

- We can use functional dependencies to show when certain decomposition are lossless.
- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

$$r = \Pi_{R1}(r) \bowtie \Pi_{R2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless decomposition if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- 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 → B, B → C)
- $R_1 = (A, B)$, $R_2 = (B, C)$
  - **Lossless decomposition:**

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

    $R_1 \cap R_2 = \{A\}$ **and** $A \rightarrow AB$
- *Note:*
  - $B \rightarrow BC$

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

Dependency Preservation

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

Dependency Preservation 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, we are forced to repeat the department name once for each time an instructor participates in a *dept_advisor* relationship.
- To fix this, we need to decompose *dept_advisor*
- Any decomposition will not include all the attributes in

  $s\_ID, dept\_name \rightarrow i\_ID$

- Thus, the composition NOT be dependency preserving

Normalization

- Need of normalization
- First Normal Form
- Second Normal Form
- Third Normal Form
- Boyce Codd Normal Form
- Fourth Normal Form
- Fifth Normal Form
- Join Dependencies - Types of Join Dependencies & Example of Join Dependencies

Normalization

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations.
- It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
- Normalization divides the larger table into smaller and links them using relationships.
- The normal form is used to reduce redundancy from the database table.

Need of normalization

- Let R be a relation scheme with a set F of functional dependencies.
- Decide whether a relation scheme R is in **"good"** form.
- In the case that a relation scheme **R is not in "good" form**, **decompose it into a set of relation scheme {R1, R2, ..., Rn}** such that
  - Each relation scheme is in **good form**
  - The decomposition is a **lossless decomposition**
  - Preferably, the decomposition should be **dependency preserving.**

Need of normalization

- The main reason for normalizing the relations is removing the Insertion, Update, and Deletion Anomalies.
- Failure to eliminate anomalies leads to data redundancy and can cause data integrity and other problems as the database grows.
- Normalization consists of a series of guidelines that helps in creating a good database structure.

Data modification anomalies can be categorized into three types:

- **Insertion Anomaly:** Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.
- **Deletion Anomaly:** The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
- **Updation Anomaly:** The update anomaly is when an update of a single data value requires multiple rows of data to be updated.

First Normal Form (1NF)

- A relation will be 1NF if it contains an atomic value.
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
- First normal form disallows the multi-valued attribute, composite attribute, and their combinations.
- **Example:** Relation **EMPLOYEE** is not in 1NF because of multi-valued attribute **EMP_PHONE**.

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

First Normal Form (1NF)

- The above EMPLOYEE table contains multiple values corresponding to EMP_PHONE attribute i.e. these values are non-atomic.
- To overcome this problem, we have to eliminate the non atomic values of EMP_PHONE attribute.

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab

- **Now the new relation does not contain any non-atomic values. so the table is said to be normalized and is in First Normal Form**.

First Normal Form (1NF) - Example

(a)

DEPARTMENT

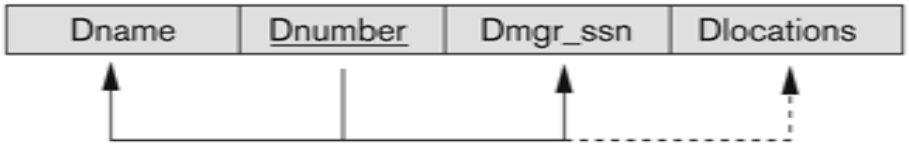


Figure 10.8
Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Example state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.

(b)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

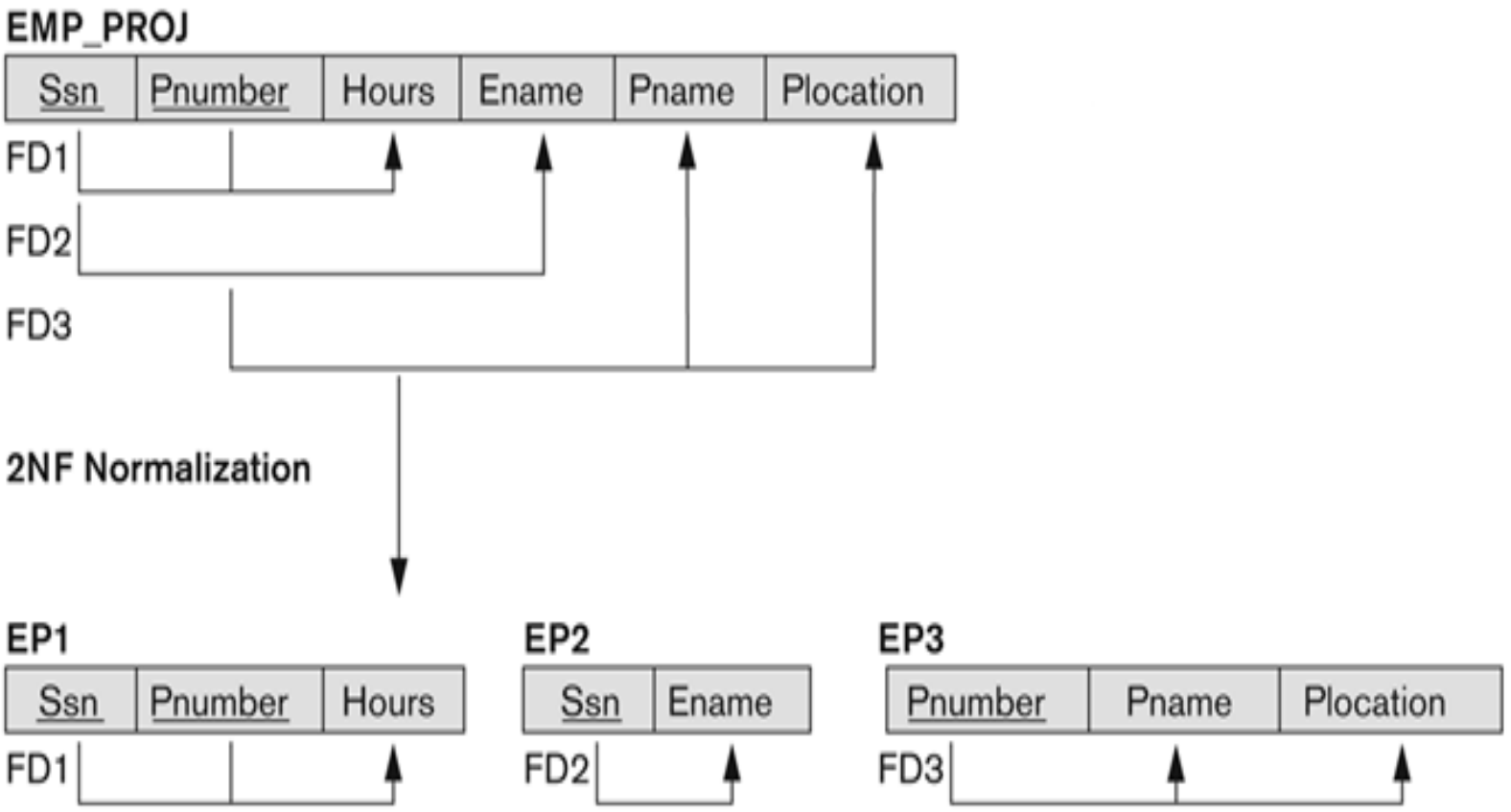
Second Normal Form (2NF)

- In the 2NF, relational must be in 1NF.
- **A relation R is in second normal form (2NF) if every non-prime attribute A in R is fully functionally dependent on the primary key (it should not have Partial Dependency)**.
- R can be decomposed into 2NF relations via the process of 2NF normalization

Partial Dependency:

- When a table has a primary key that is made up of two or more columns, then all the other columns (not included in the primary key) in that table should depend on the entire primary key and not on a part of it.
- If any column(which is not in the primary key) depends on a part of the primary key, then we say we have Partial dependency in the table.

Second Normal Form (2NF) - Example



Third Normal Form (3NF)

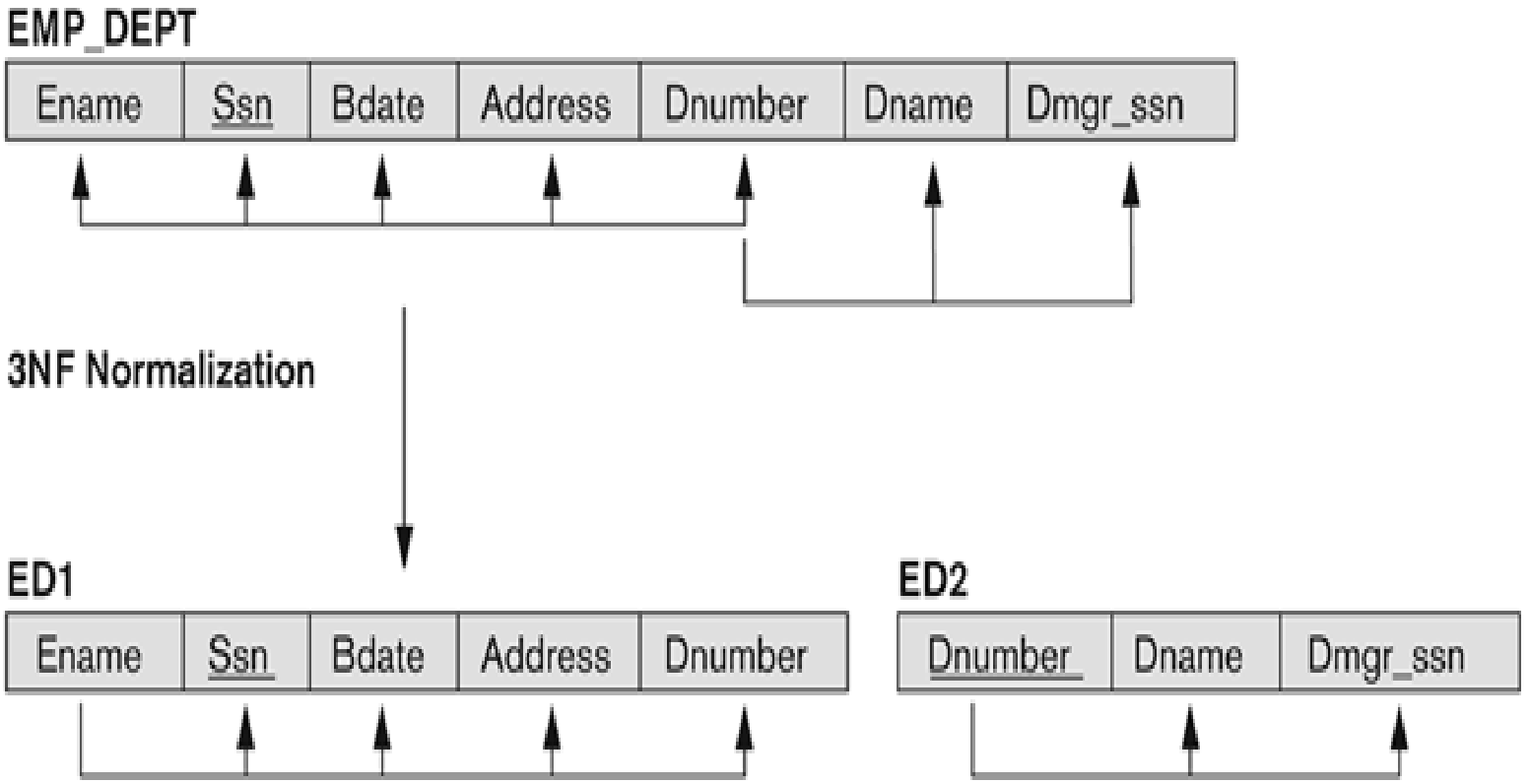
- A relation schema R is in **third normal form (3NF), if it is in 2NF *and* no non-prime attribute A in R is transitively dependent on the primary key**
- R can be decomposed into 3NF relations via the process of 3NF normalization

Transitive functional dependency:

A FD **X -> Z** is said to be **Transitive functional dependency,** if it can be derived from two FDs **X -> Y and Y -> Z.**

- **NOTE:**
  - In X -> Y and Y -> Z, with X as the primary key, we consider this a problem only if Y is not a candidate key.
  - When Y is a candidate key, there is no problem with the transitive dependency .

Third Normal Form (3NF) - Example



Boyce-Codd Normal Form

- A relation schema R is in **Boyce-Codd Normal Form (BCNF),** if whenever an **FD X -> A** holds in R, then **X is a superkey** of R.
- Each normal form is strictly stronger than the previous one
  - Every 2NF relation is in 1NF
  - Every 3NF relation is in 2NF
  - Every BCNF relation is in 3NF
- There exist relations that are in 3NF but not in BCNF

- **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 super key for *R***

Boyce-Codd Normal Form – Example 1

- 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**

Boyce-Codd Normal Form – Example 2

Consider a relation R with attributes (student, teacher, subject).

R

Student	Teacher	Subject
Shayan	R.Das	Database
Shayan	K.Raman	C
Tahira	R.Das	Database
Tahira	N.Gupta	C

FD: { (student, Teacher) -> subject, (student, subject) -> Teacher, (Teacher) -> subject}

- **Candidate keys are (student, teacher) and (student, subject).**
- **The above relation is in 3NF (since there is no transitive dependency).**
- **A relation R is in BCNF if for every non-trivial FD X->Y, X must be a key.**
- The above relation **is not in BCNF**, **because in the FD (teacher->subject), teacher is not a key.**
- This relation suffers with anomalies − For example, if we delete the student Tahira , we will also lose the information that N.Gupta teaches C. This issue occurs because the teacher is a determinant but not a candidate key.

Boyce-Codd Normal Form - Example

So, R is divided into two relations R1(Teacher, Subject) and R2(Student, Teacher).

R1

Teacher	Subject
R.Das	Database
K.Raman	C
N.Gupta	C

R2

Student	Teacher
Shayan	R.Das
Shayan	K.Raman
Tahira	R.Das
Tahira	N.Gupta

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.

Fourth Normal Form (4NF)

A relation R is in 4NF if and only if the following conditions are satisfied:

- **1. It should be in the Boyce-Codd Normal Form (BCNF).**
  2. The table should not have any Multi-valued Dependency.

- 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 $\alpha \cup \beta = R$)
  - α is a superkey for schema R
- If a relation is in 4NF, it is in BCNF

Multivalued Dependencies (MVDs)

- Multivalued dependency occurs when two attributes in a table are independent of each other but, both depend on a third attribute.
- **Multivalued dependency (MVD)** is a type of dependency that exists when a table contains more than one multivalued attribute and changes to one attribute can affect another attribute.

CAR_MODEL	MANUF_MONTH	COLOUR
S2022	FEB	Blue
S2022	MAR	Red
S2023	APR	Blue
S2023	MAY	Red

- The COLOUR and MANUF_MONTH columns are dependent on CAR_MODEL but are independent of each other. Therefore, they can be classified as multivalued, being dependent on CAR_MODEL. The dependencies can be represented as:
- **CAR_MODEL →→ MANUF_MONTH**
- **CAR_MODEL →→ COLOUR**

Fourth Normal Form (4NF) - Example

- So to make the above table into 4NF, we can decompose it into two tables:

CAR_MODEL	MANUF_MONTH
S2022	FEB
S2022	MAR
S2023	APR
S2023	MAY

CAR_MODEL	COLOUR
S2022	Blue
S2022	Red
S2023	Blue
S2023	Red

Fifth Normal Form (5NF)

- A relation is in 5NF if
- **it is in 4NF**
- **does not contain any join dependency and joining should be lossless.**
- 5NF is also known as **Project-join normal form (PJNF).**
- When we decompose the given table to remove redundancy in the data and then compose it again to create the original table , we should not lose any data. It is called a **lossless decomposition**.

Join dependency

- Join dependency for relation R can be stated as

  R=(R1 ⋈ R2 ⋈ R3 ⋈ .....Rn) where R1,R2,R3.....Rn are sub-relation of R and ⋈ is Natural Join Operator.

Fifth Normal Form (5NF) - Example

Example: <employee>

Empname	EmpSkills	EmpJob (Assigned Work)
Tom	Networking	EJ001
Harry	Web Development	EJ002
Katie	Programming	EJ002

The table can be decomposed into 3 tables

<EmployeeSkills>

Emp Name	EmpSkills
Tom	Networking
Harry	Web Development
Katie	Programming

Fifth Normal Form (5NF) - Example

<EmployeeJob>

EmpName	EmpJob
Tom	EJ001
Harry	EJ002
Katie	EJ002

<JobSkills>

EmpSkills	EmpJob
Networking	EJ001
Web Development	EJ002
Programming	EJ002

Join dependency

- Join decomposition is a further generalization of Multivalued dependencies.
- If the join of R1 and R2 over C is equal to relation R, then we can say that a join dependency (JD) exists.
- Where R1 and R2 are the decompositions R1(A, B, C) and R2(C, D) of a given relations R (A, B, C, D).
- Alternatively, R1 and R2 are a lossless decomposition of R.
- A JD ⋈ {R1, R2,..., Rn} is said to hold over a relation R if R1, R2,....., Rn is a lossless-join decomposition.
- The *(A, B, C, D), (C, D) will be a JD of R, if the join of join's attribute is equal to the relation R.
- Here, *(R1, R2, R3) is used to indicate that relation R1, R2, R3 and so on are a JD of R.

Types of Join dependency

Types of Join Dependency

There are two types of Join Dependencies:

- **Lossless Join Dependency:**

  It means that whenever the join occurs between the tables, then no information should be lost, the new table must have all the content in the original table.

- **Lossy Join Dependency:**

  In this type of join dependency, data loss may occur at some point in time which includes the absence of a tuple from the original table or duplicate tuples within the database.

Example of Join dependency

- Suppose we have a table having stats of the company, this can be decomposed into sub-tables to check for the join dependency among them.
- **Table: Company_Stats**

Company	Product	Agent
C1	TV	Aman
C1	AC	Aman
C2	Refrigerator	Mohan
C2	TV	Mohit

Example of Join dependency

- Let us create sub-tables **R1 with attributes Company & Product** and **R2 with attributes Product & Agent**. When we join them we should get the exact same attributes as the original table.

Table: R1

Company	Product
C1	TV
C1	AC
C2	Refrigerator
C2	TV

Table: R2

Product	Agent
TV	Aman
AC	Aman
Refrigerator	Mohan
TV	Mohit

Example of Join dependency

On performing join operation between R1 & R2:

- $R_1 \bowtie R_2$

Company	Product	Agent
C1	TV	Aman
~~**C1**~~	~~**TV**~~	~~**Mohan**~~
C1	AC	Aman
C2	Refrigerator	Mohan
~~**C2**~~	~~**TV**~~	~~**Aman**~~
C2	TV	Mohit

Example of Join dependency

- Here, we can see that we got two additional tuples after performing join **i.e. (C1, TV, Mohan) & (C2, TV, Aman)** these tuples are known as **Spurious Tuple**, which is not the property of Join Dependency.
- Therefore, we will create another relation R3 and perform its natural join with (R1 ⋈ R2).

Company	Agent
C1	Aman
C2	Mohan
C2	Mohit

- **Now on doing natural join of (R1 ⋈ R2) ⋈ R3, we get the original relation.**