DBMS UNIT - 2 Notes

UNIT-II: LOGICAL DATA MODELLING

Logical data modeling is the process of transforming the conceptual schema (e.g., a Fine-granular Design-Specific ER/EER model) into a schema that conforms to the rules and structures of a specific class of Database Management System (DBMS), most commonly a Relational DBMS (RDBMS). This phase bridges the high-level conceptual design with the more concrete implementation details.

1. Overview of Relational Data Model

The relational data model, proposed by E.F. Codd in 1970, is based on the mathematical concept of a relation (from set theory and first-order predicate logic). It represents data in the form of tables (relations).

- **Definition and Terminology:** (p.244)
  (Fig 6.1, p.246)

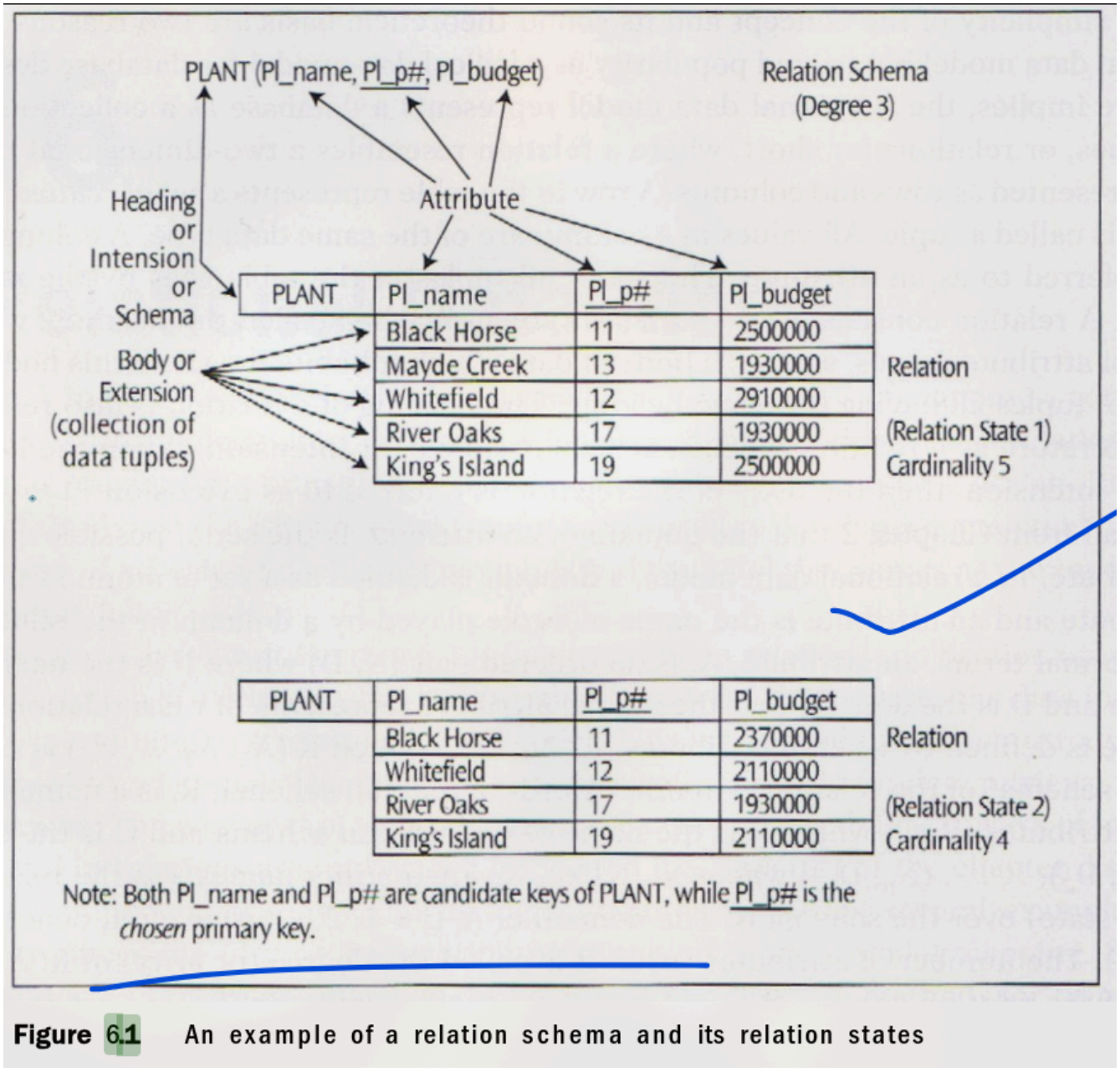


Figure 6.1 An example of a relation schema and its relation states

- *Fig 6.1 shows a relation schema `PLANT (Pl_name, Pl_p#, Pl_budget)` of degree 3, and two relation states (instances) for it, one with cardinality 5 and another with cardinality 4. It also highlights that `Pl_name` and `Pl_p#` are candidate keys, with `Pl_p#` chosen as the primary key.*
- **Relation:** A two-dimensional table consisting of rows (tuples) and columns (attributes). Mathematically, a relation is a subset of the Cartesian product of a list of domains.
- **Attribute (Column/Field):** A named column of a relation that represents a property of the entity or relationship being modeled (e.g., `Pl_name`, `Pl_budget`). Each attribute has a domain.
- **Domain:** The set of all allowable, atomic values for an attribute (or a set of attributes). It defines the data type (e.g., INTEGER, VARCHAR(20), DATE) and any constraints on those values (e.g., range, specific set of values).
- **Tuple (Row/Record):** A single row in a relation. It represents a collection of related data values, with each value corresponding to an attribute of the relation. Each tuple represents a single entity instance or a relationship instance.
- **Relation Schema (or Scheme):** The name of the relation followed by its set of attributes enclosed in parentheses (e.g., `PLANT(Pl_name, Pl_p#, Pl_budget)`). It defines the structure or "heading" of the relation (also called its intension). A relational database schema is a collection of relation schemas.
- **Relation State (or Instance):** The actual data content of a relation at a particular point in time. It is a set of tuples that conform to the relation schema. This is the "body" of the relation (also called its extension).
- **Degree of a Relation:** The number of attributes (columns) in its relation schema.
- **Cardinality of a Relation (State):** The number of tuples (rows) in its relation state.

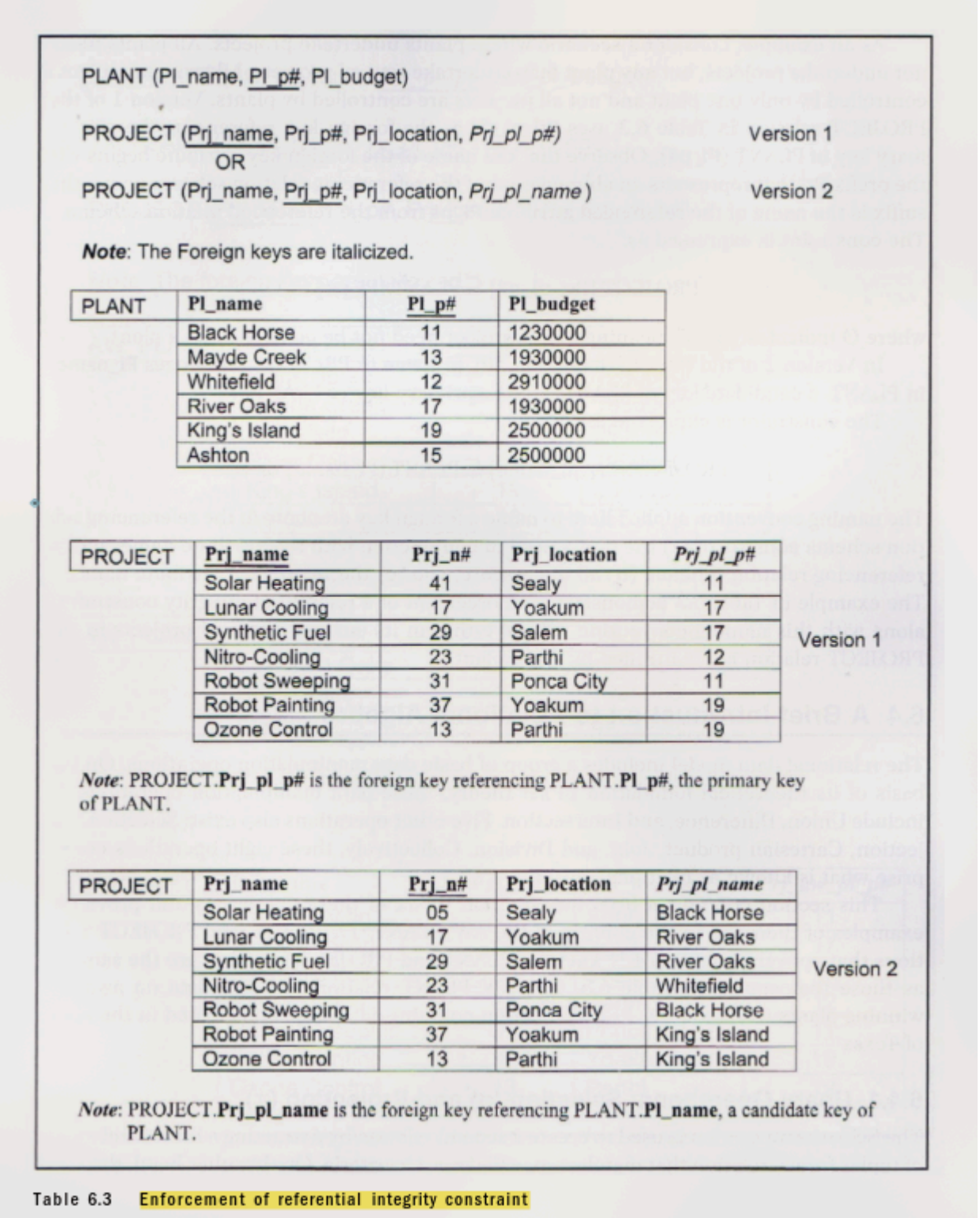
- **Characteristics of a Relation (Formal Properties):** (p.245-247)
  These properties distinguish a formal relation from a simple table or file:
  - **Order of Tuples is Immaterial:** Since a relation state is a *set* of tuples, there is no inherent order among the rows. Tuples can be retrieved in any order unless explicitly sorted.
  - **Order of Attributes is Immaterial (by Name):** The order of columns in the relation schema definition is not significant because attributes are referenced by their names, not by their position.
  - **All Attribute Values are Atomic (First Normal Form - 1NF):** Each cell (intersection of a row and column) in a relation must contain only a single, indivisible (atomic) value from the attribute's domain. This means no composite attributes (attributes that can be broken down further) or multi-valued attributes (attributes that can hold a set of values) are allowed directly within a single cell of a relation. *This is a fundamental assumption of the basic relational model.*
  - **Each Tuple is Distinct (No Duplicate Tuples):** Since a relation state is a set, all tuples within it must be unique. This is guaranteed by the presence of at least one candidate key (and thus a primary key).
  - **Attribute Naming:** Within a single relation schema, all attribute names must be unique. However, the same attribute name can appear in different relation schemas (though the

Universal Relation Schema assumption, sometimes adopted for simplicity in mapping from ER, prefers globally unique attribute names or careful qualification).

- **Null Values:** A special value, `NULL`, can be used to represent missing or inapplicable information for an attribute in a tuple, provided the attribute's domain allows for nulls (primary key attributes cannot be null).

- **Integrity Constraints [PYQ - Key concepts are important]** (p.247-254)
  Rules that data must adhere to in order to maintain accuracy, consistency, and validity.
  - **Key Constraints [PYQ Q1b]:** Ensure unique identification of tuples.
    - **Superkey:** An attribute, or a set of attributes, whose values uniquely identify each tuple in any possible relation state of a relation schema. (e.g., in `PRESCRIPTION-A(Rx_rx#, Rx_pat#, Rx_medcode, Rx_dosage)`, `{Rx_rx#, Rx_pat#, Rx_medcode, Rx_dosage}` is a superkey, as is `{Rx_rx#}`).
    - **Candidate Key:** A minimal superkey. "Minimal" means that if any attribute is removed from the candidate key, the remaining set of attributes is no longer a superkey. A relation schema can have multiple candidate keys. (e.g., `Rx_rx#` is a candidate key; `{Rx_pat#, Rx_medcode}` is also a candidate key for `PRESCRIPTION-A`).
    - **Primary Key (PK):** One candidate key that is chosen by the database designer to be the main identifier for tuples in a relation. Its attributes are underlined in the schema. The choice among candidate keys is often based on factors like minimality (fewer attributes), stability (values rarely change), and simplicity.
    - **Alternate Key (Secondary Key):** Any candidate key that is not selected as the primary key.
  - **Entity Integrity Constraint:** (p.251) States that no attribute participating in the primary key of a relation can have a null value. This ensures that every tuple in a relation is uniquely identifiable and has a complete identifier.
  - **Referential Integrity Constraint [PYQ Q1b]:** (p.252-254) Maintains consistency between tuples in two (possibly the same) relations that are related. It involves foreign keys.

(Fig 6.3, p.253)



PLANT (Pl_name, Pl_p#, Pl_budget)

PROJECT (Prj_name, Prj_p#, Prj_location, *Prj_pl_p#*) Version 1

OR

PROJECT (Prj_name, Prj_p#, Prj_location, *Prj_pl_name*) Version 2

Note: The Foreign keys are italicized.

PLANT	Pl_name	Pl_p#	Pl_budget
	Black Horse	11	1230000
	Mayde Creek	13	1930000
	Whitefield	12	2910000
	River Oaks	17	1930000
	King's Island	19	2500000
	Ashton	15	2500000

PROJECT	Prj_name	Prj_n#	Prj_location	*Prj_pl_p#*
	Solar Heating	41	Sealy	11
	Lunar Cooling	17	Yoakum	17
	Synthetic Fuel	29	Salem	17
	Nitro-Cooling	23	Parthi	12
	Robot Sweeping	31	Ponca City	11
	Robot Painting	37	Yoakum	19
	Ozone Control	13	Parthi	19

Version 1

Note: PROJECT.**Prj_pl_p#** is the foreign key referencing PLANT.**Pl_p#**, the primary key of PLANT.

PROJECT	Prj_name	Prj_n#	Prj_location	*Prj_pl_name*
	Solar Heating	05	Sealy	Black Horse
	Lunar Cooling	17	Yoakum	River Oaks
	Synthetic Fuel	29	Salem	River Oaks
	Nitro-Cooling	23	Parthi	Whitefield
	Robot Sweeping	31	Ponca City	Black Horse
	Robot Painting	37	Yoakum	King's Island
	Ozone Control	13	Parthi	King's Island

Version 2

Note: PROJECT.**Prj_pl_name** is the foreign key referencing PLANT.**Pl_name**, a candidate key of PLANT.

Table 6.3 Enforcement of referential integrity constraint

- *Fig 6.3 illustrates the relational schema for* `EMPLOYEE`, `PLANT`, *and* `BUILDING` *after mapping from an ERD.* `Emp_pl_name` *in* `EMPLOYEE` *is a foreign key referencing* `Pl_name` *in* `PLANT`. `Bld_pl_p#` *in* `BUILDING` *references* `Pl_p#` *in* `PLANT`.
- **Foreign Key (FK):** An attribute (or set of attributes) in one relation (the **referencing relation**) whose values are required to match the values of a candidate key (usually the primary key) of some tuple in another (or possibly the same) relation (the **referenced relation**), or the foreign key values must be wholly null (if permitted).
- This constraint ensures that a tuple in the referencing relation cannot refer to a non-existent tuple in the referenced relation.

- **Example:** If `PROJECT.Prj_pl_p#` is an FK referencing `PLANT.Pl_p#`, then any `Prj_pl_p#` value in `PROJECT` must either exist as a `Pl_p#` in `PLANT` or be null.
- **Self-Referencing Foreign Key:** An FK that references a PK within the same relation (e.g., `Emp_supervisor#` in `EMPLOYEE` referencing `Emp_emp#` in `EMPLOYEE`).
- **Actions on Violation (when a referenced PK is updated or deleted):**
  - **RESTRICT (or NO ACTION):** Disallow the update/delete on the referenced tuple if dependent FKs exist.
  - **CASCADE:** If the referenced PK is deleted, automatically delete all referencing FK tuples. If PK is updated, update FK values.
  - **SET NULL:** If the referenced PK is deleted/updated, set the corresponding FK values in referencing tuples to NULL (only if FK attributes allow nulls).
  - **SET DEFAULT:** Set FK values to a predefined default value (only if FK attributes have a default and allow it).

2. Mapping ER Model to a Logical Schema (Relational Schema) [PYQ Q1d, Q4a] (p.261-277)
This is the core process of transforming the conceptual ER/EER model (specifically, the Fine-granular Design-Specific ER/EER model) into a set of relation schemas. The goal is to preserve as much information (semantics) from the conceptual model as possible.
(The textbook primarily discusses information-reducing mapping first, which is the traditional approach, and then introduces an information-preserving grammar).

- **Mapping Entity Types (Base and Weak):** (p.261)
  (Fig 6.2, 6.3, p.262)
  - **Strong (Base) Entity Type:**
    1. Create a new relation schema for the entity type.
    2. Include all simple, stored attributes of the entity type as attributes of the relation.
    3. For composite attributes, include only their simple atomic components as attributes in the relation.
    4. Choose one of the candidate keys of the entity type as the primary key for the relation. Underline the primary key attributes.
    - Derived attributes are generally not included directly but their derivation logic is noted.
  - **Weak Entity Type:**
    1. Create a new relation schema for the weak entity type.
    2. Include all simple, stored attributes of the weak entity type as attributes of the relation.
    3. Include the primary key attribute(s) of the owner (identifying) entity type(s) as foreign key attribute(s) in this new relation.
    4. The primary key of this new relation is the combination of the primary key(s) of the owner entity type(s) and the partial key (discriminator) of the weak entity type.

    - Example: `BUILDING(Bld_building, Bld_pl_p#)` where `Bld_pl_p#` is FK from `PLANT` and `Bld_building` is the partial key. PK is `{Bld_pl_p#, Bld_building}`.

- **Mapping Relationship Types:** (p.262-265) The mapping depends on the cardinality ratio and participation constraints.
    - **1:N (One-to-Many) Binary Relationship:**
        - **Foreign Key Approach (Standard & Most Common):**
            1. Identify the relation schemas corresponding to the entity types participating on the '1-side' (parent) and 'N-side' (child).
            2. Include the primary key of the '1-side' (parent) relation as a foreign key in the 'N-side' (child) relation. This FK links child tuples to their parent tuple.
            3. Any attributes of the 1:N relationship itself are also included as attributes in the 'N-side' (child) relation.
            4. If participation of the child in the relationship is optional, the FK in the child relation can accept null values. If total, it cannot be null (unless business rules allow an orphan child temporarily).
        - **Cross-Referencing Design (Separate Relationship Relation):** (p.264, *Fig 6.5, 6.6, 6.7, p.265*)
          Used less frequently, typically when trying to avoid nulls in FKs if the child's participation is optional, or if the relationship has many attributes and migrating them to the child clutters it.
            1. Create a new relation schema (often called a "relationship relation" or "junction table" though it's 1:N) specifically for the 1:N relationship.
            2. Include the primary key of the '1-side' entity as a foreign key in this new relation.
            3. Include the primary key of the 'N-side' entity as a foreign key in this new relation.
            4. The primary key of this new relationship relation is typically the primary key of the 'N-side' entity (as each N-side instance is related to at most one 1-side instance).
            5. Attributes of the relationship become attributes of this new relation.
               (Fig 6.6, p.265 illustrates `WORKS_IN` as a separate relation for the 1:N relationship between `EMPLOYEE` (N) and `PLANT` (1) when `EMPLOYEE` participation is optional).

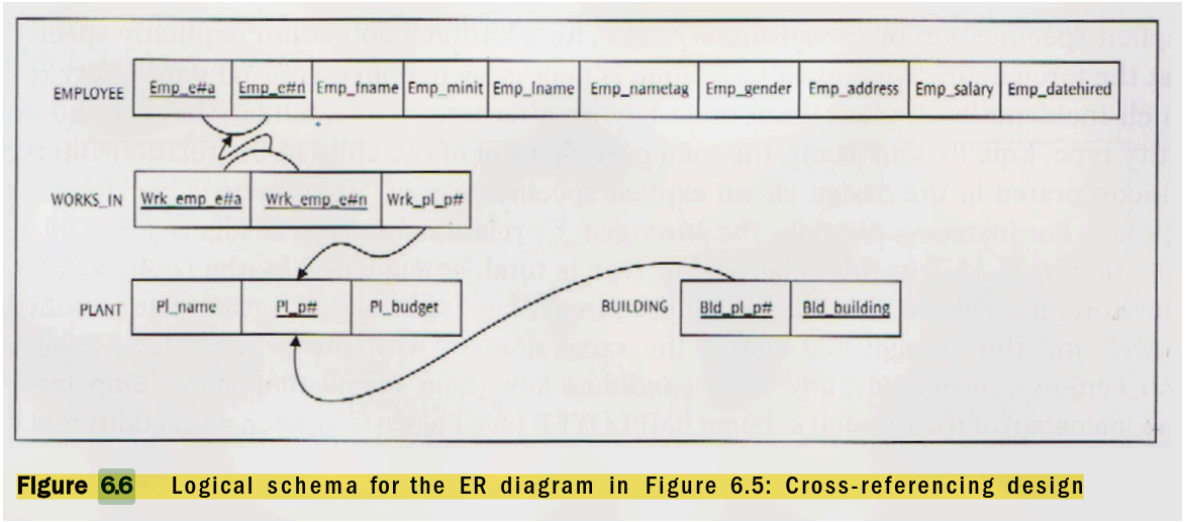


Figure 6.6 Logical schema for the ER diagram in Figure 6.5: Cross-referencing design

        - *Fig 6.6 shows a cross-referencing design where `WORKS_IN` becomes a separate table to map the relationship between `EMPLOYEE` and `PLANT` when an employee may or may not work in a plant (optional participation).*
- **M:N (Many-to-Many) Binary Relationship:**
    1. Create a new relation schema (often called a junction table, associative table, or intersection table) to represent the M:N relationship.
    2. Include the primary key attributes from *both* participating entity types as foreign keys in this new relation.
    3. The primary key of this new relationship relation is usually the combination of these two (or more, if composite PKs) foreign key attributes.
    4. Any attributes of the M:N relationship itself become attributes of this new relation.
    - Participation constraints (total/partial) influence whether the FKs in this new table can be null (though typically they form the PK and thus cannot be null).
- **1:1 (One-to-One) Binary Relationship:** (p.264-265) Several options exist, choice depends on participation constraints.
    - **Option 1 (Foreign Key in One Relation):**
        1. Choose one of the participating relations (say R1, corresponding to entity type E1) and include the primary key of the other relation (R2, for E2) as a foreign key in R1.
        2. This FK in R1 must also be constrained to be unique (e.g., by defining it as an alternate key) to enforce the 1:1 nature.
        3. If participation of E1 in the relationship is total, R1 is a good choice to host the FK (as it won't be null). If participation of E1 is optional, the FK in R1 can be null.
        4. Attributes of the 1:1 relationship can be placed in the relation that hosts the FK.
           (Fig 6.8-6.10, p.266-267 show mapping 1:1 where `PLANT` has total participation in `Managed_by` with `EMPLOYEE`. FK `Mgr_emp_id` (referencing `EMPLOYEE`) is placed in `PLANT` and is NOT NULL and UNIQUE).
    - **Option 2 (Merged Relation):**
      If both entity types have *total participation* in the 1:1 relationship, and they are semantically very close, consider merging them into a single relation schema. The PK can be chosen from either original entity type. This eliminates the need for an FK but might lead to many nulls if attributes are not shared.
    - **Option 3 (Separate Relationship Relation / Cross-Referencing):**
      Similar to M:N mapping. Create a new relation for the 1:1 relationship. Include PKs of both participating entities as FKs. The PK of this new relation can be the PK of either entity (both must be constrained unique). Used if both participations are optional, or if the relationship has significant attributes.
      (Fig 6.11-6.16, p.267-269 show mapping 1:1 where both `EMPLOYEE` and `PLANT` have partial participation in `Managed_by`. A separate `MANAGED_BY` relation is created).
- **Recursive (Unary) Relationship:**

- **1:N Recursive:** Add the primary key of the entity type as a foreign key *within the same relation schema*. This FK attribute must be given a different name (role name) to distinguish it from the PK (e.g., `EMPLOYEE(EmpID (PK), EmpName, SupervisorID (FK references EmpID))`).
- **M:N Recursive:** Create a new relation schema (junction table) for the relationship. This new relation will have two foreign key attributes, both referencing the primary key of the original entity type. These two FK attributes must have different role names (e.g., `PREREQUISITE(CourseID (FK), PrerequisiteID (FK))`). The PK of this junction table is the combination of these two FKs.
- **1:1 Recursive:** Similar to 1:N recursive, but the FK added to the same relation must also be constrained to be unique.

- **Mapping Multi-valued Attributes:** (Already resolved in Fine-Granular ER by creating a weak entity).
  If still present conceptually at this stage: Create a new relation schema for the multi-valued attribute. This new relation will include:
  1. The primary key of the original entity type (as a foreign key).
  2. The multi-valued attribute itself.
  3. The primary key of this new relation is the combination of the FK (from original entity) and the multi-valued attribute.
- **Mapping N-ary Relationships (Higher Degree, e.g., Ternary) [PYQ Q5b]:** (Already resolved in Fine-Granular ER by creating a gerund entity).
  If still present conceptually at this stage:
  1. Create a new relation schema to represent the n-ary relationship.
  2. Include the primary key attributes from *all* participating entity types as foreign keys in this new relation.
  3. The primary key of this new relationship relation is typically the combination of all these foreign key attributes.
  4. Any attributes of the n-ary relationship itself become attributes of this new relation.
- **Information-Preserving Mapping Grammar:** (p.277-281)
  The textbook introduces a more formal grammar for logical schemes that aims to preserve more of the conceptual model's semantics than traditional relational schema notation. This includes explicitly representing:
  - Atomic attributes with their type (t) and size (s).
  - Primary keys (underlined).
  - Mandatory attributes (marked with •).
  - Alternate keys (Q[...]).
  - Derived attributes (dotted underline).
  - Foreign keys (italicized), along with (min,max) participation constraints from the ERD and deletion rules (C, N, D, R).

- The label of the parent scheme (e.g., L2) is coded on top of the FK box.
  (Fig 6.23, p.280 shows an example of this grammar applied to a simple ERD. Fig 6.24, p.282 shows the Bearcat Inc. logical schema using this information-preserving grammar).

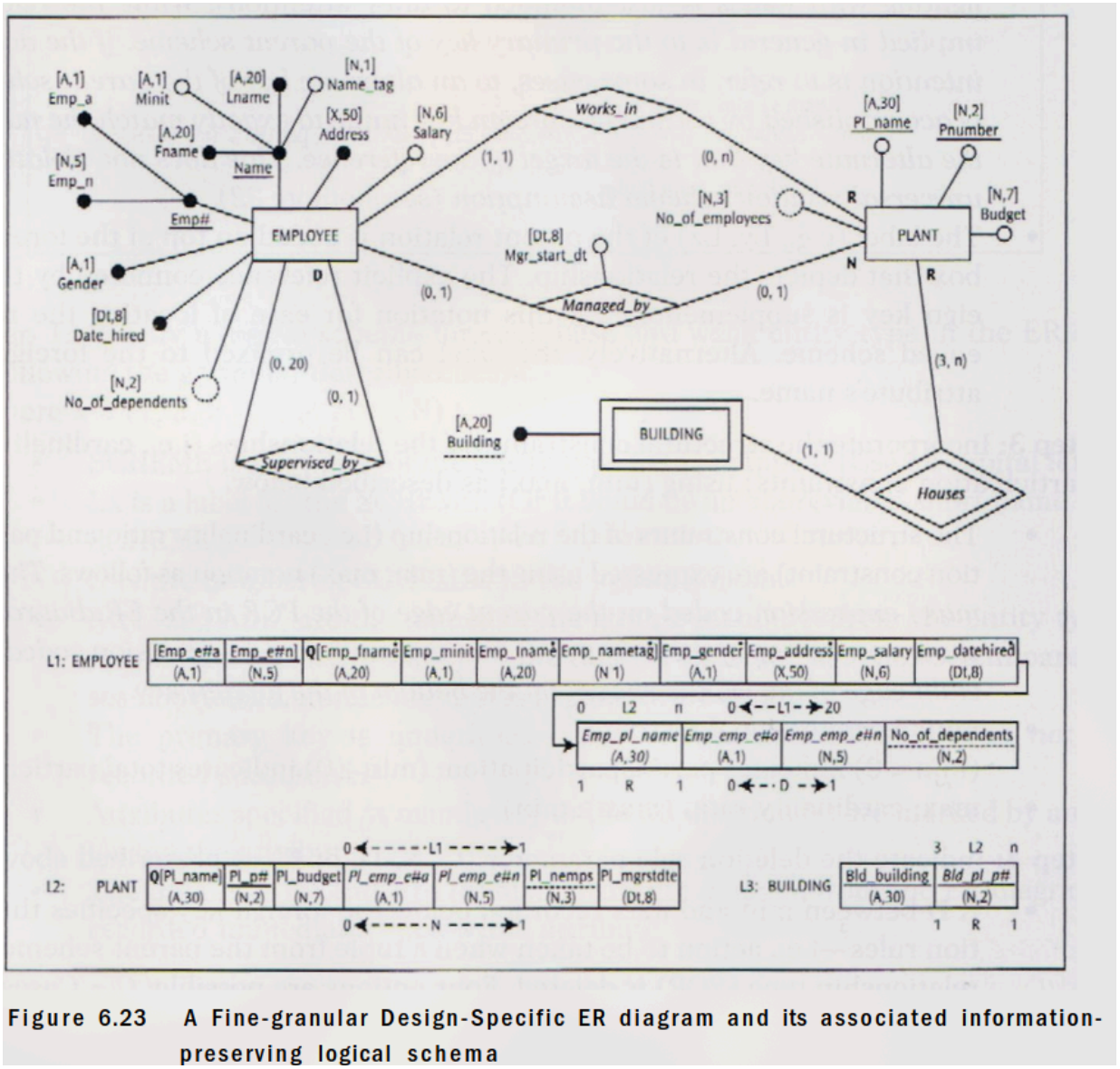


Figure 6.23 A Fine-granular Design-Specific ER diagram and its associated information-preserving logical schema

- *Fig 6.24 is the comprehensive logical schema for Bearcat Inc. using the information-preserving grammar, capturing details like alternate keys (e.g., `Q[Pl_name]` in PLANT), mandatory attributes (e.g., `Emp_e#a•`), foreign keys with (min,max) and deletion rules (e.g., `Emp_pl_p#` in EMPLOYEE with `0 L2 N` above it and `1 R 1` below it).*

3. Mapping EER Model (Enhanced ER) Constructs to a Logical Schema (Relational Schema) [PYQ Q4a] (p.281-295)

EER models introduce superclass/subclass (SC/sc) relationships (specialization/generalization, hierarchy, lattice, categorization) and aggregation. Mapping these to relations requires specific strategies.

- **Mapping Specialization/Generalization:** (p.281-284)
  The core idea is to represent the "is-a" relationship. Several options exist, and the best choice depends on the specific constraints (disjoint/overlapping, total/partial) and the characteristics of the subclasses.

(Fig 6.25, p.283 shows an EER diagram for `STUDENT_ATHLETE` specialized into `FOOTBALL_PLAYER`, `BASKETBALL_PLAYER`, `BASEBALL_PLAYER`. Fig 6.26, p.284 shows three mapping solutions for this).

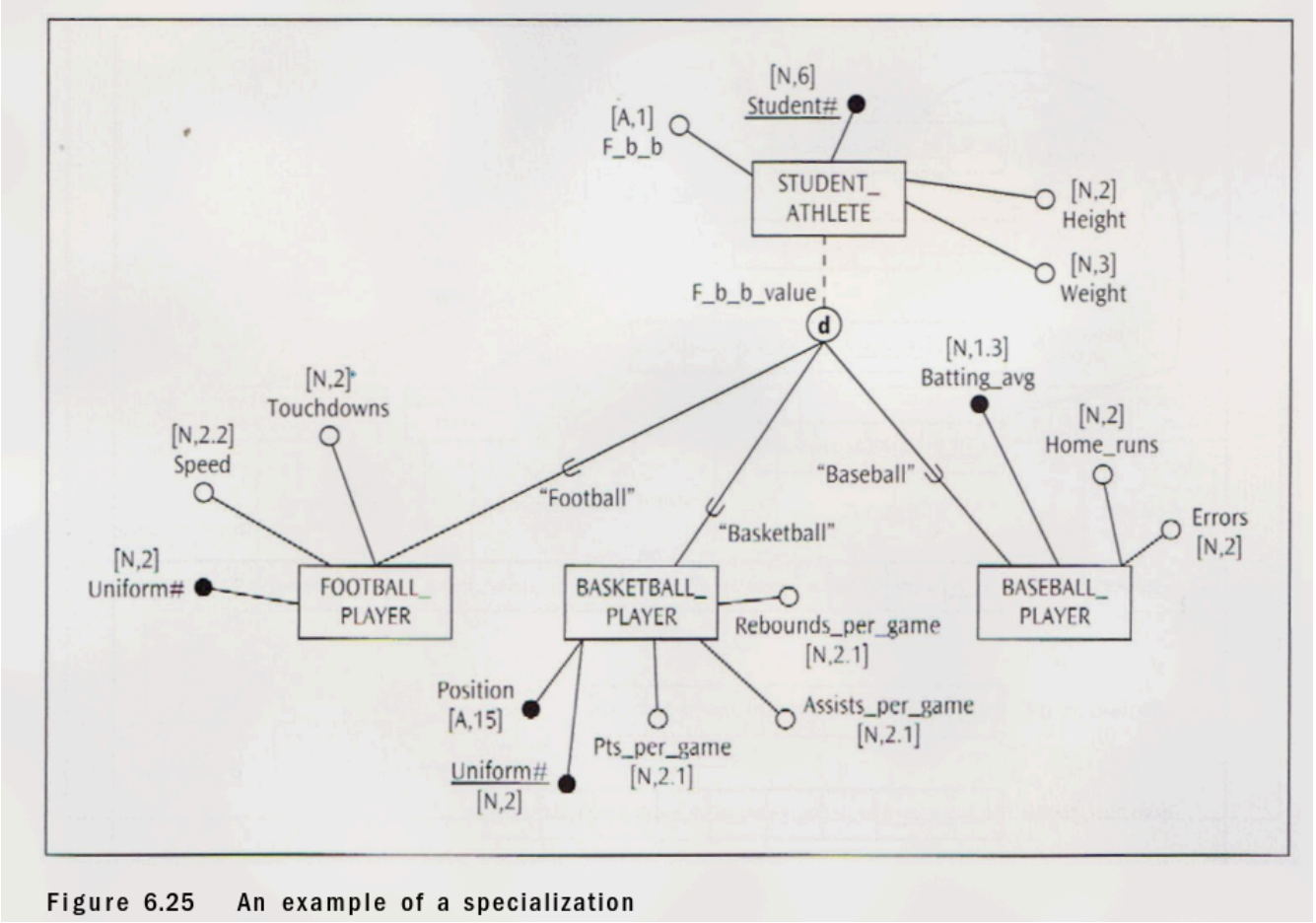


Figure 6.25 An example of a specialization

- *Fig 6.25: `STUDENT_ATHLETE` (SC) with specific attributes. Specialization into `FOOTBALL_PLAYER`, `BASKETBALL_PLAYER`, `BASEBALL_PLAYER` (sc's), each with their own specific attributes. The specialization is overlapping ('o' in circle) and partial (dotted line from*

SC).

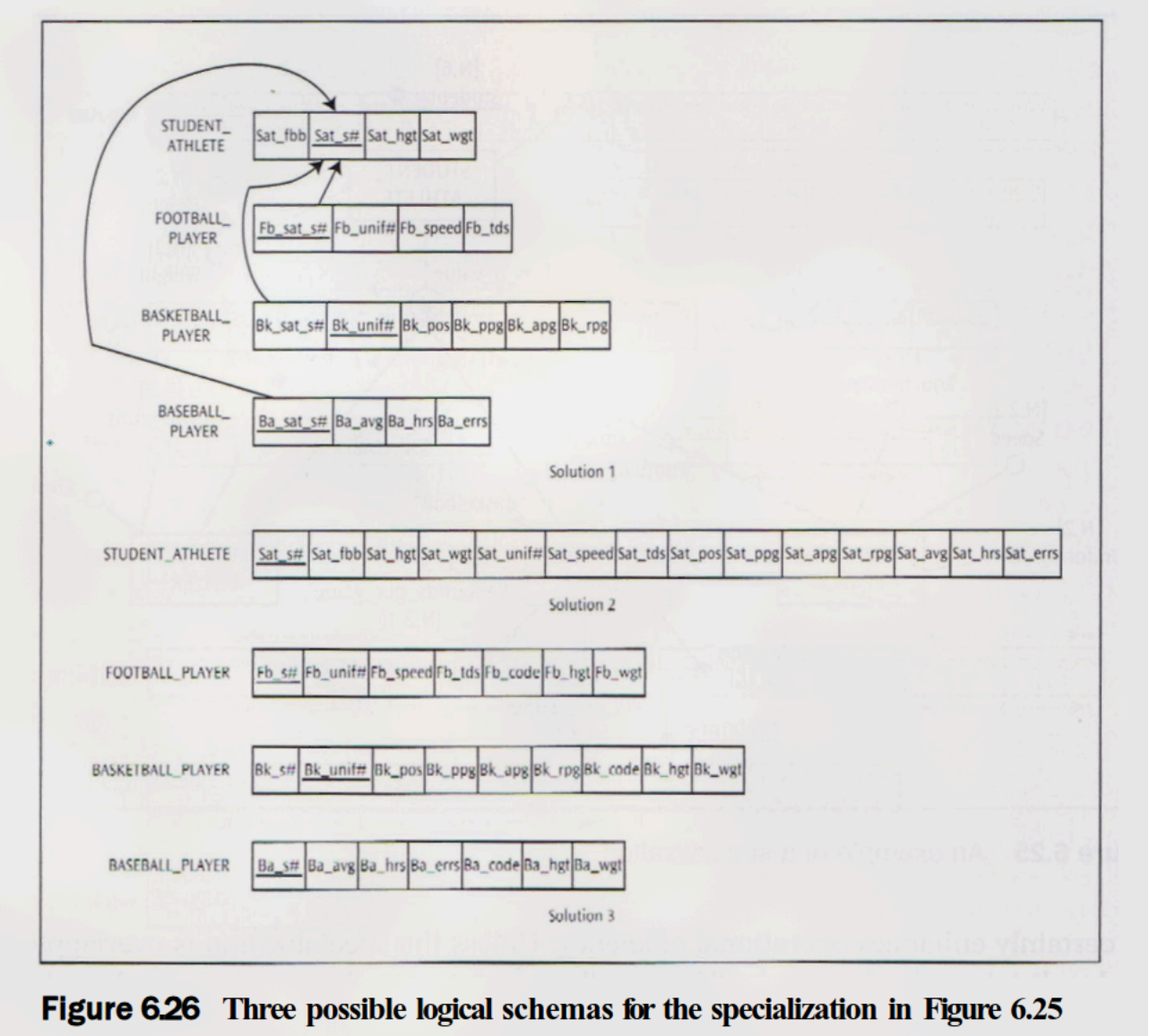


Figure 6.26 Three possible logical schemas for the specialization in Figure 6.25

- *Fig 6.26 illustrates three solutions for mapping the specialization in Fig 6.25:*
- **Option 1 (Multiple Relations - One for SC, One for each SC):** This is the most general and often preferred approach.
  1. Create a relation for the superclass (SC) with its attributes. The PK of the SC relation is its own PK.
  2. Create a separate relation for each subclass (sc) that includes only the specific attributes of that subclass.
  3. The primary key of each subclass relation is the *same* as the primary key of the superclass relation. This PK in the subclass relation also acts as a foreign key referencing the superclass relation.

  - **Pros:** Clearly separates SC and sc attributes. No attribute redundancy for SC attributes. Efficient if subclasses have many specific attributes or participate in distinct relationships. Works for all disjoint/overlapping and total/partial constraints.
  - **Cons:** Retrieving all information for a subclass instance requires a join between the SC relation and the specific sc relation.
- **Option 2 (Single Relation - SC Only, with All SC and SC Attributes):**

1. Create a single relation that includes all attributes of the superclass *and* all specific attributes of *all* its subclasses.
2. Add a "type" attribute (discriminator attribute) to indicate which subclass an instance belongs to (if disjoint) or to indicate membership in multiple subclasses (if overlapping, might need multiple boolean type attributes).
3. The PK is the PK of the original superclass.

- **Pros:** Avoids joins to retrieve complete information for any instance. Simple structure.
- **Cons:** Can lead to many null values if subclasses have many distinct attributes and an instance belongs to only one or few subclasses. This wastes space and can make queries complex (e.g., checking for nulls). Less clear separation of concerns. Less suitable if subclasses participate in very different relationships.

- **Option 3 (Multiple Relations - One for Each SC Only, No SC Relation):**
  1. Create a separate relation for *each subclass*.
  2. Each subclass relation includes *all attributes of the superclass* (inherited) plus its own specific attributes.
  3. The PK of each subclass relation is the PK of the original superclass.

  - **Pros:** No joins needed to get full subclass information.
  - **Cons:** This option is generally viable **only if the specialization is total and disjoint**. If partial, SC instances not belonging to any sc would be lost. If overlapping, SC attributes would be duplicated across multiple subclass relations for an instance belonging to multiple sc's. Significant redundancy of superclass attributes if many subclasses.

- **Mapping Specialization Hierarchy:** (p.284-285)
  A hierarchy involves multiple levels of specialization.
  1. Apply one of the SC/sc mapping options (usually Option 1 or Option 2 from above) for the top-level specialization.
  2. Then, for any subclass that itself is a superclass in a lower-level specialization, repeat the mapping process.

  - If Option 1 is used consistently, each entity type in the hierarchy (SC or sc) becomes a separate relation. An instance at a lower level will have its PK referencing the PK of its parent relation, and so on, up to the root SC.
    (Fig 6.27, p.285 shows an EER hierarchy. Fig 6.28, p.285 shows its mapping using Option 1 – separate relations for `STUDENT_ATHLETE`*,* `FOOTBALL_PLAYER`*,* `TEAM_CAPTAIN`*, etc., with PKs*

acting as FKs up the hierarchy).

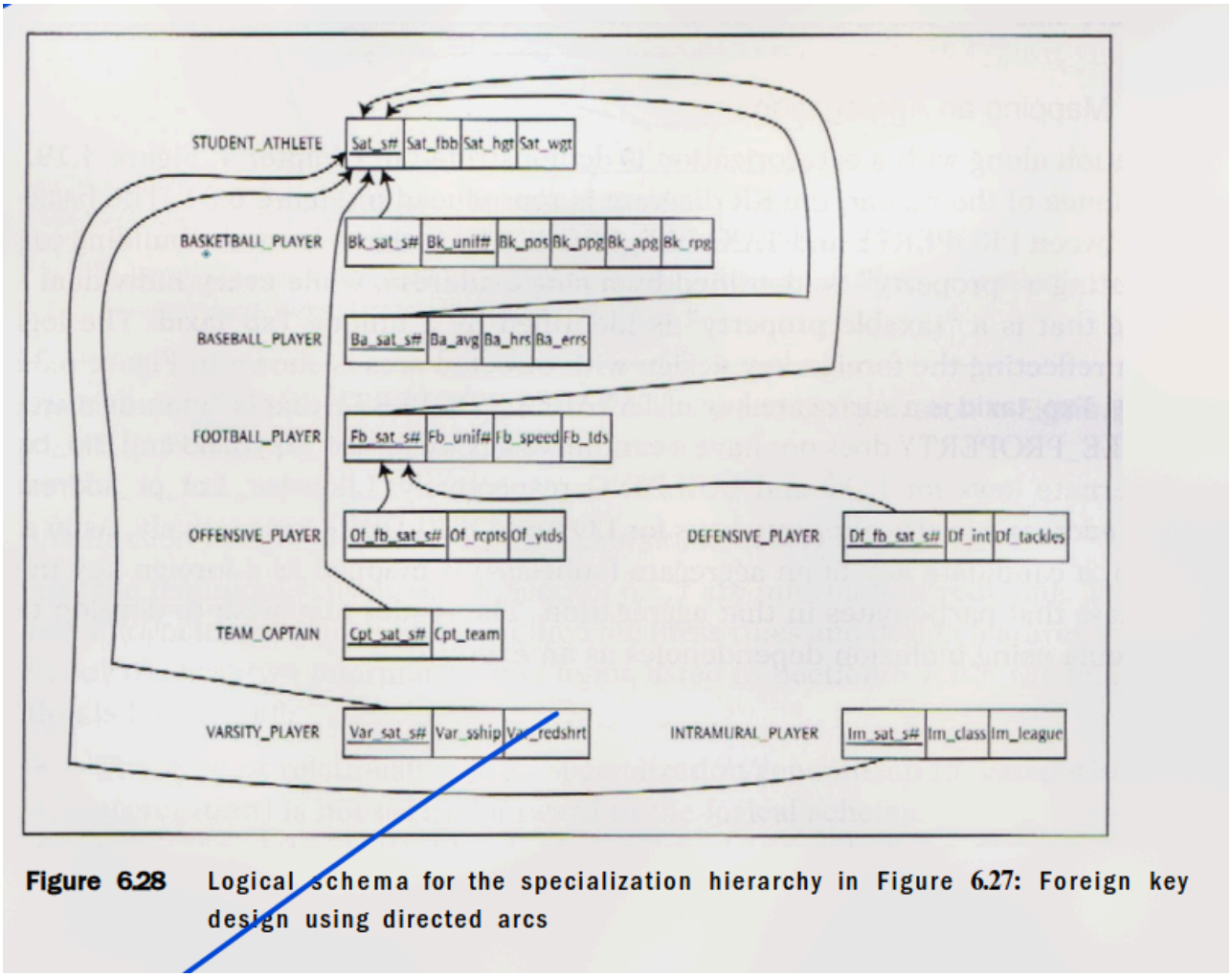


Figure 6.28 Logical schema for the specialization hierarchy in Figure 6.27: Foreign key design using directed arcs

- *Fig 6.28 shows the mapping of the hierarchy from Fig 6.27:* `Football_Player(SA_s#, Fb_unif#, ...)` *where* `SA_s#` *is PK and FK to* `Student_Athlete`. `Offensive_Player(SA_s#, Of_rcpts, ...)` *where* `SA_s#` *is PK and FK to* `Football_Player`.

- **Mapping Specialization Lattice (Shared Subclass):** (p.285-287)
  A shared subclass inherits from multiple superclasses (from different specializations).
  1. Create a relation for the shared subclass.
  2. Its primary key is typically derived from one of its "primary" superclass paths or a new surrogate key if identification is complex.
  3. Include the primary keys of *all* its direct superclasses as foreign keys in the shared subclass relation. This handles the multiple inheritance.

  - The other superclasses and non-shared subclasses are mapped using standard SC/sc options. *(Fig 6.29, p.286 EER with lattice. Fig 6.30, p.287 shows its mapping.* `PUBLIC_SCHOOL` *is a shared subclass of* `NOT_FOR_PROFIT_ORGANIZATION` *and* `SCHOOL`*. In the* `PUBLIC_SCHOOL` *relation,* `Psh_nfp_sp_s#` *(FK to* `NOT_FOR_PROFIT_ORG.Sp_s#`*) and* `Psh_sch_name` *(FK to* `SCHOOL.Sch_name`*) would be included).*

- **Mapping Categorization:** (p.286-287)
  A category (subclass) is a union of different superclass types.
  1. Create a relation for the category with its specific attributes.

2. The primary key of the category relation is typically a **surrogate key** (a newly generated unique ID), as instances come from different superclasses which might have different PK structures.
3. For each superclass that contributes to the category:
    - Add the primary key of the *category* relation as a foreign key in the *superclass* relation. This FK in the superclass identifies which instance of the superclass (if any) belongs to the category. This is opposite to specialization where the subclass PK references the superclass PK.
    - This FK in the superclass relations should be constrained to be unique if an SC instance can belong to the category at most once.
      (Fig 6.29 EER. Fig 6.30 mapping. `SPONSOR` *is a category of* `CHURCH`, `SCHOOL`, `INDIVIDUAL`*. A surrogate key* `Sp_s#` *is created for* `SPONSOR`*. Then,* `Ch_sp_s#` *(FK to* `SPONSOR.Sp_s#`*) is added to* `CHURCH`, `Sch_sp_s#` *to* `SCHOOL`, `Ind_sp_s#` *to* `INDIVIDUAL`*.)*

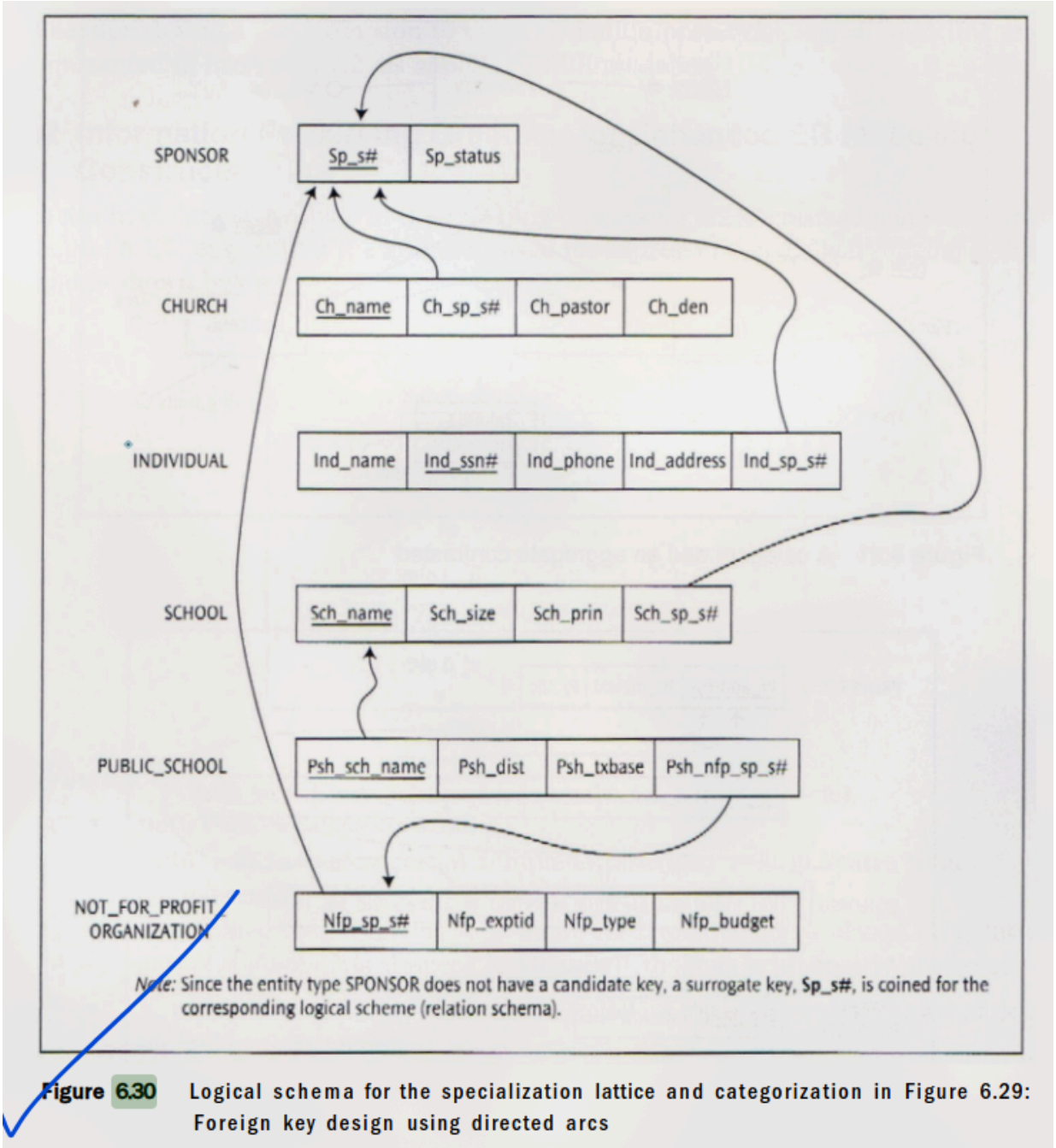


Figure 6.30 Logical schema for the specialization lattice and categorization in Figure 6.29: Foreign key design using directed arcs

- *Fig 6.30 shows* `SPONSOR(Sp_s#, Sp_status, ...)`. `CHURCH(Ch_name, Ch_sp_s# (FK to SPONSOR), ...)`. `SCHOOL(Sch_name, Sch_sp_s# (FK to SPONSOR), ...)`. `INDIVIDUAL(Ind_ssn#, Ind_sp_s# (FK to SPONSOR), ...)`.

- **Mapping Aggregation [PYQ Q5b]:** (p.287)
  Aggregation represents a "whole-part" or "is-part-of" relationship where a "whole" entity is composed of "part" entities.
  1. Create a relation for the aggregate (the "whole" entity type acting as a subclass).
  2. Include the primary keys of all the superclasses (the "part" entity types) as foreign keys in the aggregate relation.
  3. The primary key of the aggregate relation is often the combination of these foreign keys, or a surrogate key if preferred.
  4. The aggregate relation will also have its own specific attributes.
     (Fig 6.31, p.288 EER with aggregation and category. Fig 6.32, p.288 shows mapping. `TAXABLE_PROPERTY` *is an aggregate of* `LOT` *and* `BUILDING`*. The* `TAXABLE_PROPERTY` *relation might have* `Txp_taxid (PK)`, `Lot_lot# (FK)`, `Bld_bldg# (FK)`, `Txp_appraisal`*.)*

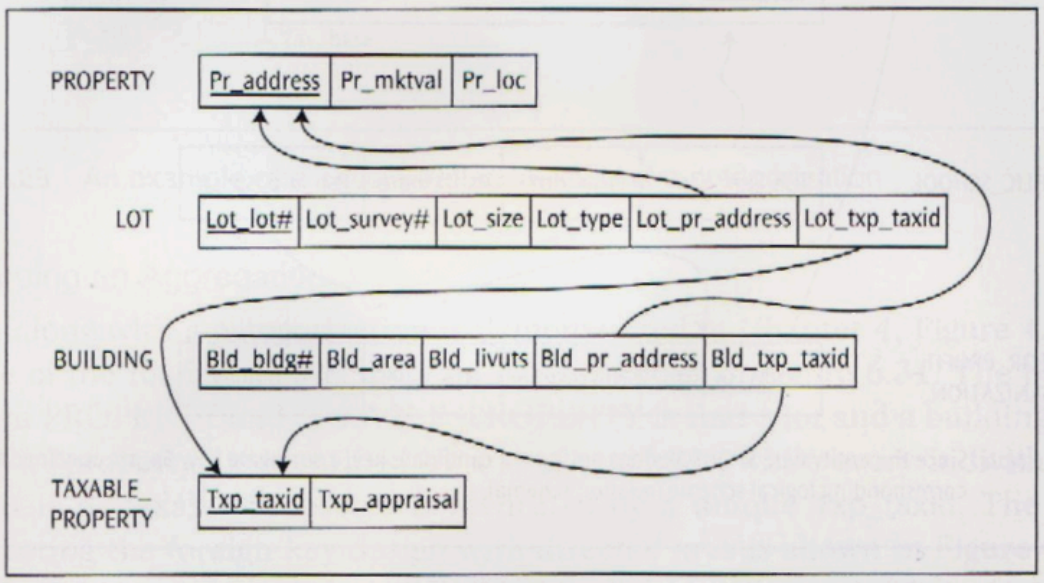


Figure 6.32 Logical schema for the category and aggregate in Figure 6.31: Foreign key design using directed arcs

  - *Fig 6.32:* `TAXABLE_PROPERTY(Txp_taxid, Lot_lot#(FK), Bld_bldg#(FK), Txp_appraisal)`. `LOT` *and* `BUILDING` *relations remain separate.*

- **Information Loss in Traditional EER Mapping:** (p.287-288)
  Standard relational mapping techniques for EER constructs can lose some semantic information present in the EER model:
  - The explicit type of relationship (specialization, categorization, aggregation) is not directly stored in the relational schema structure.
  - SC/sc (intra-entity class) relationships might become structurally indistinguishable from regular (inter-entity class) relationships (both use FKs).

    - Disjointness constraints (d or o) for specializations are often not directly representable declaratively and might require application logic or triggers.
    - Completeness constraints (total or partial) for specializations (especially on the superclass side) might be lost or only implicitly handled by nullability of FKs.
    - Multiple specializations of the same superclass might not be clearly differentiated in the structure.
    - Specialization lattices (shared subclasses) can be complex to query.
- **Information-Preserving Grammar for EER Constructs:** (p.289-295)
  The textbook extends its information-preserving logical schema grammar to better capture EER semantics.
    - The foreign key notation is augmented with:
        - `#`: An integer above the FK indicating the number of subclasses in the specific specialization/categorization/aggregation instance.
        - `Jx`: A label below the FK indicating the type and instance of the SC/sc relationship.
            - `J` can be: `d` (disjoint specialization), `o` (overlapping specialization), `u` (union for categorization), `a` (aggregation), `L` (specialization lattice).
            - `x` is a marker (e.g., 1, 2, 3) to distinguish different specializations/categorizations involving the same superclass or subclass.
    - This grammar aims to make more EER metadata explicit in the logical schema definition, reducing reliance on separate textual constraint lists.
      (Fig 6.33-6.38, p.291-295 provide examples of applying this extended grammar to map EER hierarchies, lattices, categorizations, and aggregations. For instance, Fig 6.34 maps the hierarchy from Fig 6.33, and Fig 6.36 maps the lattice/categorization from Fig 6.35).

4. Mapping of Higher Degree Relationships [PYQ Q5b]

* (This was also covered under ER Mapping. The process is the same whether the ER model is basic or enhanced with EER constructs, as higher-degree relationships are an ER concept).

* **Recap:**

1. Conceptually, in the Fine-Granular Design-Specific ER model, an n-ary relationship (where n > 2) is typically first decomposed into a **gerund entity type**.
2. This gerund entity type has (n) identifying binary relationships (usually 1:N) with each of the (n) original participating entity types.
3. When mapping to the relational model:

* Create a new relation for this gerund entity type.
* The primary key of this gerund relation is the composite of the primary keys of all (n) original participating entity types. These PKs from original entities are included as foreign keys in the gerund relation.
* Any attributes of the original n-ary relationship become attributes of this gerund relation.

(Example: A ternary `SUPPLIES(SUPPLIER, PART, PROJECT)` with attribute `Quantity` would map to a relation `SUPPLIES(SupplierID (FK), PartID (FK), ProjectID (FK), Quantity)`, where the PK is `{SupplierID, PartID, ProjectID}`).

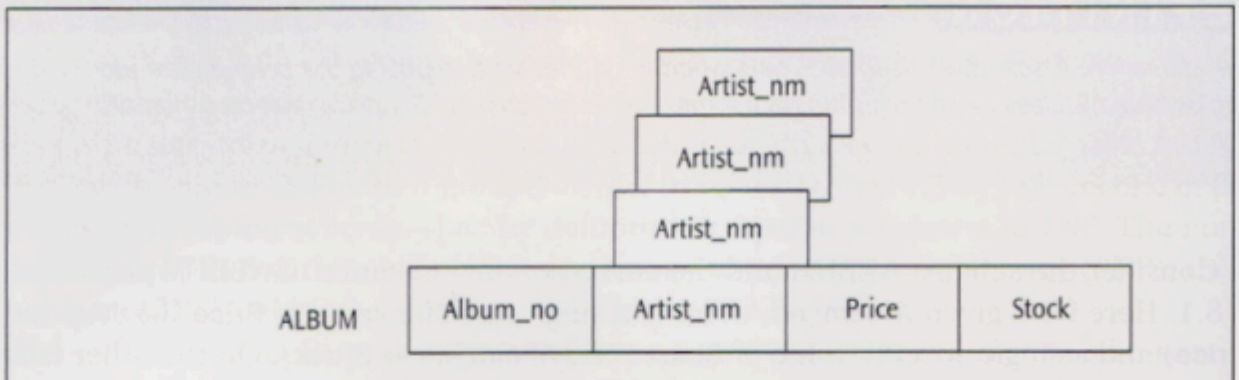
5. Mapping of Aggregation [PYQ Q5b]
* (This was covered under EER Mapping).
* **Recap:** Aggregation defines an "is-part-of" relationship, where an aggregate entity (the "whole") is formed from several component entities (the "parts").
* **Mapping Strategy:**
1. Create a relation for the aggregate entity type (the "whole").
2. The primary key of this aggregate relation can be its own unique identifier if it has one, or a surrogate key.
3. Include the primary keys of all the component ("part") entity types as foreign keys in the aggregate relation. These FKs link the "whole" to its "parts."
4. If an instance of a "part" can only belong to one "whole" instance, then these FKs in the aggregate relation might be part of a candidate key or have uniqueness constraints, depending on the exact semantics of the aggregation.
5. The aggregate relation also includes any specific attributes of the "whole" that are not attributes of its "parts."
(Example from textbook: `PERSONAL_COMPUTER` (aggregate/whole) is an aggregation of `HARDWARE` and `OPERATING_SYSTEM` (parts). The `PERSONAL_COMPUTER` relation would include FKs to `HARDWARE` and `OPERATING_SYSTEM`.)

6. Mapping Complex ER Model Constructs to a Logical Schema
(These are advanced ER constructs from Chapter 5, often used to capture more nuanced business rules directly in the ERD before detailed logical design).
* **Cluster Entities:** (Ch 5.5.3, p.204-205)
* A cluster entity is a conceptual grouping of several entities and their relationships.
* If a cluster entity itself needs to participate in relationships with entities *outside* the cluster, it must be decomposed before mapping.
* **Decomposition Strategy:** The "nucleus" relationship within the cluster (often an M:N or n-ary relationship) is typically transformed into a gerund entity type (or a weak entity type if it has a partial key). This gerund/weak entity then represents the cluster when interacting with external entities.
* This gerund/weak entity is then mapped to a relation as per standard rules.
* **Weak Relationships (Inter-relationship Dependencies):** (Ch 5.5.4, p.206-208)
* **Inclusion Dependency (e.g., `Manages` ⊆ `Works_in`):**
* If the dependency doesn't create an M:N structure between the core entities involved in the two relationships, it can often be handled by careful foreign key placement and ensuring the FK for the "dependent" relationship refers to entities already satisfying the "precedent" relationship. This often translates to application logic or database triggers for strict enforcement.
* Sometimes, as shown in *Fig 5.45-5.47 (p.208)*, if `TEACHING` ⊆ `CAN_TEACH`, and both are M:N gerunds from (`INSTRUCTOR`, `COURSE`), this can be modeled conceptually using EER specialization (`TEACHING` as a subclass of `CAN_TEACH`). Then map this EER structure.
* **Exclusion Dependency (Mutually Exclusive Relationships):**
* This is difficult to enforce declaratively in standard relational models using only PK/FK constraints.
* It usually requires application-level validation logic or database triggers to ensure that an entity (or pair of entities) participates in only one of the specified mutually exclusive relationships.

* The conceptual model (ERD with exclusive arc or dotted line between weak relationships) serves as documentation for this constraint.

7. Normalization [PYQ Q5a]
Normalization is a systematic technique for organizing attributes into relations (tables) to minimize data redundancy and reduce the likelihood of data anomalies (insertion, deletion, update). It's a crucial part of logical database design for relational databases. It is based on the concepts of functional dependencies and primary keys.

Introduction & Anomalies: (Ch 8.1, p.345)
* **Functional Dependency (FD):** A constraint between two sets of attributes X and Y in a relation. X → Y means that the value(s) of X uniquely determine the value(s) of Y. X is the determinant; Y is the dependent.
* **Undesirable FD:** An FD where the determinant is *not* a candidate key. These are the FDs that cause problems.
* **Desirable FD:** An FD where the determinant *is* a candidate key.
* **Normalization Goal:** To decompose relations with undesirable FDs into a set of smaller relations where all (or most) FDs are desirable, while preserving data and dependencies.
* **Anomalies (without proper normalization):**
* **Insertion Anomaly:** Inability to add a new piece of information about one entity until information about another related entity is available (e.g., cannot add a new course until at least one student enrolls, if all course info is in an unnormalized student-course table).
* **Deletion Anomaly:** Unintentional loss of information about one entity when a tuple representing information about another entity is deleted (e.g., deleting the last student enrolled in a course might delete the course information itself if not normalized).
* **Modification/Update Anomaly:** To change a fact, multiple rows might need to be updated. If not all are updated consistently, the database becomes inconsistent (e.g., changing a course title might require updating many student enrollment records).
* **Normal Forms (NF):** A sequence of rules or conditions that relations must satisfy. Higher normal forms generally mean less redundancy and fewer anomalies.
* **1NF (First Normal Form):** (Ch 8.1.1, p.346)
* **Condition:** All attribute values must be atomic (indivisible). No repeating groups or multi-valued attributes within a single cell of a table. Each row-column intersection must contain exactly one value from the applicable domain.
* A relation in the formal relational model is, by definition, in 1NF.
* **Resolution of Violation:** If you have non-1NF data (e.g., an attribute `Artist_nms` that stores a list of artists for an album), you expand the relation so there's a separate tuple for each atomic value (e.g., each artist for that album). The primary key often becomes composite. *(Fig 8.1, p.349)*



ALBUM

Album_no	Artist_nm	Price	Stock
BS123	Britney Spears	17.95	1000
JT111	Justin Timberlake	17.95	1200
BTL007	{John Lennon, Paul McCartney, George Harrison, Ringo Star}	23.95	
MJ100	Michael Jackson	17.95	
JM456	John Mayer	16.95	1000
JM151	John Mayer	16.95	1000
MX789	Madonna	11.95	500
DJM237	{John Denver, Michael Jackson, Madonna}	11.95	2000
DR711	Diana Ross	12.95	1000
PM137	Paul McCartney	19.95	

NEW_ALBUM

Album_no	Artist_nm	Price	Stock
BS123	Britney Spears	17.95	1000
JT111	Justin Timberlake	17.95	1200
BTL007	John Lennon	23.95	
BTL007	Paul McCartney	23.95	
BTL007	George Harrison	23.95	
BTL007	Ringo Star	23.95	
MJ100	Michael Jackson	17.95	
JM456	John Mayer	16.95	1000
JM151	John Mayer	16.95	1000
MX789	Madonna	11.95	500
DJM237	John Denver	11.95	2000
DJM237	Michael Jackson	11.95	2000
DJM237	Madonna	11.95	2000
DR711	Diana Ross	12.95	1000
PM137	Paul McCartney	19.95	

Figure 8.1 An example of 1NF violation and resolution

* *Fig 8.1:* `ALBUM` *with multi-valued* `Artist_nm` *is not 1NF.* `NEW_ALBUM` *is 1NF with PK {*`Album_no`*,* `Artist_nm`*} by creating separate rows for each artist of an album.*

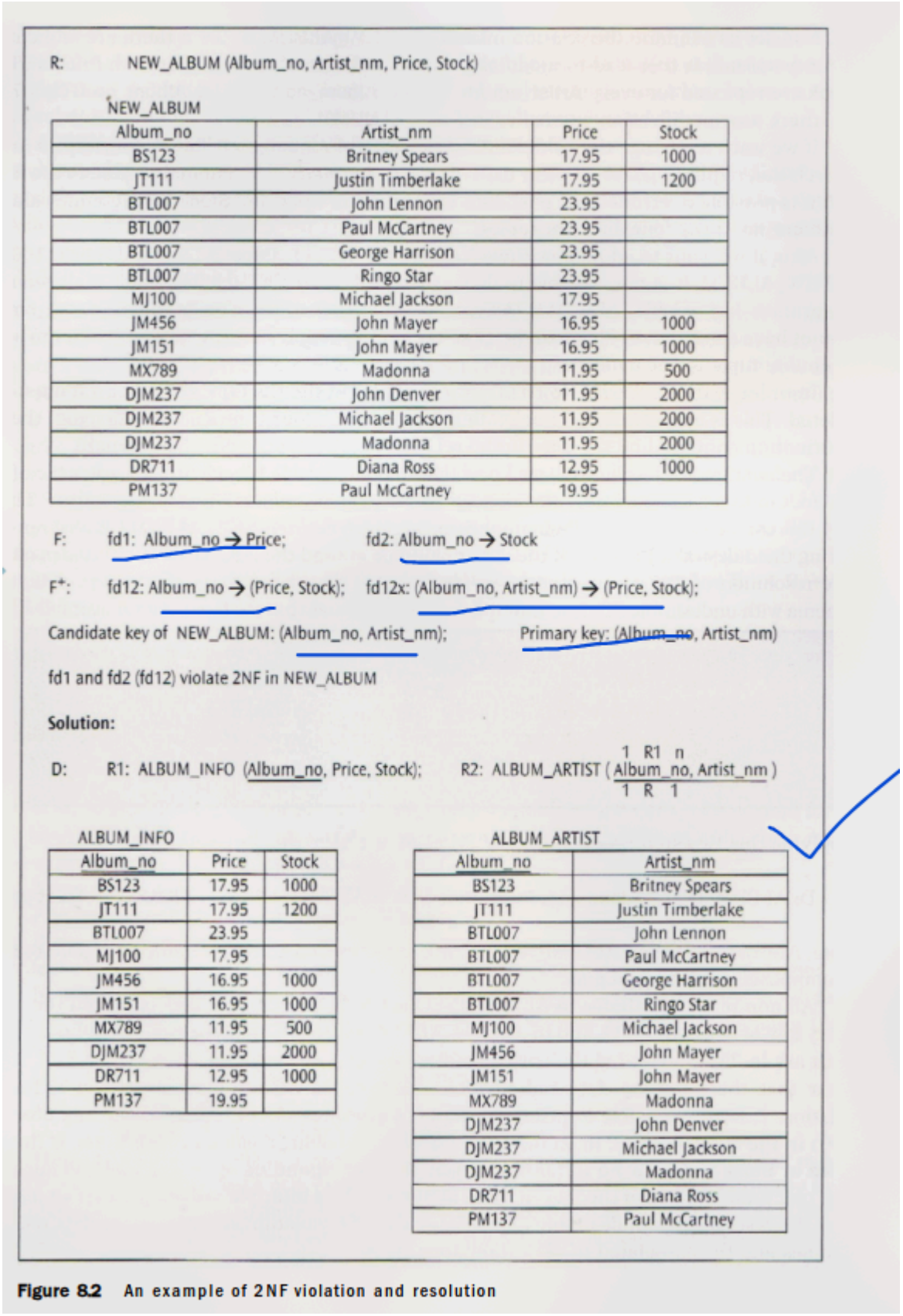
* **2NF (Second Normal Form):** (Ch 8.1.2, p.347-350)

* **Condition:** Must be in 1NF, AND every non-key (non-prime) attribute must be *fully functionally dependent* on the *entire* primary key.

* **Full Functional Dependency:** An attribute Y is fully functionally dependent on a set of attributes X if X → Y, and no proper subset of X also functionally determines Y.

* **Partial Dependency:** A non-key attribute is functionally dependent on only a *part* of a composite primary key. 2NF aims to eliminate partial dependencies.

* If the PK is a single attribute (not composite), the relation is automatically in 2NF if it's in 1NF.
* **Resolution:** Decompose the relation. Create a new relation for the partially dependent attribute(s) with the part of the PK they depend on as the PK of the new relation. Remove the partially dependent attribute(s) from the original relation. The part of the PK remains in the original and also acts as an FK to the new relation. *(Fig 8.2, p.350)*



R: NEW_ALBUM (Album_no, Artist_nm, Price, Stock)

NEW_ALBUM

Album_no	Artist_nm	Price	Stock
BS123	Britney Spears	17.95	1000
JT111	Justin Timberlake	17.95	1200
BTL007	John Lennon	23.95	
BTL007	Paul McCartney	23.95	
BTL007	George Harrison	23.95	
BTL007	Ringo Star	23.95	
MJ100	Michael Jackson	17.95	
JM456	John Mayer	16.95	1000
JM151	John Mayer	16.95	1000
MX789	Madonna	11.95	500
DJM237	John Denver	11.95	2000
DJM237	Michael Jackson	11.95	2000
DJM237	Madonna	11.95	2000
DR711	Diana Ross	12.95	1000
PM137	Paul McCartney	19.95	

F: fd1: Album_no → Price; fd2: Album_no → Stock

F+: fd12: Album_no → (Price, Stock); fd12x: (Album_no, Artist_nm) → (Price, Stock);

Candidate key of NEW_ALBUM: (Album_no, Artist_nm); Primary key: (Album_no, Artist_nm)

fd1 and fd2 (fd12) violate 2NF in NEW_ALBUM

Solution:

D: R1: ALBUM_INFO (Album_no, Price, Stock); R2: ALBUM_ARTIST (Album_no, Artist_nm) — 1 R1 n; 1 R 1

ALBUM_INFO

Album_no	Price	Stock
BS123	17.95	1000
JT111	17.95	1200
BTL007	23.95	
MJ100	17.95	
JM456	16.95	1000
JM151	16.95	1000
MX789	11.95	500
DJM237	11.95	2000
DR711	12.95	1000
PM137	19.95	

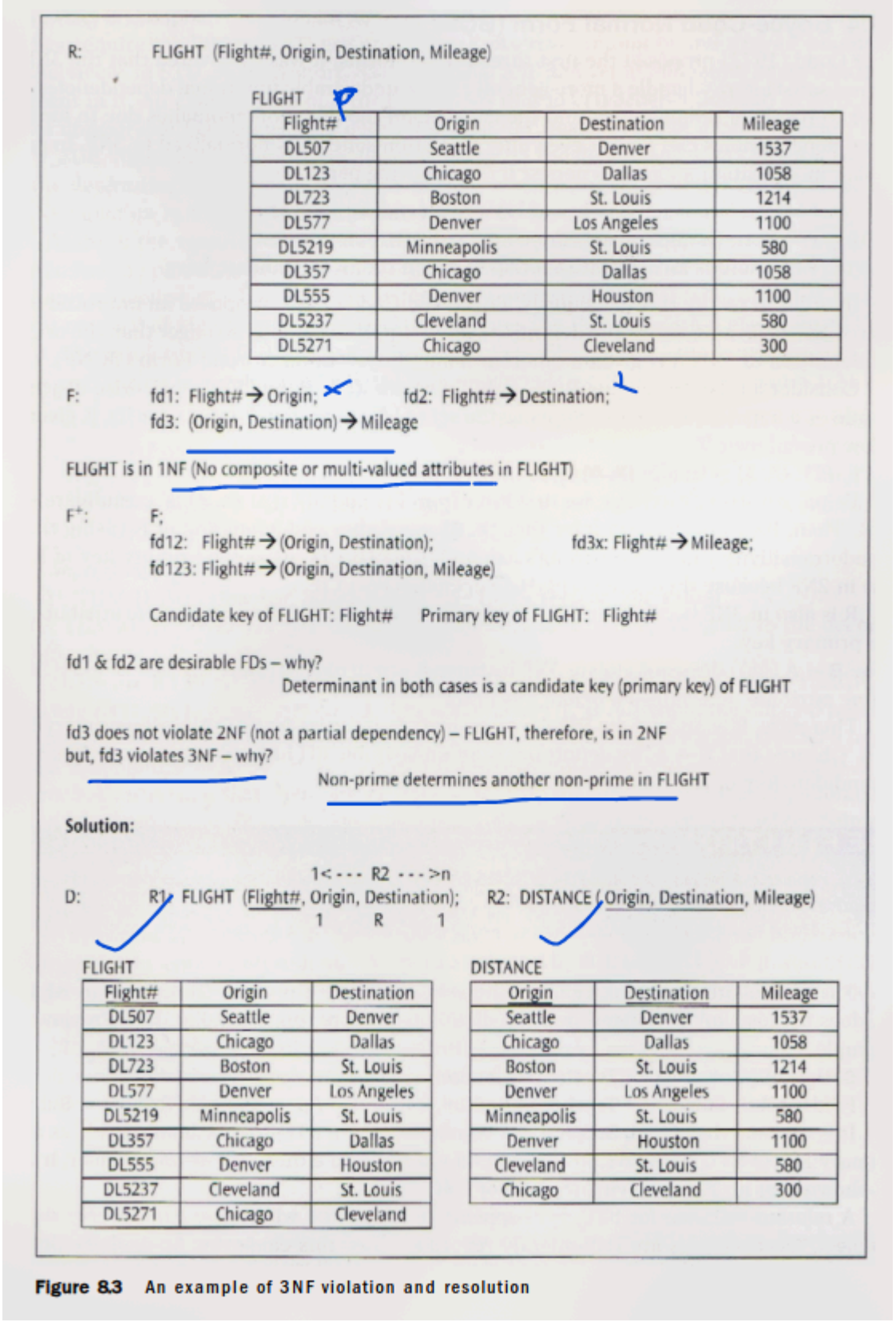
ALBUM_ARTIST

Album_no	Artist_nm
BS123	Britney Spears
JT111	Justin Timberlake
BTL007	John Lennon
BTL007	Paul McCartney
BTL007	George Harrison
BTL007	Ringo Star
MJ100	Michael Jackson
JM456	John Mayer
JM151	John Mayer
MX789	Madonna
DJM237	John Denver
DJM237	Michael Jackson
DJM237	Madonna
DR711	Diana Ross
PM137	Paul McCartney

Figure 8.2 An example of 2NF violation and resolution

* *Fig 8.2:* `NEW_ALBUM(Album_no, Artist_nm, Price, Stock)`. *PK={Album_no, Artist_nm}. FDs:* `Album_no` → `Price`, `Album_no` → `Stock`. `Price` *and* `Stock` *are partially dependent on PK. Decomposed into:* `ALBUM_INFO(Album_no (PK), Price, Stock)` *and* `ALBUM_ARTIST(Album_no`

`(PK,FK), Artist_nm (PK))`.
* **3NF (Third Normal Form):** (Ch 8.1.3, p.351-353)
* **Condition:** Must be in 2NF, AND no non-key (non-prime) attribute should be *transitively dependent* on the primary key.
* **Transitive Dependency:** If PK → A and A → B, where A is a non-key attribute and B is a non-key attribute (and A is not functionally dependent on B, and B is not part of PK), then B is transitively dependent on PK via A.
* Essentially, 3NF aims to ensure that non-key attributes depend *only* on the PK, the whole PK, and nothing but the PK (excluding other non-key attributes).
* **Resolution:** Decompose the relation. Create a new relation for the transitively dependent attribute(s) with their determinant (the intermediate non-key attribute) as the PK of this new relation. Remove the transitively dependent attribute(s) from the original relation, but keep their determinant (which now acts as an FK to the new relation). *(Fig 8.3, p.353)*



R: FLIGHT (Flight#, Origin, Destination, Mileage)

FLIGHT

Flight#	Origin	Destination	Mileage
DL507	Seattle	Denver	1537
DL123	Chicago	Dallas	1058
DL723	Boston	St. Louis	1214
DL577	Denver	Los Angeles	1100
DL5219	Minneapolis	St. Louis	580
DL357	Chicago	Dallas	1058
DL555	Denver	Houston	1100
DL5237	Cleveland	St. Louis	580
DL5271	Chicago	Cleveland	300

F: fd1: Flight# → Origin; fd2: Flight# → Destination;
fd3: (Origin, Destination) → Mileage

FLIGHT is in 1NF (No composite or multi-valued attributes in FLIGHT)

F^+: F;
fd12: Flight# → (Origin, Destination); fd3x: Flight# → Mileage;
fd123: Flight# → (Origin, Destination, Mileage)

Candidate key of FLIGHT: Flight# Primary key of FLIGHT: Flight#

fd1 & fd2 are desirable FDs – why?
Determinant in both cases is a candidate key (primary key) of FLIGHT

fd3 does not violate 2NF (not a partial dependency) – FLIGHT, therefore, is in 2NF
but, fd3 violates 3NF – why?
Non-prime determines another non-prime in FLIGHT

Solution:

D: R1: FLIGHT (Flight#, Origin, Destination); R2: DISTANCE (Origin, Destination, Mileage)
1 <- - - R2 - - - >n
1 R 1

FLIGHT

Flight#	Origin	Destination
DL507	Seattle	Denver
DL123	Chicago	Dallas
DL723	Boston	St. Louis
DL577	Denver	Los Angeles
DL5219	Minneapolis	St. Louis
DL357	Chicago	Dallas
DL555	Denver	Houston
DL5237	Cleveland	St. Louis
DL5271	Chicago	Cleveland

DISTANCE

Origin	Destination	Mileage
Seattle	Denver	1537
Chicago	Dallas	1058
Boston	St. Louis	1214
Denver	Los Angeles	1100
Minneapolis	St. Louis	580
Denver	Houston	1100
Cleveland	St. Louis	580
Chicago	Cleveland	300

Figure 8.3 An example of 3NF violation and resolution

* *Fig 8.3:* `FLIGHT(Flight# (PK), Origin, Destination, Mileage)`. *FD: {*`Origin`, `Destination`*} →* `Mileage`. `Mileage` *is transitively dependent on* `Flight#` *via {*`Origin`, `Destination`*}. Decomposed into:* `DISTANCE(Origin (PK), Destination (PK), Mileage)` *and* `FLIGHT(Flight# (PK), Origin (FK), Destination (FK))`.

* **BCNF (Boyce-Codd Normal Form):** (Ch 8.1.4, p.354-356)

* A stricter version of 3NF. Most relations in 3NF are also in BCNF.

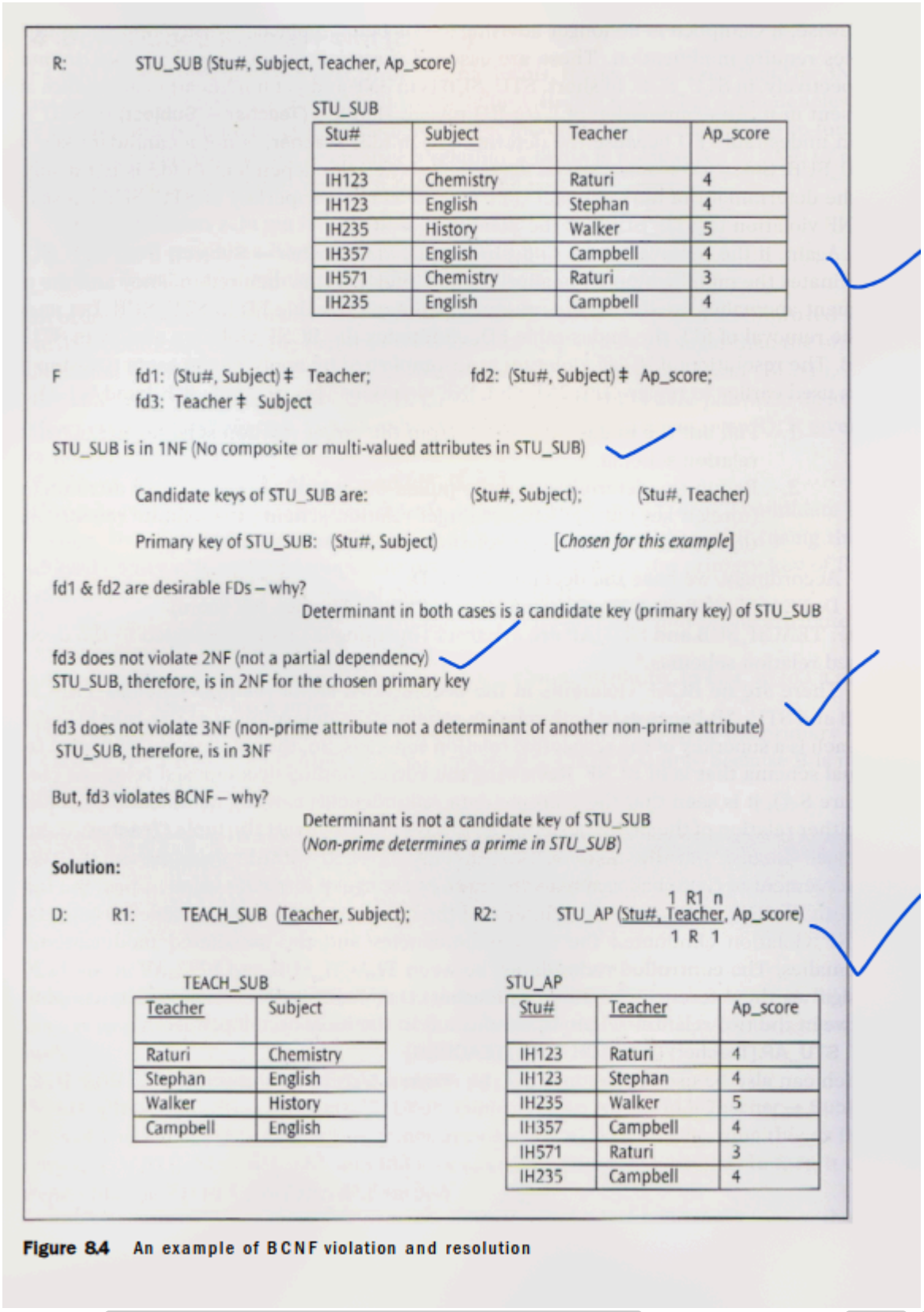
* **Condition:** For every non-trivial functional dependency X → Y in the relation, X must be a **superkey** (or candidate key) of the relation.

* BCNF handles certain anomalies that 3NF might not, especially in relations with multiple overlapping

composite candidate keys.

* **Resolution:** If X → Y violates BCNF (X is not a superkey), decompose R into R1(X, Y) and R2(R - Y). X becomes the PK of R1.

(Fig 8.4, p.357)



R: STU_SUB (Stu#, Subject, Teacher, Ap_score)

STU_SUB

Stu#	Subject	Teacher	Ap_score
IH123	Chemistry	Raturi	4
IH123	English	Stephan	4
IH235	History	Walker	5
IH357	English	Campbell	4
IH571	Chemistry	Raturi	3
IH235	English	Campbell	4

F fd1: (Stu#, Subject) ‡ Teacher; fd2: (Stu#, Subject) ‡ Ap_score;
fd3: Teacher ‡ Subject

STU_SUB is in 1NF (No composite or multi-valued attributes in STU_SUB)

Candidate keys of STU_SUB are: (Stu#, Subject); (Stu#, Teacher)

Primary key of STU_SUB: (Stu#, Subject) [*Chosen for this example*]

fd1 & fd2 are desirable FDs – why?
Determinant in both cases is a candidate key (primary key) of STU_SUB

fd3 does not violate 2NF (not a partial dependency)
STU_SUB, therefore, is in 2NF for the chosen primary key

fd3 does not violate 3NF (non-prime attribute not a determinant of another non-prime attribute)
STU_SUB, therefore, is in 3NF

But, fd3 violates BCNF – why?
Determinant is not a candidate key of STU_SUB
(*Non-prime determines a prime in STU_SUB*)

Solution:

D: R1: TEACH_SUB (Teacher, Subject); R2: STU_AP (Stu#, Teacher, Ap_score)
1 R1 n
1 R 1

TEACH_SUB

Teacher	Subject
Raturi	Chemistry
Stephan	English
Walker	History
Campbell	English

STU_AP

Stu#	Teacher	Ap_score
IH123	Raturi	4
IH123	Stephan	4
IH235	Walker	5
IH357	Campbell	4
IH571	Raturi	3
IH235	Campbell	4

Figure 8.4 An example of BCNF violation and resolution

* *Fig 8.4:* `STU_SUB(Stu#, Subject, Teacher, Ap_score)`. *Candidate Keys: {*`Stu#`, `Subject`*} and {*`Stu#`, `Teacher`*}. If PK={*`Stu#`, `Subject`*}, then FD:* `Teacher` → `Subject` *violates BCNF because* `Teacher` *is not a superkey. Decomposed into:* `TEACH_SUB(Teacher (PK), Subject)` *and* `STU_AP(Stu# (PK), Teacher (PK,FK), Ap_score)`.

* **4NF (Fourth Normal Form):** (Deals with Multi-Valued Dependencies - MVDs) (Ch 9.2, p.422-428)

* **Condition:** Must be in BCNF, AND must not have any non-trivial multi-valued dependencies (MVDs) X –» Y, unless X is a superkey.
* **Multi-Valued Dependency (MVD):** X –» Y states that for a given value of X, the set of Y values associated with it is independent of the set of Z values associated with it (where Z = R - (X ∪ Y)). This often arises when trying to represent two or more independent M:N relationships in a single table.
* 4NF aims to eliminate redundancy due to independent multi-valued facts about the same entity.
* **Resolution:** If X –» Y violates 4NF, decompose R into R1(X, Y) and R2(X, R - (X ∪ Y)). *(Fig 9.1, p.424 for MVD test; example p.426 MUSIC_VEHICLE(Name, Music, Vehicle) has MVD Name –» Music | Vehicle. Decomposed into N_MUSIC(Name, Music) and N_VEHICLE(Name, Vehicle). Fig 9.2, p.427 for MUSIC_SKILL example).*

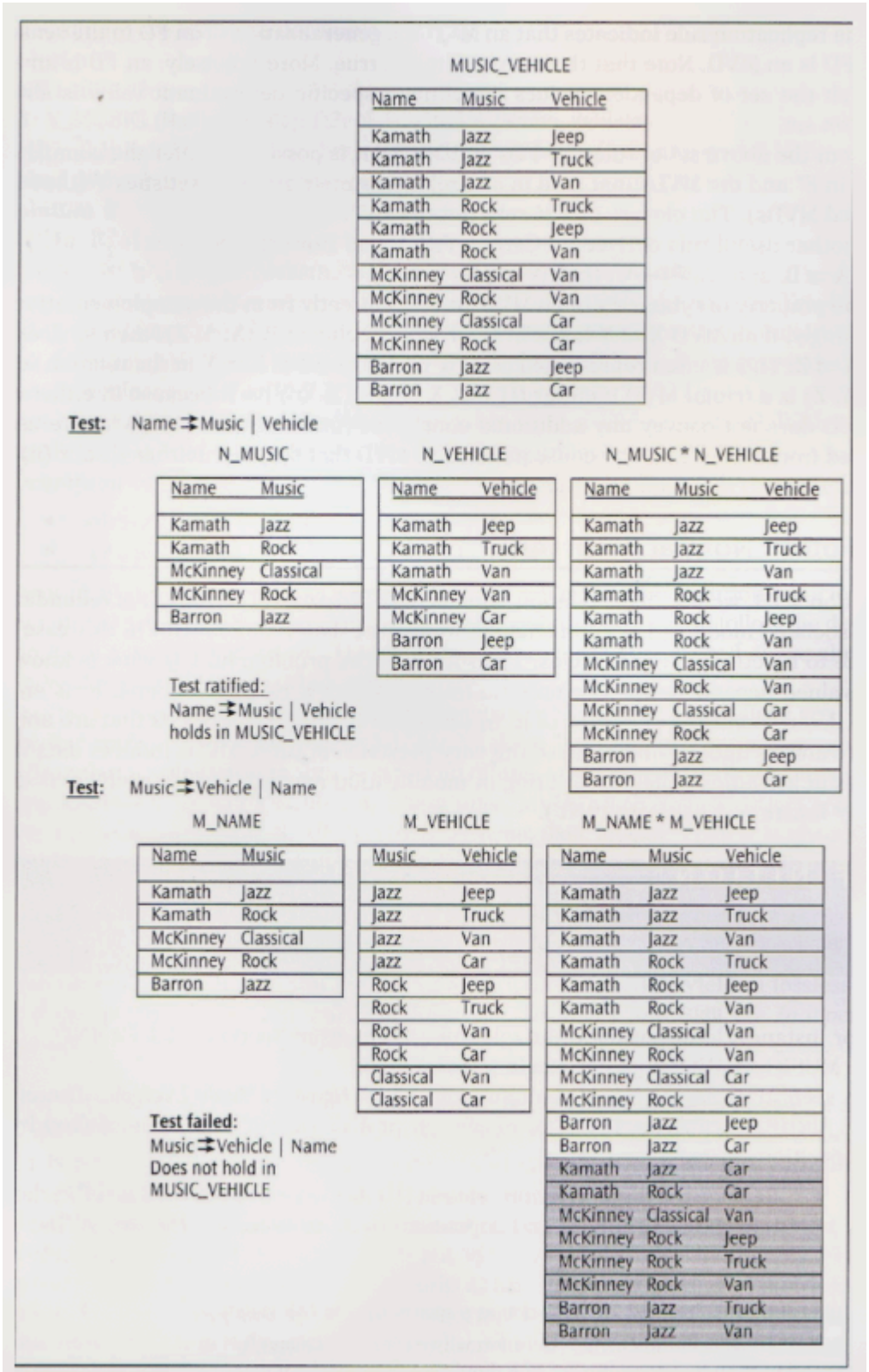
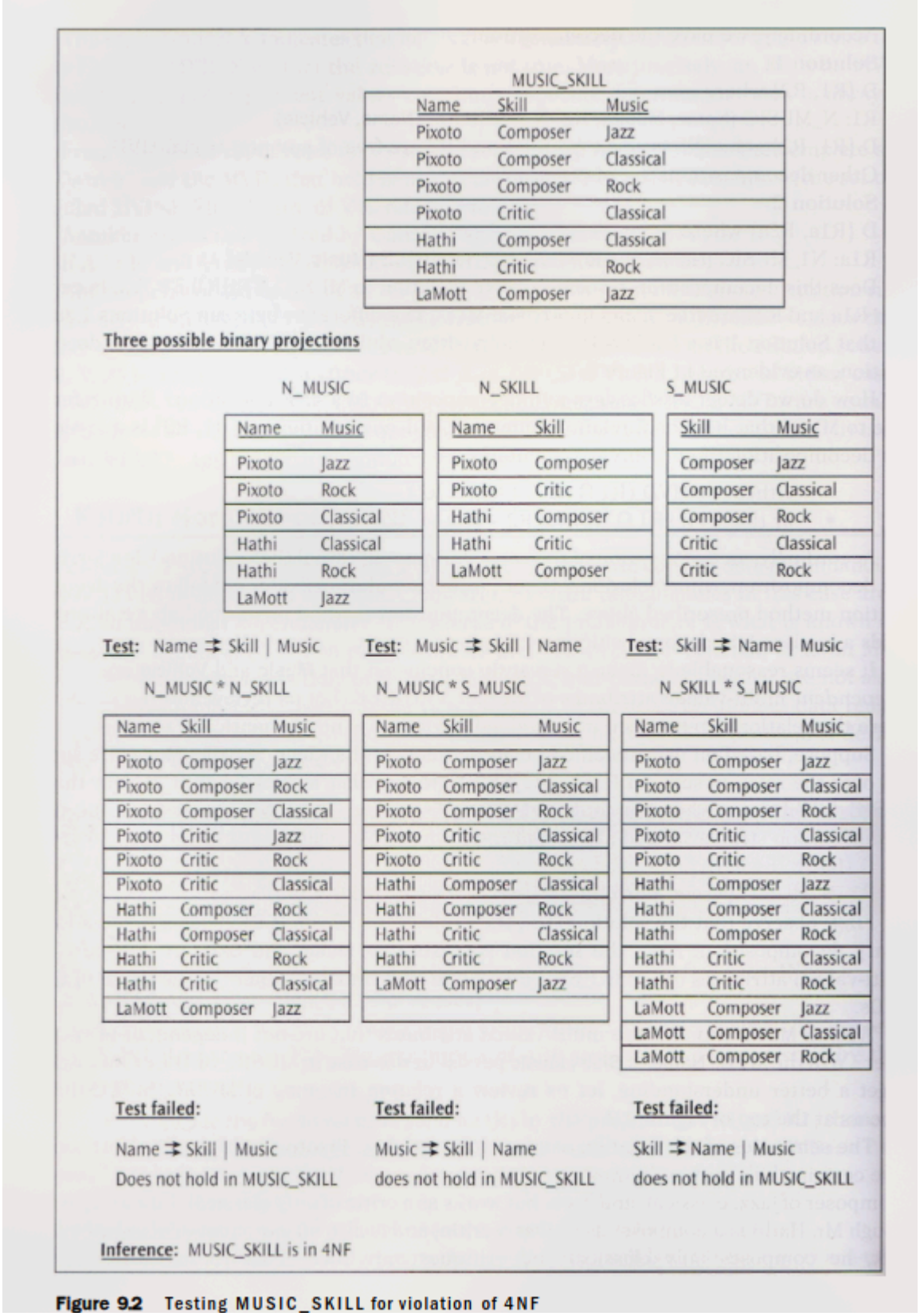


Figure 9.1 Test to check for the presence of multi-valued dependencies



MUSIC_SKILL

Name	Skill	Music
Pixoto	Composer	Jazz
Pixoto	Composer	Classical
Pixoto	Composer	Rock
Pixoto	Critic	Classical
Hathi	Composer	Classical
Hathi	Critic	Rock
LaMott	Composer	Jazz

Three possible binary projections

N_MUSIC

Name	Music
Pixoto	Jazz
Pixoto	Rock
Pixoto	Classical
Hathi	Classical
Hathi	Rock
LaMott	Jazz

N_SKILL

Name	Skill
Pixoto	Composer
Pixoto	Critic
Hathi	Composer
Hathi	Critic
LaMott	Composer

S_MUSIC

Skill	Music
Composer	Jazz
Composer	Classical
Composer	Rock
Critic	Classical
Critic	Rock

Test: Name ⇉ Skill | Music

N_MUSIC * N_SKILL

Name	Skill	Music
Pixoto	Composer	Jazz
Pixoto	Composer	Rock
Pixoto	Composer	Classical
Pixoto	Critic	Jazz
Pixoto	Critic	Rock
Pixoto	Critic	Classical
Hathi	Composer	Rock
Hathi	Composer	Classical
Hathi	Critic	Rock
Hathi	Critic	Classical
LaMott	Composer	Jazz

Test failed:

Name ⇉ Skill | Music

Does not hold in MUSIC_SKILL

Test: Music ⇉ Skill | Name

N_MUSIC * S_MUSIC

Name	Skill	Music
Pixoto	Composer	Jazz
Pixoto	Composer	Classical
Pixoto	Composer	Rock
Pixoto	Critic	Classical
Pixoto	Critic	Rock
Hathi	Composer	Classical
Hathi	Composer	Rock
Hathi	Critic	Classical
Hathi	Critic	Rock
LaMott	Composer	Jazz

Test failed:

Music ⇉ Skill | Name

does not hold in MUSIC_SKILL

Test: Skill ⇉ Name | Music

N_SKILL * S_MUSIC

Name	Skill	Music
Pixoto	Composer	Jazz
Pixoto	Composer	Classical
Pixoto	Composer	Rock
Pixoto	Critic	Classical
Pixoto	Critic	Rock
Hathi	Composer	Jazz
Hathi	Composer	Classical
Hathi	Composer	Rock
Hathi	Critic	Classical
Hathi	Critic	Rock
LaMott	Composer	Jazz
LaMott	Composer	Classical
LaMott	Composer	Rock

Test failed:

Skill ⇉ Name | Music

does not hold in MUSIC_SKILL

Inference: MUSIC_SKILL is in 4NF

Figure 9.2 Testing MUSIC_SKILL for violation of 4NF

* **5NF (Fifth Normal Form / Project-Join Normal Form - PJ/NF):** (Deals with Join Dependencies - JDs) (Ch 9.5, p.429-433)

* **Condition:** Must be in 4NF, AND every join dependency (JD) in the relation must be implied by the candidate keys.

* **Join Dependency (JD):** A relation R satisfies a JD * (R1, R2, ..., Rn) if R is equal to the natural join of its projections R1, R2, ..., Rn. A JD that is not implied by candidate keys can cause redundancy that cannot be removed by 4NF.

* 5NF is rarely violated if the design is already in BCNF/4NF. It deals with very specific, complex cases of redundancy often involving cyclic dependencies.

* **Resolution:** Decompose R into its projections R1, R2, ..., Rn.

(Fig 9.5, p.430 `SCHEDULE_X` violates 5NF. Fig 9.6, p.432 `SCHEDULE_Y` is in 5NF).

Business rule: If an instructor teaches a course, and that course is offered in certain quarters, then the instructor must teach that course in those quarters if s/he is teaching during those quarters.

R: SCHEDULE_X (Prof_name, Course#, Quarter); R1x: TAUGHT_X (Prof_name, Course#);

R2x: TAUGHT_DURING_X (Prof_name, Quarter); R3x: COURSE_OFFERING_X (Course#, Quarter)

SCHEDULE_X

Prof_name	Course#	Quarter
Verstrate	IS812	Fall
Verstrate	IS812	Spring
Verstrate	IS832	Winter
Verstrate	IS330	Fall
Verstrate	IS330	Spring
Surendra	IS812	Fall
Surendra	IS812	Spring
Surendra	IS430	Fall
Surendra	IS430	Spring
Kim	IS821	Winter
Kim	IS430	Spring
Kim	IS430	Summer

1. Does the natural join of any two of the three tables below strictly yield the table above? The answer is "No." Therefore, SCHEDULE_X is in 4NF.

TAUGHT_X

Prof_name	Course#
Verstrate	IS812
Verstrate	IS832
Verstrate	IS330
Surendra	IS812
Surendra	IS430
Kim	IS821
Kim	IS430

TAUGHT_DURING_X

Prof_name	Quarter
Verstrate	Fall
Verstrate	Winter
Verstrate	Spring
Surendra	Fall
Surendra	Spring
Kim	Winter
Kim	Spring
Kim	Summer

COURSE_OFFERING_X

Course#	Quarter
IS330	Fall
IS330	Spring
IS430	Fall
IS430	Spring
IS430	Summer
IS812	Fall
IS812	Spring
IS821	Winter
IS832	Winter

2. Does the natural join of all the three tables above strictly yield SCHEDULE_X above? The answer is "Yes," indicating presence of join dependencies. Therefore, SCHEDULE_X violates 5NF. In order to achieve 5NF in the design, SCHEDULE_X must be replaced by the set of relation schemas {TAUGHT_X, TAUGHT_DURING_X, COURSE_OFFERING_X}.

Figure 9.5 An illustration of 5NF violation and resolution