Chapter 1: Introduction

Database System Concepts, 6th Ed.

Database Management System (DBMS)

- DBMS contains information about a particular enterprise
  - Collection of interrelated data
  - Set of programs to access the data
  - An environment that is both *convenient* and *efficient* to use
- Database Applications:
  - Banking: transactions
  - Airlines: reservations, schedules
  - Universities: registration, grades
  - Sales: customers, products, purchases
  - Online retailers: order tracking, customized recommendations
  - Manufacturing: production, inventory, orders, supply chain
  - Human resources: employee records, salaries, tax deductions
- Databases can be very large.
- Databases touch all aspects of our lives

University Database Example

- Application program examples
  - Add new students, instructors, and courses
  - Register students for courses, and generate class rosters
  - Assign grades to students, compute grade point averages (GPA) and generate transcripts
- In the early days, database applications were built directly on top of file systems

Drawbacks of using file systems to store data

- Data redundancy and inconsistency
  - Multiple file formats, duplication of information in different files
- Difficulty in accessing data
  - Need to write a new program to carry out each new task
- Data isolation — multiple files and formats
- Integrity problems
  - Integrity constraints (e.g., account balance > 0) become "buried" in program code rather than being stated explicitly
  - Hard to add new constraints or change existing ones

Drawbacks of using file systems to store data (Cont.)

- Atomicity of updates
  - Failures may leave database in an inconsistent state with partial updates carried out
  - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users
  - Concurrent access needed for performance
  - Uncontrolled concurrent accesses can lead to inconsistencies
    - Example: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- Security problems
  - Hard to provide user access to some, but not all, data

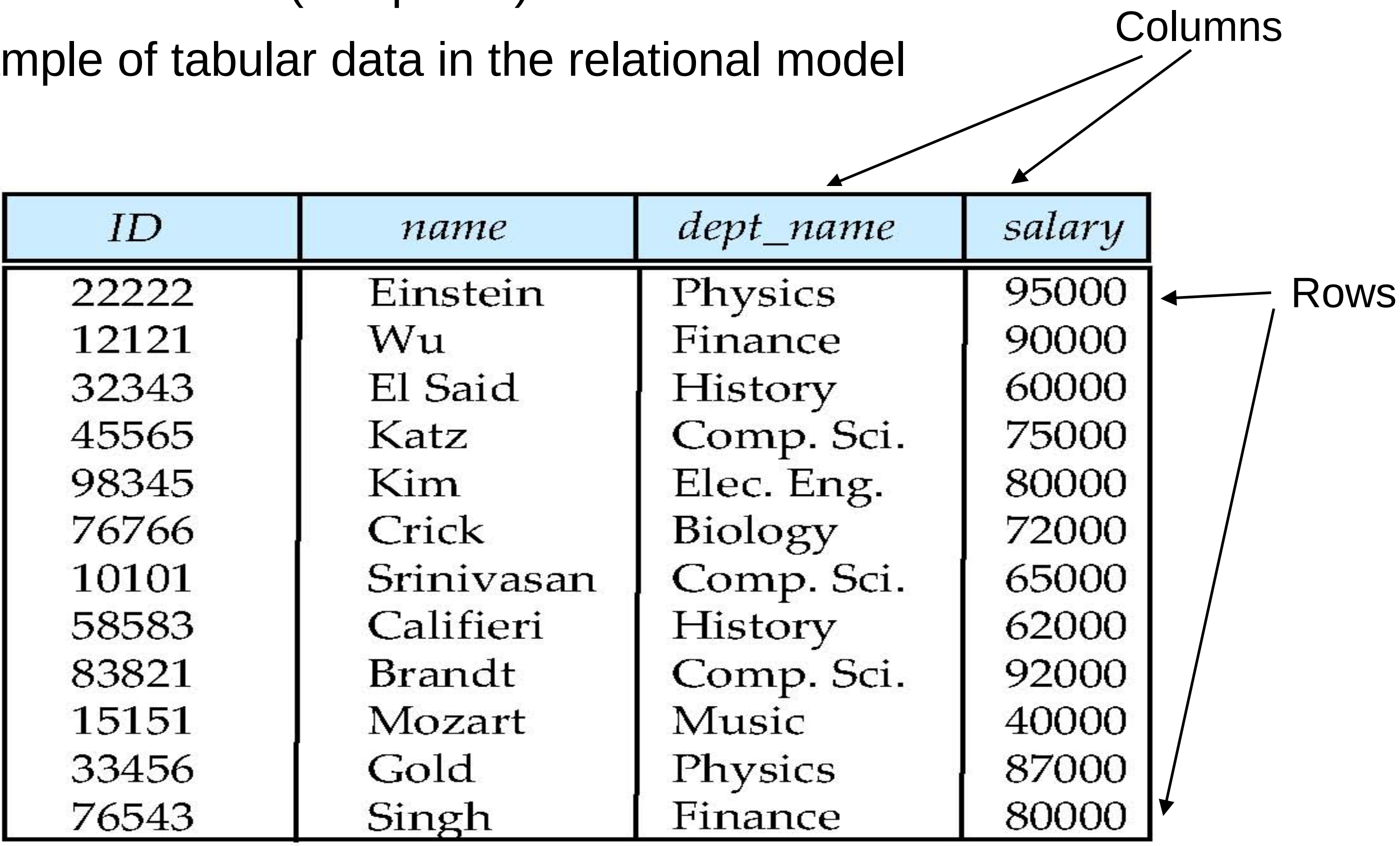
Database systems offer solutions to all the above problems

Data Models

- A collection of tools for describing
  - Data
  - Data relationships
  - Data semantics
  - Data constraints
- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semistructured data model (XML)
- Other older models:
  - Network model
  - Hierarchical model

Relational Model

- Relational model (Chapter 2)
- Example of tabular data in the relational model

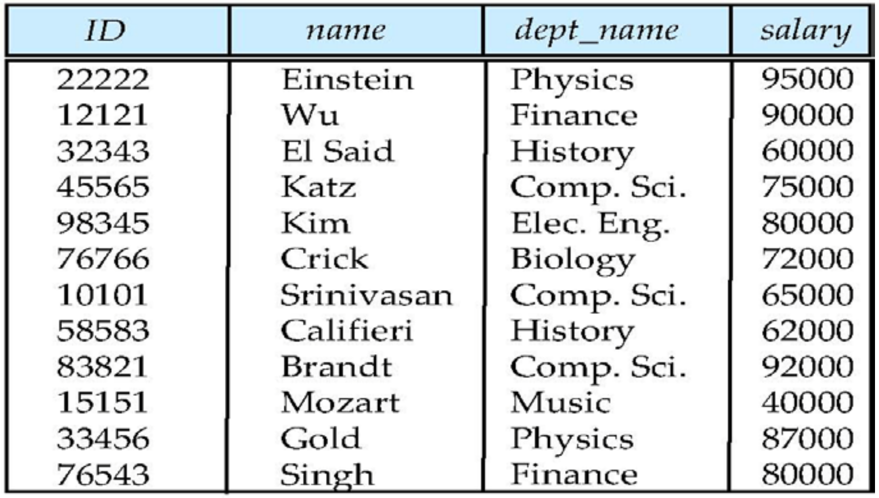


ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

Columns
Rows

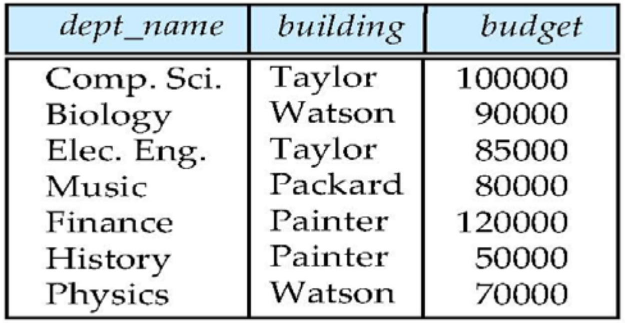
(a) The *instructor* table

A Sample Relational Database



ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table



dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table

Data Definition Language (DDL)

- Specification notation for defining the database schema

  Example:
  ```
  create table instructor (
          ID          char(5),
          name        varchar(20),
          dept_name   varchar(20),
          salary      numeric(8,2))
  ```

- DDL compiler generates a set of table templates stored in a ***data dictionary***
- Data dictionary contains metadata (i.e., data about data)
  - Database schema
  - Integrity constraints
    - Primary key (ID uniquely identifies instructors)
    - Referential integrity (**references** constraint in SQL)
      - e.g. *dept_name* value in any *instructor* tuple must appear in *department* relation
  - Authorization

SQL

- **SQL**: widely used non-procedural language
  - Example: Find the name of the instructor with ID 22222

```
  select    name
  from      instructor
  where     instructor.ID = ‘22222’
```

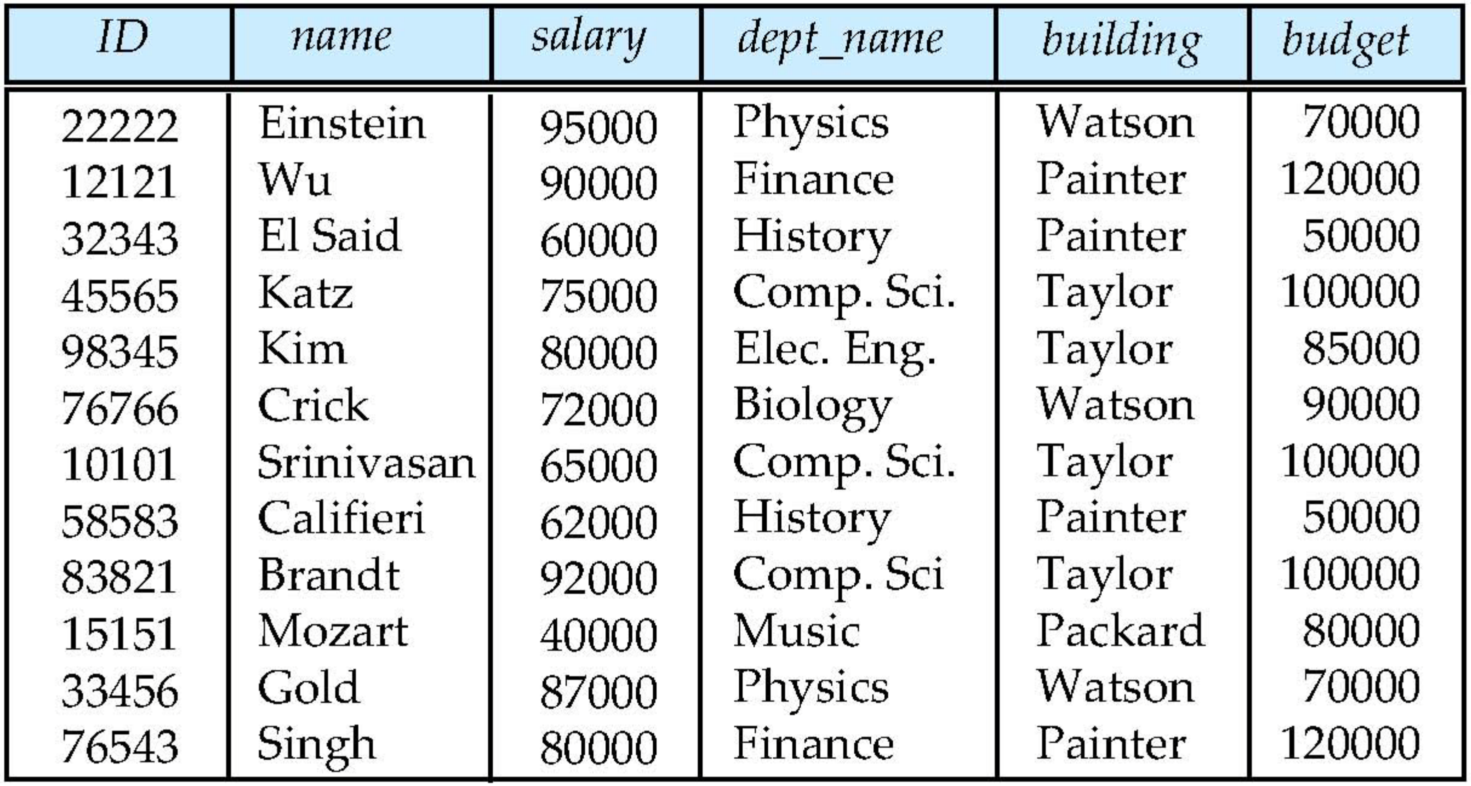
  - Example: Find the ID and building of instructors in the Physics dept.

```
select instructor.ID, department.building
from instructor, department
where instructor.dept_name = department.dept_name and
          department.dept_name = ‘Physics’
```

- Application programs generally access databases through one of
  - Language extensions to allow embedded SQL
  - Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database
- Chapters 3, 4 and 5

Database Design?

- Is there any problem with this design?



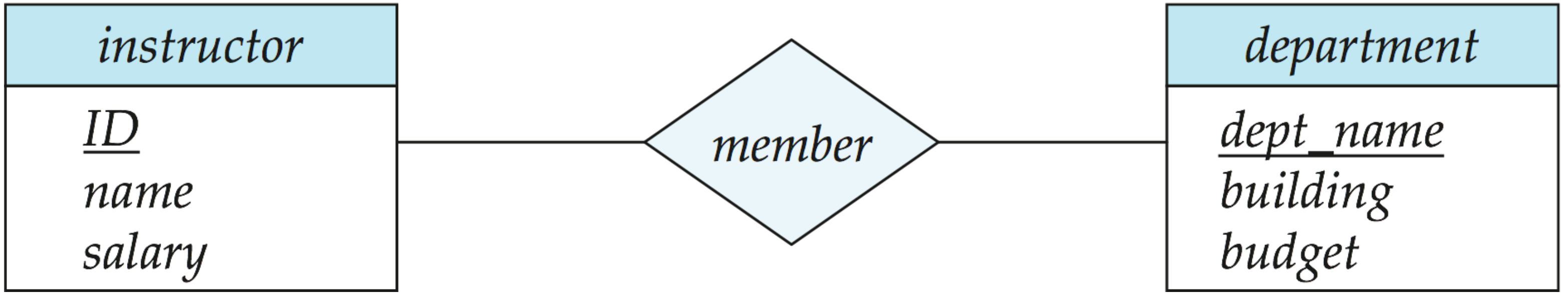
ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	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

Design Approaches

- Normalization Theory (Chapter 8)
  - Formalize what designs are bad, and test for them
- Entity Relationship Model (Chapter 7)
  - Models an enterprise as a collection of *entities* and *relationships*
    - Entity: a “thing” or “object” in the enterprise that is distinguishable from other objects
      - Described by a set of *attributes*
    - Relationship: an association among several entities
  - Represented diagrammatically by an *entity-relationship diagram:*

The Entity-Relationship Model

- Models an enterprise as a collection of *entities* and *relationships*
  - Entity: a “thing” or “object” in the enterprise that is distinguishable from other objects
    - Described by a set of *attributes*
  - Relationship: an association among several entities
- Represented diagrammatically by an *entity-relationship diagram:*



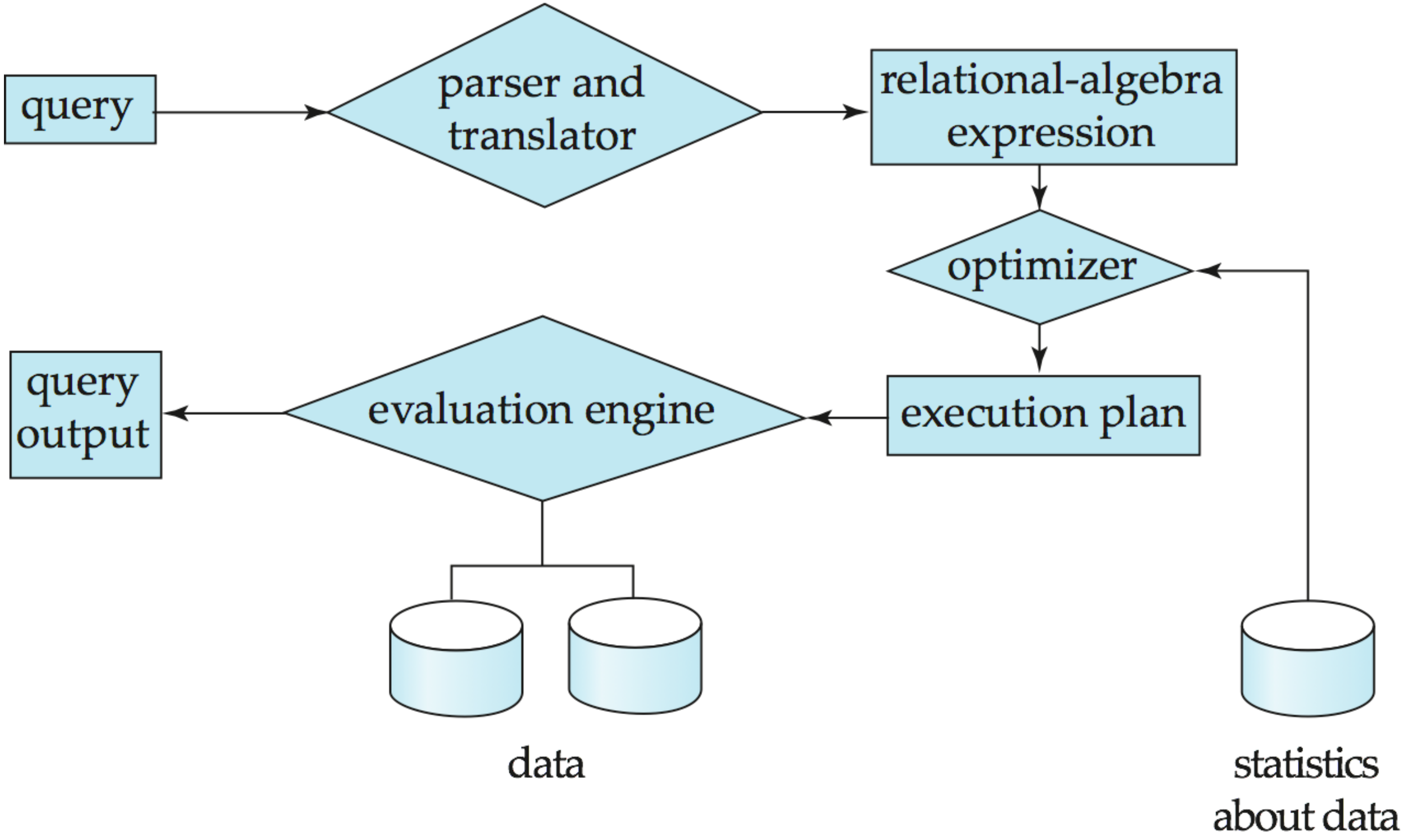
What happened to dept_name of instructor and student?

Storage Management

- **Storage manager** is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible to the following tasks:
  - Interaction with the file manager
  - Efficient storing, retrieving and updating of data
- Issues:
  - Storage access
  - File organization
  - Indexing and hashing

Query Processing

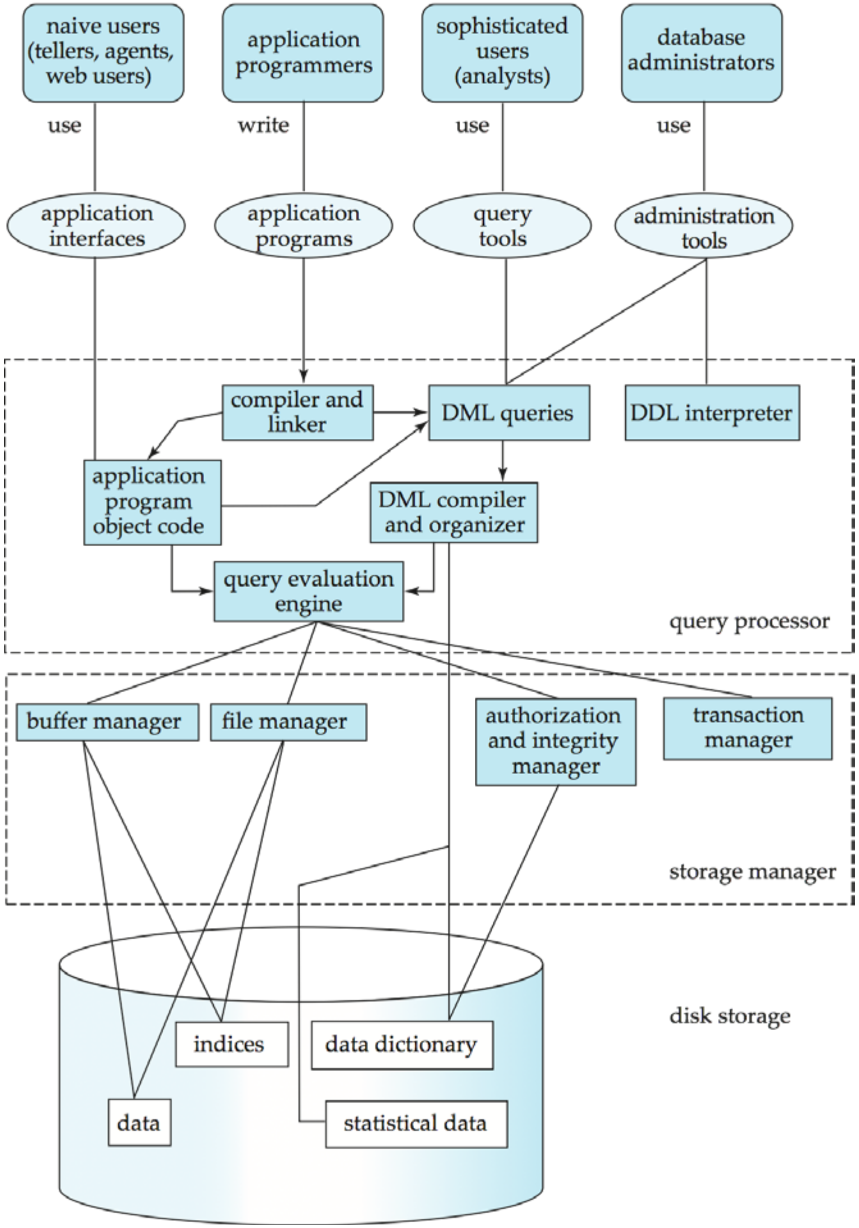
1. Parsing and translation
2. Optimization
3. Evaluation



Transaction Management

- What if the system fails?
- What if more than one user is concurrently updating the same data?
- A **transaction** is a collection of operations that performs a single logical function in a database application
- **Transaction-management component** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
- **Concurrency-control manager** controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Database System Internals



Database Architecture

The architecture of a database systems is greatly influenced by the underlying computer system on which the database is running:

- Centralized
- Client-server
- Parallel (multi-processor)
- Distributed

History of Database Systems

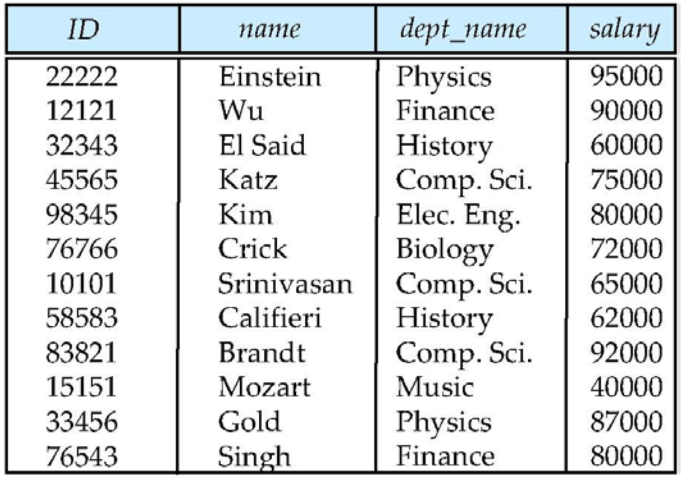
- 1950s and early 1960s:
  - Data processing using magnetic tapes for storage
    - Tapes provided only sequential access
  - Punched cards for input
- Late 1960s and 1970s:
  - Hard disks allowed direct access to data
  - Network and hierarchical data models in widespread use
  - Ted Codd defines the relational data model
    - Would win the ACM Turing Award for this work
    - IBM Research begins System R prototype
    - UC Berkeley begins Ingres prototype
  - High-performance (for the era) transaction processing

History (cont.)

- 1980s:
  - Research relational prototypes evolve into commercial systems
    - SQL becomes industrial standard
  - Parallel and distributed database systems
  - Object-oriented database systems
- 1990s:
  - Large decision support and data-mining applications
  - Large multi-terabyte data warehouses
  - Emergence of Web commerce
- Early 2000s:
  - XML and XQuery standards
  - Automated database administration
- Later 2000s:
  - Giant data storage systems
    - Google BigTable, Yahoo PNuts, Amazon, ..

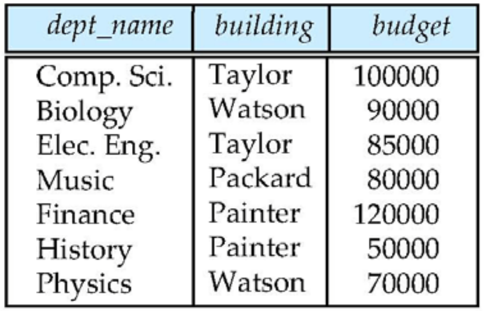
End of Chapter 1

Figure 1.02



ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

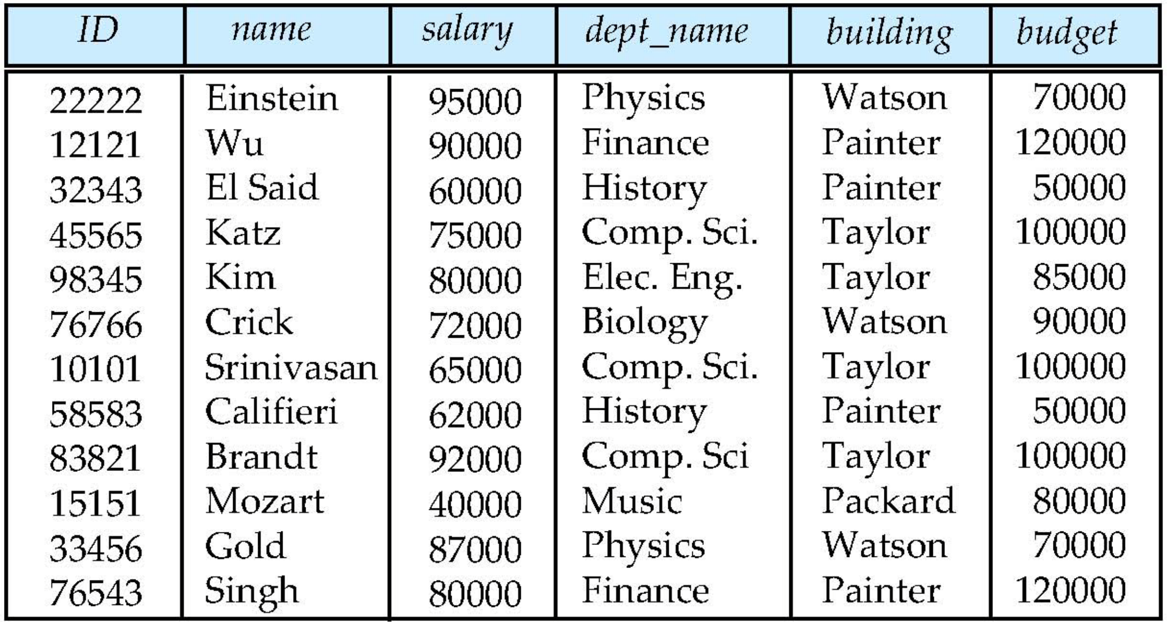
(a) The *instructor* table



dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

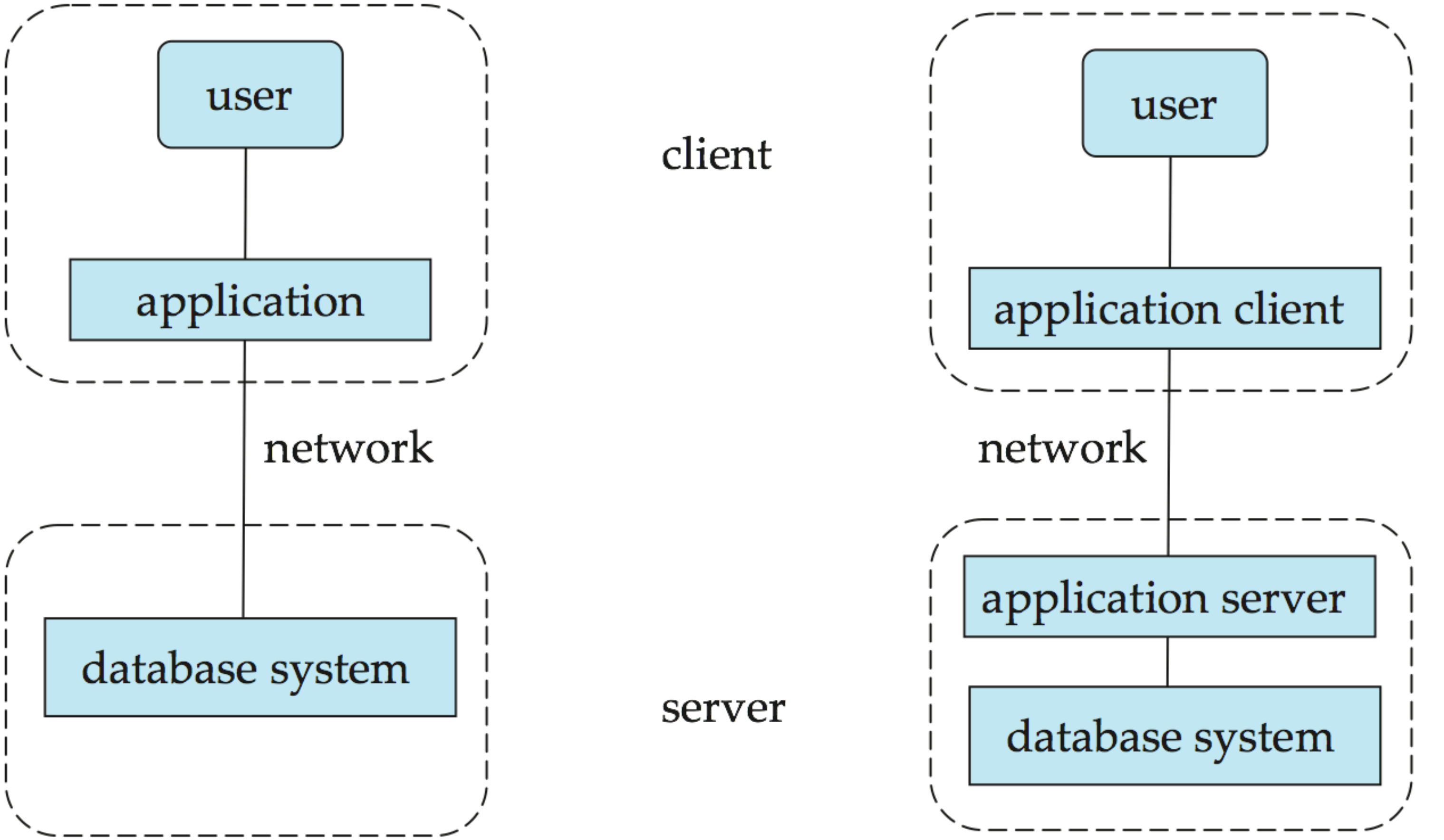
(b) The *department* table

Figure 1.04



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

Figure 1.06



(a) Two-tier architecture

(b) Three-tier architecture

This image cannot currently be displayed.

Chapter 7: Entity-Relationship Model

Database System Concepts, 6th Ed.

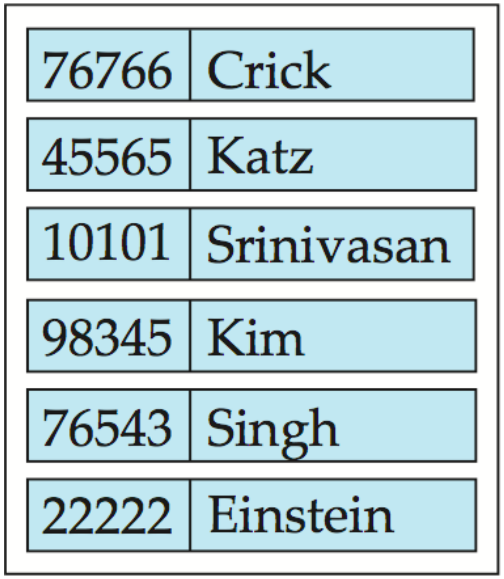
Chapter 7: Entity-Relationship Model

- Design Process
- Modeling
- Constraints
- E-R Diagram
- Design Issues
- Weak Entity Sets
- Extended E-R Features
- Design of the Bank Database
- Reduction to Relation Schemas
- Database Design
- UML

Modeling

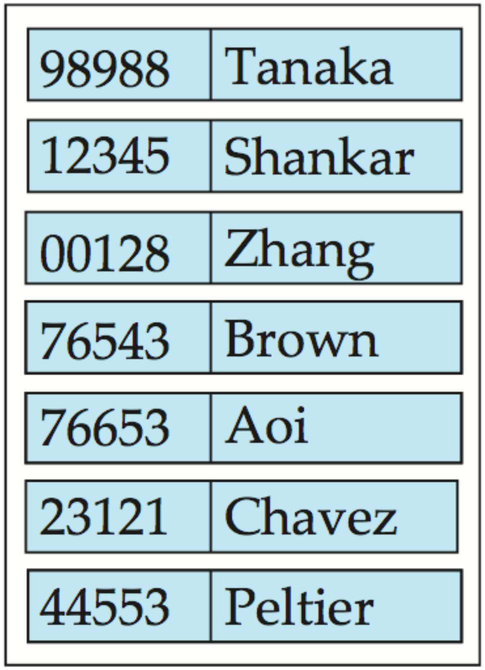
- A *database* can be modeled as:
  - a collection of entities,
  - relationship among entities.
- An **entity** is an object that exists and is distinguishable from other objects.
  - Example: specific person, company, event, plant
- Entities have **attributes**
  - Example: people have *names* and *addresses*
- An **entity set** is a set of entities of the same type that share the same properties.
  - Example: set of all persons, companies, trees, holidays

Entity Sets *instructor* and *student*



instructor_ID	instructor_name
76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor



student-ID	student_name
98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

Relationship Sets

- A **relationship** is an association among several entities

  Example:

44553 (Peltier)	*advisor*	22222 (Einstein)
student entity	relationship set	*instructor* entity

- A **relationship set** is a mathematical relation among $n \geq 2$ entities, each taken from entity sets

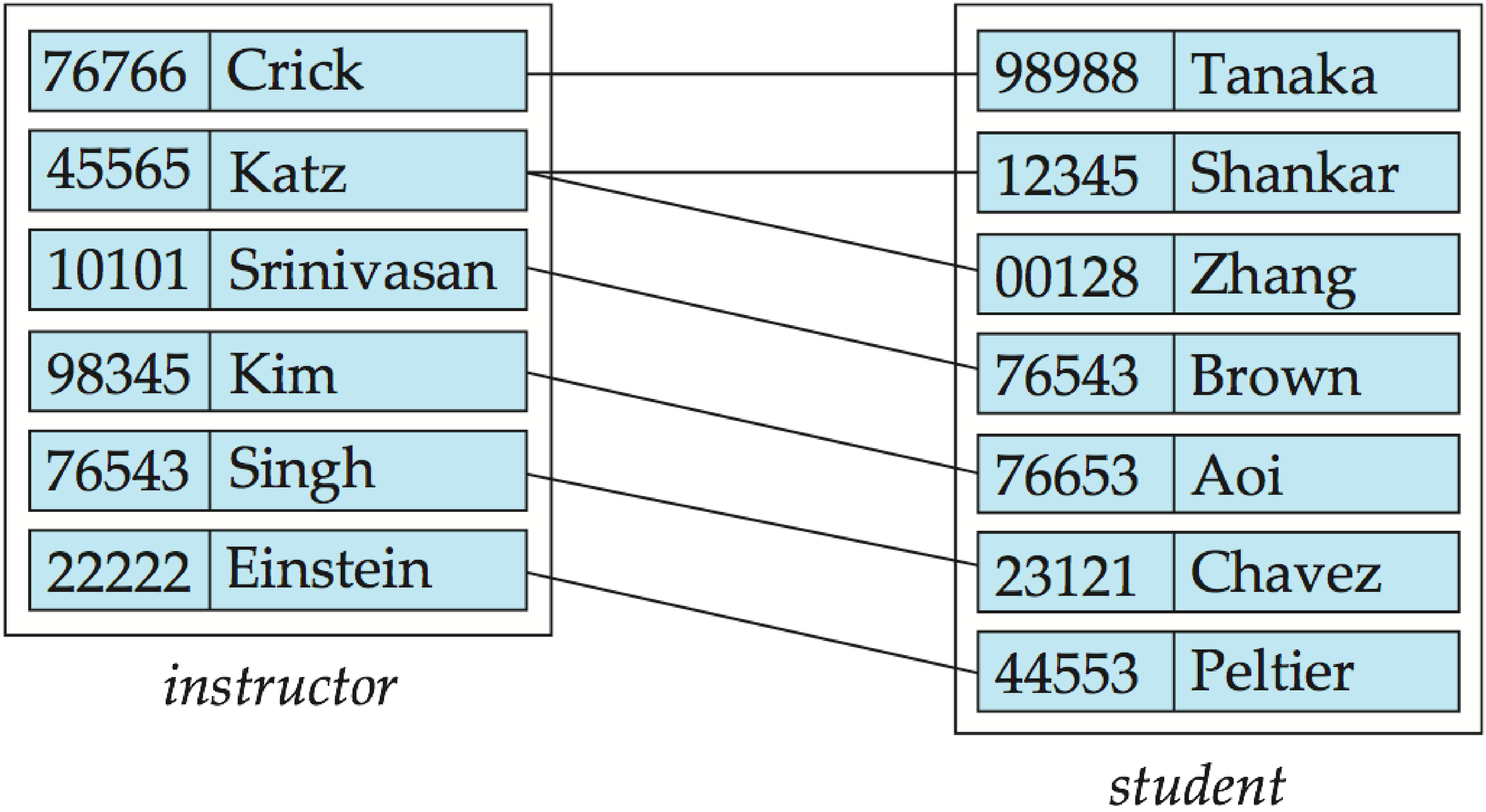
  $$\{(e_1, e_2, \ldots e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

  where $(e_1, e_2, \ldots, e_n)$ is a relationship

  - Example:

    $(44553,22222) \in$ *advisor*

Relationship Set *advisor*



76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

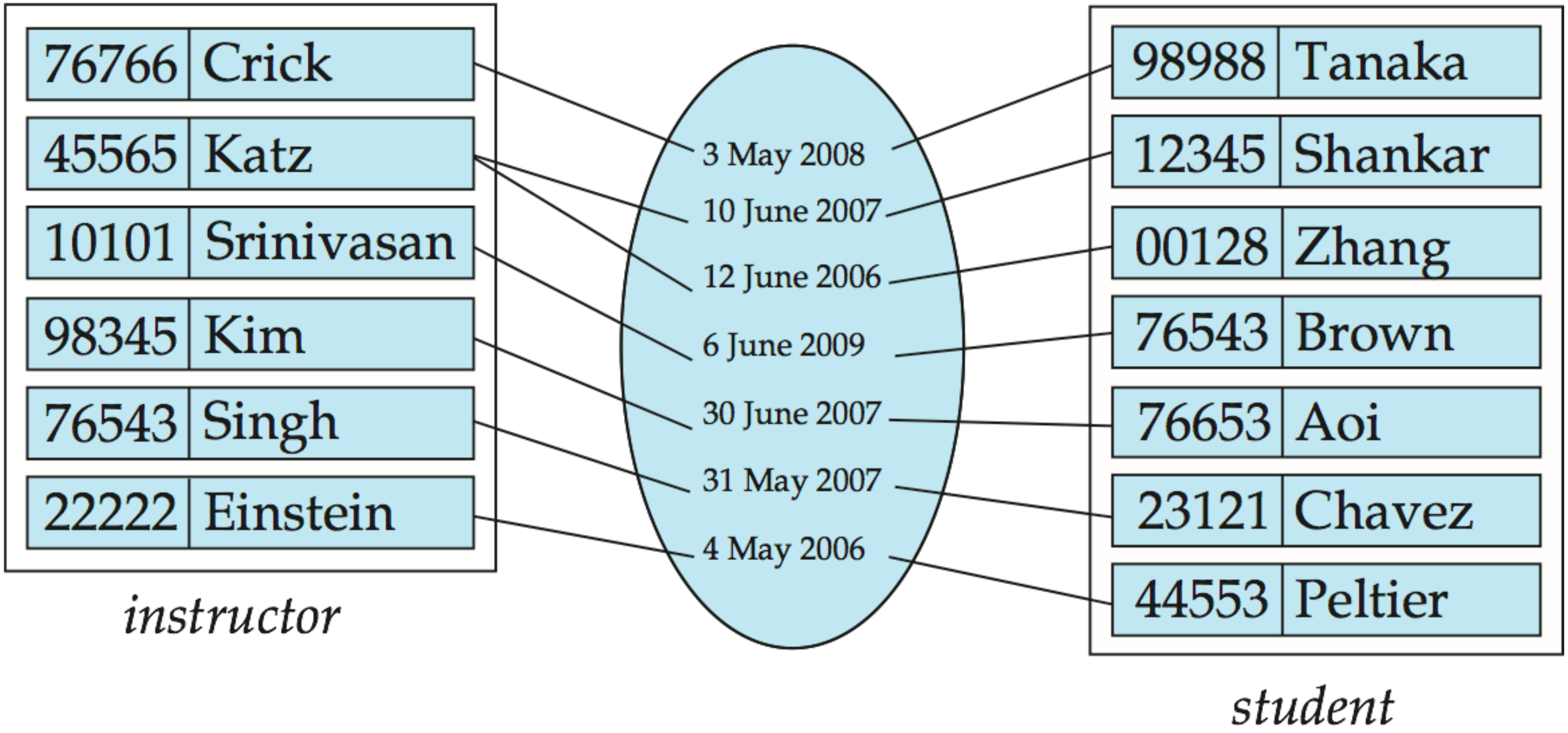
instructor

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

Relationship Sets (Cont.)

- An **attribute** can also be property of a relationship set.
- For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have the attribute *date* which tracks when the student started being associated with the advisor



76766 Crick
45565 Katz
10101 Srinivasan
98345 Kim
76543 Singh
22222 Einstein

instructor

3 May 2008
10 June 2007
12 June 2006
6 June 2009
30 June 2007
31 May 2007
4 May 2006

98988 Tanaka
12345 Shankar
00128 Zhang
76543 Brown
76653 Aoi
23121 Chavez
44553 Peltier

student

Degree of a Relationship Set

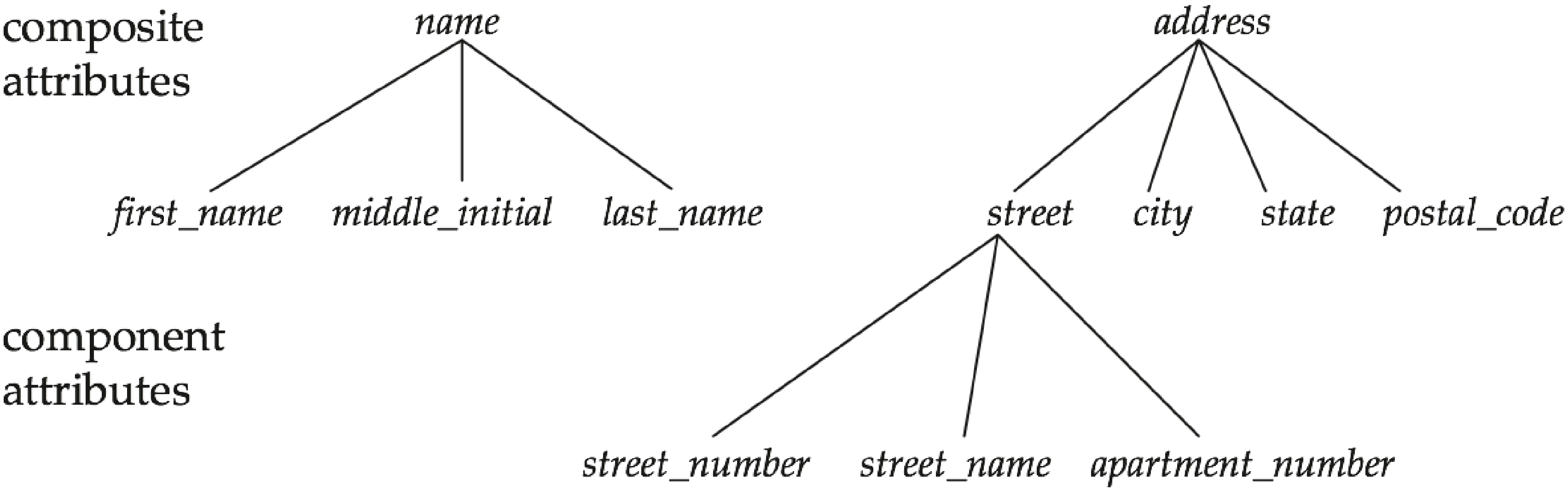
- **binary relationship**
  - involve two entity sets (or degree two).
  - most relationship sets in a database system are binary.
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)
  - Example: *students* work on research *projects* under the guidance of an *instructor*.
  - relationship *proj_guide* is a ternary relationship between *instructor, student,* and *project*

Attributes

- An entity is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set.
  - Example:

    instructor = (*ID, name, street, city, salary*)
    course= (*course_id, title, credits*)
- **Domain** – the set of permitted values for each attribute
- Attribute types:
  - **Simple** and **composite** attributes.
  - **Single-valued** and **multivalued** attributes
    - Example: multivalued attribute: *phone_numbers*
  - **Derived** attributes
    - Can be computed from other attributes
    - Example: age, given date_of_birth

Composite Attributes



composite attributes

component attributes

name

first_name *middle_initial* *last_name*

address

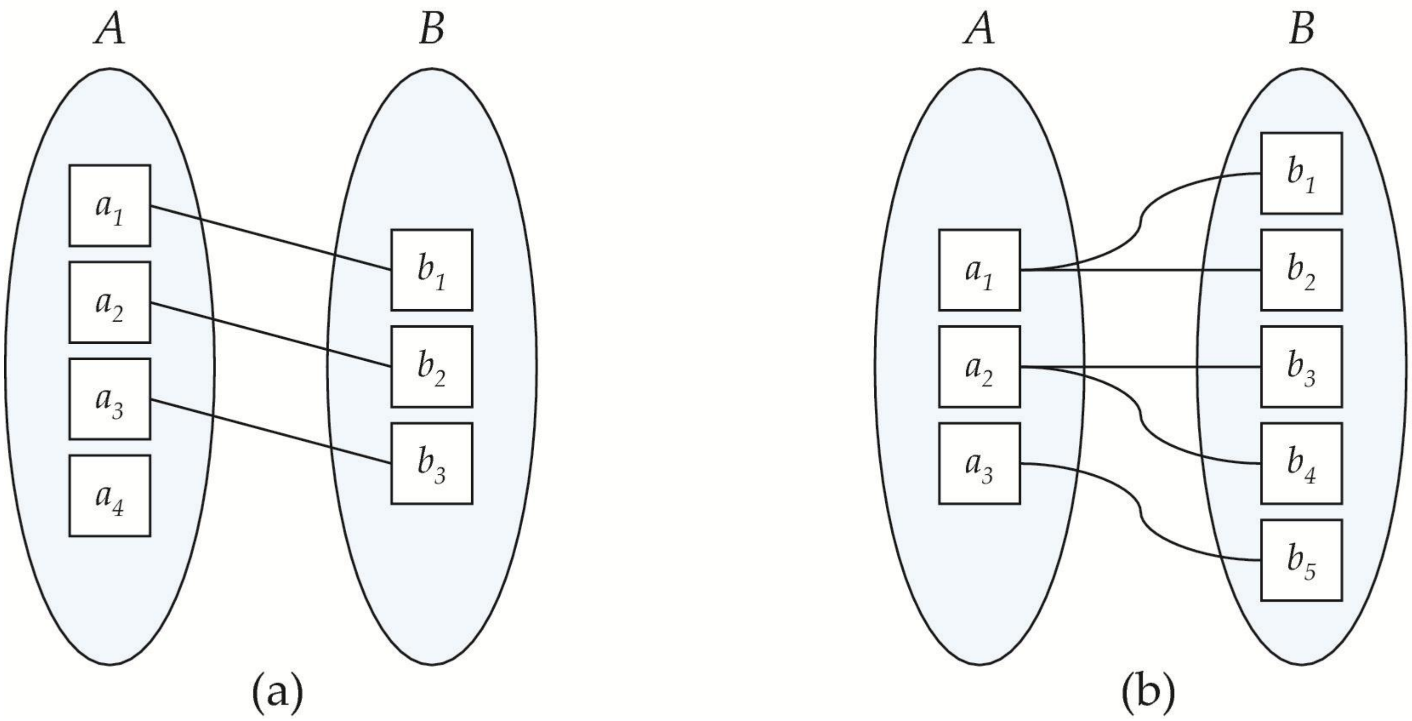
street *city* *state* *postal_code*

street_number *street_name* *apartment_number*

Mapping Cardinality Constraints

- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
  - One to one
  - One to many
  - Many to one
  - Many to many

Mapping Cardinalities

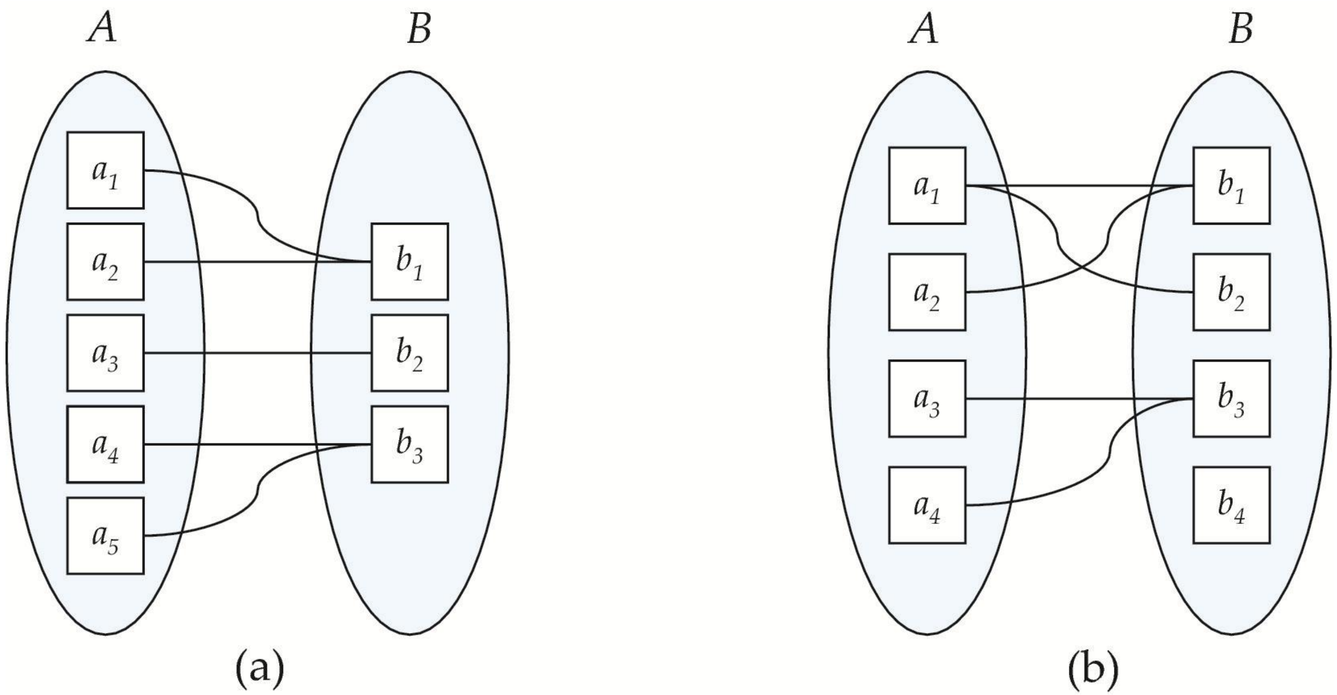


One to one

One to many

Note: Some elements in *A* and *B* may not be mapped to any elements in the other set

Mapping Cardinalities



(a)

(b)

Many to one

Many to many

Note: Some elements in A and B may not be mapped to any elements in the other set

Keys

- A **super key** of an entity set is a set of one or more attributes whose values uniquely determine each entity.
- A **candidate key** of an entity set is a minimal super key
  - *ID* is candidate key of *instructor*
  - *course_id* is candidate key of *course*
- Although several candidate keys may exist, one of the candidate keys is selected to be the **primary key**.

Keys for Relationship Sets

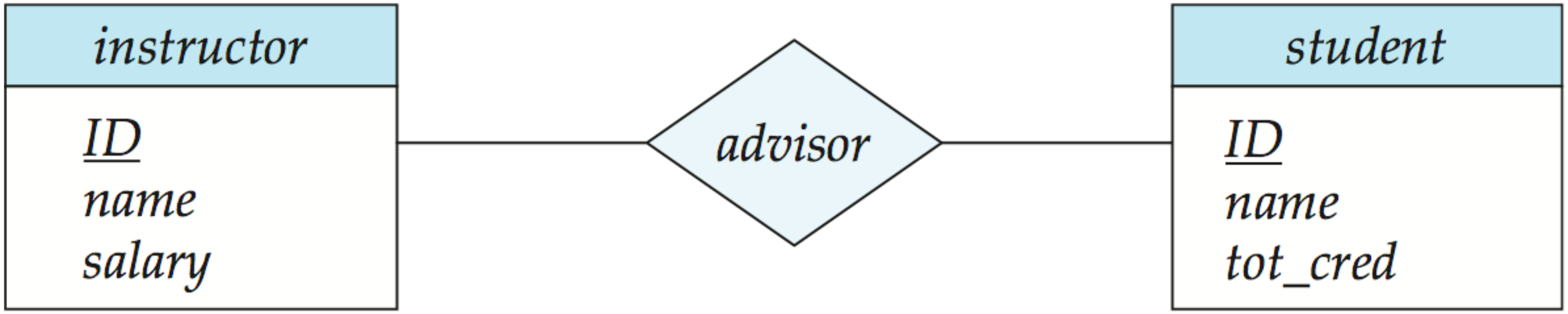
- The combination of primary keys of the participating entity sets forms a super key of a relationship set.
  - (*s_id, i_id*) is the super key of *advisor*
  - *NOTE: this means* ***a pair of entity sets can have at most one relationship in a particular relationship set.***
    - Example: if we wish to track multiple meeting dates between a student and her advisor, we cannot assume a relationship for each meeting. We can use a multivalued attribute though
- Must consider the mapping cardinality of the relationship set when deciding what are the candidate keys
- Need to consider semantics of relationship set in selecting the *primary key* in case of more than one candidate key

Redundant Attributes

- Suppose we have entity sets
  - *instructor*, with attributes including *dept_name*
  - *department*

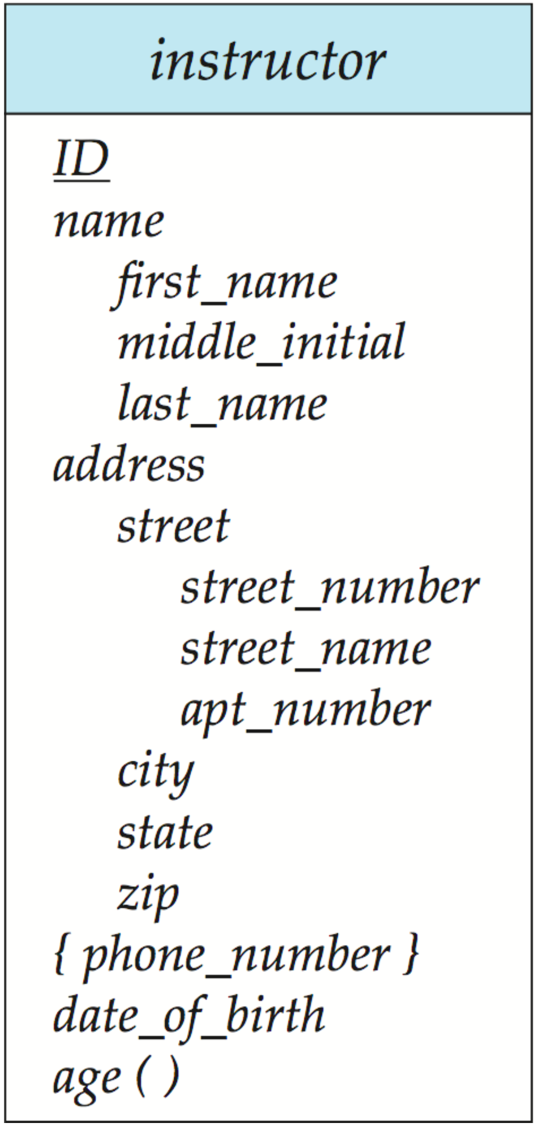
  and a relationship
  - *inst_dept* relating *instructor* and *department*
- Attribute *dept_name* in entity *instructor* is redundant since there is an explicit relationship *inst_dept* which relates instructors to departments
  - The attribute replicates information present in the relationship, and should be removed from *instructor*
  - BUT: when converting back to tables, in some cases the attribute gets reintroduced, as we will see.

E-R Diagrams



- Rectangles represent entity sets.
- Diamonds represent relationship sets.
- Attributes listed inside entity rectangle
- Underline indicates primary key attributes

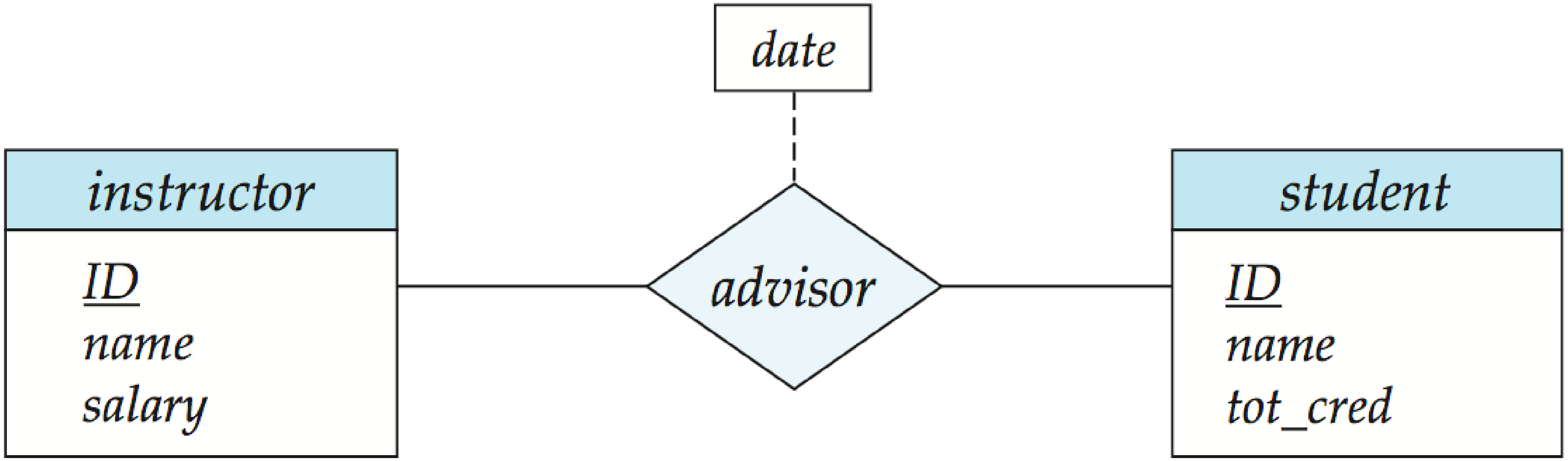
Entity With Composite, Multivalued, and Derived Attributes



instructor

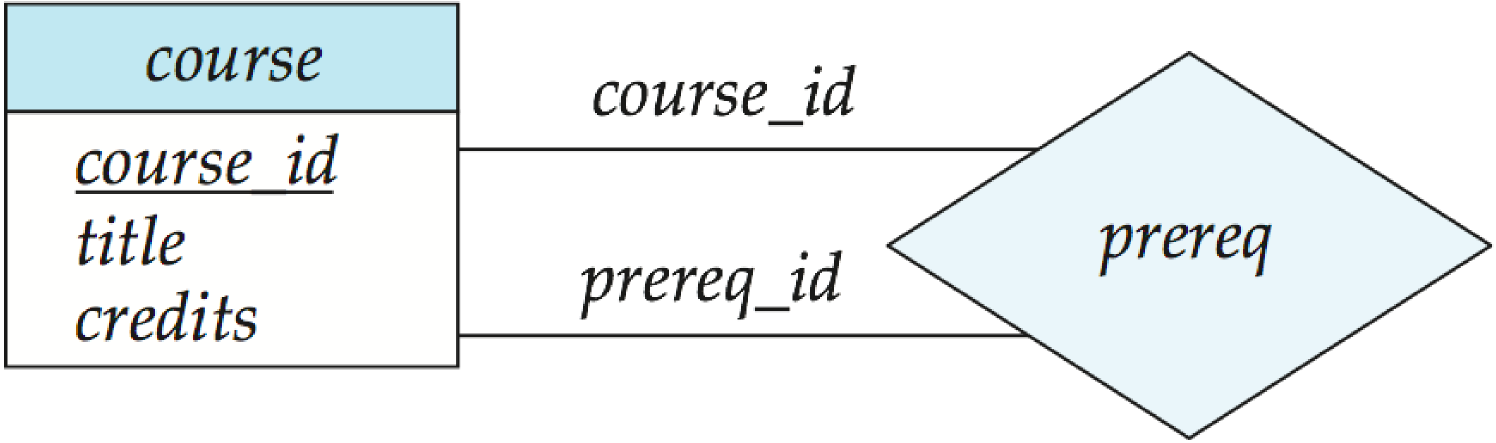
- *ID*
- *name*
    - *first_name*
    - *middle_initial*
    - *last_name*
- *address*
    - *street*
        - *street_number*
        - *street_name*
        - *apt_number*
    - *city*
    - *state*
    - *zip*
- *{ phone_number }*
- *date_of_birth*
- *age ( )*

Relationship Sets with Attributes



Roles

- Entity sets of a relationship need not be distinct
  - Each occurrence of an entity set plays a "role" in the relationship
- The labels "*course_id*" and "*prereq_id*" are called **roles**.

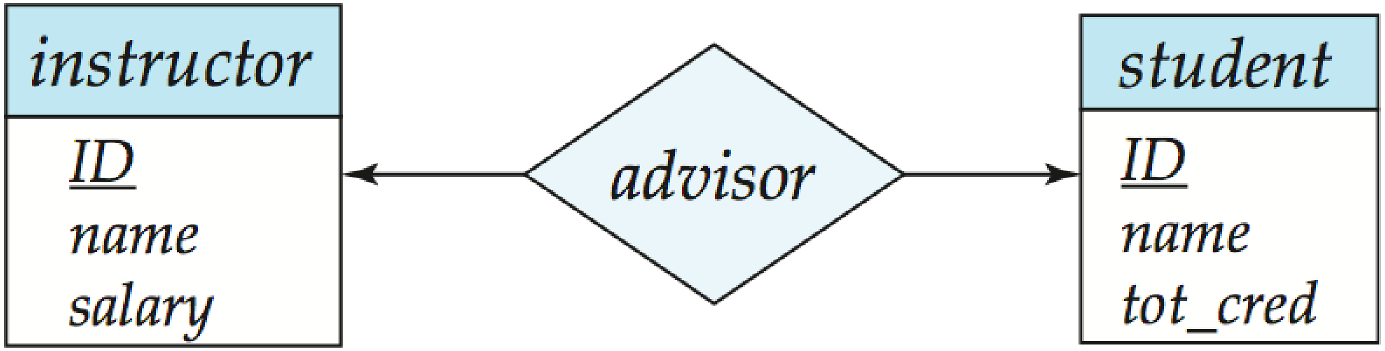


Cardinality Constraints

- We express cardinality constraints by drawing either a directed line (→), signifying "one," or an undirected line (—), signifying "many," between the relationship set and the entity set.
- One-to-one relationship:
  - A student is associated with at most one *instructor* via the relationship *advisor*
  - A *student* is associated with at most one *department* via *stud_dept*

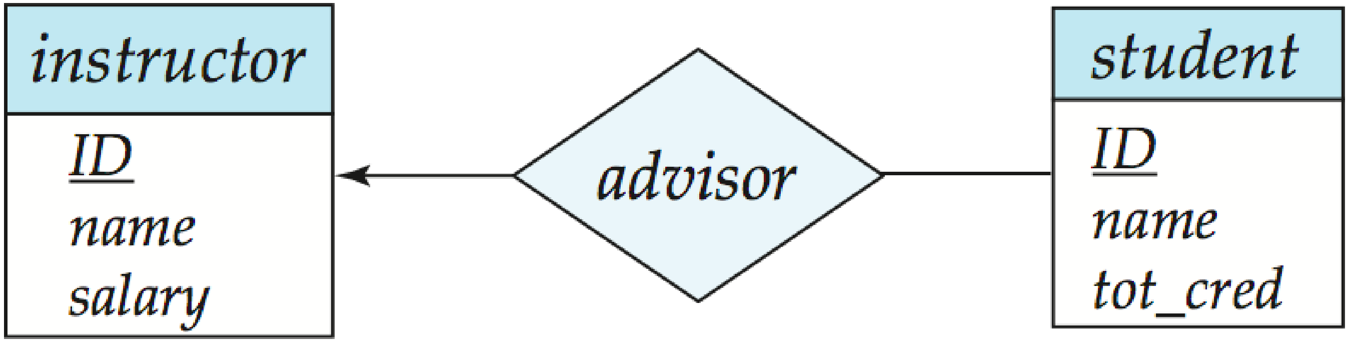
One-to-One Relationship

- one-to-one relationship between an *instructor* and a *student*
  - an instructor is associated with at most one student via *advisor*
  - and a student is associated with at most one instructor via *advisor*



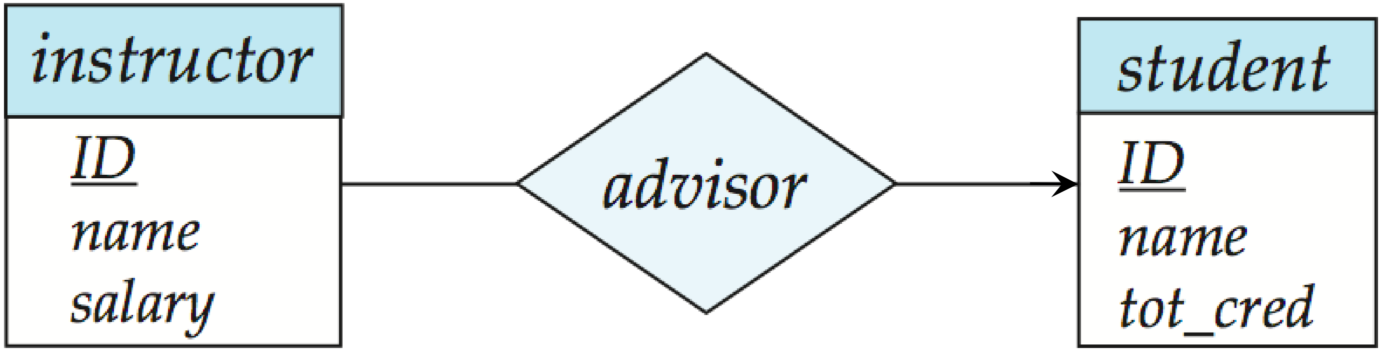
One-to-Many Relationship

- one-to-many relationship between an *instructor* and a *student*
  - an instructor is associated with several (including 0) students via *advisor*
  - a student is associated with at most one instructor via advisor,



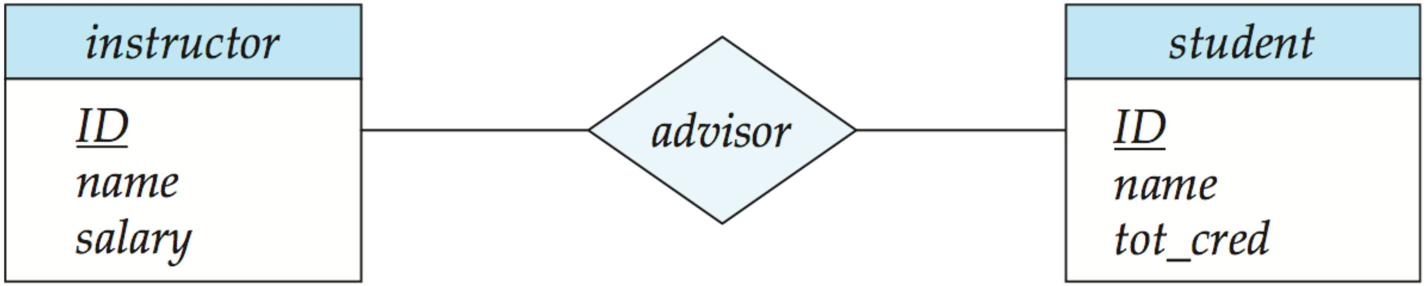
Many-to-One Relationships

- In a many-to-one relationship between an *instructor* and a *student,*
  - an instructor is associated with at most one student via *advisor*,
  - and a student is associated with several (including 0) instructors via *advisor*



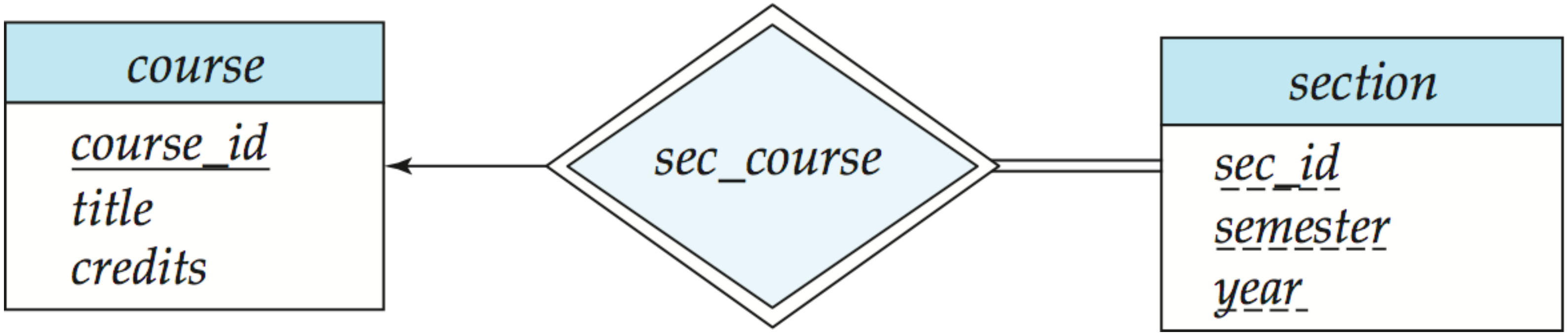
Many-to-Many Relationship

- An instructor is associated with several (possibly 0) students via *advisor*
- A student is associated with several (possibly 0) instructors via *advisor*



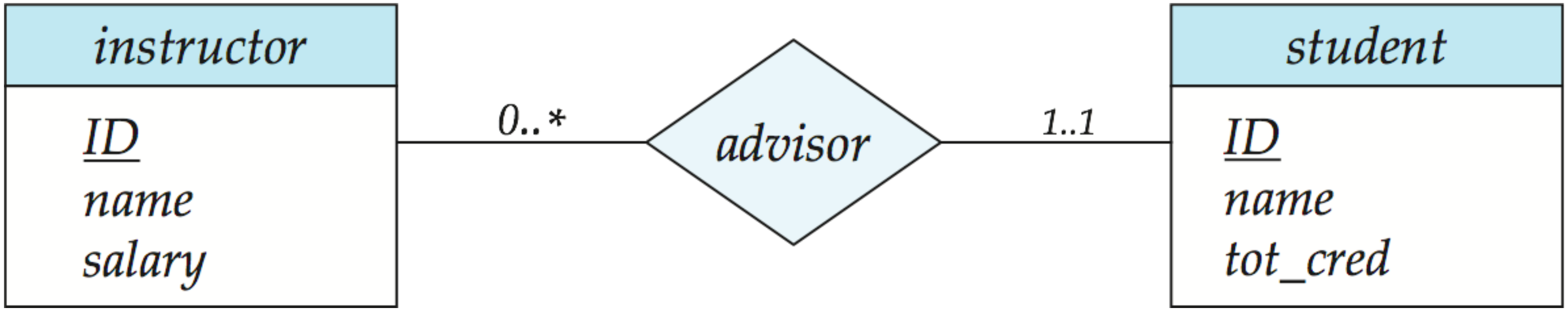
Participation of an Entity Set in a Relationship Set

- Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
  - E.g., participation of *section* in *sec_course* is total
    - every *section* must have an associated course
- Partial participation: some entities may not participate in any relationship in the relationship set
  - Example: participation of *instructor* in *advisor* is partial

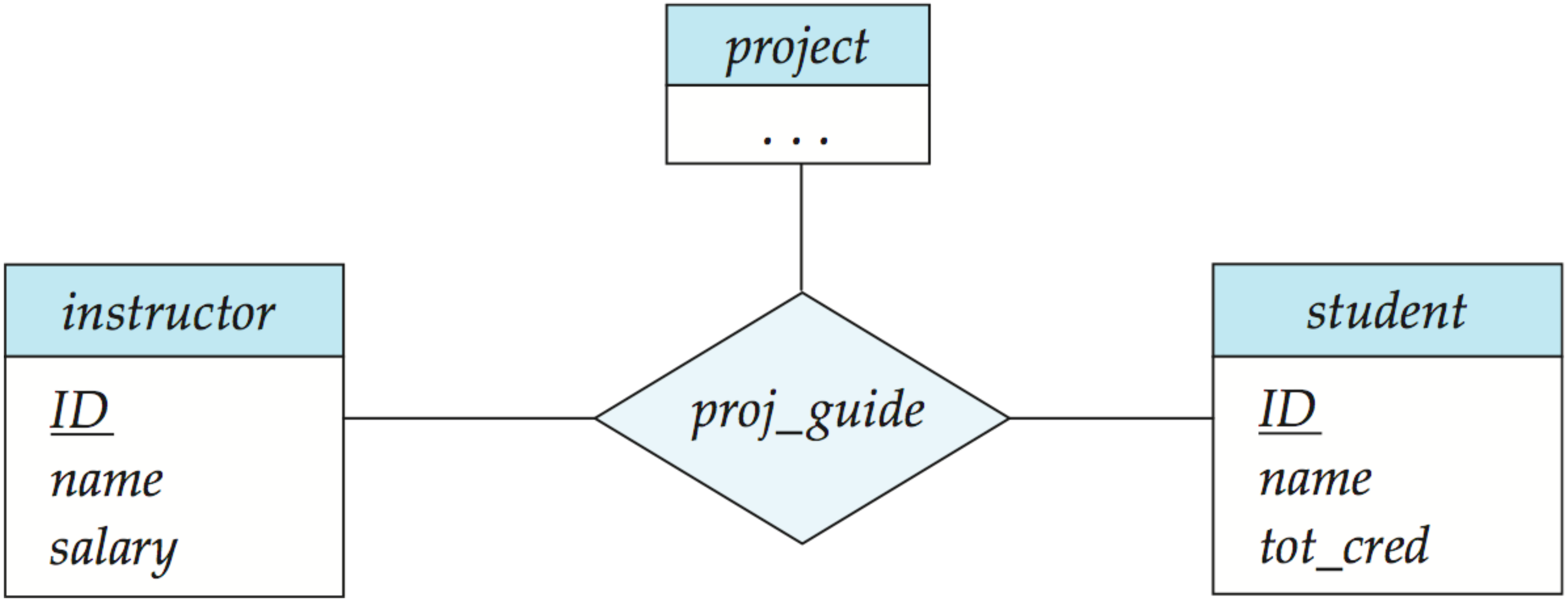


Alternative Notation for Cardinality Limits

- Cardinality limits can also express participation constraints



E-R Diagram with a Ternary Relationship



Cardinality Constraints on Ternary Relationship

- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- E.g., an arrow from *proj_guide* to *instructor* indicates each student has at most one guide for a project
- If there is more than one arrow, there are two ways of defining the meaning.
  - E.g., a ternary relationship *R* between *A*, *B* and *C* with arrows to *B* and *C* could mean

    1. each *A* entity is associated with a unique entity from *B* and *C* or

    2. each pair of entities from (*A, B*) is associated with a unique *C* entity, and each pair (*A, C*) is associated with a unique *B*
  - Each alternative has been used in different formalisms
  - To avoid confusion we outlaw more than one arrow

This image cannot currently be displayed.

How about doing an ER design interactively on the board? Suggest an application to be modeled.

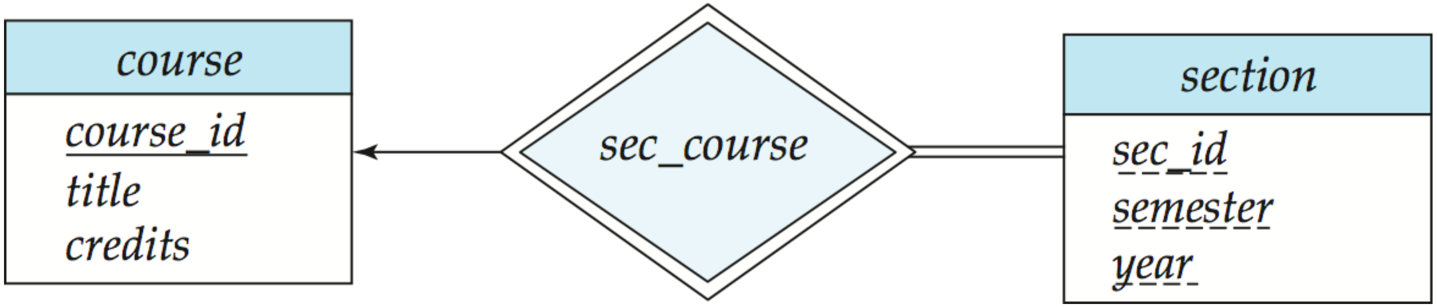
Database System Concepts, 6th Ed.

Weak Entity Sets

- An entity set that does not have a primary key is referred to as a **weak entity set**.
- The existence of a weak entity set depends on the existence of a **identifying entity set**
  - It must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set
  - **Identifying relationship** depicted using a double diamond
- The **discriminator** (*or partial key)* of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set.
- The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.

Weak Entity Sets (Cont.)

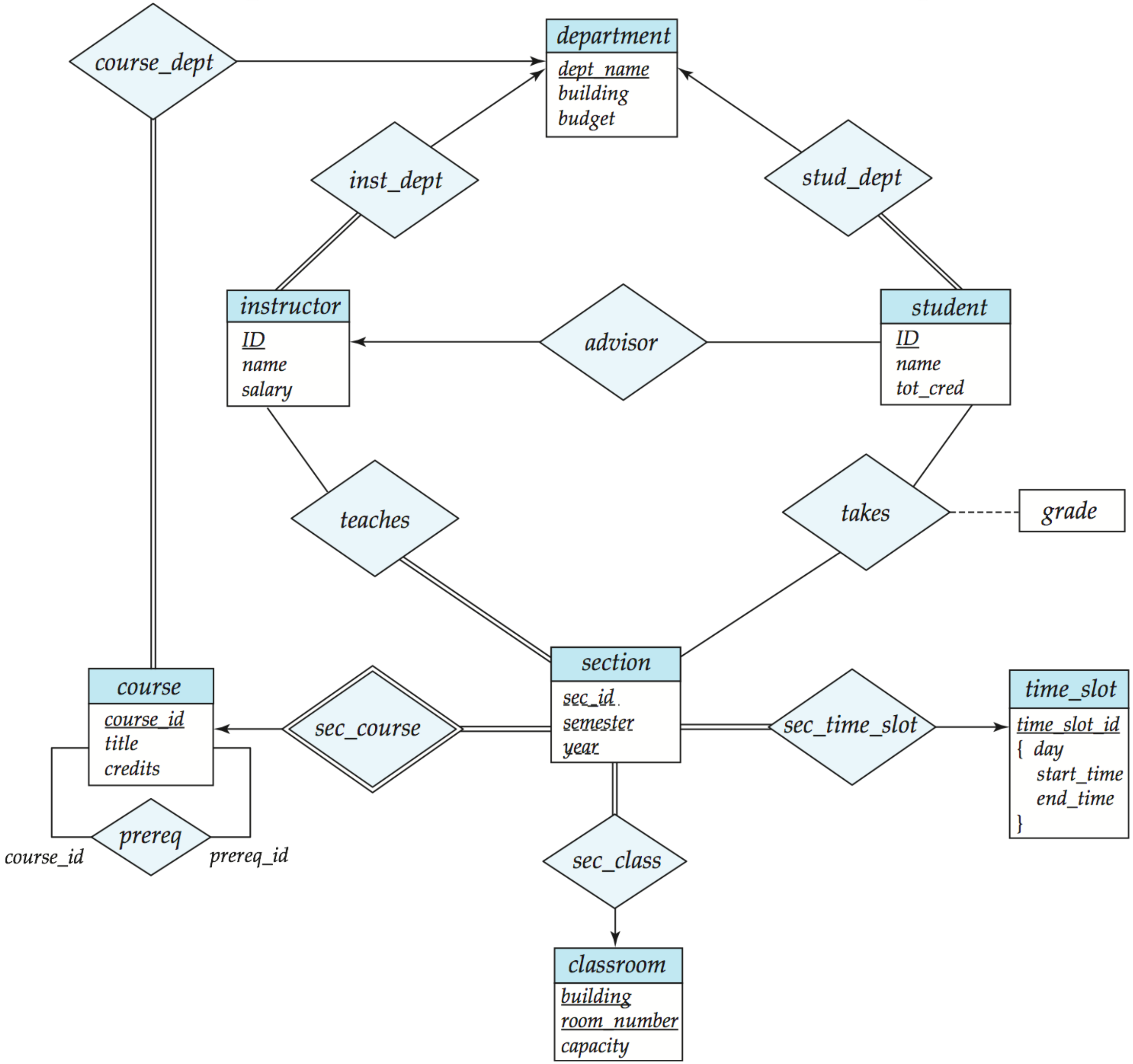
- We underline the discriminator of a weak entity set with a dashed line.
- We put the identifying relationship of a weak entity in a double diamond.
- Primary key for *section* – (*course_id, sec_id, semester, year*)



Weak Entity Sets (Cont.)

- Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship.
- If *course_id* were explicitly stored, *section* could be made a strong entity, but then the relationship between *section* and *course* would be duplicated by an implicit relationship defined by the attribute *course_id* common to *course* and *section*

E-R Diagram for a University Enterprise



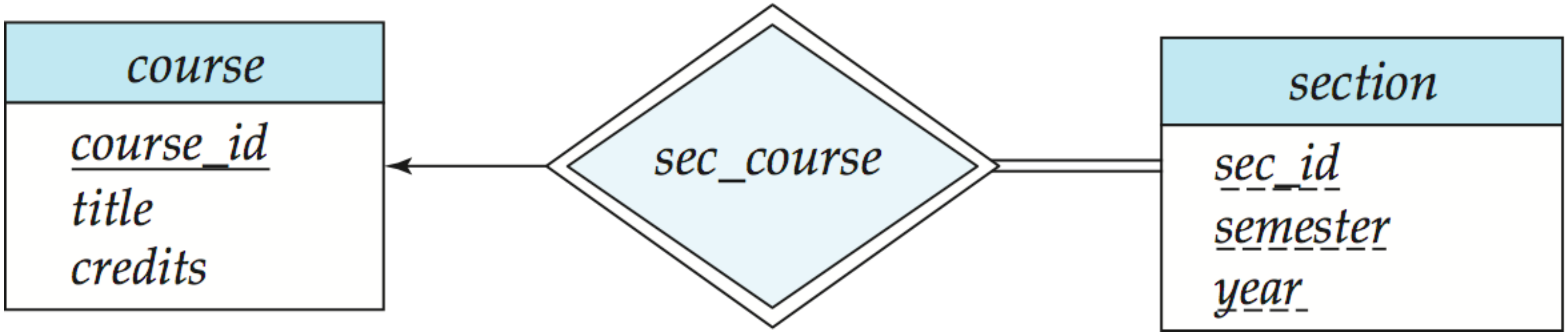
Reduction to Relational Schemas

Reduction to Relation Schemas

- Entity sets and relationship sets can be expressed uniformly as *relation schemas* that represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of schemas.
- For each entity set and relationship set there is a unique schema that is assigned the name of the corresponding entity set or relationship set.
- Each schema has a number of columns (generally corresponding to attributes), which have unique names.

Representing Entity Sets With Simple Attributes

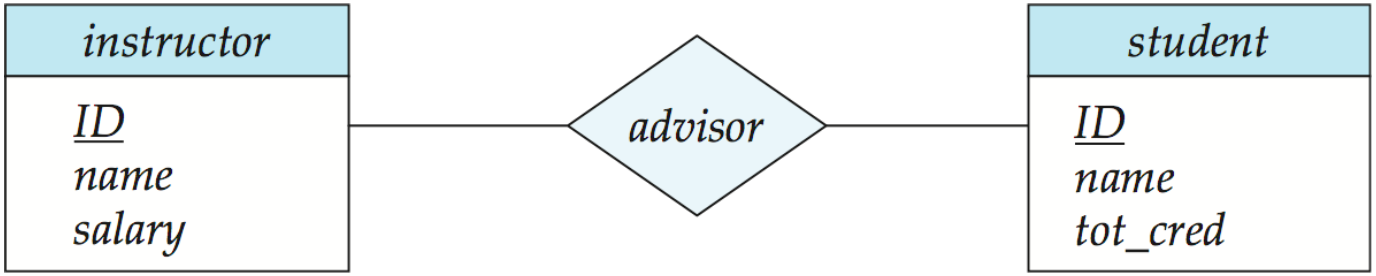
- A strong entity set reduces to a schema with the same attributes
  student(ID, name, tot_cred)
- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set
  section (course_id, sec_id, sem, year)



Representing Relationship Sets

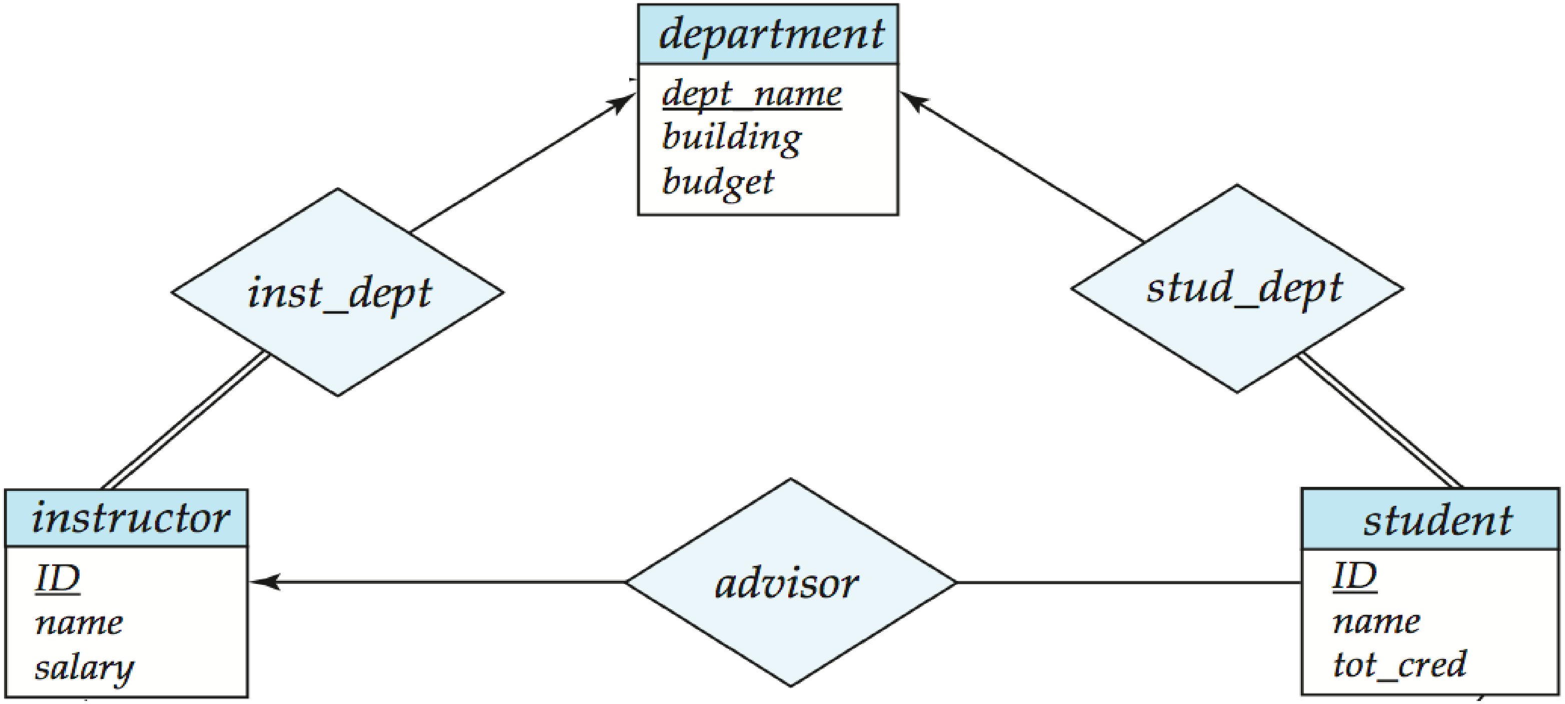
- A many-to-many relationship set is represented as a schema with attributes for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
- Example: schema for relationship set *advisor*

  advisor = (*s_id, i_id*)



Redundancy of Schemas

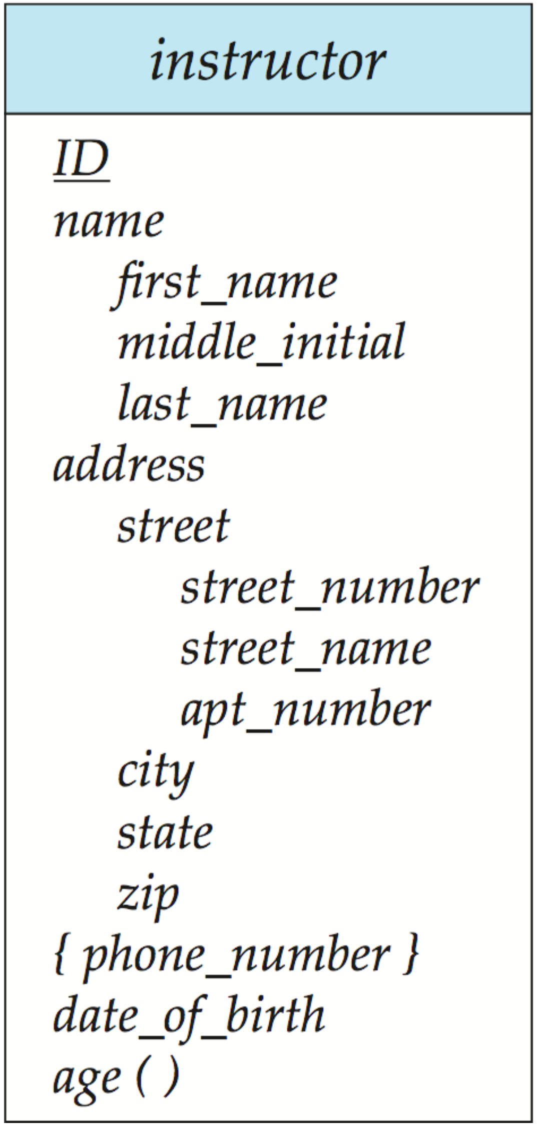
- Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the "many" side, containing the primary key of the "one" side
- Example: Instead of creating a schema for relationship set *inst_dept*, add an attribute *dept_name* to the schema arising from entity set *instructor*



Redundancy of Schemas (Cont.)

- For one-to-one relationship sets, either side can be chosen to act as the “many” side
  - That is, extra attribute can be added to either of the tables corresponding to the two entity sets
- If participation is *partial* on the “many” side, replacing a schema by an extra attribute in the schema corresponding to the “many” side could result in null values
- The schema corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.
  - Example: The *section* schema already contains the attributes that would appear in the *sec_course* schema

Composite and Multivalued Attributes



instructor

ID
name
  first_name
  middle_initial
  last_name
address
  street
    street_number
    street_name
    apt_number
  city
  state
  zip
{ phone_number }
date_of_birth
age ( )

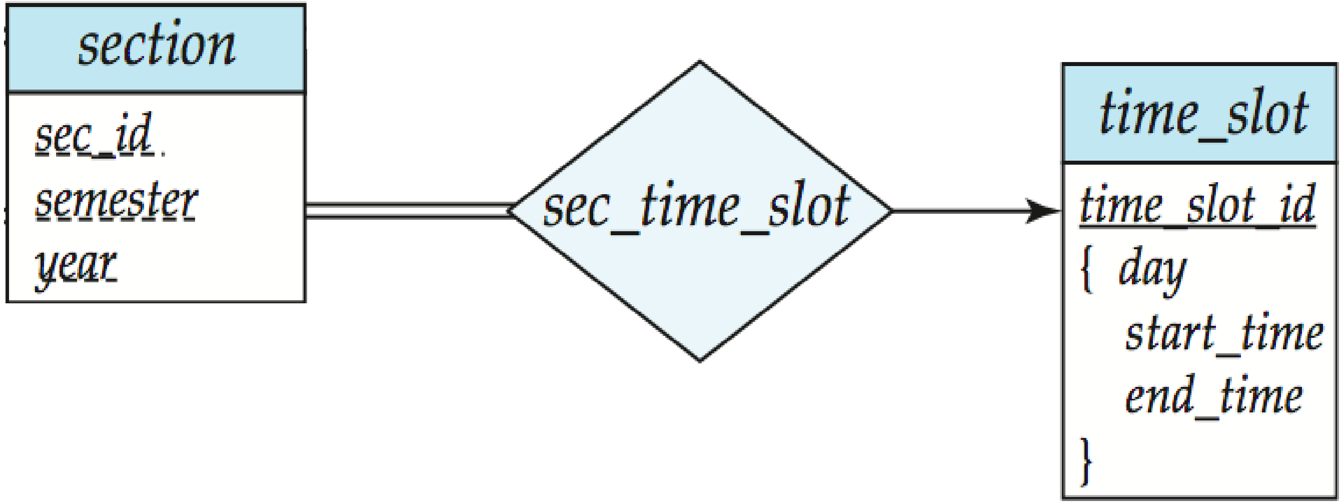
- Composite attributes are flattened out by creating a separate attribute for each component attribute
  - Example: given entity set *instructor* with composite attribute *name* with component attributes *first_name* and *last_name* the schema corresponding to the entity set has two attributes *name_first_name* and *name_last_name*
    - *Prefix omitted if there is no ambiguity*
- Ignoring multivalued attributes, extended instructor schema is
  - *instructor(ID,*
    first_name, middle_initial, last_name,
    street_number, street_name,
    apt_number, city, state, zip_code,
    date_of_birth)

Composite and Multivalued Attributes

- A multivalued attribute *M* of an entity *E* is represented by a separate schema *EM*
  - Schema *EM* has attributes corresponding to the primary key of *E* and an attribute corresponding to multivalued attribute *M*
  - Example: Multivalued attribute *phone_number* of *instructor* is represented by a schema:
    inst_phone= (*<u>ID</u>*, *<u>phone_number</u>*)
  - Each value of the multivalued attribute maps to a separate tuple of the relation on schema *EM*
    - For example, an *instructor* entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples:
      (22222, 456-7890) and (22222, 123-4567)

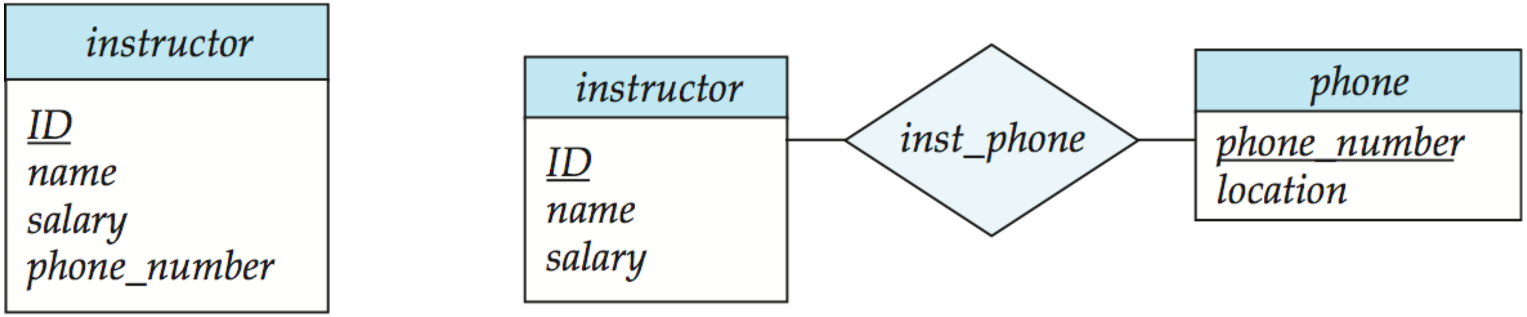
Multivalued Attributes (Cont.)

- Special case:entity *time_slot* has only one attribute other than the primary-key attribute, and that attribute is multivalued
  - Optimization: Don't create the relation corresponding to the entity, just create the one corresponding to the multivalued attribute
  - *time_slot*(<u>*time_slot_id, day, start_time*</u>, *end_time*)
  - Caveat: *time_slot* attribute of *section* (from *sec_time_slot*) cannot be a foreign key due to this optimization



Design Issues

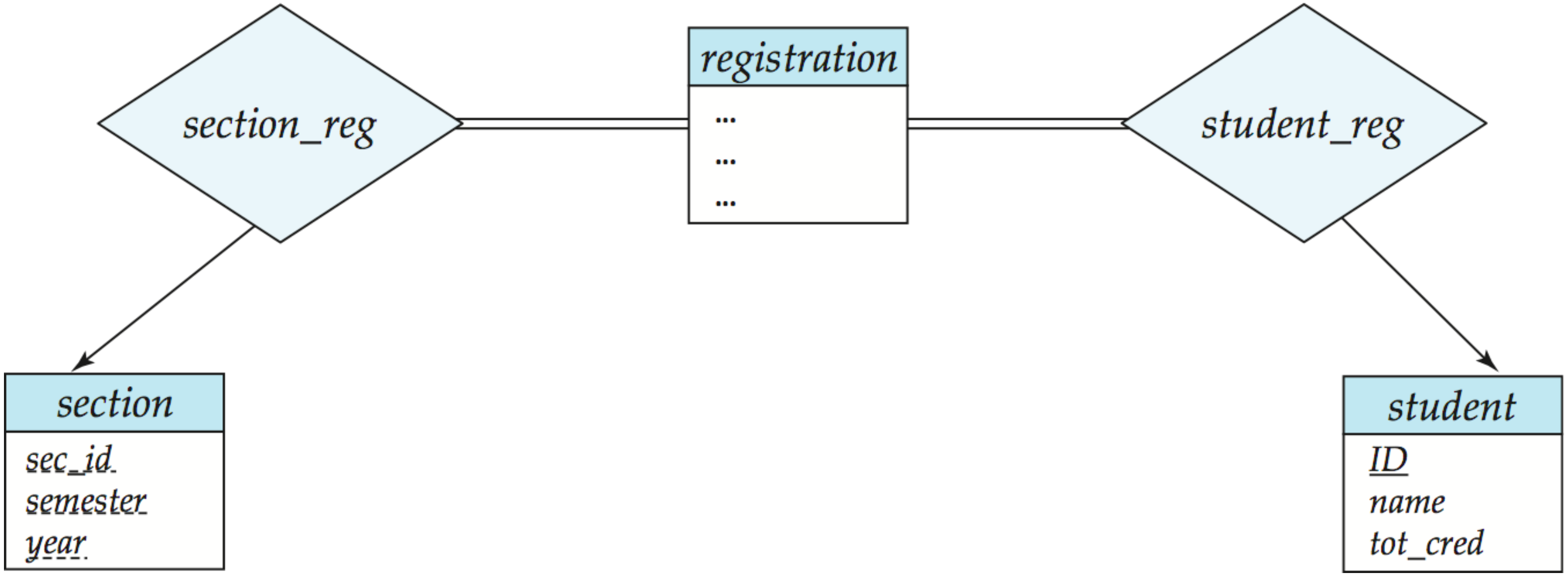
- **Use of entity sets vs. attributes**



- Use of phone as an entity allows extra information about phone numbers (plus multiple phone numbers)

Design Issues

- **Use of entity sets vs. relationship sets**
  Possible guideline is to designate a relationship set to describe an action that occurs between entities



Design Issues

- **Binary versus n-ary relationship sets**
  Although it is possible to replace any nonbinary (*n*-ary, for *n* > 2) relationship set by a number of distinct binary relationship sets, a *n*-ary relationship set shows more clearly that several entities participate in a single relationship.
- **Placement of relationship attributes**

  e.g., attribute *date* as attribute of *advisor* or as attribute of *student*

Binary Vs. Non-Binary Relationships

- Some relationships that appear to be non-binary may be better represented using binary relationships
  - E.g., A ternary relationship *parents*, relating a child to his/her father and mother, is best replaced by two binary relationships, *father* and *mother*
    - Using two binary relationships allows partial information (e.g., only mother being know)
  - But there are some relationships that are naturally non-binary
    - Example: *proj_guide*

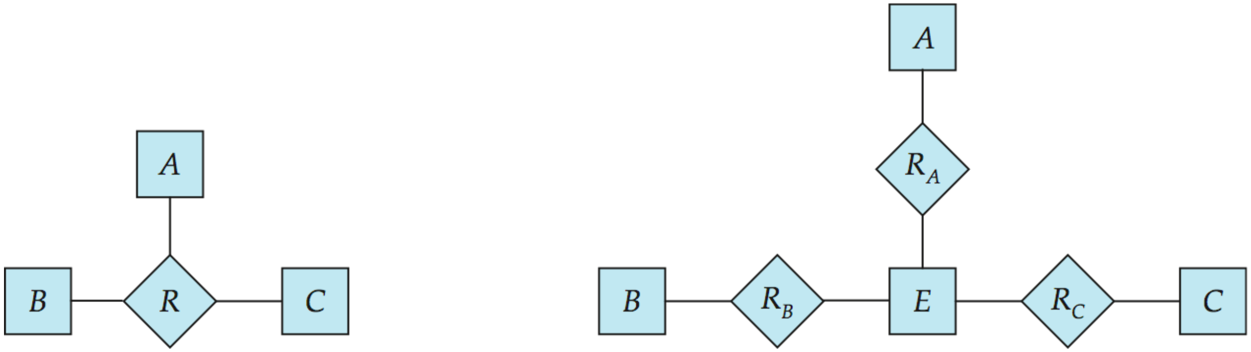
Converting Non-Binary Relationships to Binary Form

- In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
  - Replace R between entity sets A, B and C by an entity set E, and three relationship sets:

    1. R_A, relating E and A 2. R_B, relating E and B
    3. R_C, relating E and C
  - Create a special identifying attribute for E
  - Add any attributes of R to E
  - For each relationship (a_i , b_i , c_i) in R, create

    1. a new entity e_i in the entity set E 2. add (e_i , a_i) to R_A

    3. add (e_i , b_i) to R_B 4. add (e_i , c_i) to R_C



(a) (b)

Converting Non-Binary Relationships (Cont.)

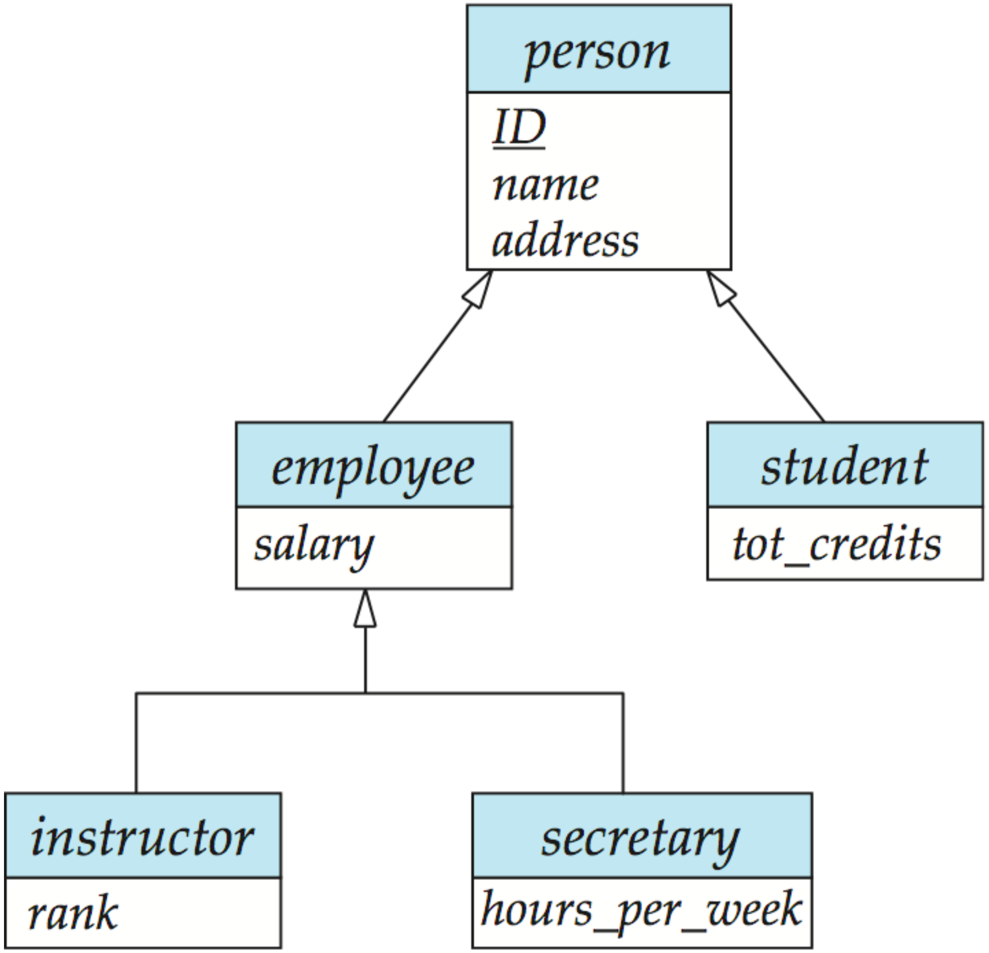
- Also need to translate constraints
  - Translating all constraints may not be possible
  - There may be instances in the translated schema that cannot correspond to any instance of *R*
    - Exercise: *add constraints to the relationships R_A, R_B and R_C* to ensure that a newly created entity corresponds to exactly one entity in each of entity sets *A, B* and *C*
  - We can avoid creating an identifying attribute by making E a weak entity set (described shortly) identified by the three relationship sets

Extended ER Features

Extended E-R Features: Specialization

- Top-down design process; we designate subgroupings within an entity set that are distinctive from other entities in the set.
- These subgroupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labeled ISA (E.g., *instructor* "is a" *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

Specialization Example



Extended ER Features: Generalization

- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably.

Specialization and Generalization (Cont.)

- Can have multiple specializations of an entity set based on different features.
- E.g., *permanent_employee* vs. *temporary_employee*, in addition to *instructor* vs. *secretary*
- Each particular employee would be
  - a member of one of *permanent_employee* or *temporary_employee*,
  - and also a member of one of *instructor*, *secretary*
- The ISA relationship also referred to as **superclass - subclass** relationship

Design Constraints on a Specialization/Generalization

- Constraint on which entities can be members of a given lower-level entity set.
  - condition-defined
    - Example: all customers over 65 years are members of *senior-citizen* entity set; *senior-citizen* ISA *person*.
  - user-defined
- Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.
  - **Disjoint**
    - an entity can belong to only one lower-level entity set
    - Noted in E-R diagram by having multiple lower-level entity sets link to the same triangle
  - **Overlapping**
    - an entity can belong to more than one lower-level entity set

Design Constraints on a Specialization/Generalization (Cont.)

- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
  - **total**: an entity must belong to one of the lower-level entity sets
  - **partial**: an entity need not belong to one of the lower-level entity sets

Representing Specialization via Schemas

- Method 1:
  - Form a schema for the higher-level entity
  - Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

schema	attributes
person	*ID, name, street, city*
student	*ID, tot_cred*
employee	*ID, salary*

  - Drawback: getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema

Representing Specialization as Schemas (Cont.)

- Method 2:
  - Form a schema for each entity set with all local and inherited attributes

schema	attributes
person	*ID, name, street, city*
student	*ID, name, street, city, tot_cred*
employee	*ID, name, street, city, salary*

  - If specialization is total, the schema for the generalized entity set (*person*) not required to store information
    - Can be defined as a "view" relation containing union of specialization relations
    - But explicit schema may still be needed for foreign key constraints
  - Drawback: *name, street* and *city* may be stored redundantly for people who are both students and employees

Schemas Corresponding to Aggregation

- To represent aggregation, create a schema containing
  - primary key of the aggregated relationship,
  - the primary key of the associated entity set
  - any descriptive attributes

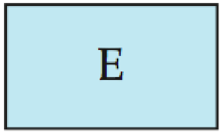
E-R Design Decisions

- The use of an attribute or entity set to represent an object.
- Whether a real-world concept is best expressed by an entity set or a relationship set.
- The use of a ternary relationship versus a pair of binary relationships.
- The use of a strong or weak entity set.
- The use of specialization/generalization – contributes to modularity in the design.
- The use of aggregation – can treat the aggregate entity set as a single unit without concern for the details of its internal structure.

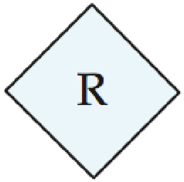
This image cannot currently be displayed.

How about doing another ER design interactively on the board?

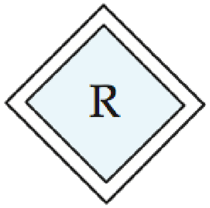
Summary of Symbols Used in E-R Notation



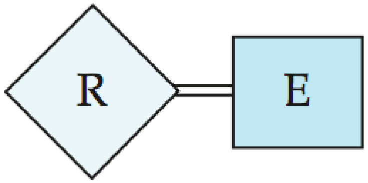
entity set



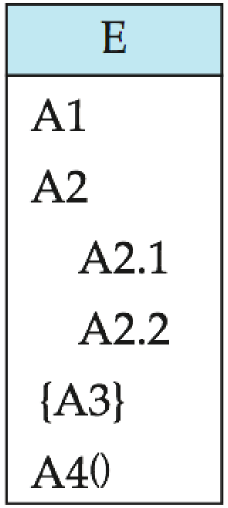
relationship set



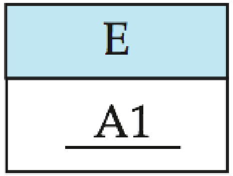
identifying
relationship set
for weak entity set



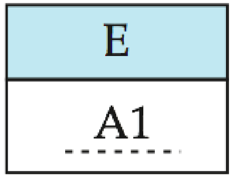
total participation
of entity set in
relationship



attributes:
simple (A1),
composite (A2) and
multivalued (A3)
derived (A4)

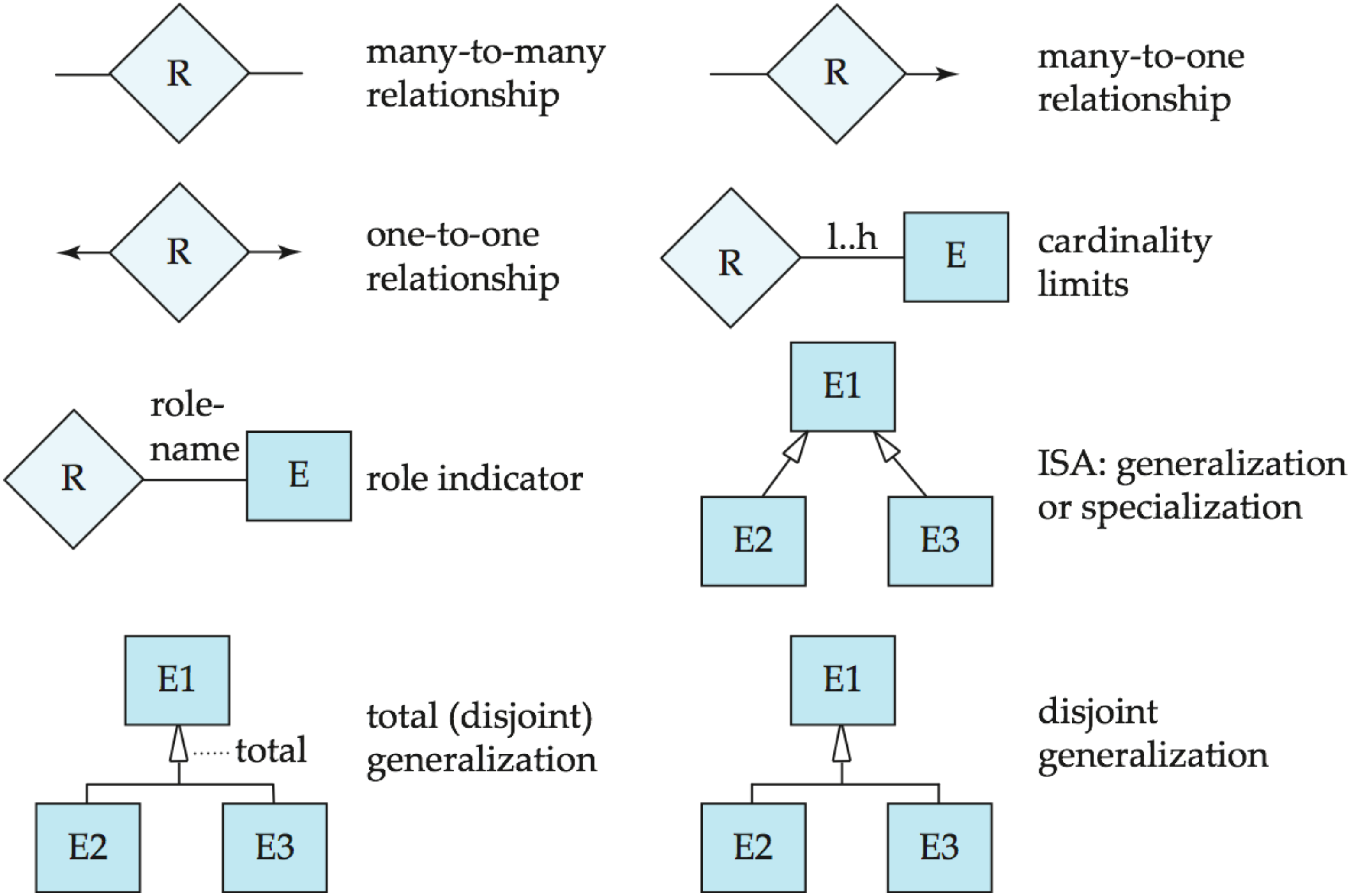


primary key



discriminating
attribute of
weak entity set

Symbols Used in E-R Notation (Cont.)



many-to-many relationship

many-to-one relationship

one-to-one relationship

cardinality limits

role indicator

ISA: generalization or specialization

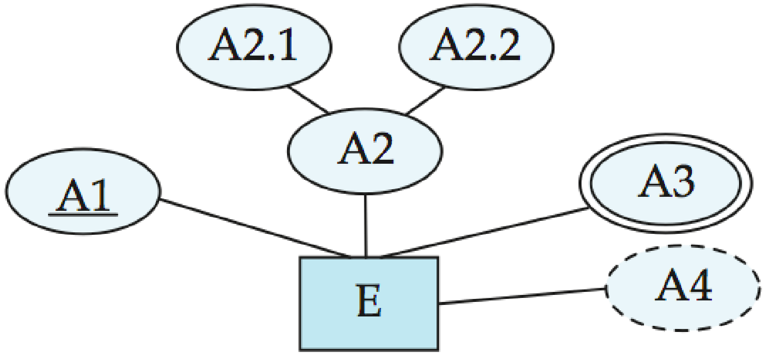
total (disjoint) generalization

disjoint generalization

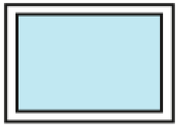
Alternative ER Notations

- Chen, IDE1FX, …

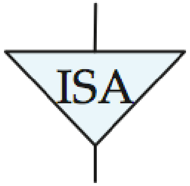
entity set E with
simple attribute A1,
composite attribute A2,
multivalued attribute A3,
derived attribute A4,
and primary key A1



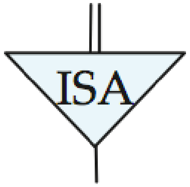
weak entity set



generalization

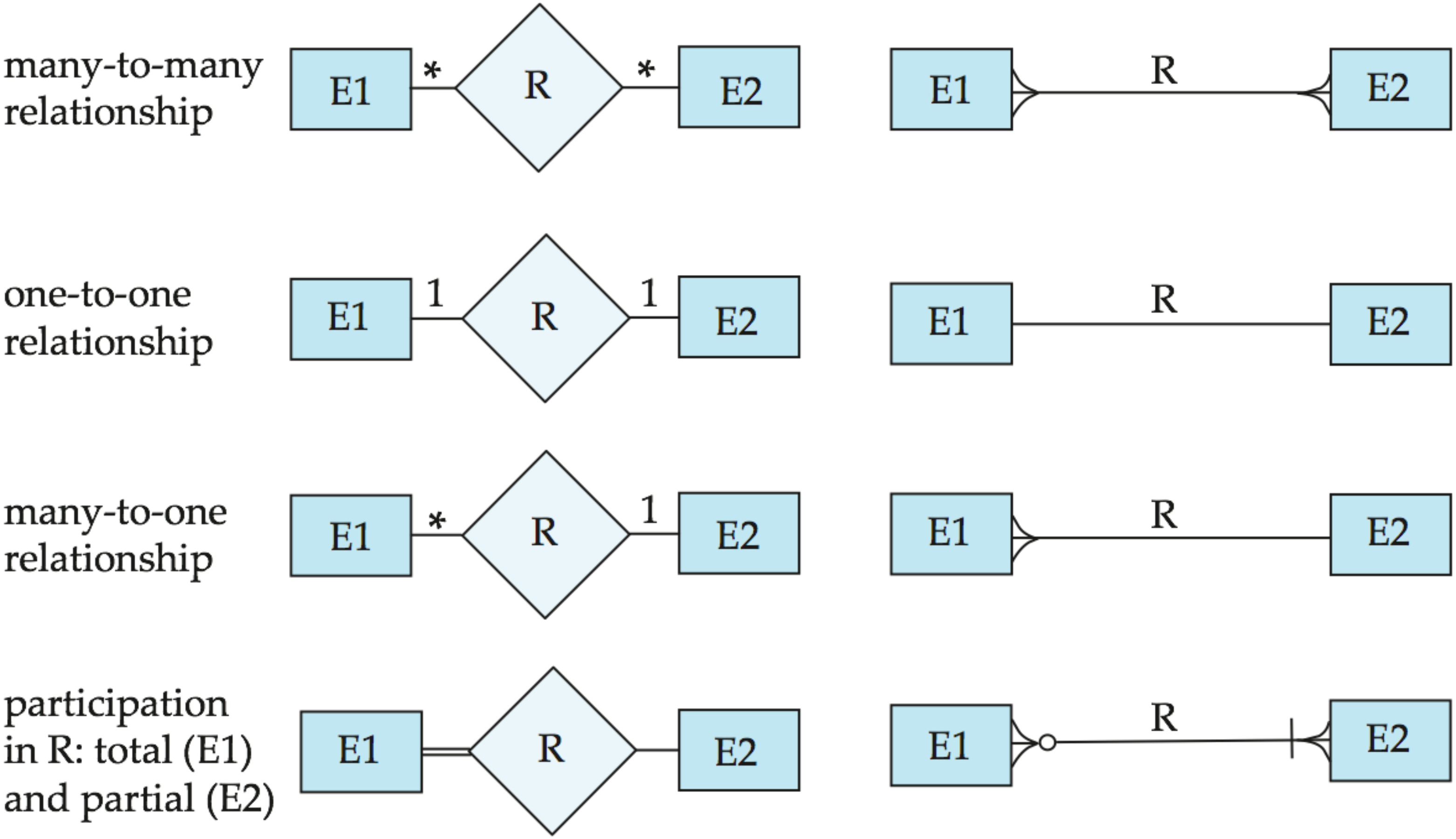


total
generalization



Alternative ER Notations

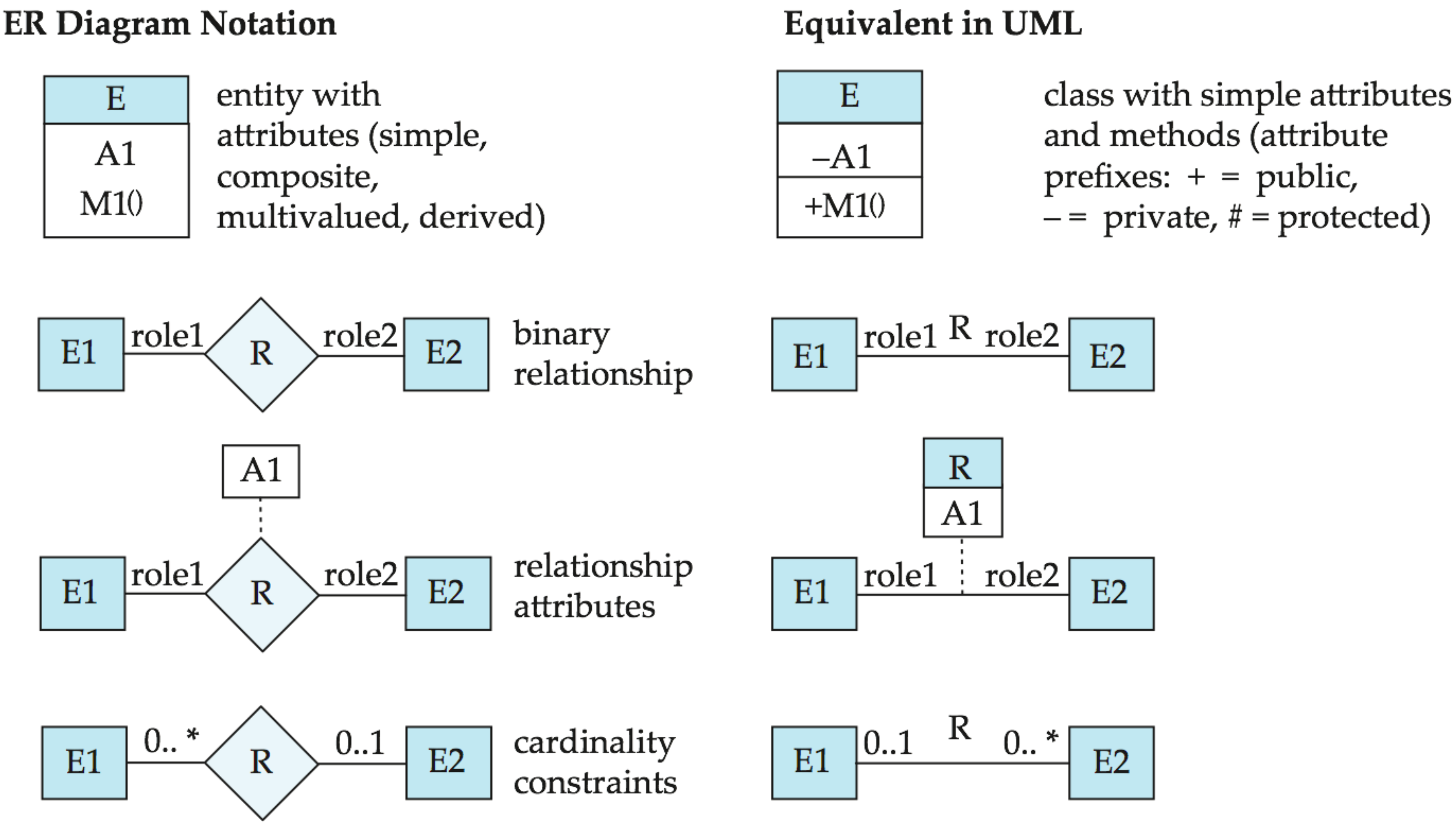
Chen | IDE1FX (Crows feet notation)



UML

- **UML**: Unified Modeling Language
- UML has many components to graphically model different aspects of an entire software system
- UML Class Diagrams correspond to E-R Diagram, but several differences.

ER vs. UML Class Diagrams



ER Diagram Notation

Equivalent in UML

entity with attributes (simple, composite, multivalued, derived)

class with simple attributes and methods (attribute prefixes: + = public, – = private, # = protected)

binary relationship

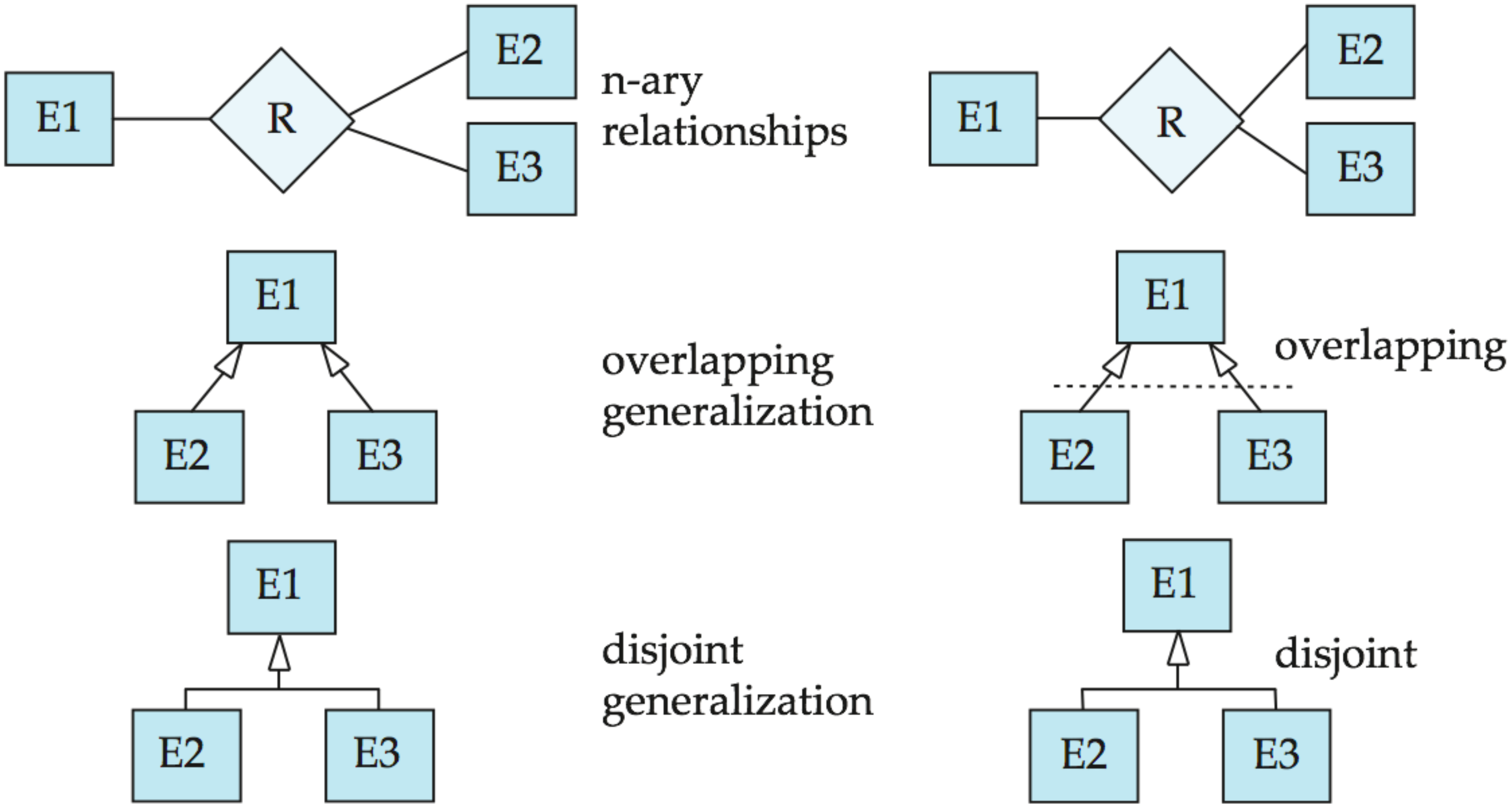
relationship attributes

cardinality constraints

*Note reversal of position in cardinality constraint depiction

ER vs. UML Class Diagrams

ER Diagram Notation **Equivalent in UML**



*Generalization can use merged or separate arrows independent of disjoint/overlapping

UML Class Diagrams (Cont.)

- Binary relationship sets are represented in UML by just drawing a line connecting the entity sets. The relationship set name is written adjacent to the line.
- The role played by an entity set in a relationship set may also be specified by writing the role name on the line, adjacent to the entity set.
- The relationship set name may alternatively be written in a box, along with attributes of the relationship set, and the box is connected, using a dotted line, to the line depicting the relationship set.

This image cannot currently be displayed.

End of Chapter 7

Database System Concepts, 6th Ed.

Figure 7.01

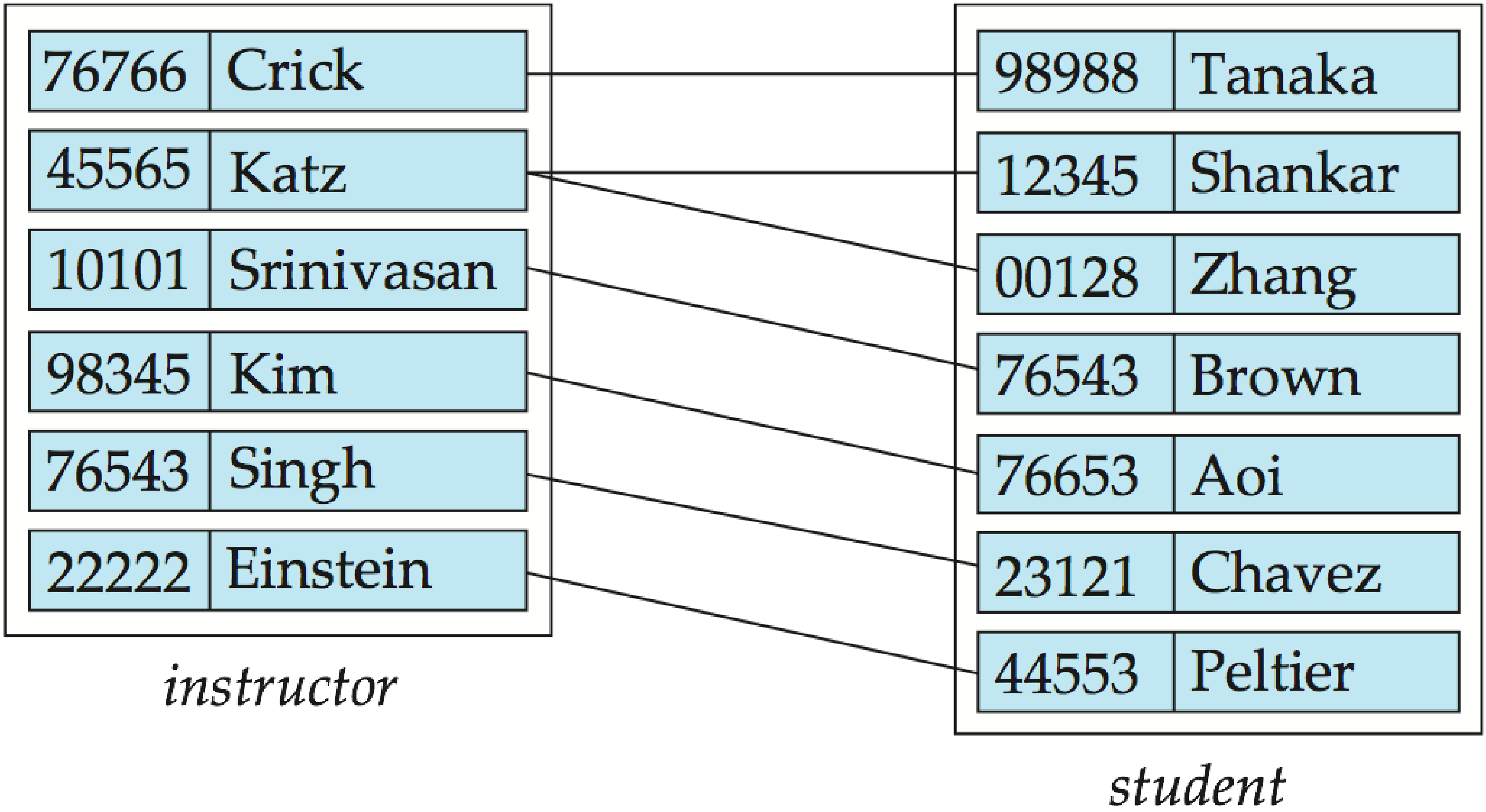
76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

Figure 7.02



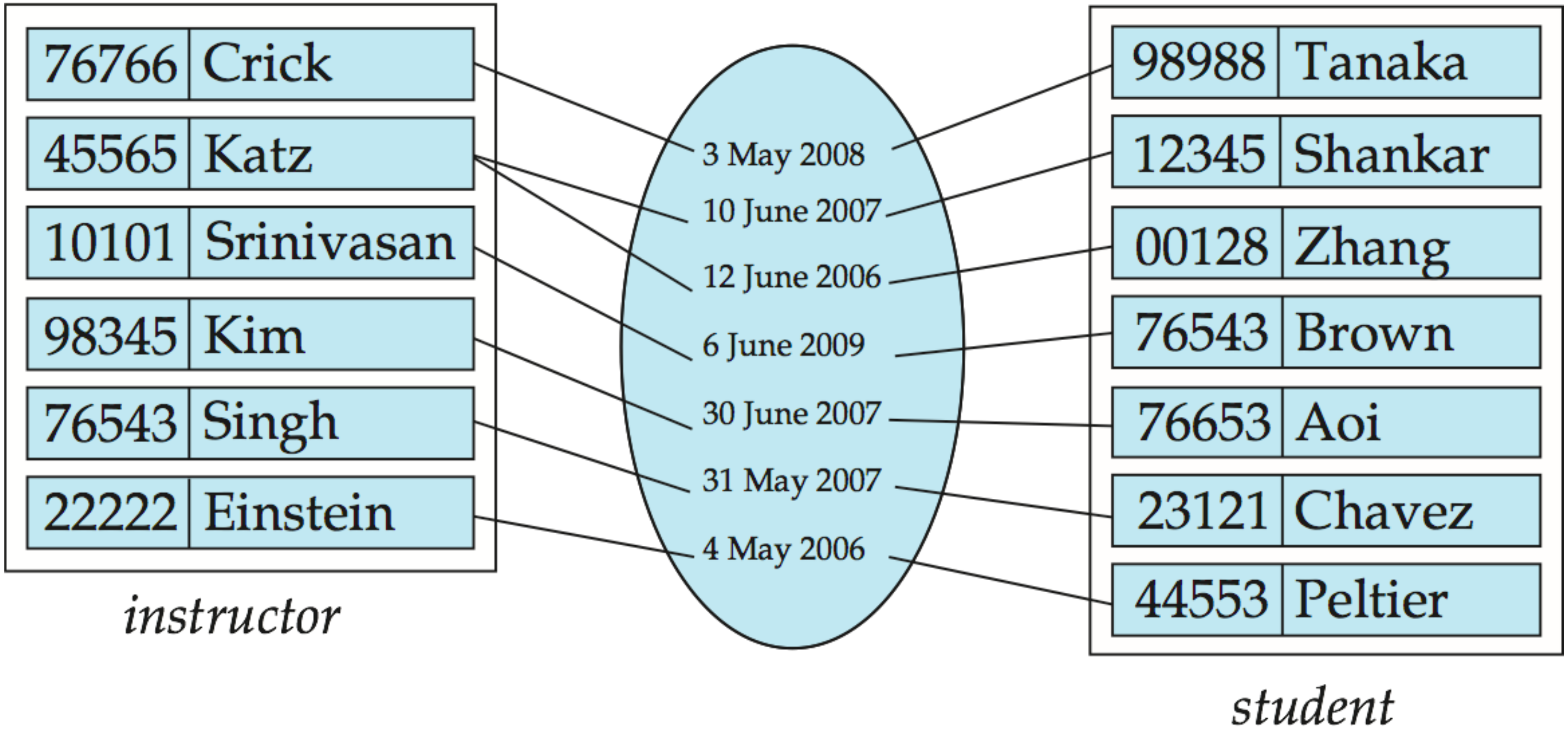
76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

Figure 7.03



76766 Crick
45565 Katz
10101 Srinivasan
98345 Kim
76543 Singh
22222 Einstein

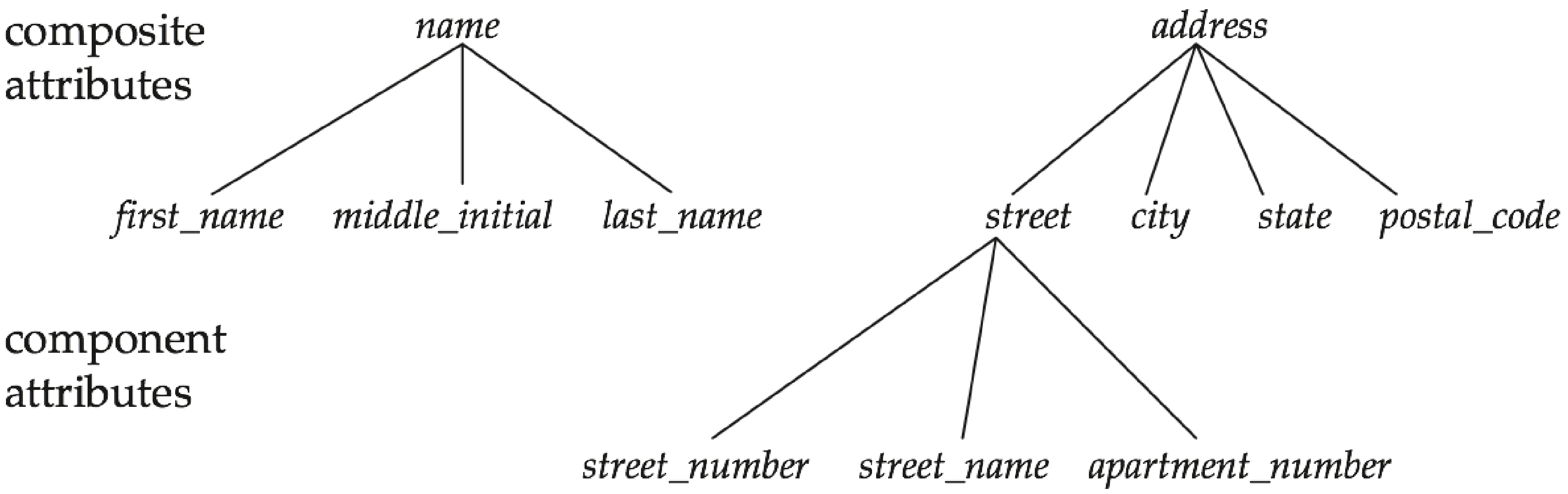
instructor

3 May 2008
10 June 2007
12 June 2006
6 June 2009
30 June 2007
31 May 2007
4 May 2006

98988 Tanaka
12345 Shankar
00128 Zhang
76543 Brown
76653 Aoi
23121 Chavez
44553 Peltier

student

Figure 7.04



composite attributes

name

first_name *middle_initial* *last_name*

address

street *city* *state* *postal_code*

component attributes

street_number *street_name* *apartment_number*

Figure 7.05

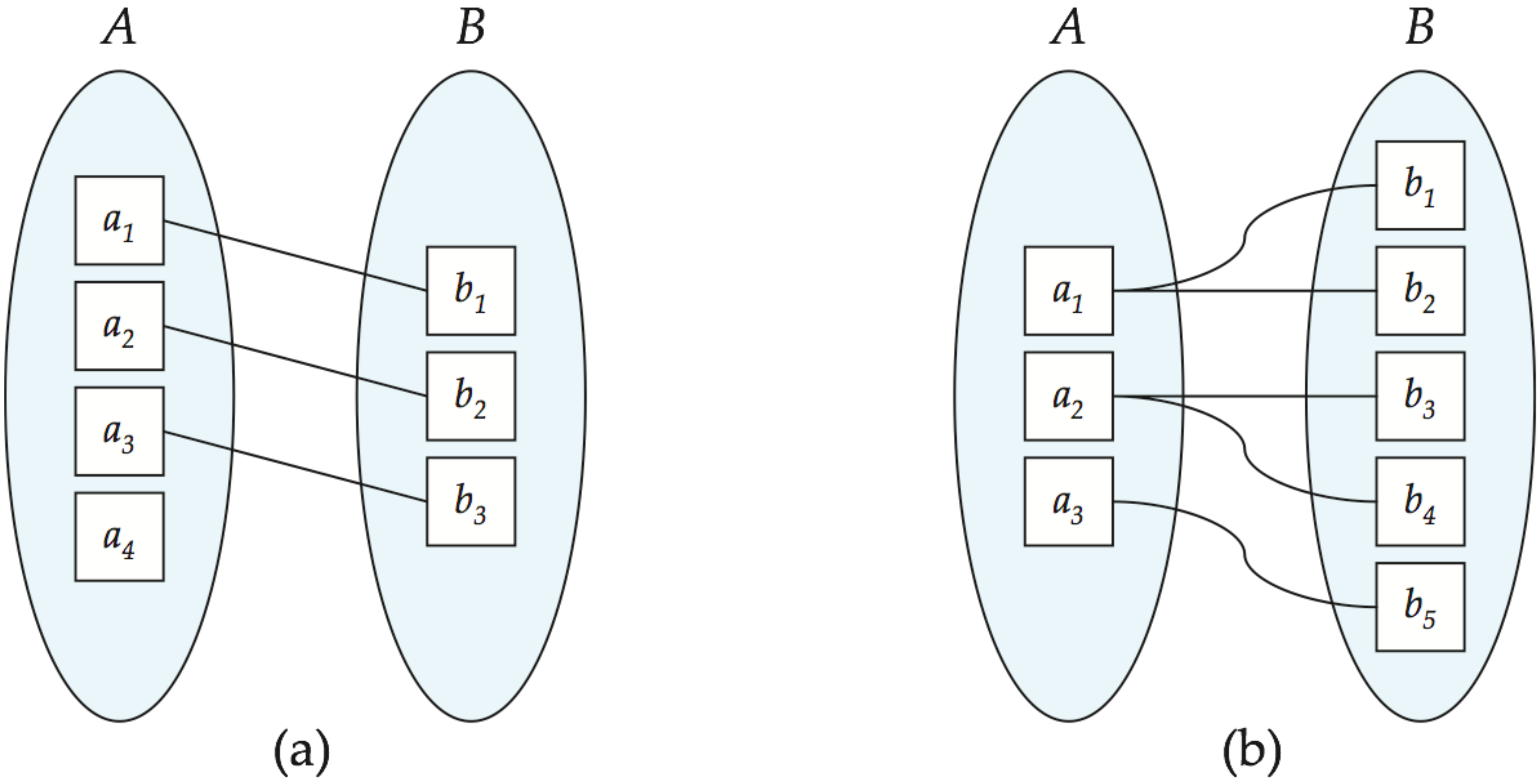
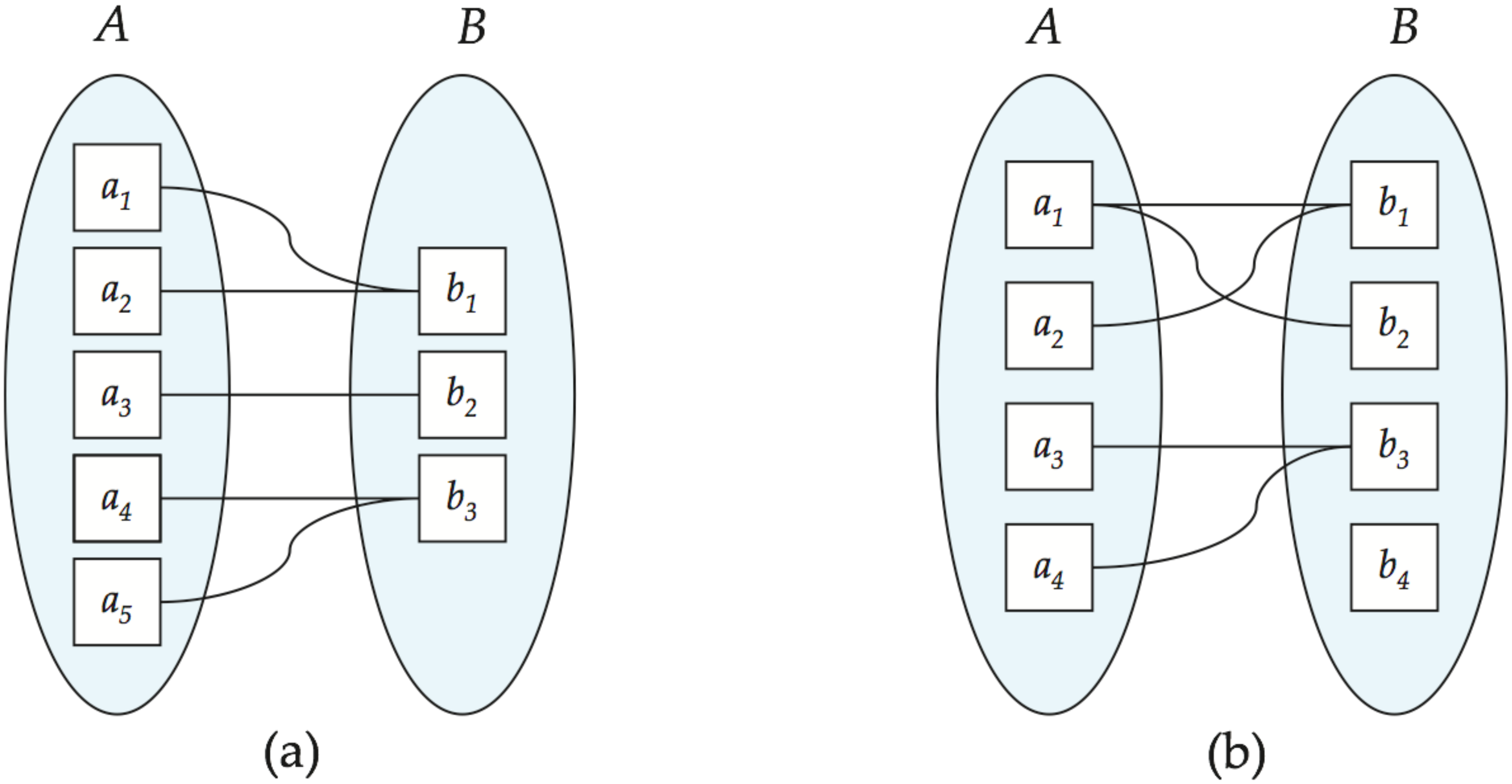


Figure 7.06



(a)

(b)

Figure 7.07

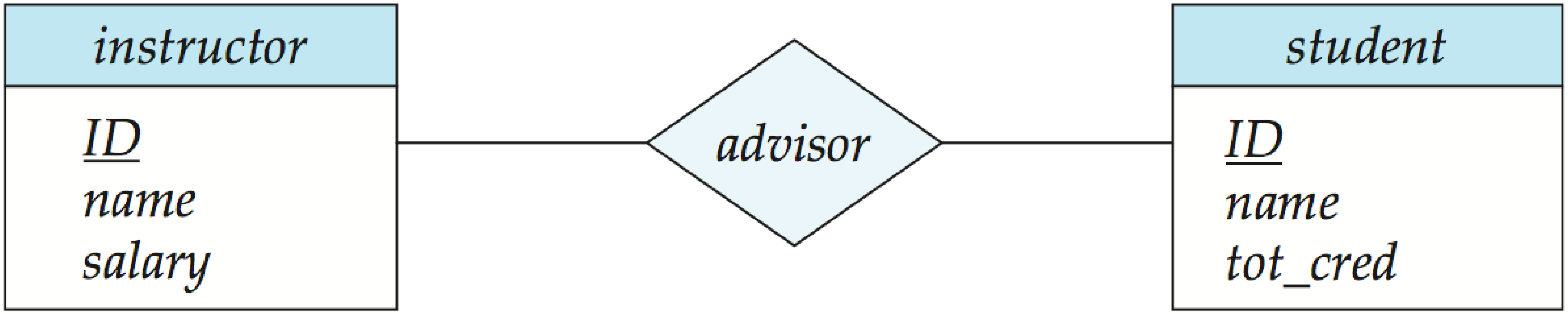


Figure 7.08

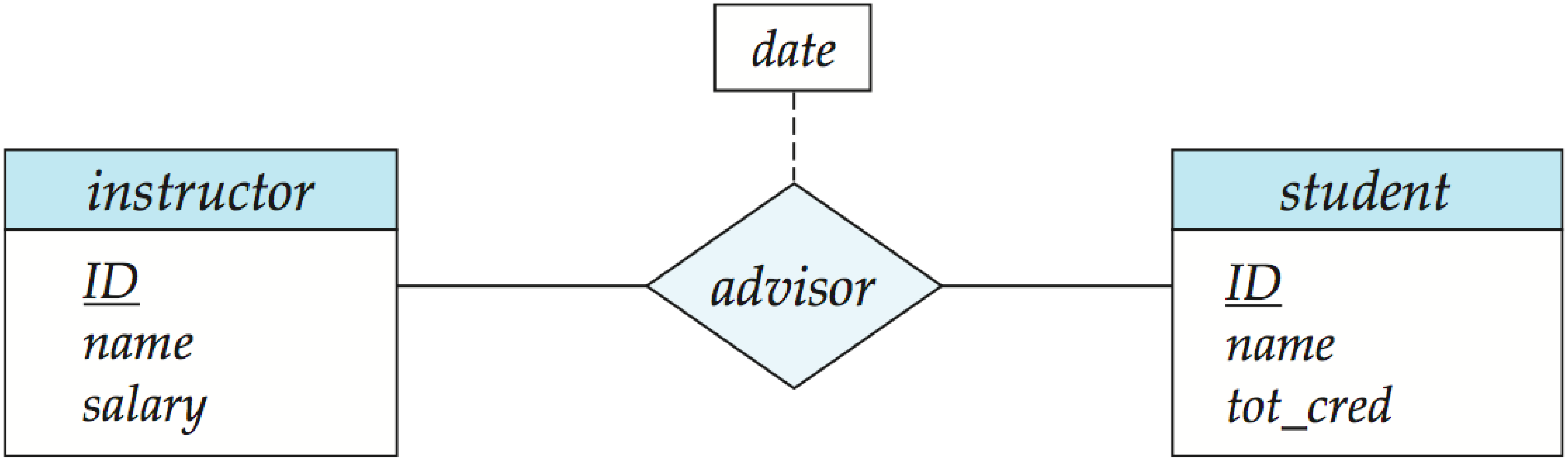
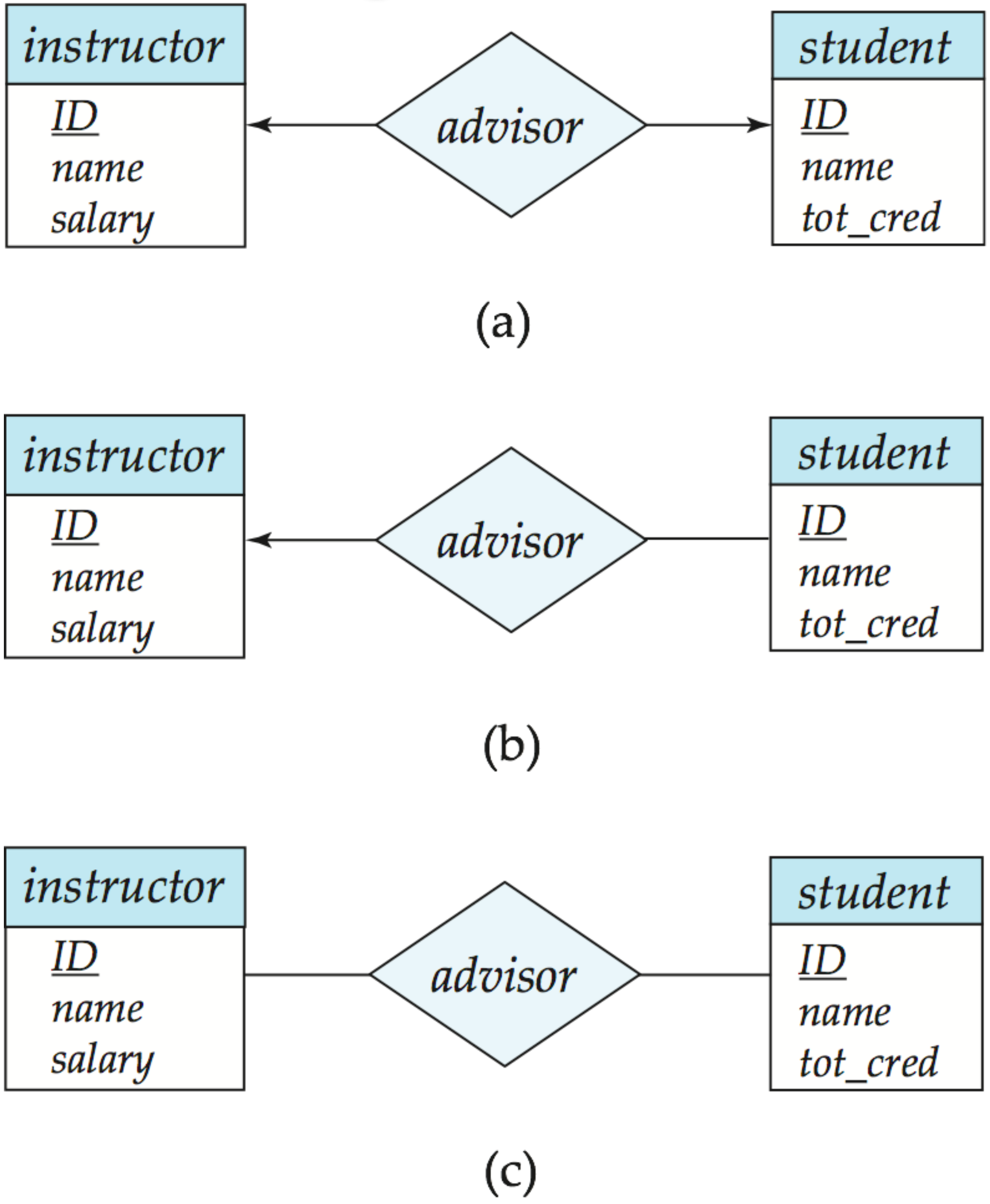

date
instructor
ID
name
salary
advisor
student
ID
name
tot_cred

Figure 7.09



(a)

(b)

(c)

Figure 7.10

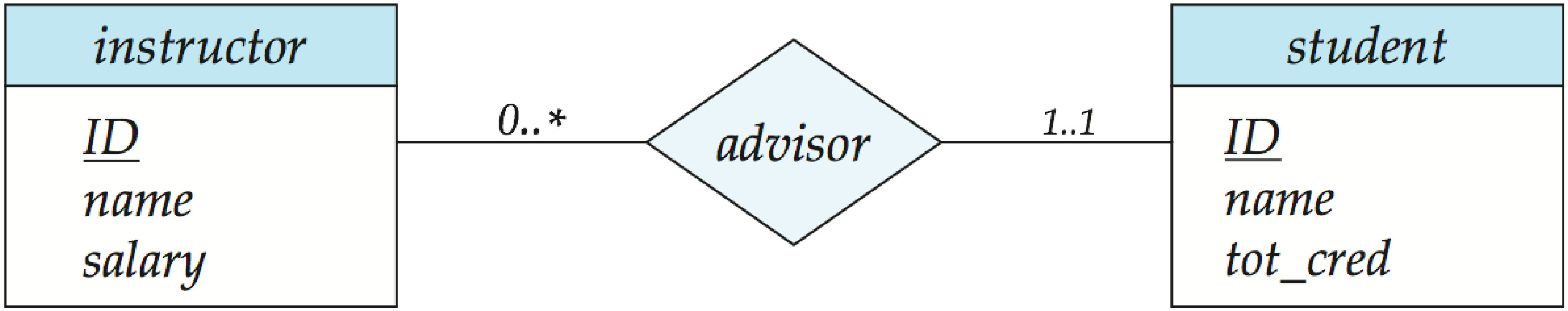
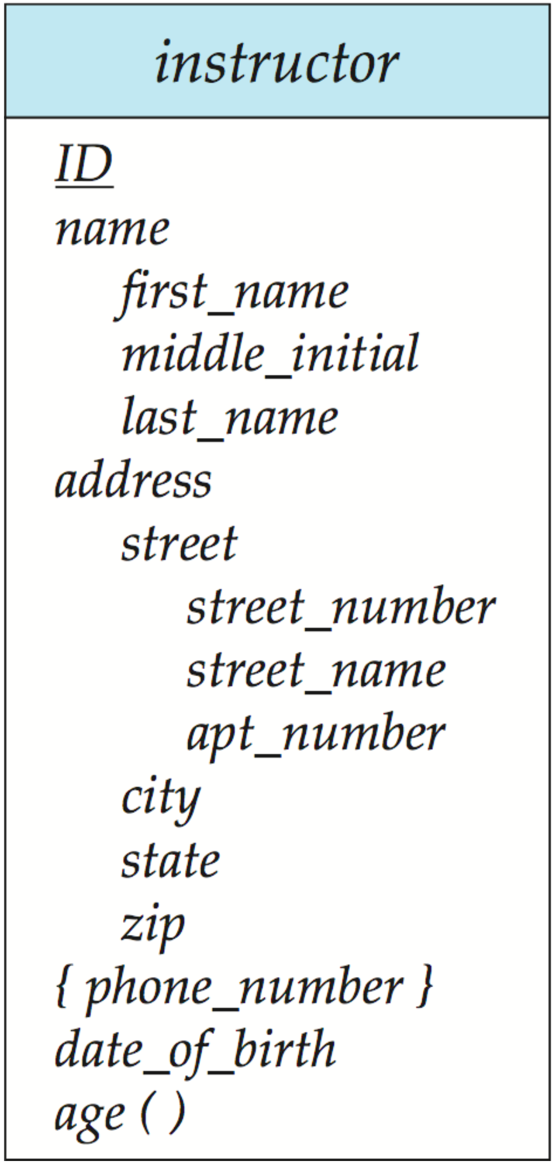

instructor
ID
name
salary
0..*
advisor
1..1
student
ID
name
tot_cred

Figure 7.11



instructor

- *ID*
- *name*
    - *first_name*
    - *middle_initial*
    - *last_name*
- *address*
    - *street*
        - *street_number*
        - *street_name*
        - *apt_number*
    - *city*
    - *state*
    - *zip*
- *{ phone_number }*
- *date_of_birth*
- *age ( )*

Figure 7.12

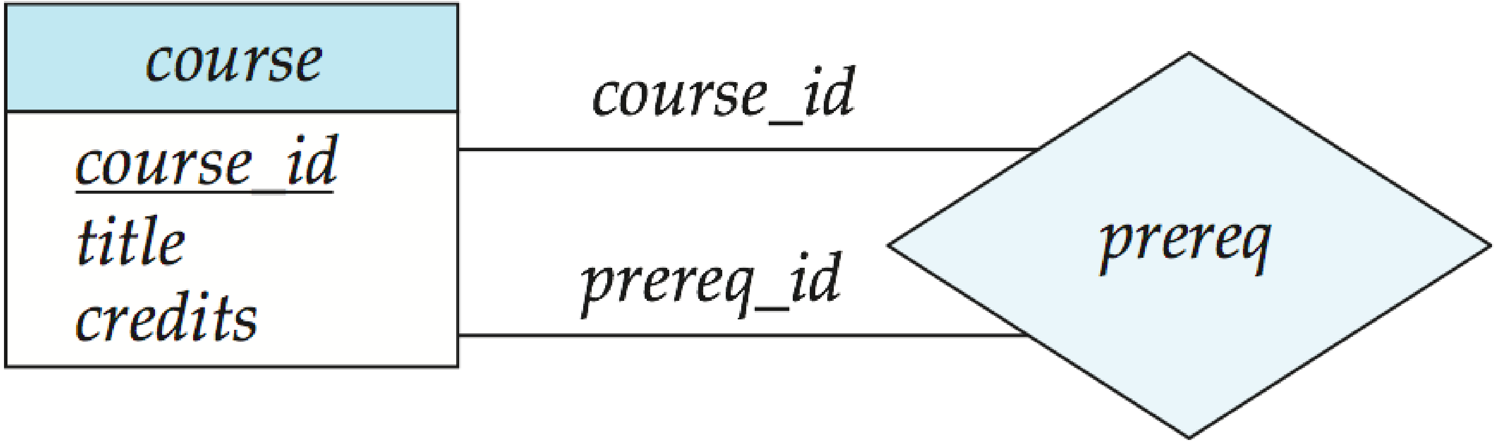


Figure 7.13

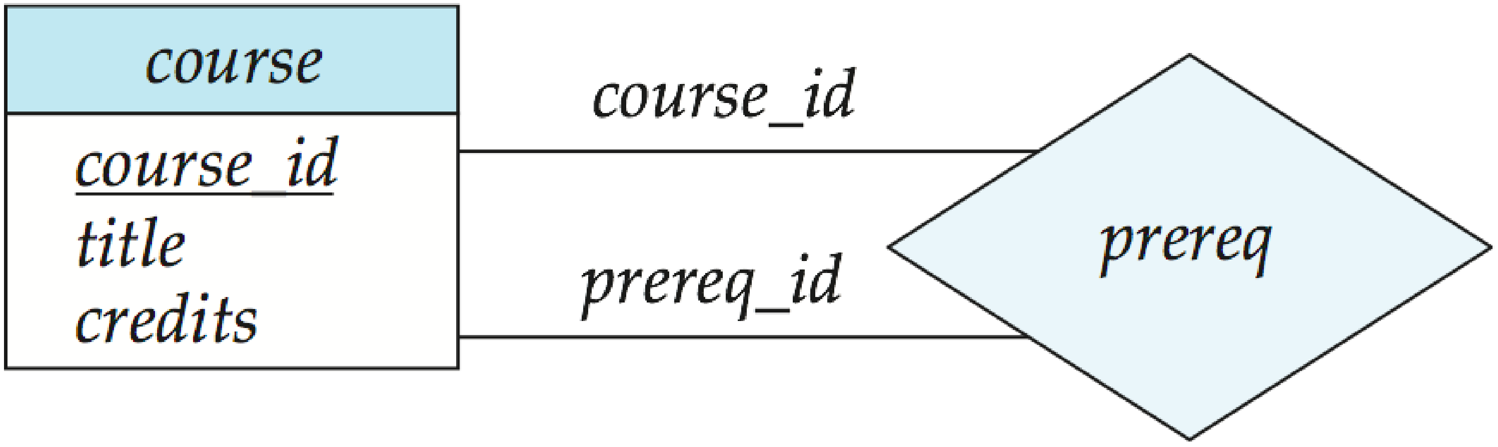


Figure 7.14

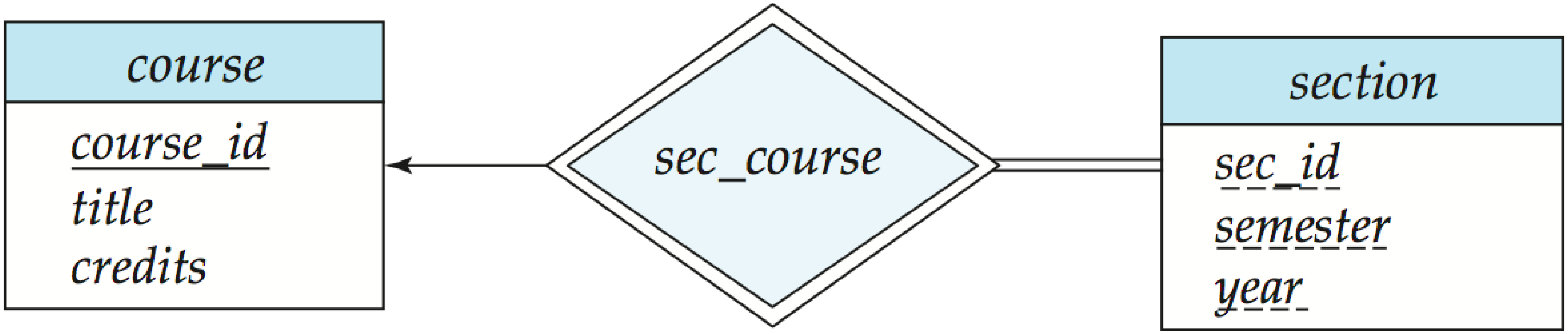

course
course_id
title
credits
sec_course
section
sec_id
semester
year

Figure 7.15

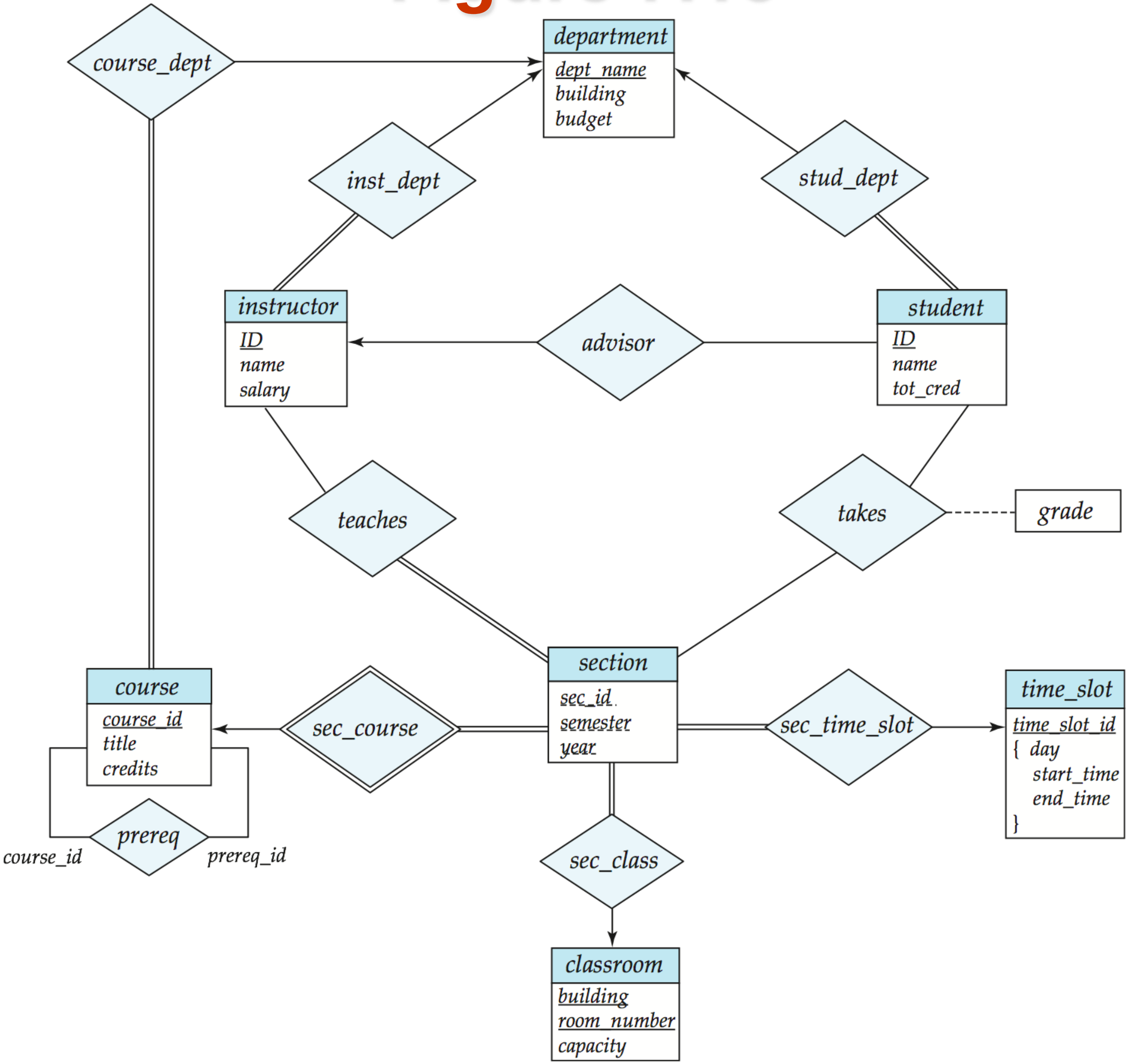
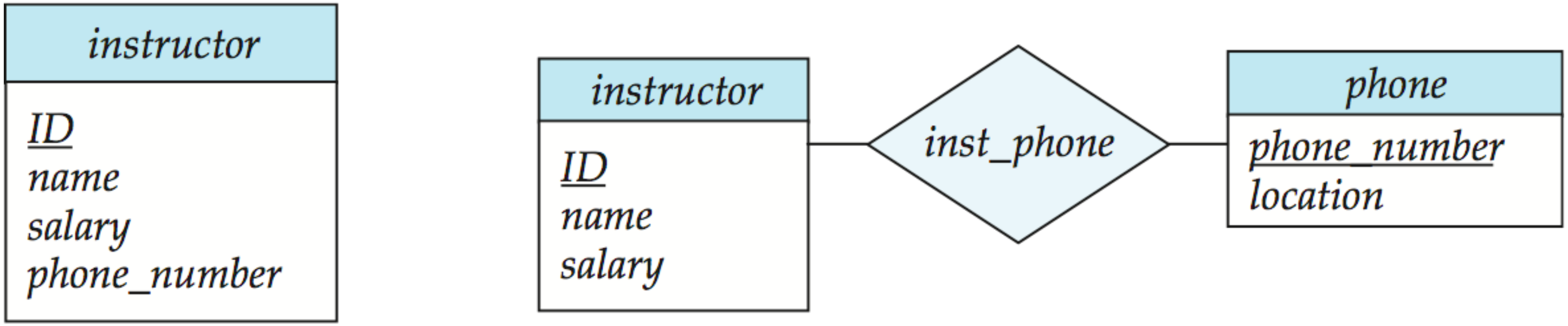

course_dept
department
dept_name
building
budget
inst_dept
stud_dept
instructor
ID
name
salary
advisor
student
ID
name
tot_cred
teaches
takes
grade
section
sec_id
semester
year
course
course_id
title
credits
sec_course
sec_time_slot
time_slot
time_slot_id
{ day
start_time
end_time
}
prereq
course_id
prereq_id
sec_class
classroom
building
room_number
capacity

Figure 7.17



(a)

(b)

Figure 7.18

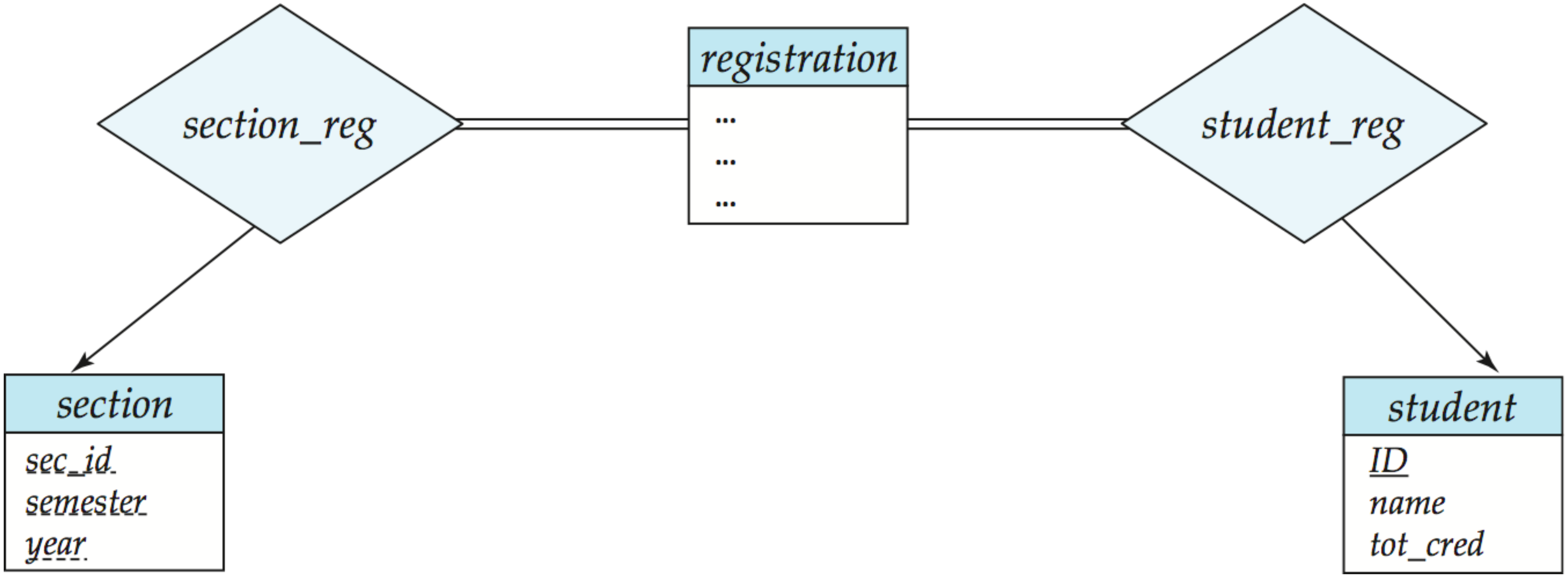

section_reg
registration
...
...
...
student_reg
section
sec_id
semester
year
student
ID
name
tot_cred

Figure 7.19

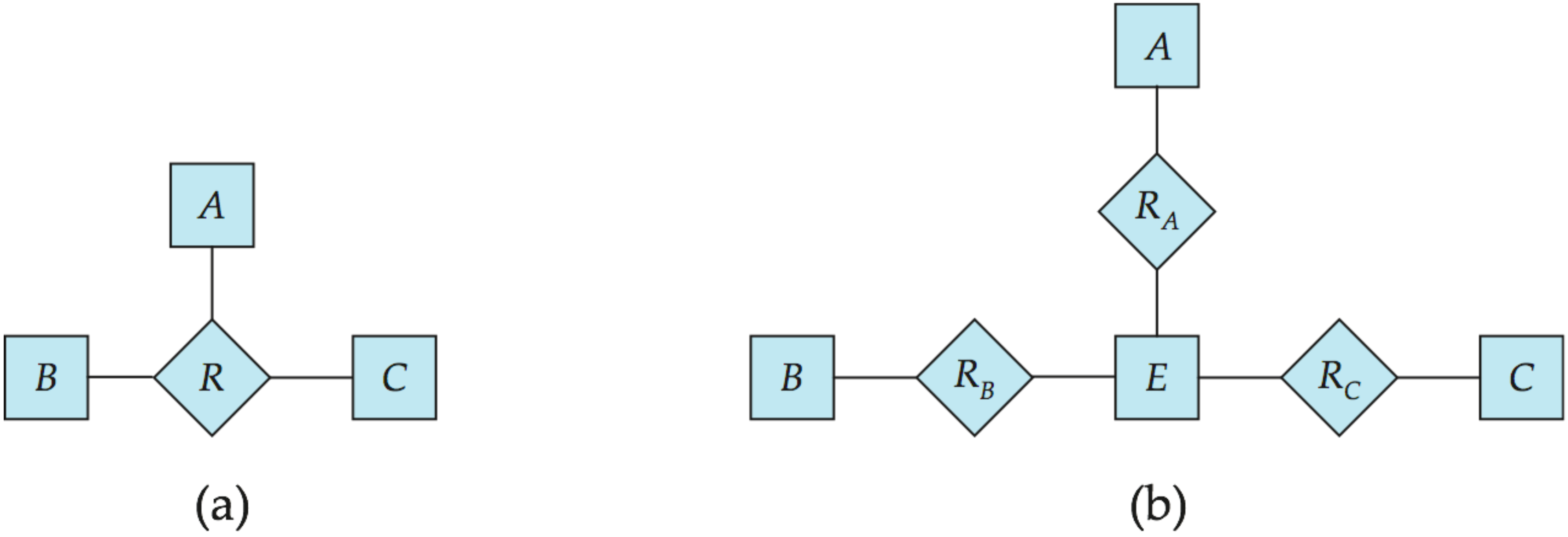
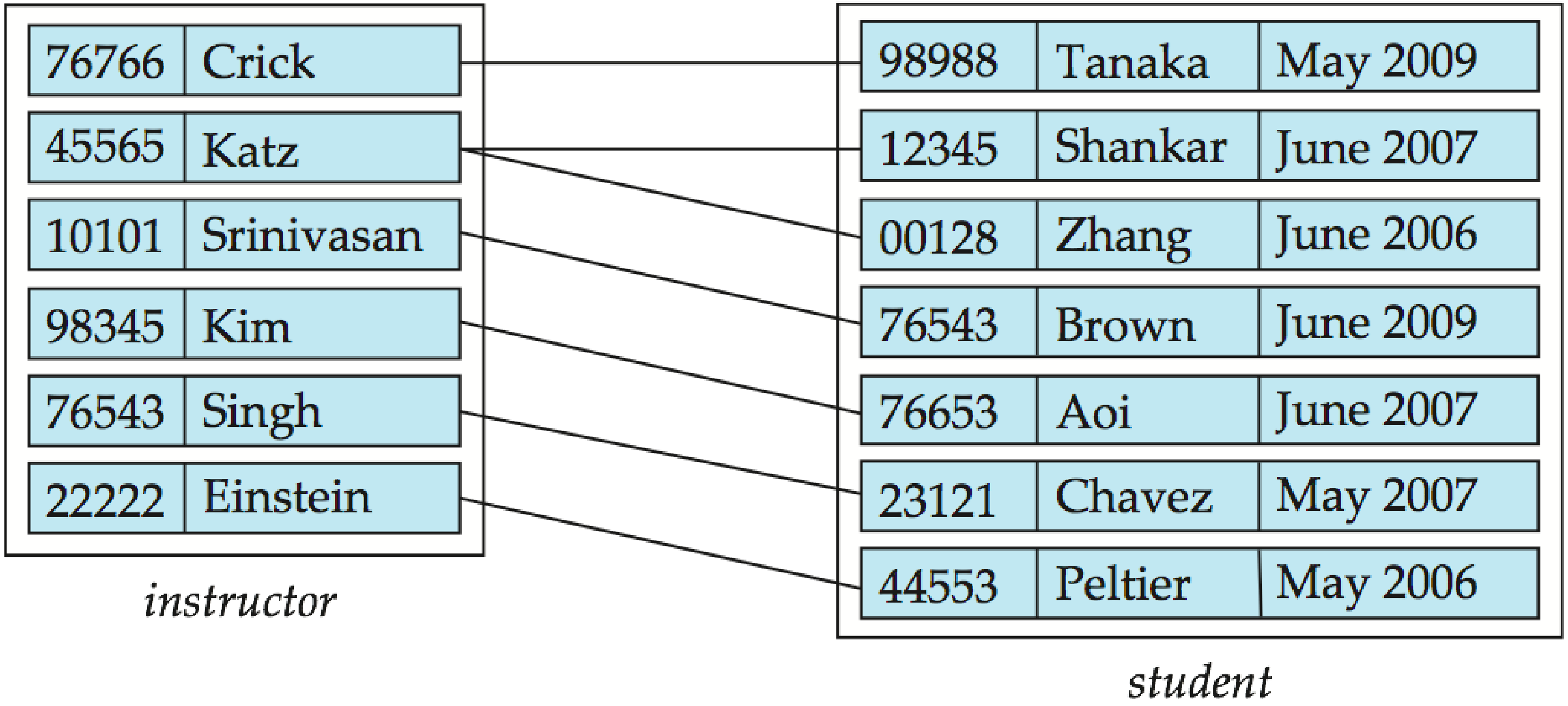


Figure 7.20



76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

98988	Tanaka	May 2009
12345	Shankar	June 2007
00128	Zhang	June 2006
76543	Brown	June 2009
76653	Aoi	June 2007
23121	Chavez	May 2007
44553	Peltier	May 2006

student

Figure 7.21

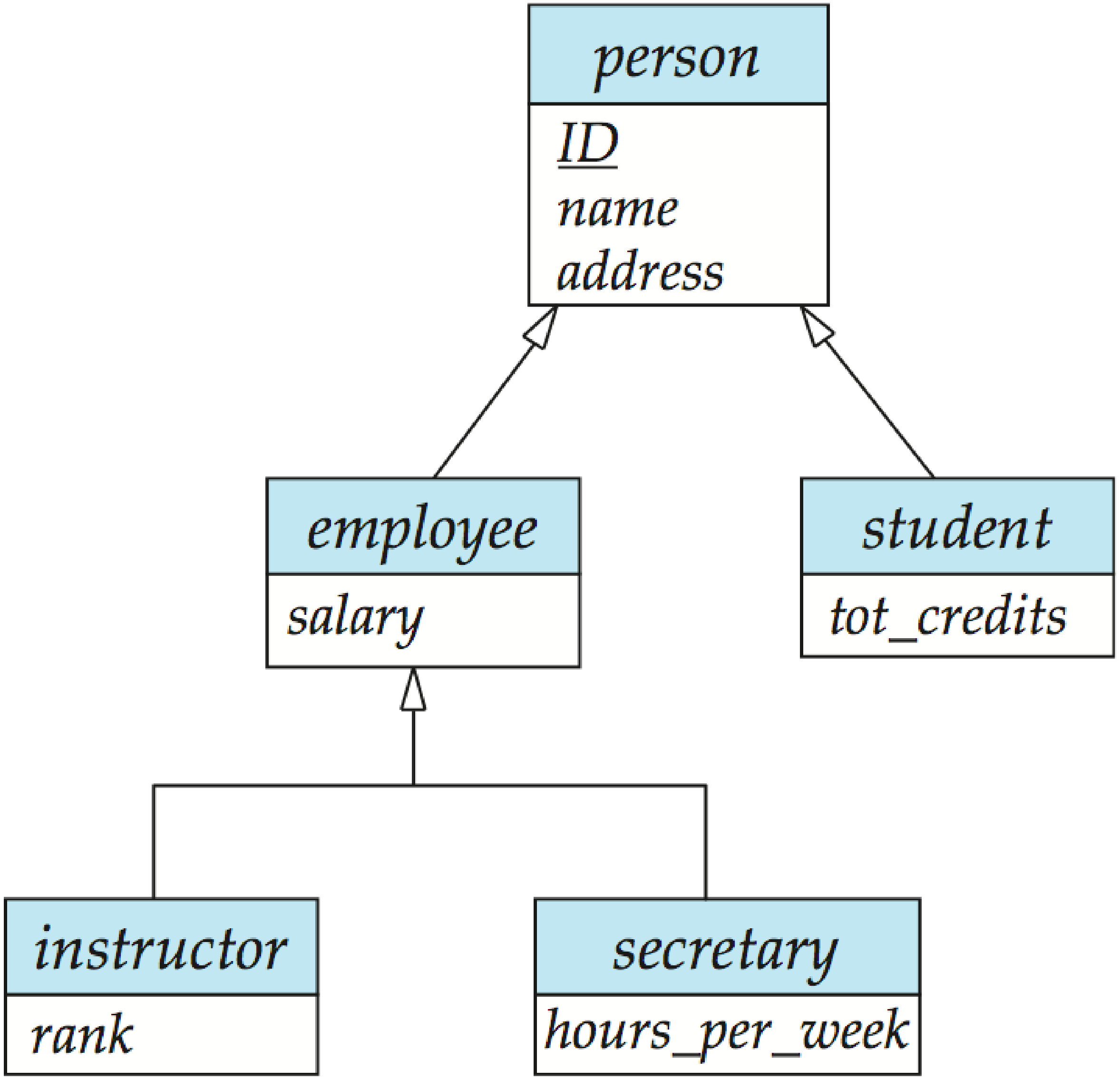

person
ID
name
address
employee
salary
student
tot_credits
instructor
rank
secretary
hours_per_week

Figure 7.22

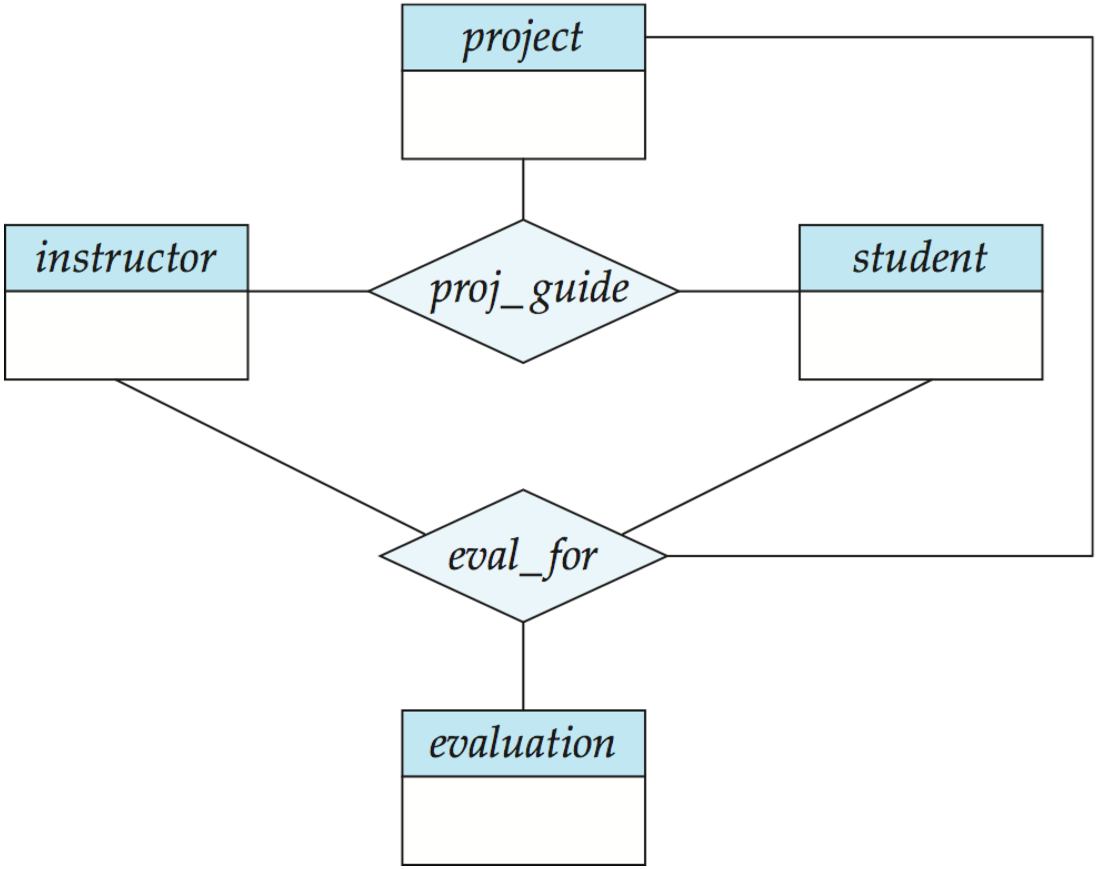

project
instructor
proj_guide
student
eval_for
evaluation

Figure 7.23

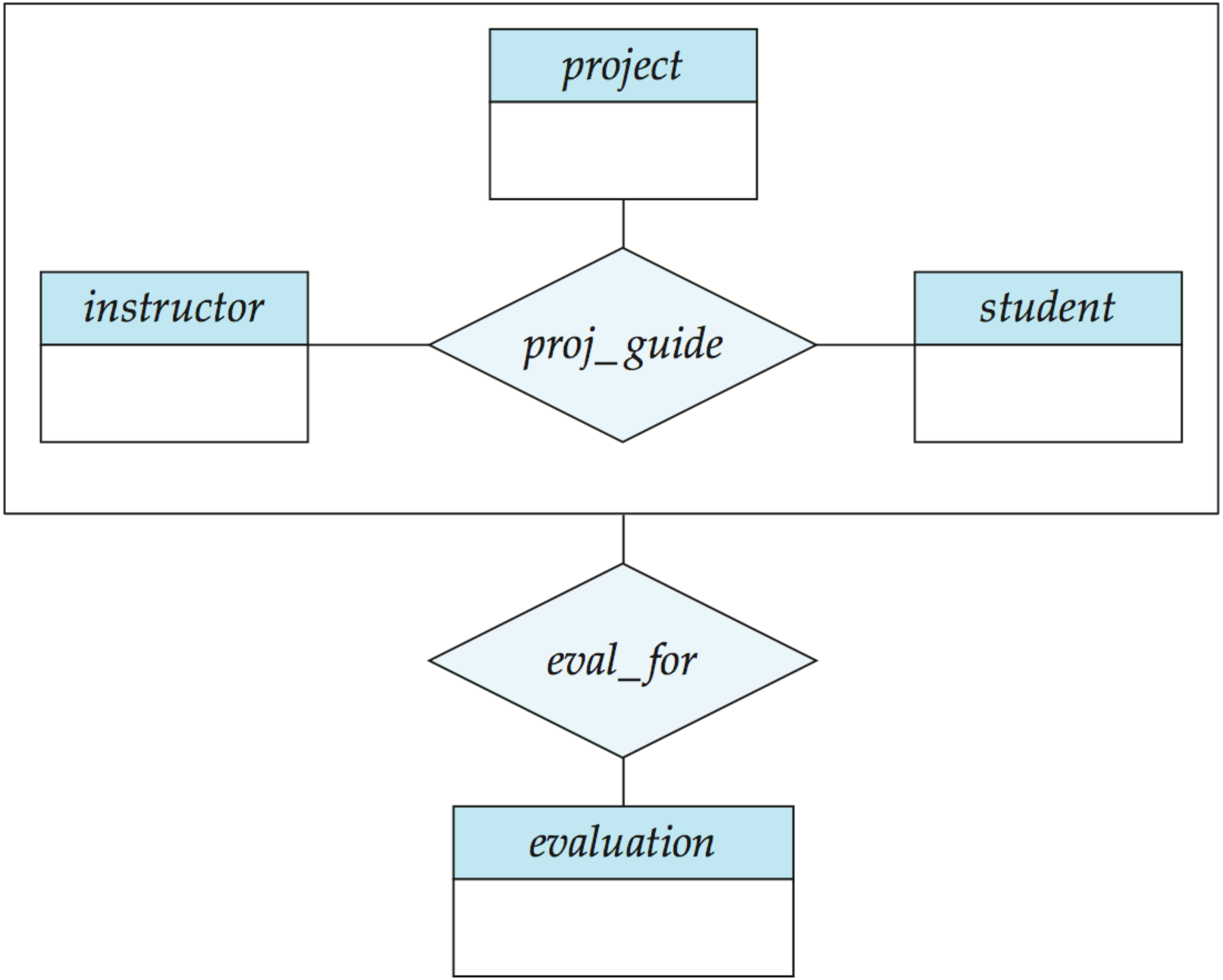
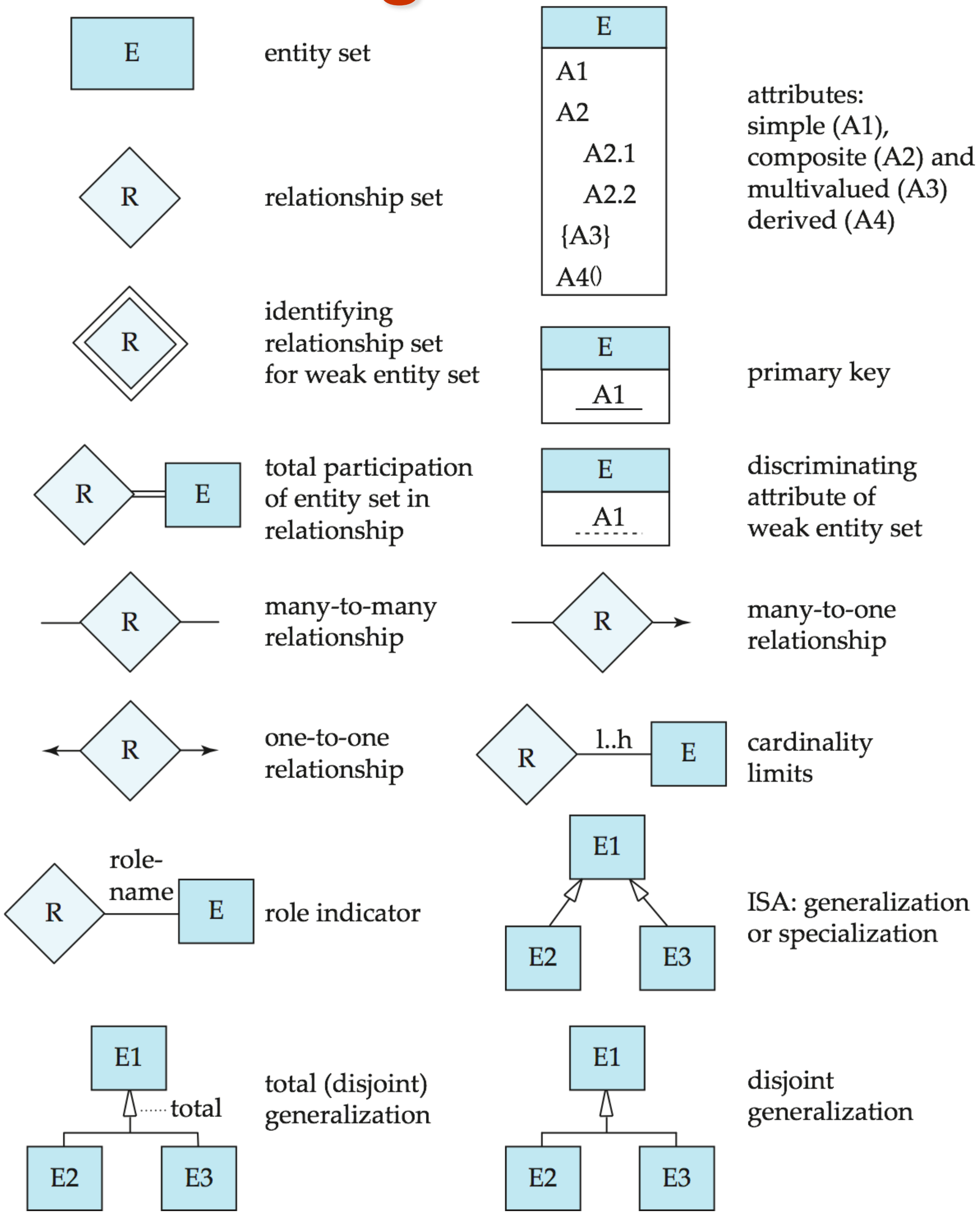

project
instructor
proj_guide
student
eval_for
evaluation

Figure 7.24



entity set

relationship set

identifying
relationship set
for weak entity set

total participation
of entity set in
relationship

many-to-many
relationship

one-to-one
relationship

role indicator

total (disjoint)
generalization

attributes:
simple (A1),
composite (A2) and
multivalued (A3)
derived (A4)

primary key

discriminating
attribute of
weak entity set

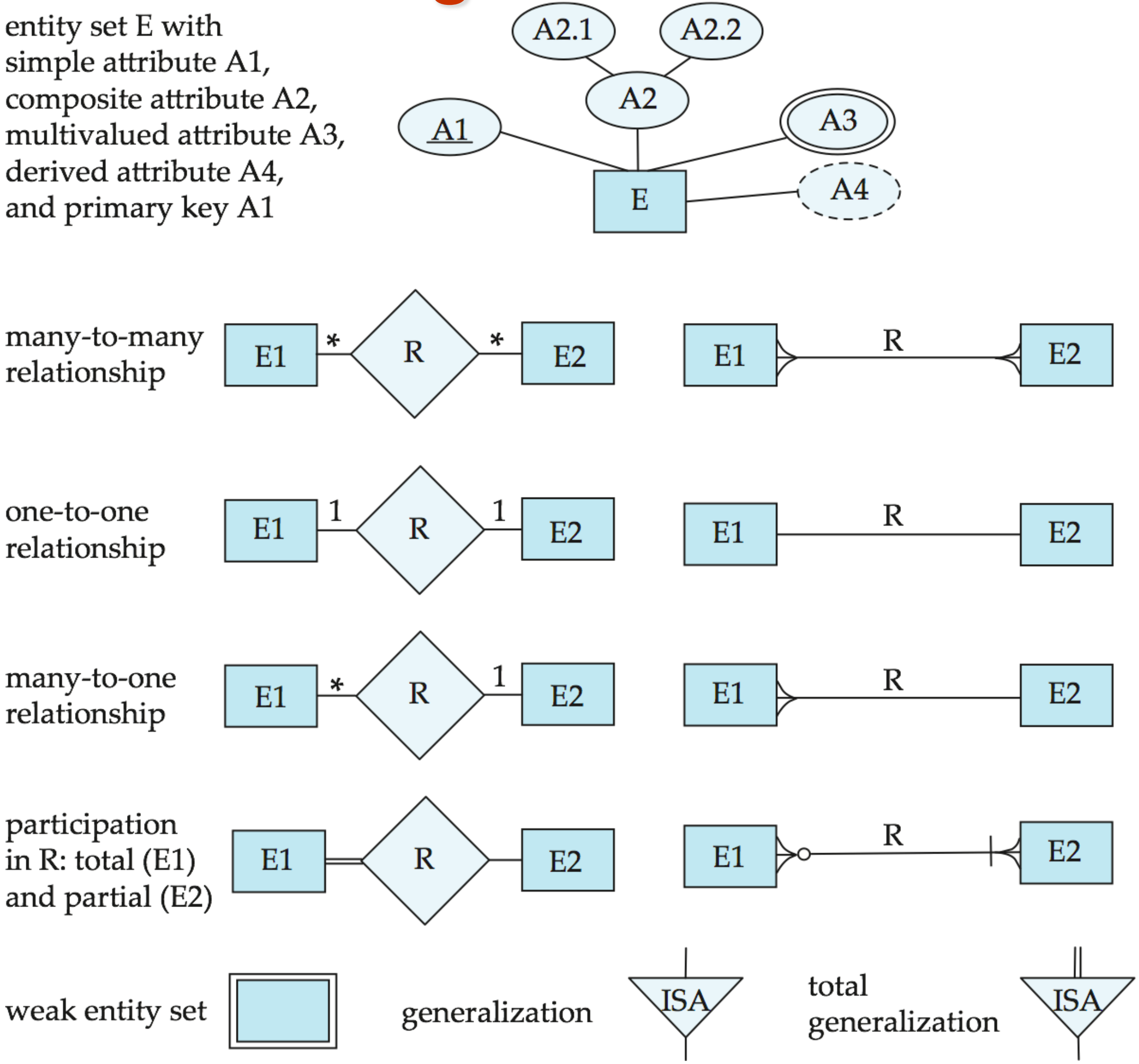
many-to-one
relationship

cardinality
limits

ISA: generalization
or specialization

disjoint
generalization

Figure 7.25



entity set E with
simple attribute A1,
composite attribute A2,
multivalued attribute A3,
derived attribute A4,
and primary key A1

many-to-many relationship

one-to-one relationship

many-to-one relationship

participation in R: total (E1) and partial (E2)

weak entity set

generalization

total generalization

Figure 7.26

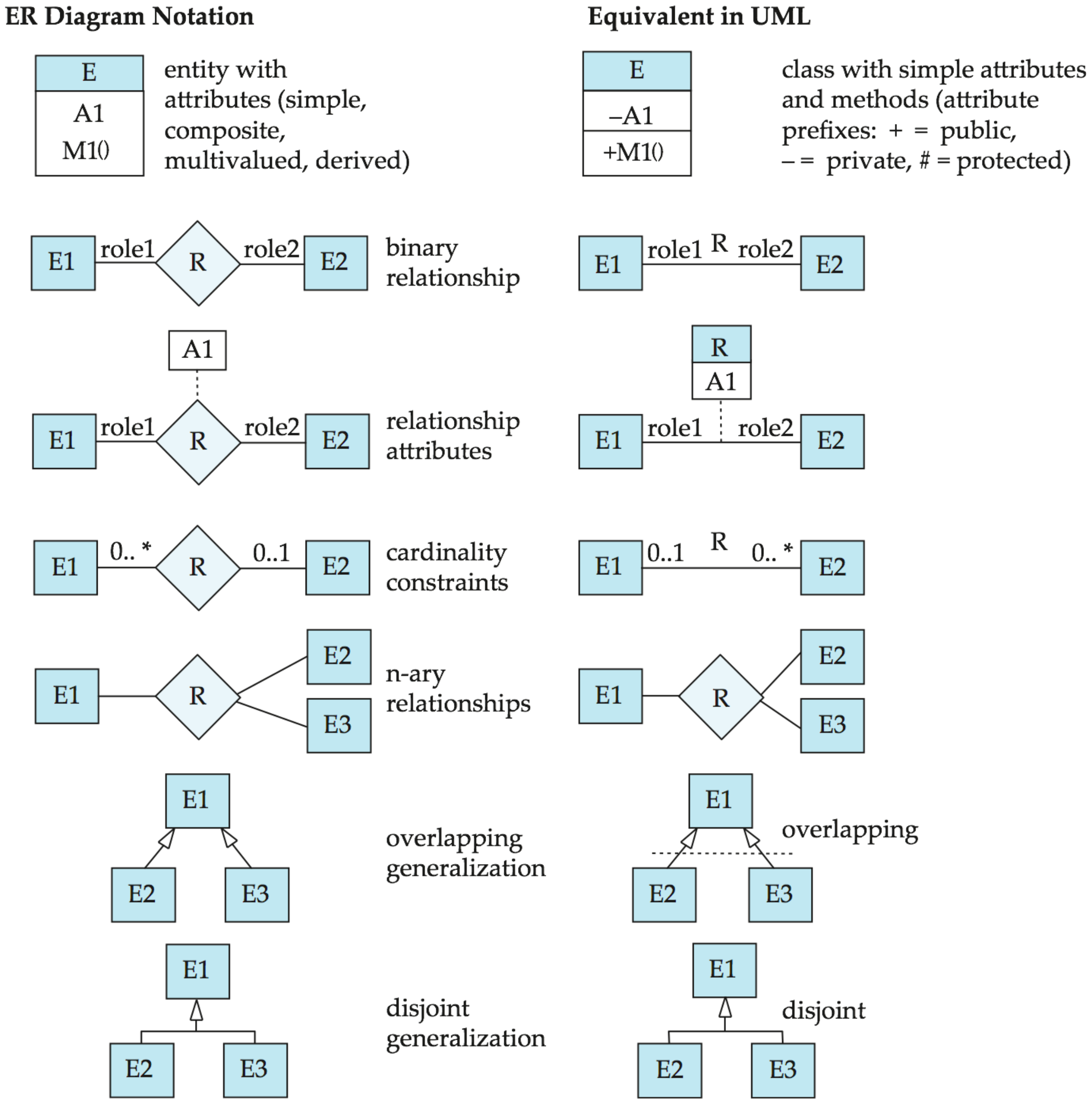
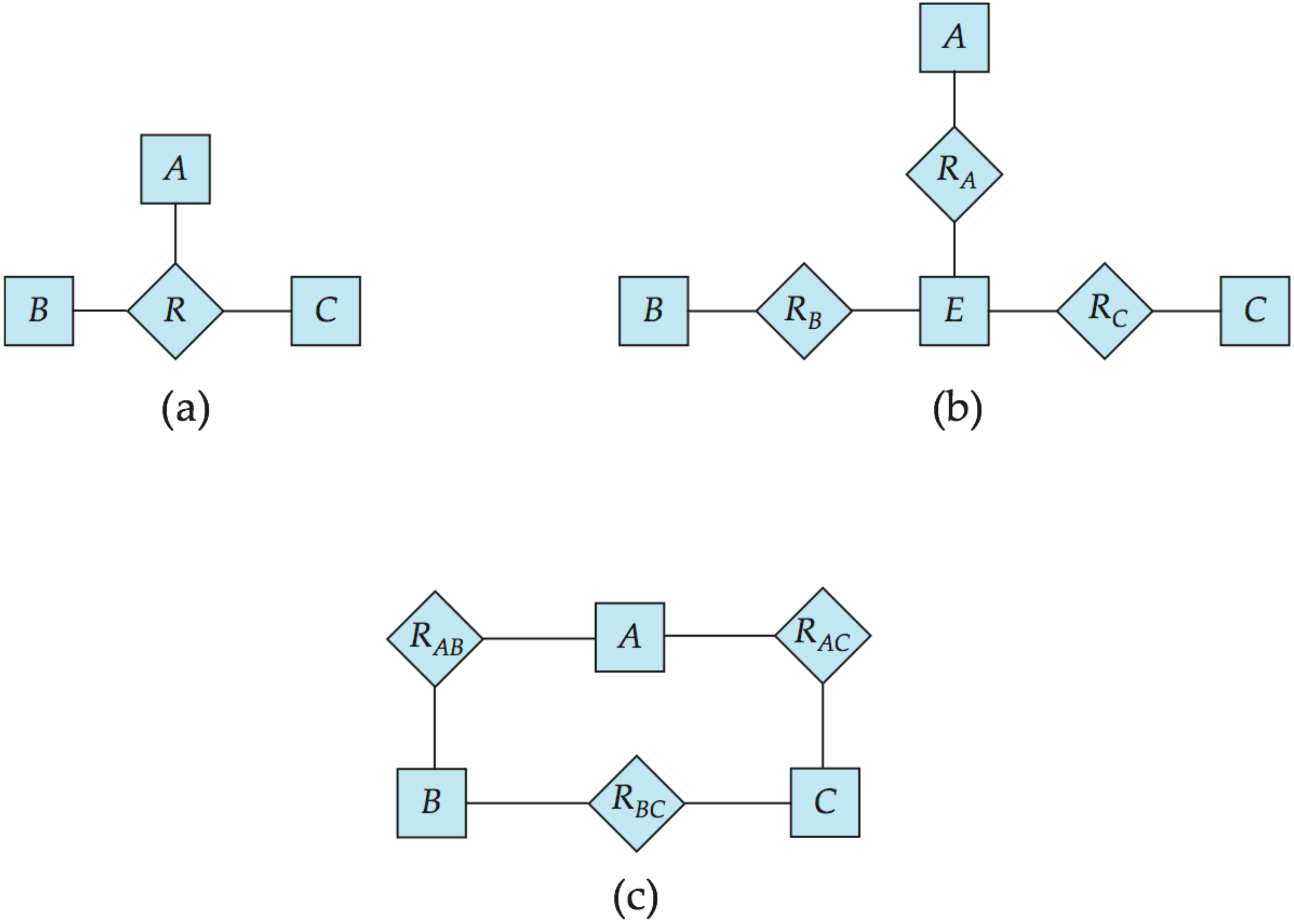

ER Diagram Notation
Equivalent in UML
E
A1
M1()
entity with attributes (simple, composite, multivalued, derived)
E
–A1
+M1()
class with simple attributes and methods (attribute prefixes: + = public, – = private, # = protected)
E1
role1
R
role2
E2
binary relationship
E1
role1
R
role2
E2
A1
E1
role1
R
role2
E2
relationship attributes
R
A1
E1
role1
role2
E2
E1
0.. *
R
0..1
E2
cardinality constraints
E1
0..1
R
0.. *
E2
E1
R
E2
E3
n-ary relationships
E1
R
E2
E3
E1
E2
E3
overlapping generalization
E1
E2
E3
overlapping
E1
E2
E3
disjoint generalization
E1
E2
E3
disjoint

Figure 7.27



(a)

(b)

(c)

Figure 7.28

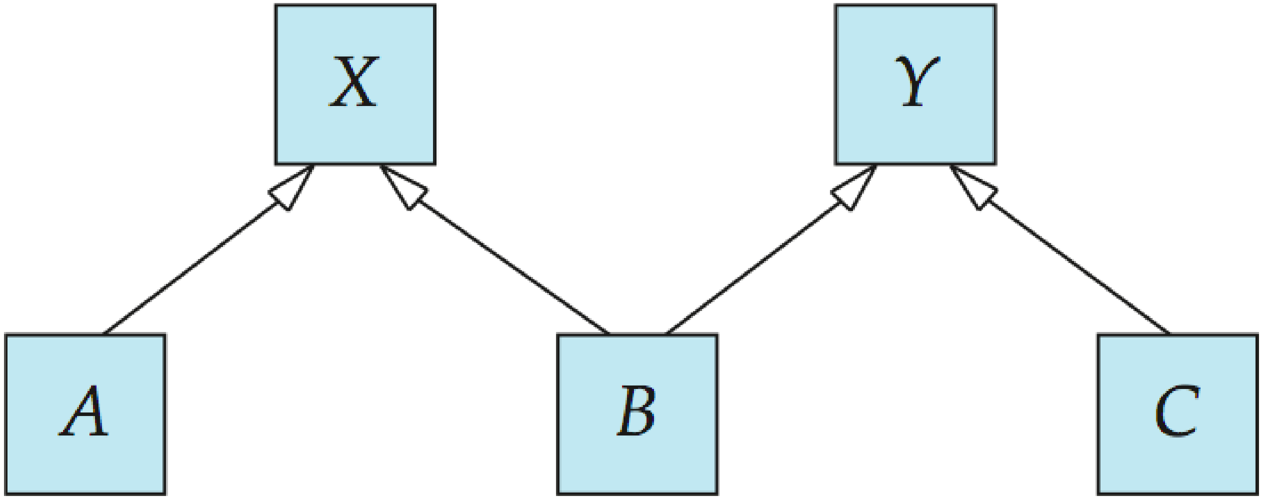
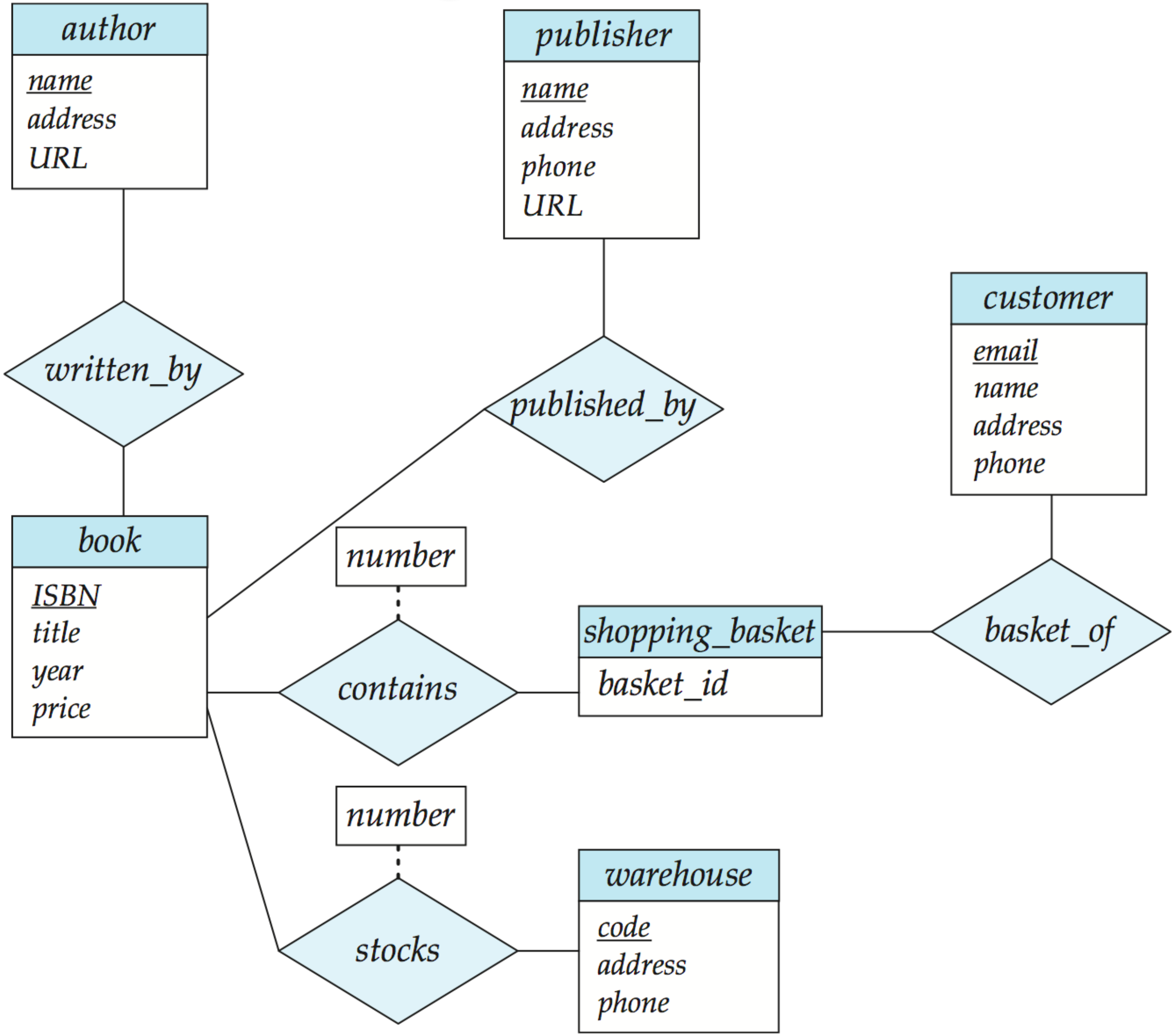


Figure 7.29


author
name
address
URL
publisher
name
address
phone
URL
written_by
published_by
customer
email
name
address
phone
book
ISBN
title
year
price
number
contains
shopping_basket
basket_id
basket_of
number
stocks
warehouse
code
address
phone

Logical Database Design - Mapping ERD to Relational

Objective

To be able to:

- Transform an E-R diagram to a logically equivalent set of relations

Topics

- Transforming ER Diagrams into Relations

Transforming ERD into Relations

Transforming (mapping) E-R diagrams to relations is a relatively straightforward process with a well-defined set of rule

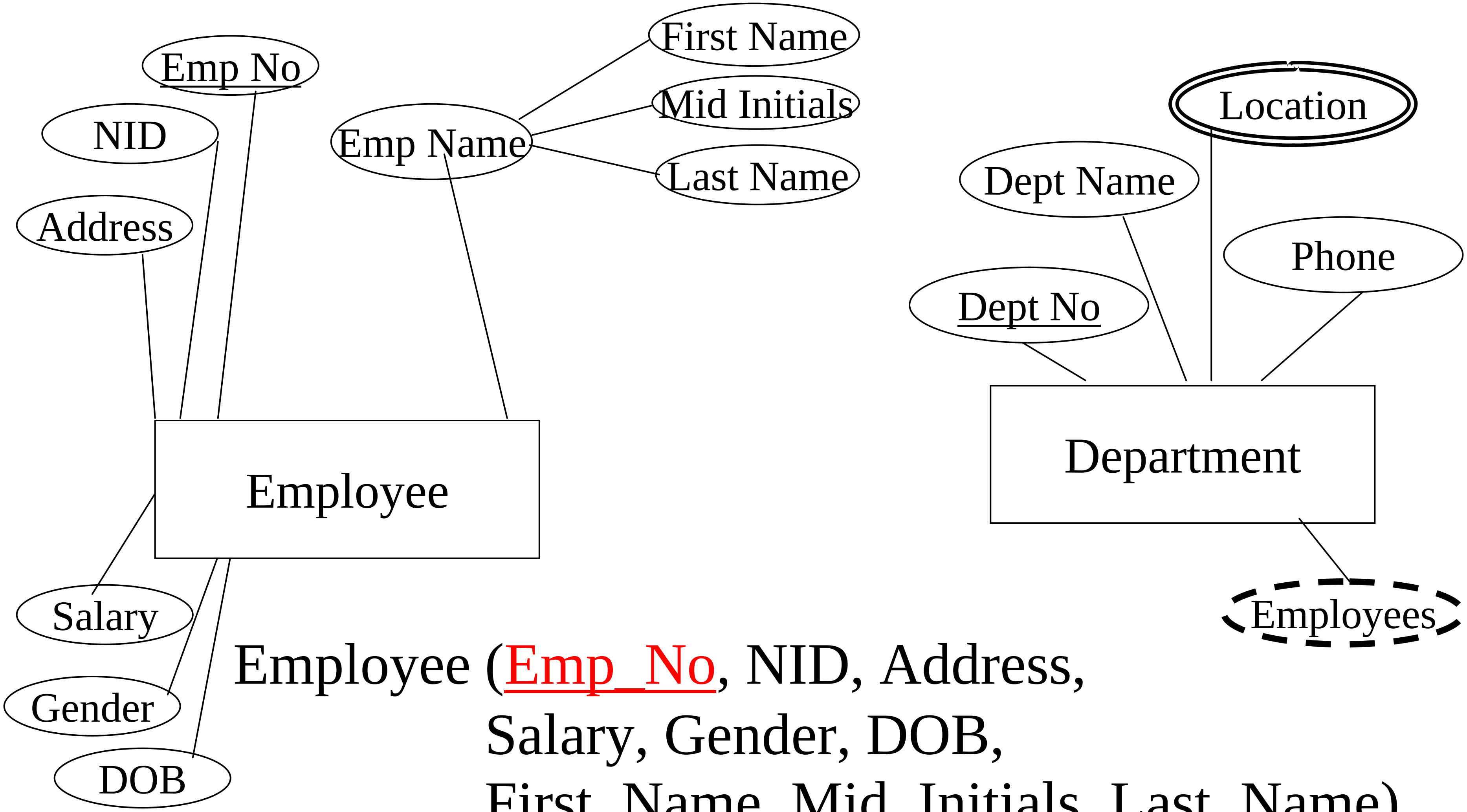
Steps:

1: Map Regular Entities
2: Map Weak Entities
3: Map Binary Relationships
4: Map Unary Relationships
5: Map EER

1. Map Regular Entities

Each regular entity type in an ERD is transformed into a relation.

Entity	**Relation**
entity name	relation name
simple attribute	attribute of the relation
entity identifier	primary key of relation
composite attribute	component attributes
multi-valued attribute	new relation with PK



Employee (Emp_No, NID, Address,
Salary, Gender, DOB,
First_Name, Mid_Initials, Last_Name)

Department(Dept_No, Dept_Name, Phone)

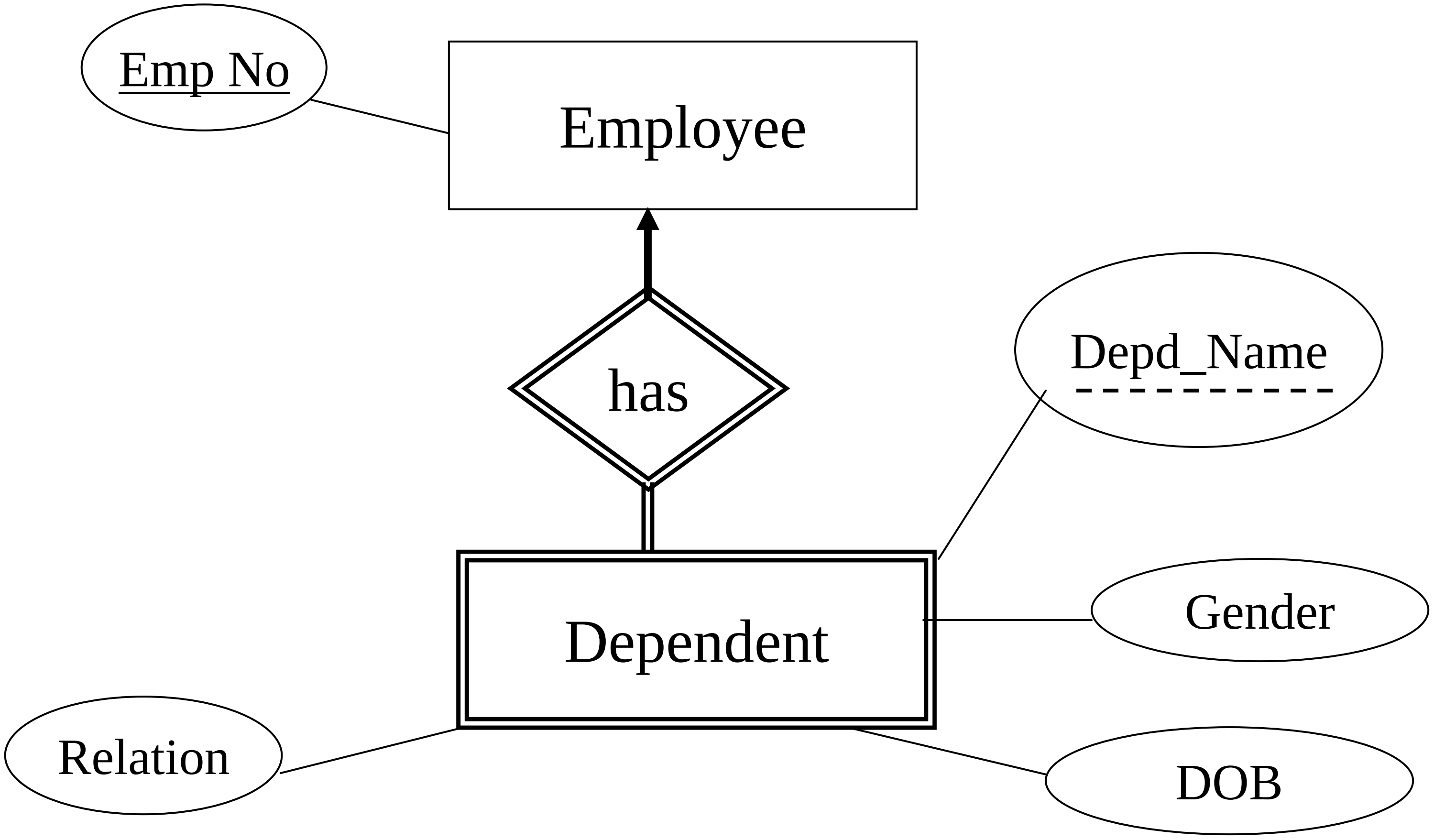
FK

Dept_Location(*Dept_No*, Location)

2. Map Weak Entities

Each weak entity type in an ERD is transformed into a relation.

Entity	**Relation**
entity name	relation name
simple attribute	attribute of the relation
owner entity identifier	foreign key attribute
entity identifier (partial)	composite key together with PK of owner as FK
composite attribute	component attributes
multi-valued attribute	new relation with PK



Dependent(*Emp_No*, Depd_Name, Gender, DOB, Relation)

FK

Employee (Emp_No, ….)

3. Map Binary Relationships

Procedure depends on both the degree of the binary relationships and the cardinalities of the relationships

- Map Binary One-to-Many Relationships
- Map Binary Many-to-Many Relationships
- Map Binary One-to-One Relationships

Map Binary One-to-Many Relationships

Create a relation for the two entity types participating in the relationships (step 1)

include PK of the entity in the one-side of the relationship as a foreign key in the relation of the many side of the relationship

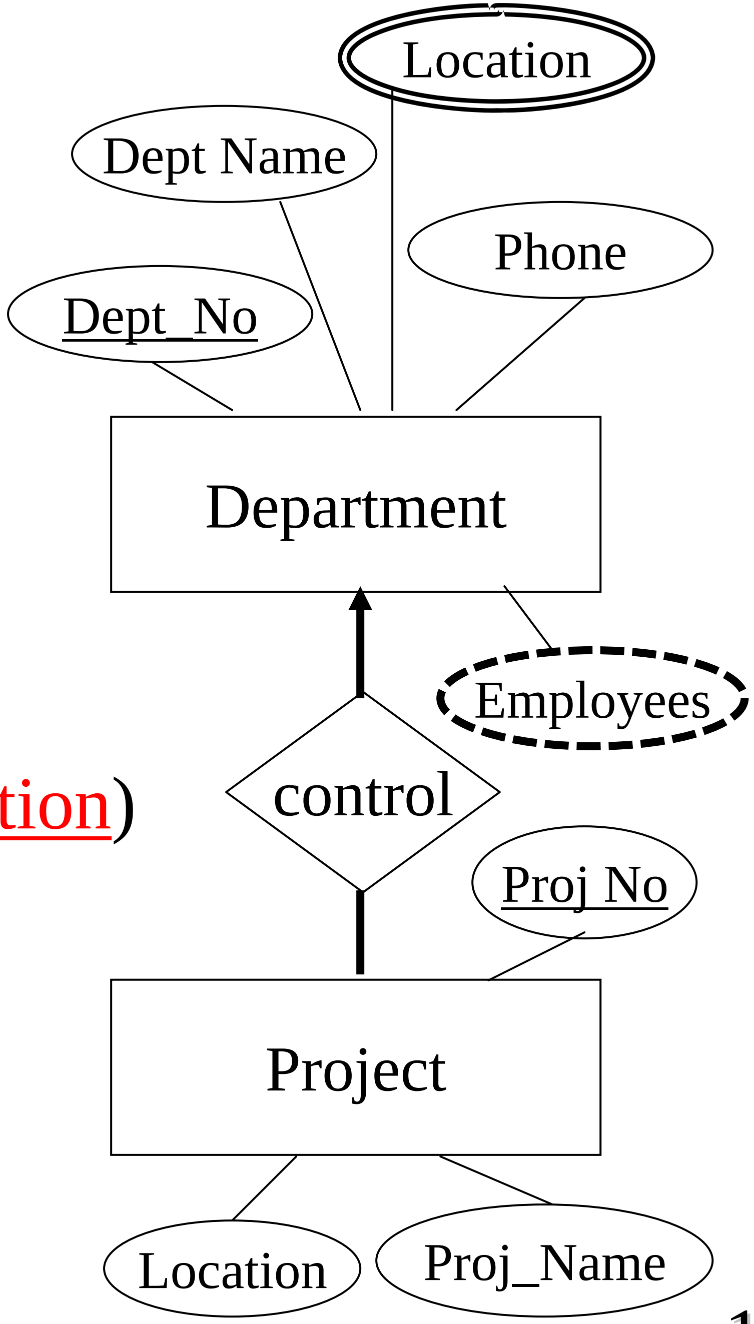
include any attributes of the relationship to the relation of the many side

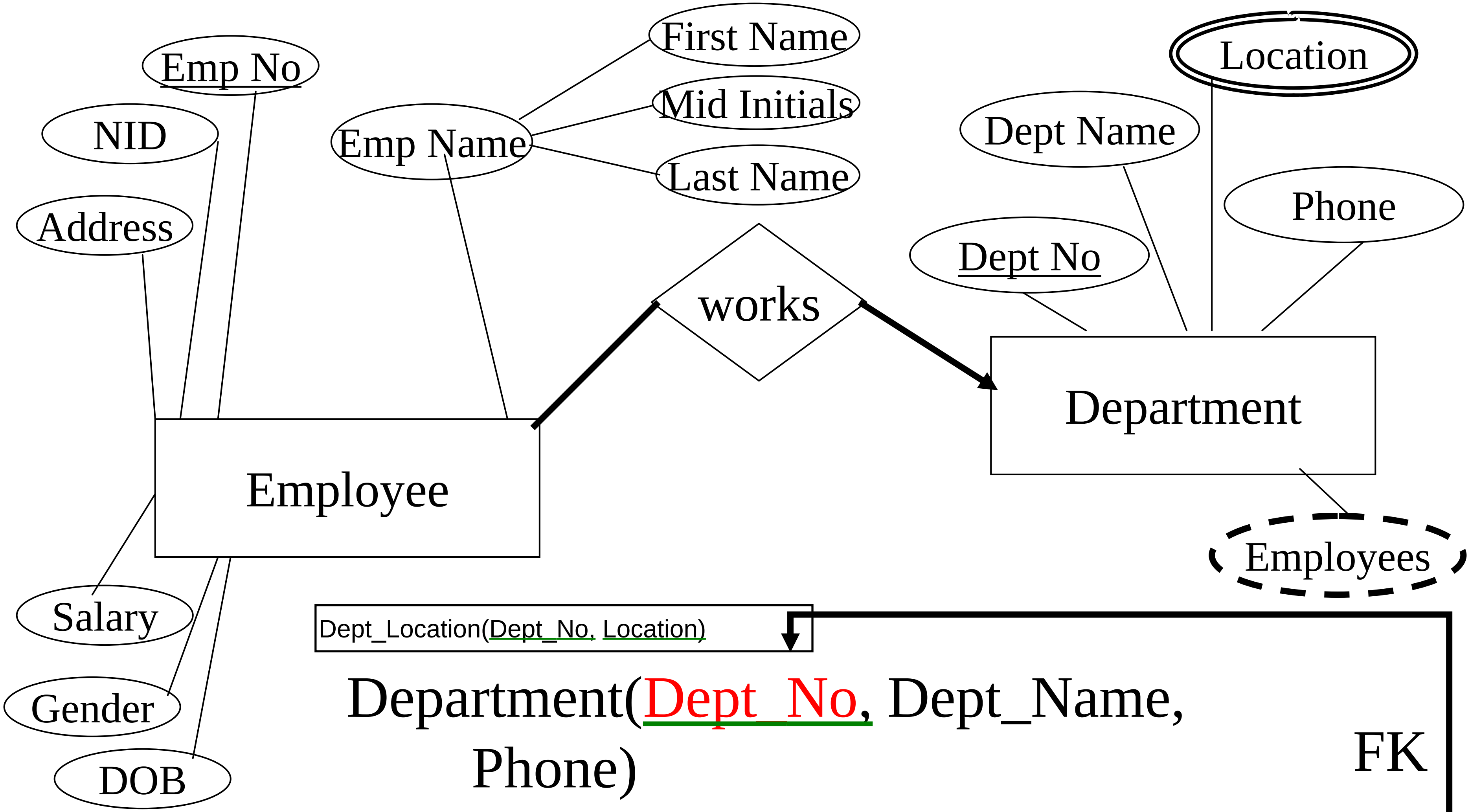
FK

Department(Dept_No,
Dept_Name, Phone)

Dept_Location(*Dept_No*, Location)

Project(Proj_No, Proj_Name,
Location, *Dept_No*)





Dept_Location(Dept_No, Location)

Department(Dept_No, Dept_Name, Phone)

FK

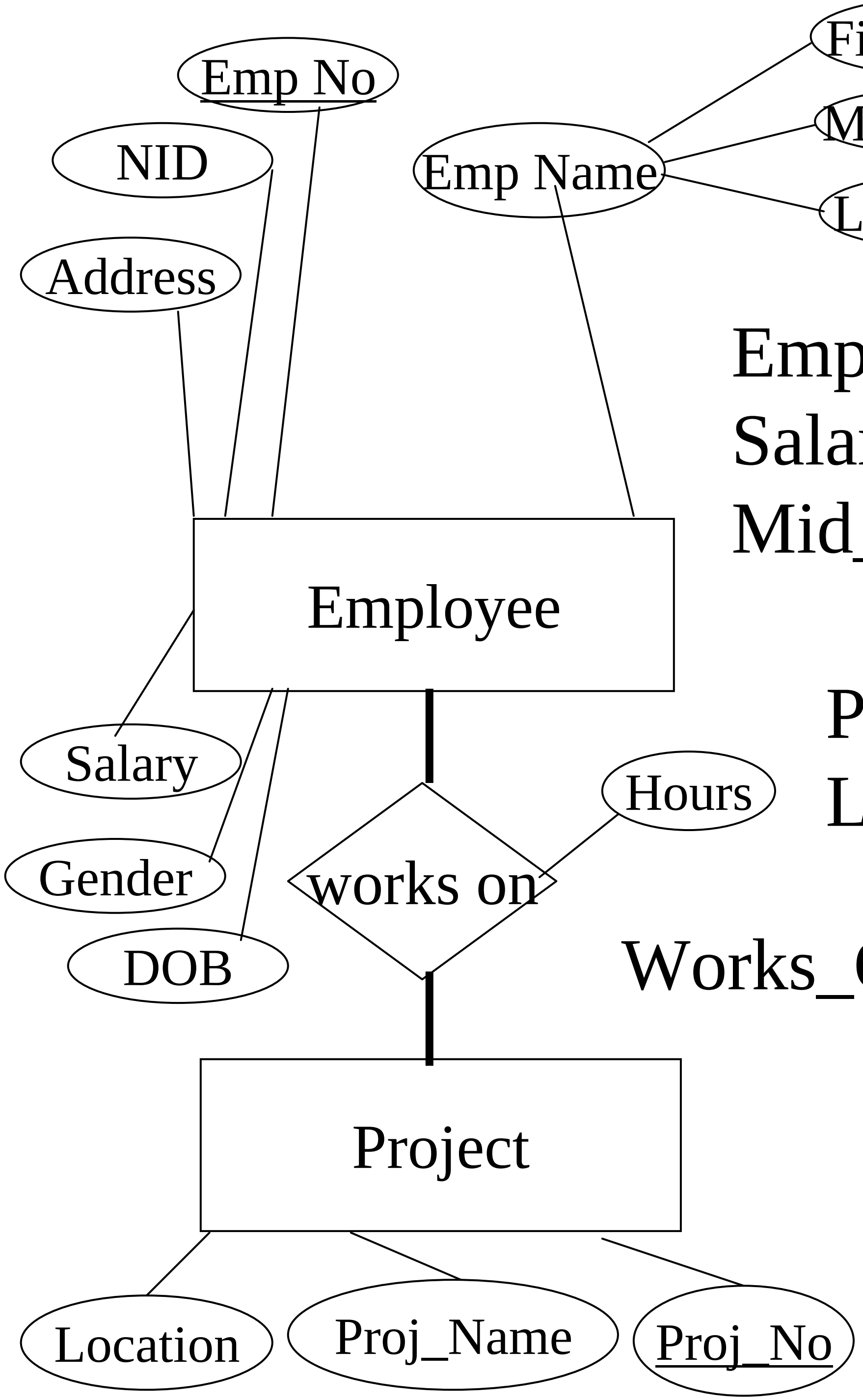
Employee(Emp_No, NID, Address, Salary, Gender, DOB, First_Name, Mid_Initials, Last_Name, *Dept_No*)

Map Binary Many-to-Many Relationships

Create a relation for the two entity types participating in the relationships (step 1)

Create new relation and include PK of each of the two participating entity types as FK. These attributes become the PK (composite)

include any attributes of the relationship to the new relation



FK

Employee(Emp_No, NID, Address, Salary, Gender, DOB, First_Name, Mid_Initials, Last_Name)

Project(Proj_ No, Proj_Name, Location)

FK

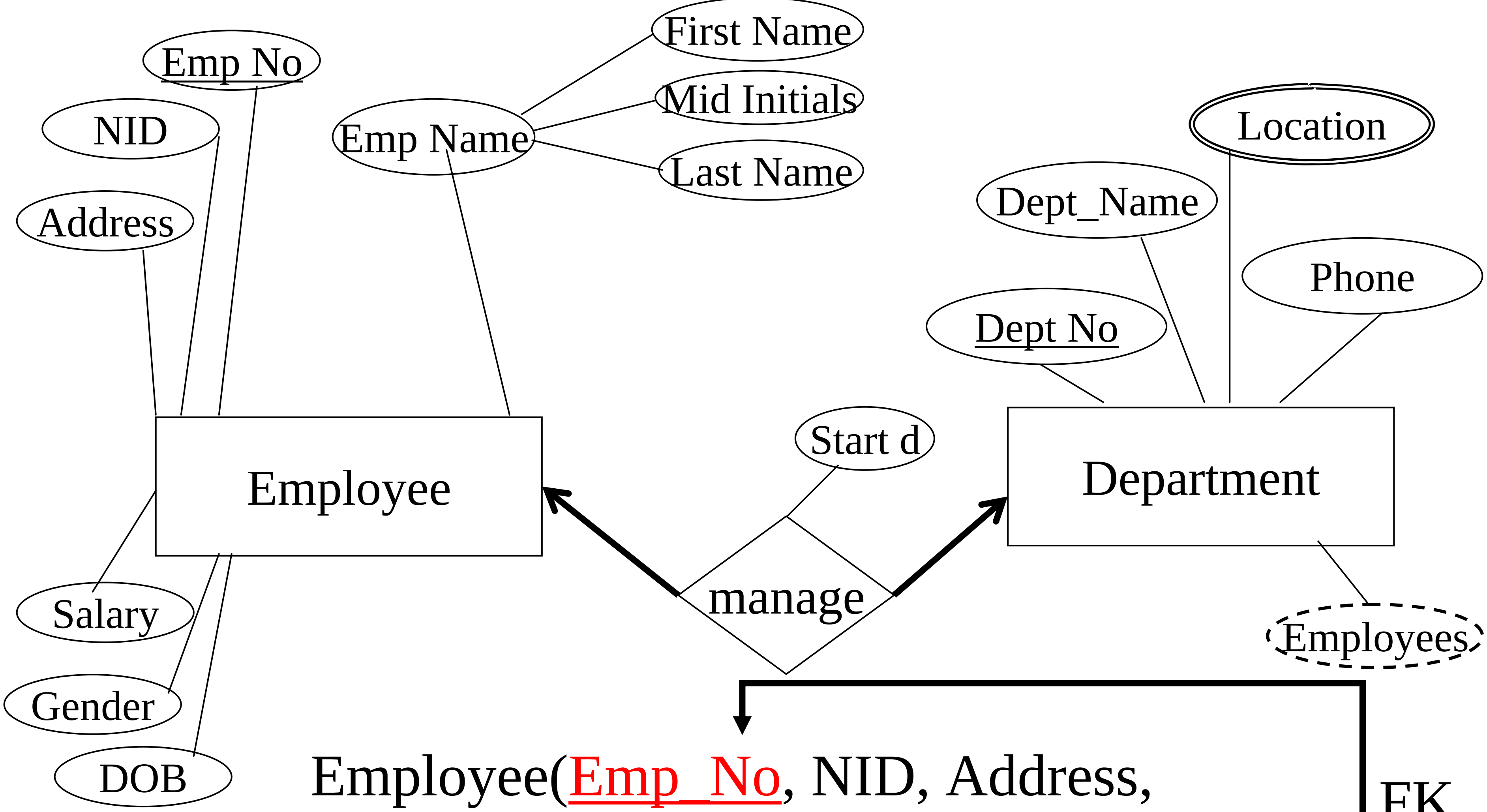
Works_On(*Emp_No*, *Proj_No*, Hours)

Map Binary One-to-One Relationships

Specialised case of One-to-Many
Create a relation for the two entity types participating in the relationships (step 1)

Include PK of one of the relations as a foreign key of the other relation (include in optional side of the relationship that has mandatory participation in the 1:1)

include any attributes of the relationship to the same relation



Employee(Emp_No, NID, Address, Salary, Gender, DOB, First_Name, Mid_Initials, Last_Name)

FK

Department(Dept_No, Dept_Name, Phone, *Manager*, Start_D)

4. Map Unary Relationships

Procedure depends on both the degree of the binary relationships and the cardinalities of the relationships

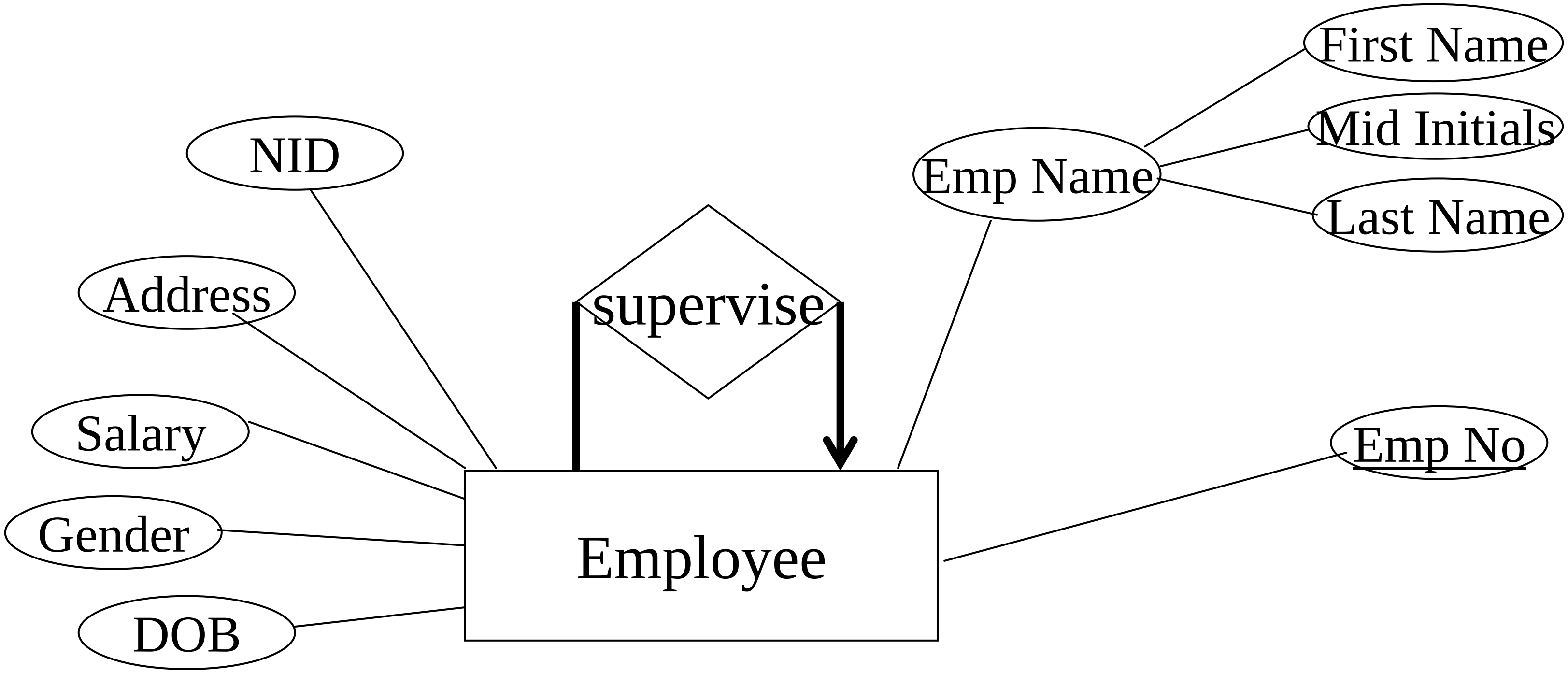
- Map Unary One-to-Many Relationships
- Map Unary Many-to-Many Relationships
- Map Unary One-to-One Relationships

Map Unary One-to-Many Relationships

Create a relation for the entity type (step 1)

include PK of the entity as a foreign key within the same relation

include any attributes of the relationship to the relation of the many side



Employee(Emp_No, NID, Address,
Salary, Gender, DOB, First_Name,
Mid_Initials, Last_Namet_No,
Supervisor)

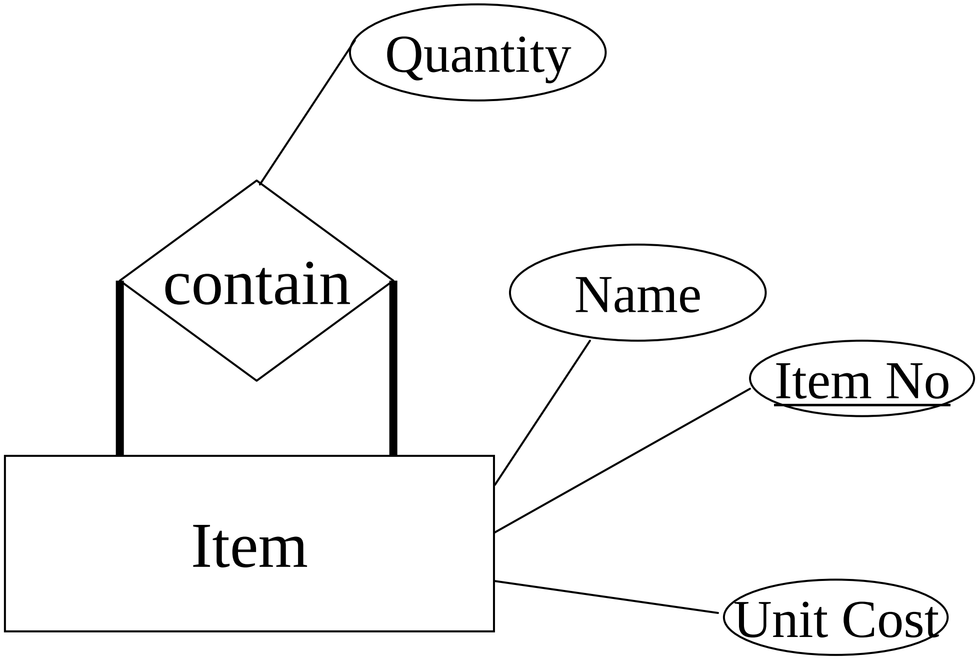
FK

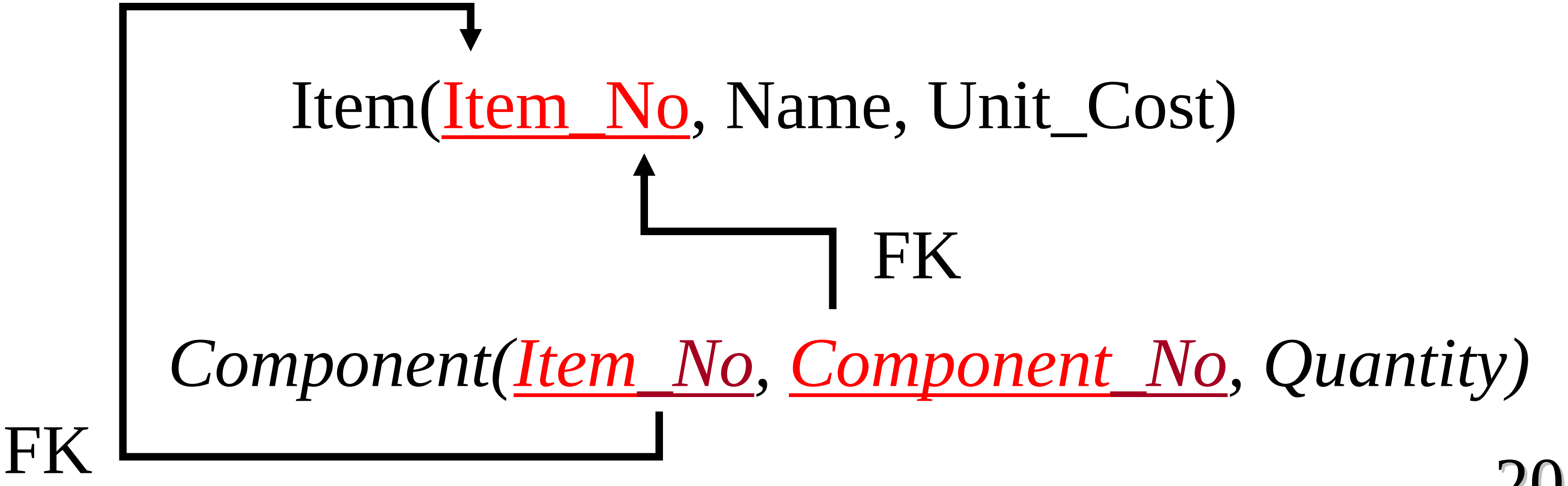
Map Unary Many-to-Many Relationships

Create a relation for the entity type (step 1)

Create new relation and include PK of the entity type as FK twice. These attributes become the PK (composite)

include any attributes of the relationship to the new relation





Item(Item_No, Name, Unit_Cost)

FK

Component(Item_No, Component_No, Quantity)

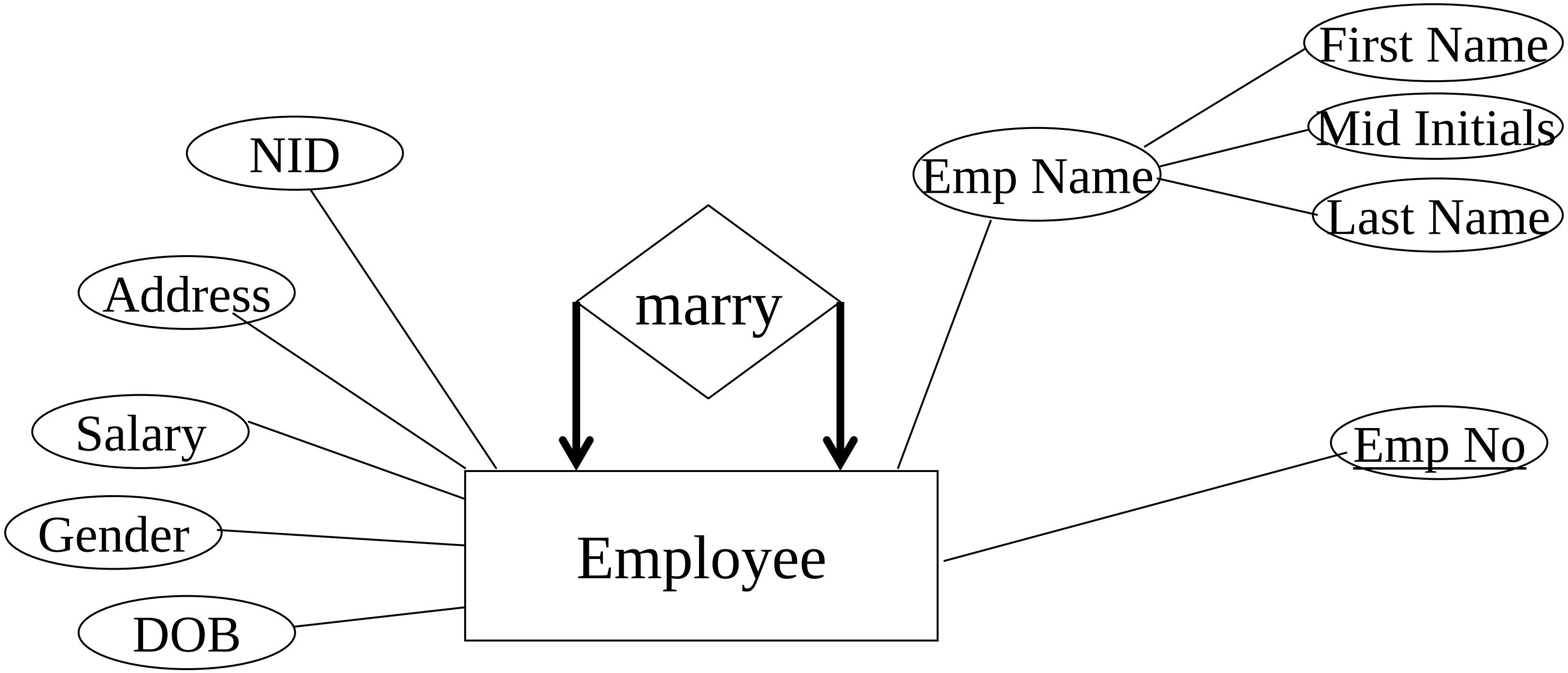
FK

Map Unary One-to-One Relationships

Create a relation for the entity type (step 1)

include PK of the entity as a foreign key within the same relation

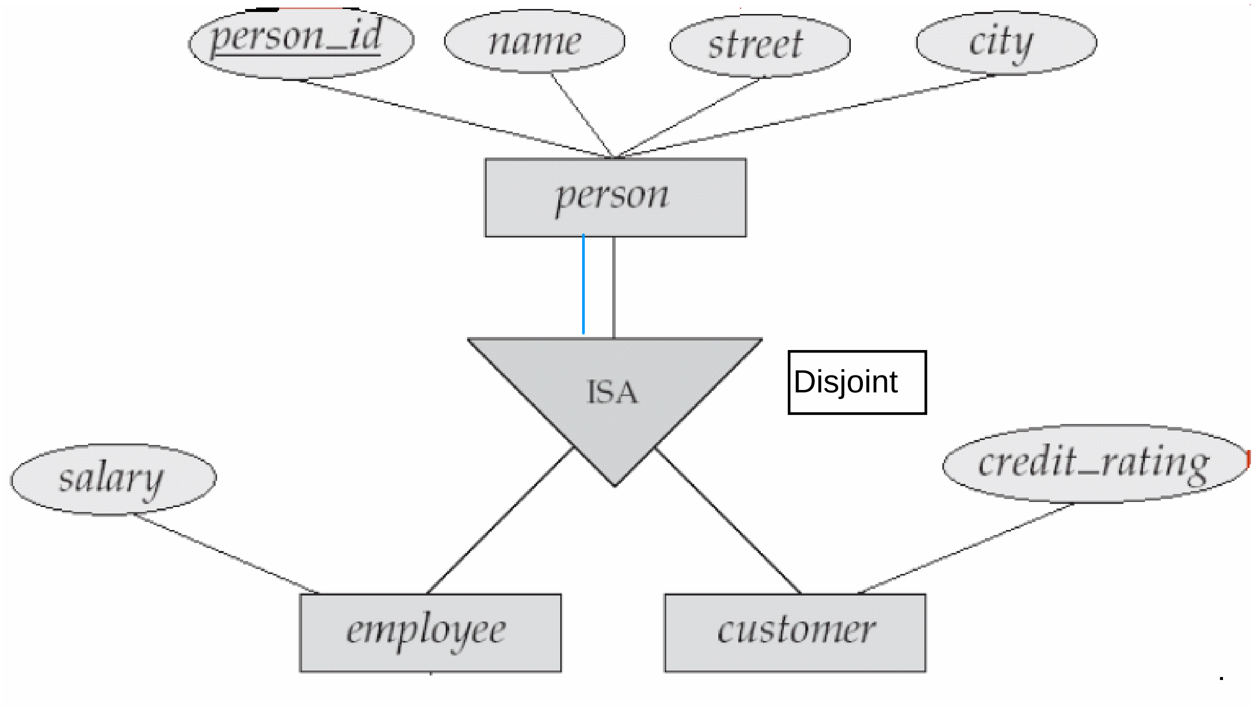
include any attributes of the relationship to the relation of the many side



Employee(Emp_No, NID, Address, Salary, Gender, DOB, First_Name, Mid_Initials, Last_Name, *Marry*)

FK

5. EER Mapping: Case#1 (Total and Disjoint)



We need to create 2 tables:

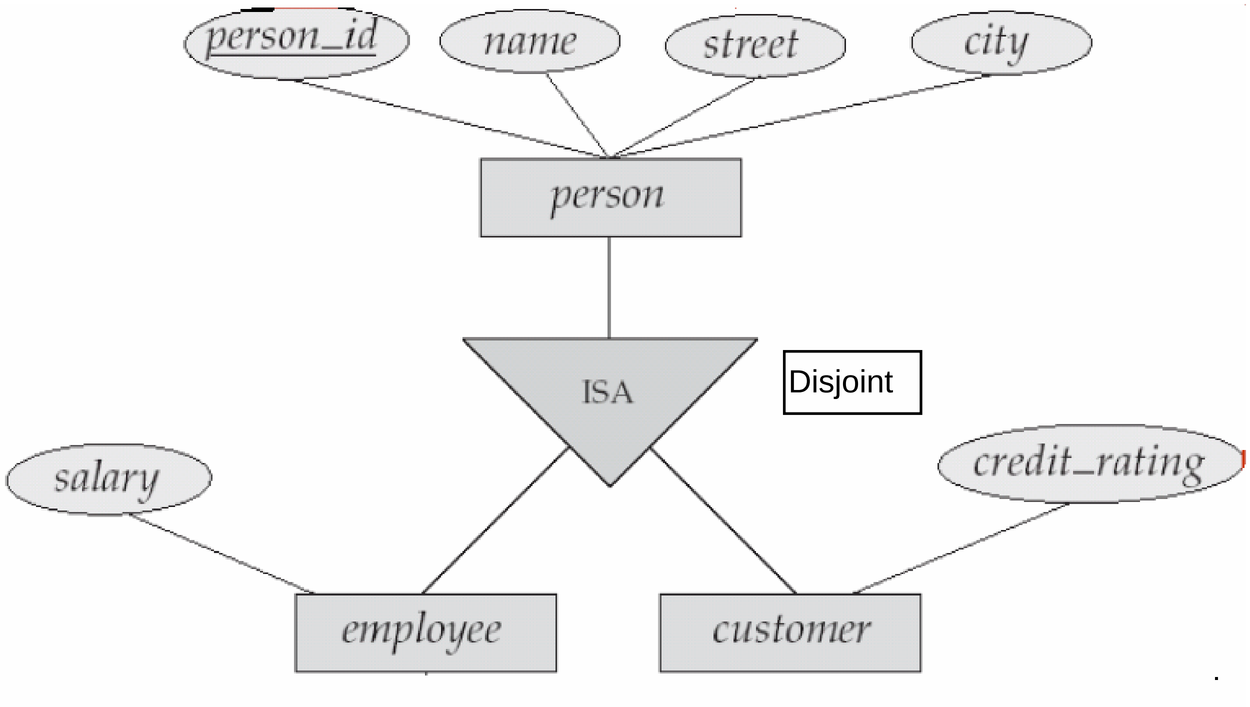
employee = (<u>Person_id</u>, name, street, city, salary)

Customer = (<u>Person_id</u>, name, street, city, credit_rating)

As it is total, no need to create person table

As it is disjoint, there is no redundancy as no employee is customer and no customer is employee.

5. EER Mapping: Case#2 (Partial and Disjoint)



We need to create 3 tables:

***Person* = (*<u>Person_id</u>*, *name, street, city*)**

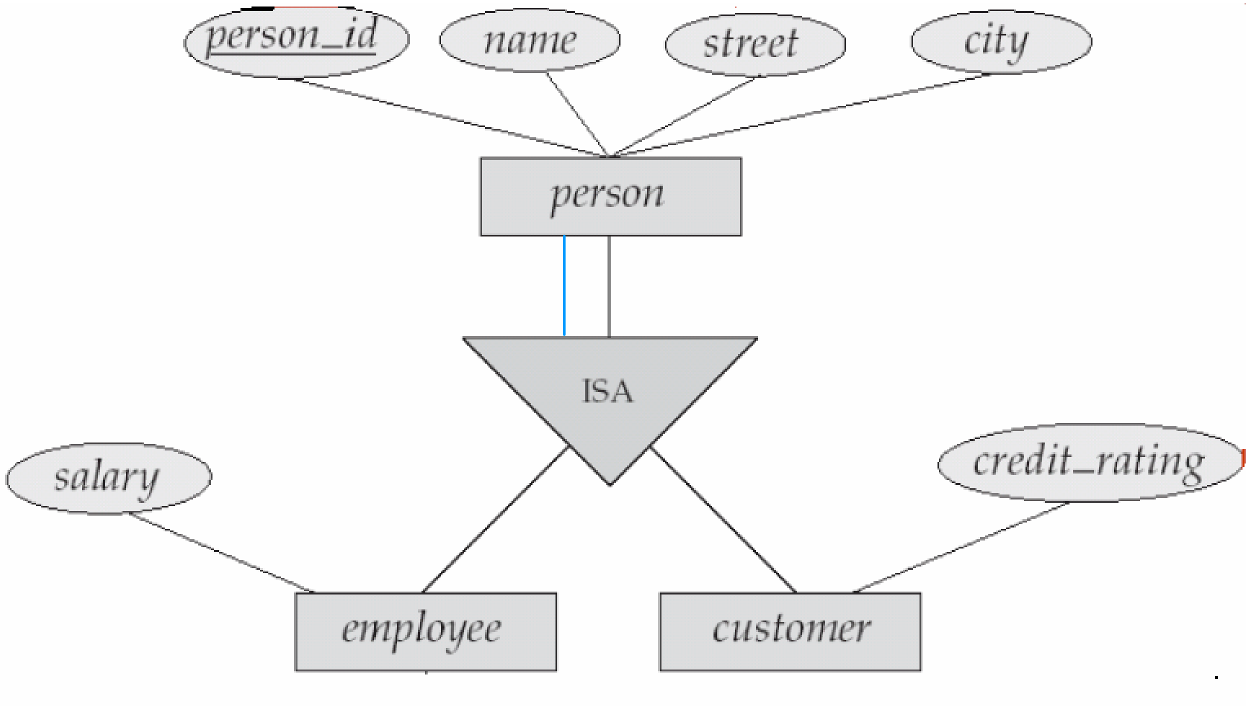
***employee* = (*<u>Person_id</u>*, *name, street, city, salary*)**

***Customer* = (*<u>Person_id</u>*, *name, street, city, credit_rating*)**

As it is partial, we have to use person table.

As it is disjoint, there is no redundancy as no employee is customer and no customer is employee.

5. EER Mapping: Case#3 (Total and Overlapping)



We need to create 3 tables:

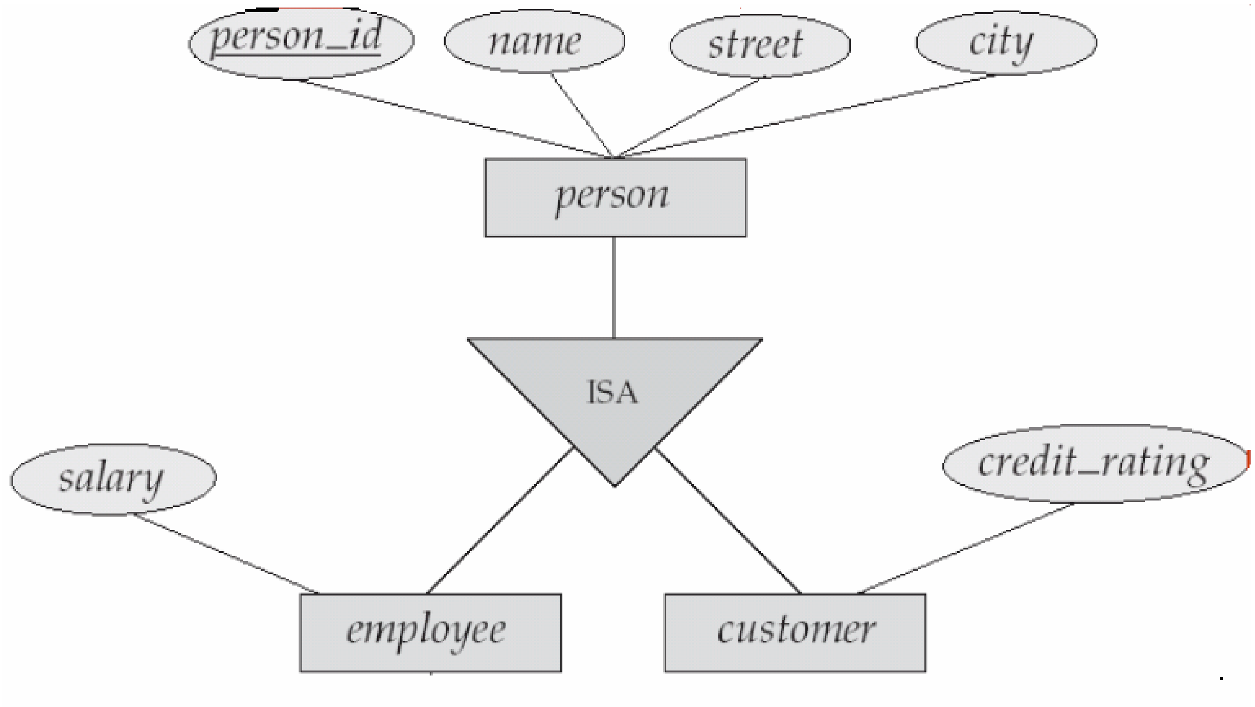
***Person* = (<u>Person_id</u>, name, street, city)**

***employee* = (<u>Person_id</u>, salary)**

***Customer* = (<u>Person_id</u>, credit_rating)**

Although it is total, but we still need to create person table because otherwise, there will be some redundancy.

5. EER Mapping: Case#4 (Partial and Overlapping)



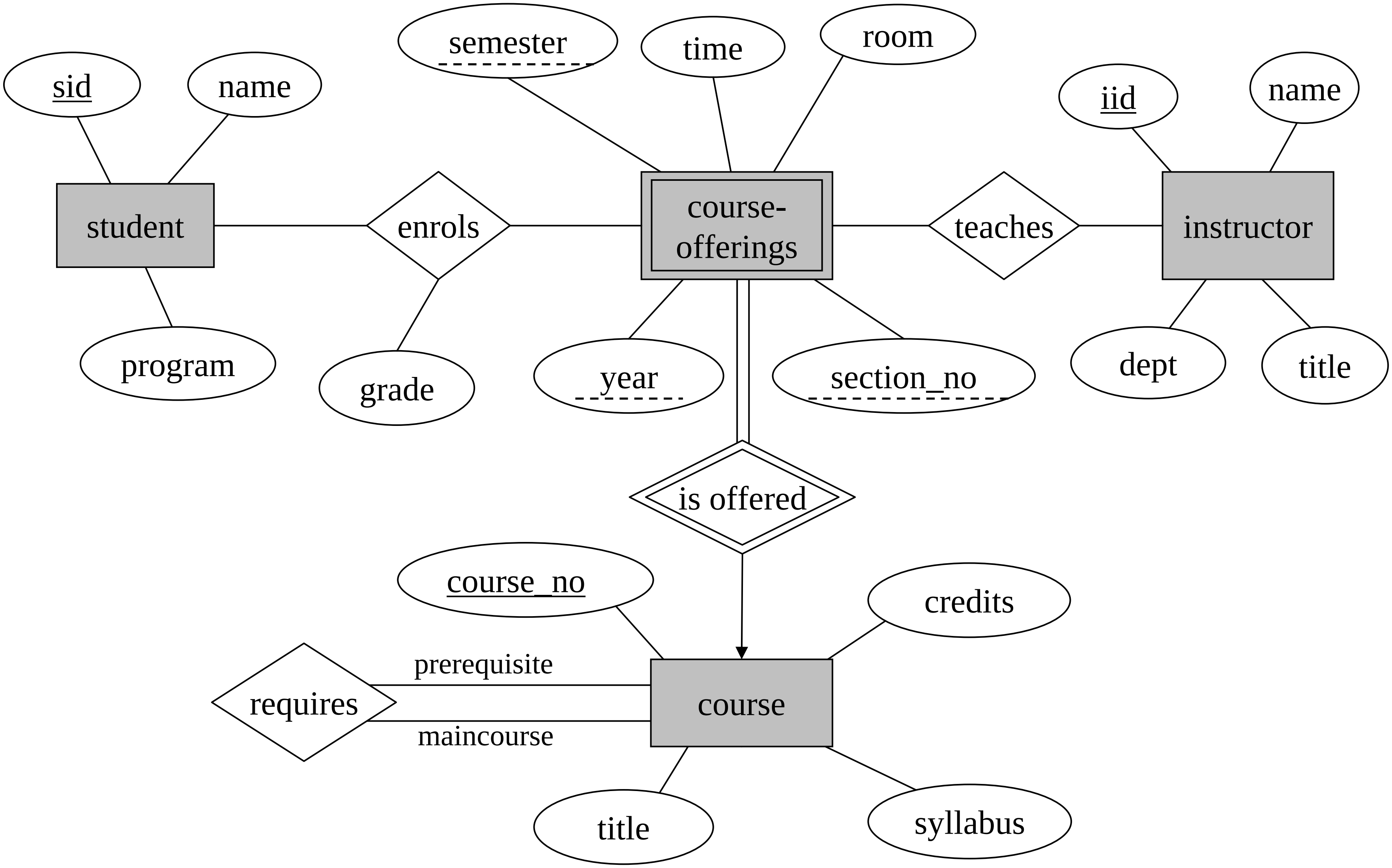
We need to create 3 tables:

Person = (Person_id, name, street, city)

employee = (Person_id, salary)

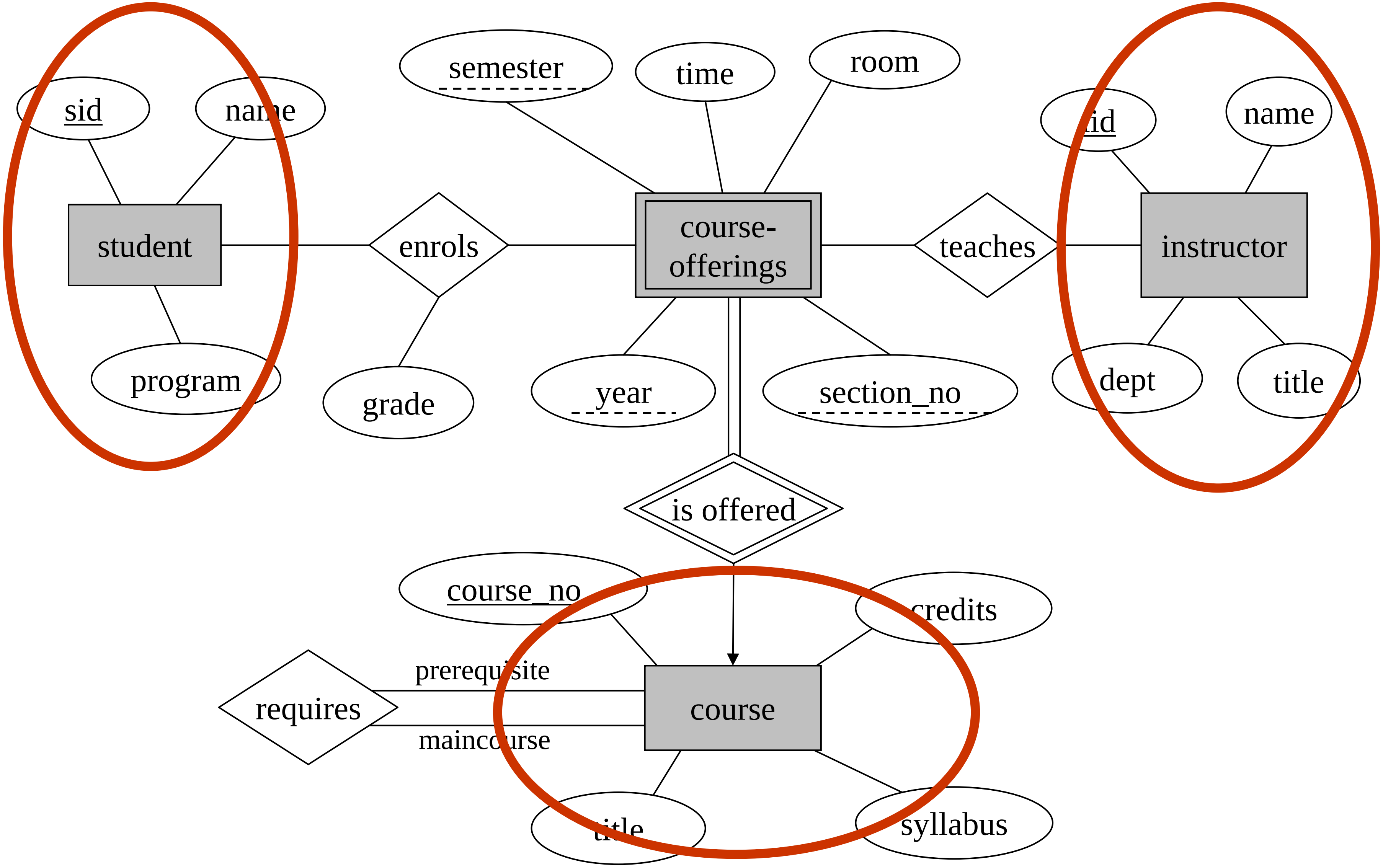
Customer = (Person_id, credit_rating)

Exercise: Convert the E-R Diagram into tables



E-R diagram for a university.

Entities (not weak)



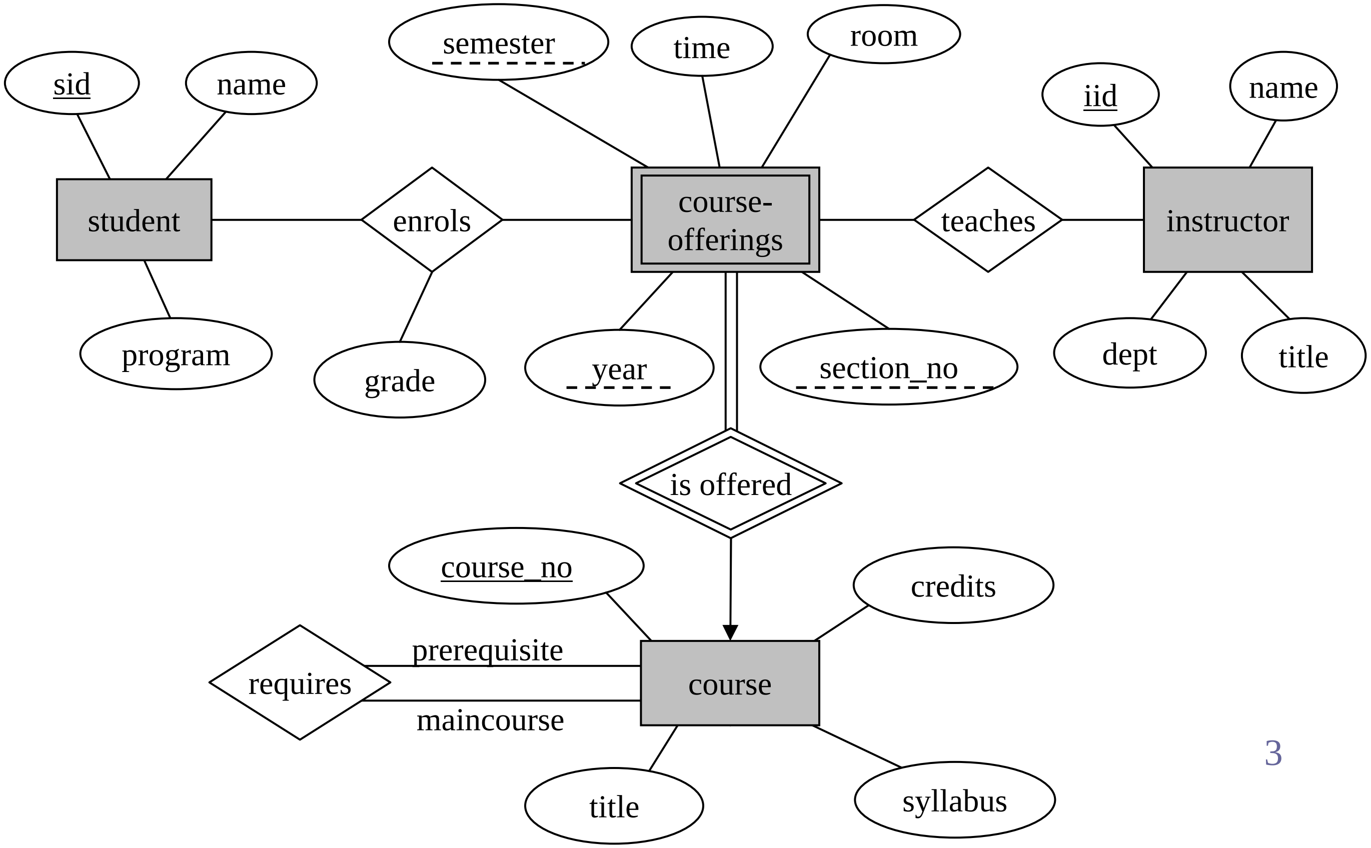
E-R diagram for a university.

Tables of Entities (not weak)

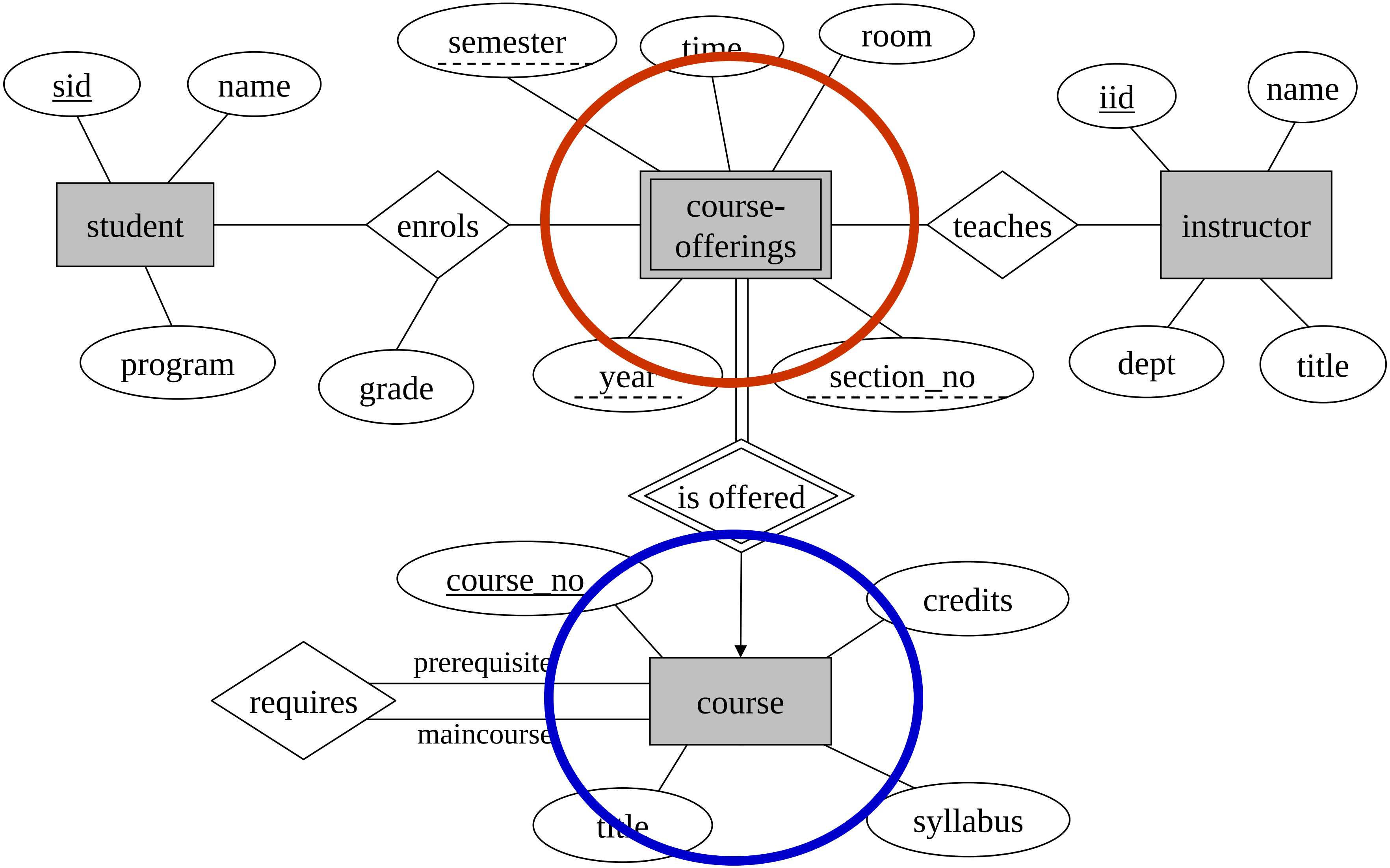
student (<u>sid</u>, name, program)

course (<u>course_no</u>, title, syllabus, credits)

instructor (<u>iid</u>, name, dept, title)



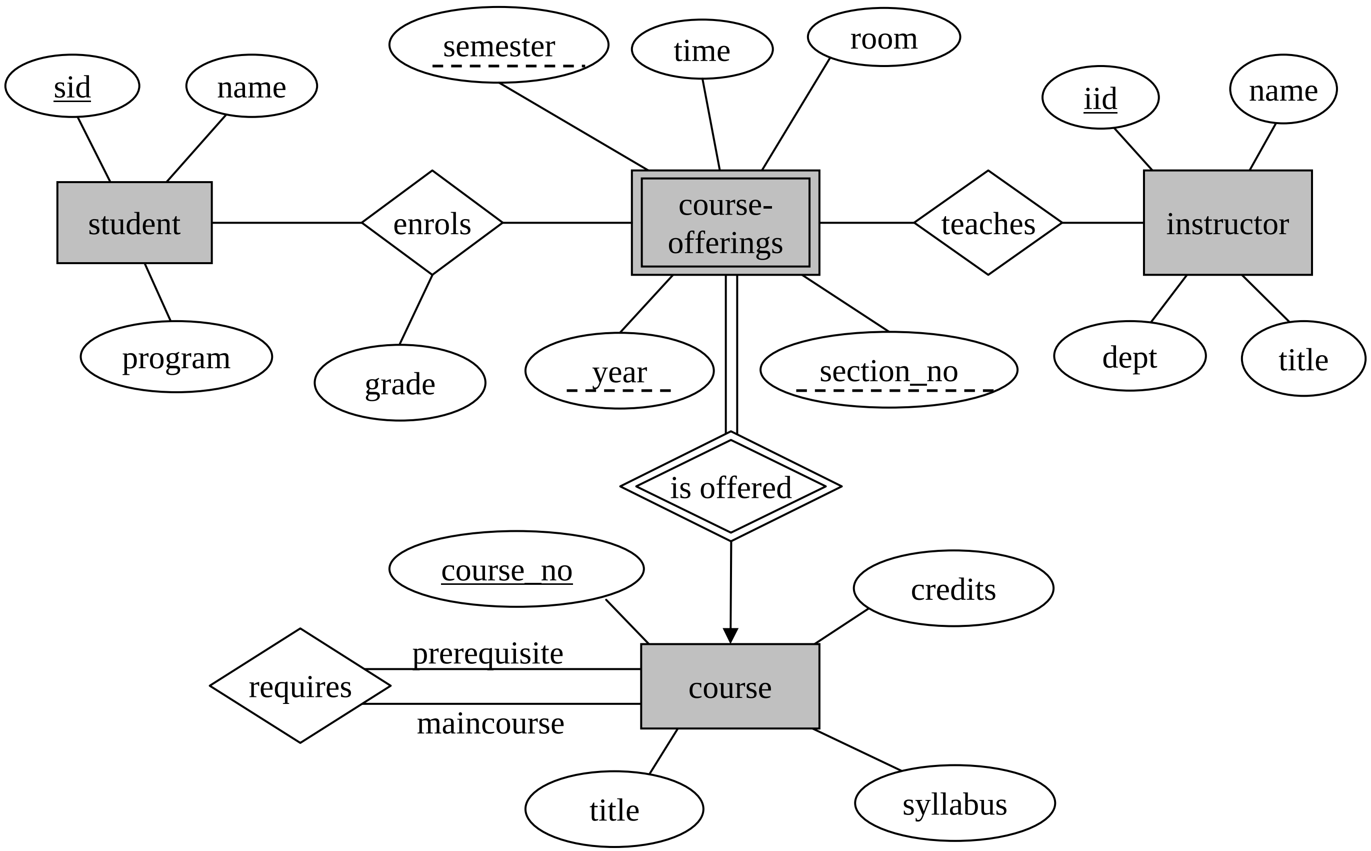
Weak Entity



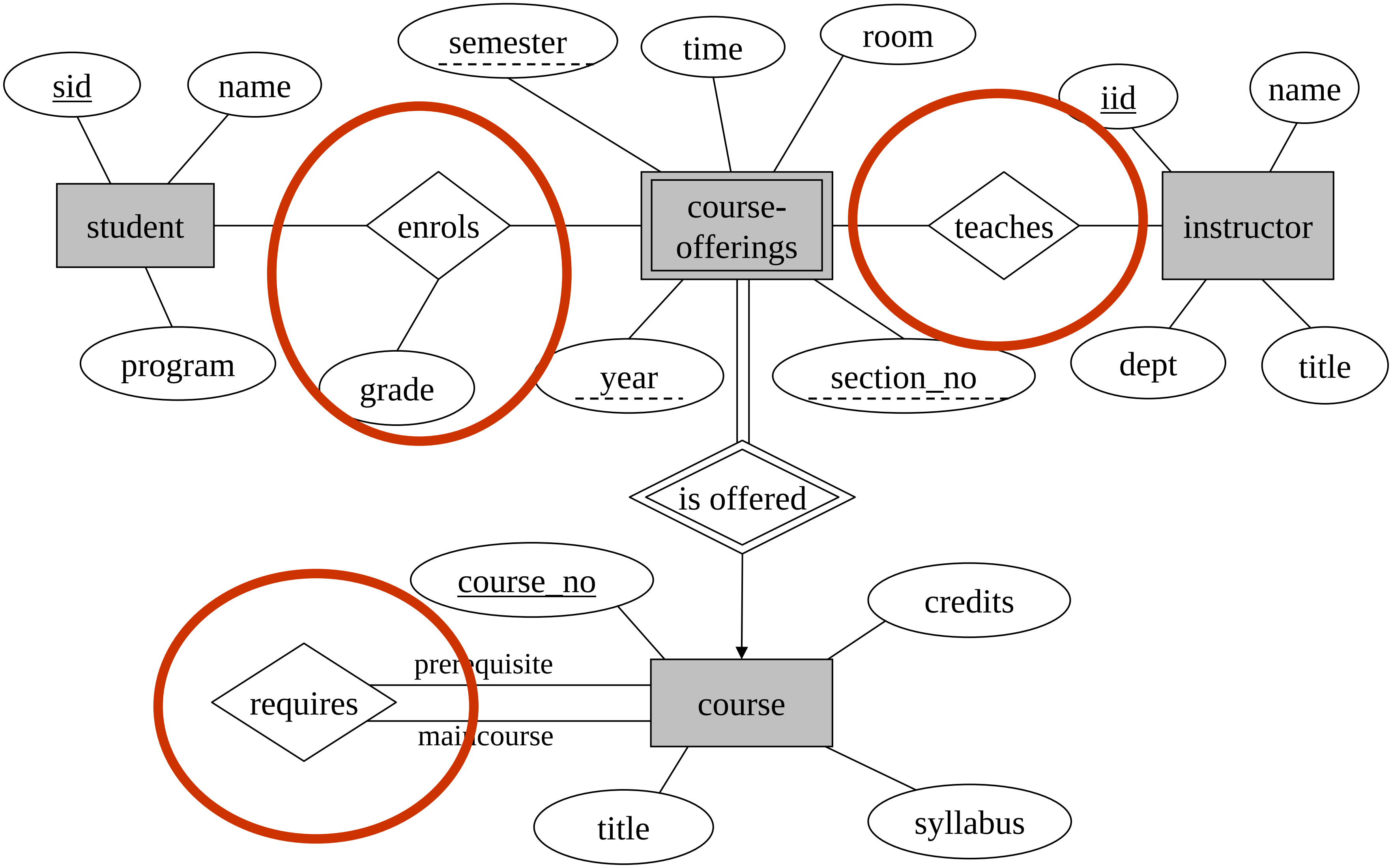
E-R diagram for a university.

Table of Weak Entity

course-offerings (course_no, section_no, year, semester, time, room)



Relationships (without weak entities)



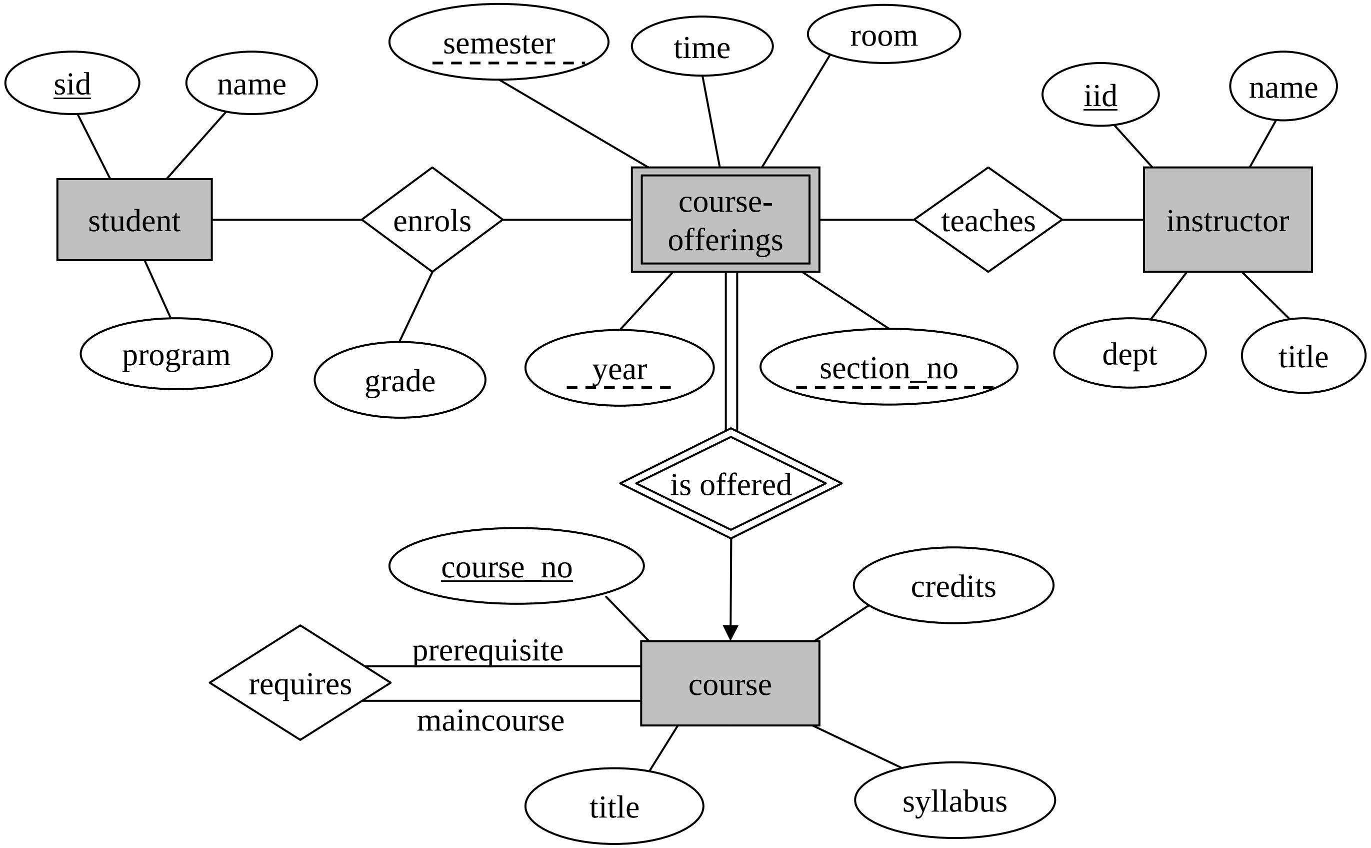
E-R diagram for a university.

Tables of Relationships (without weak entities)

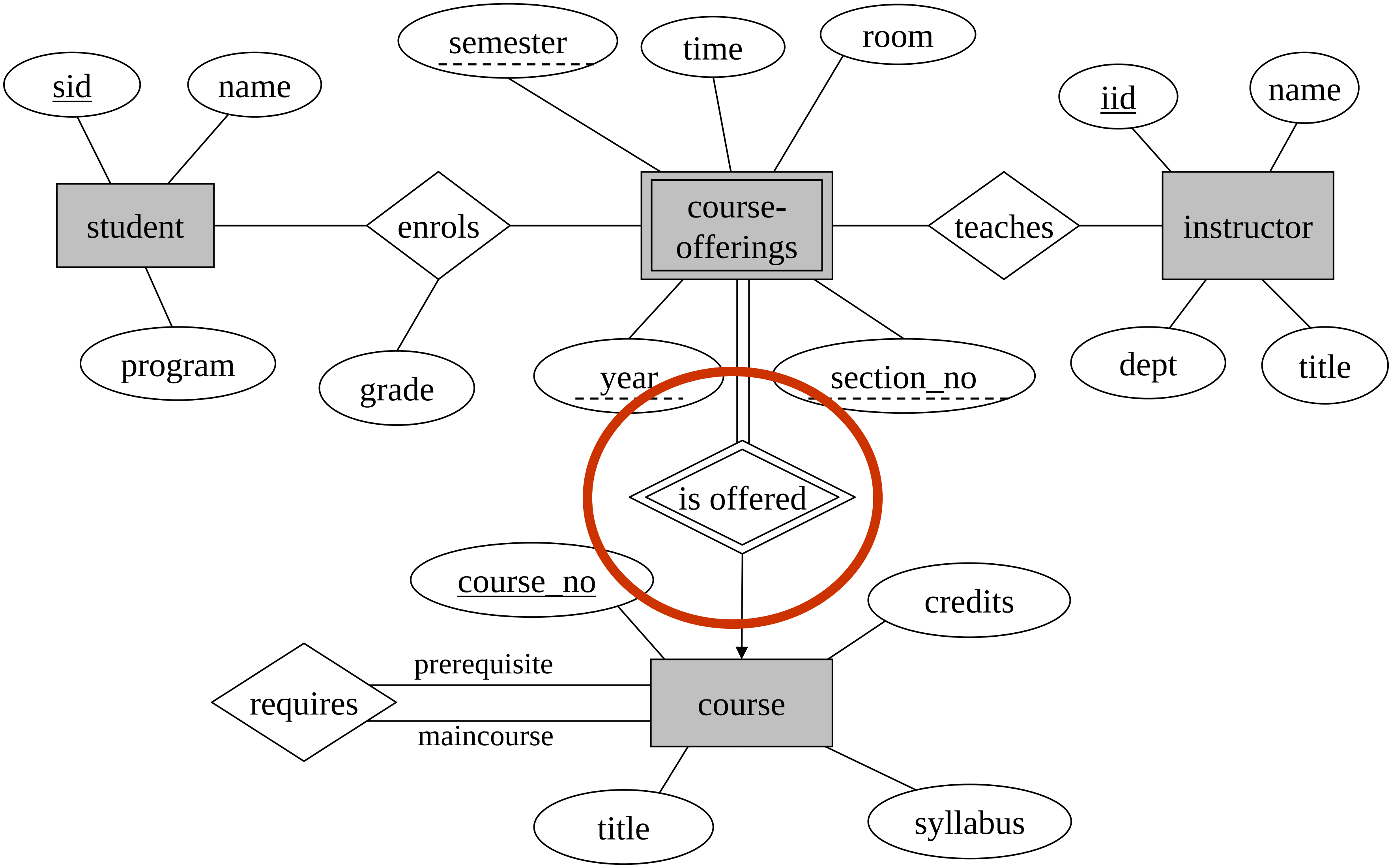
enrols (sid, course_no, section_no, semester, year, grade)

teaches (iid, course_no, section_no, semester, year)

requires (main_course_no, prerequisite_no)



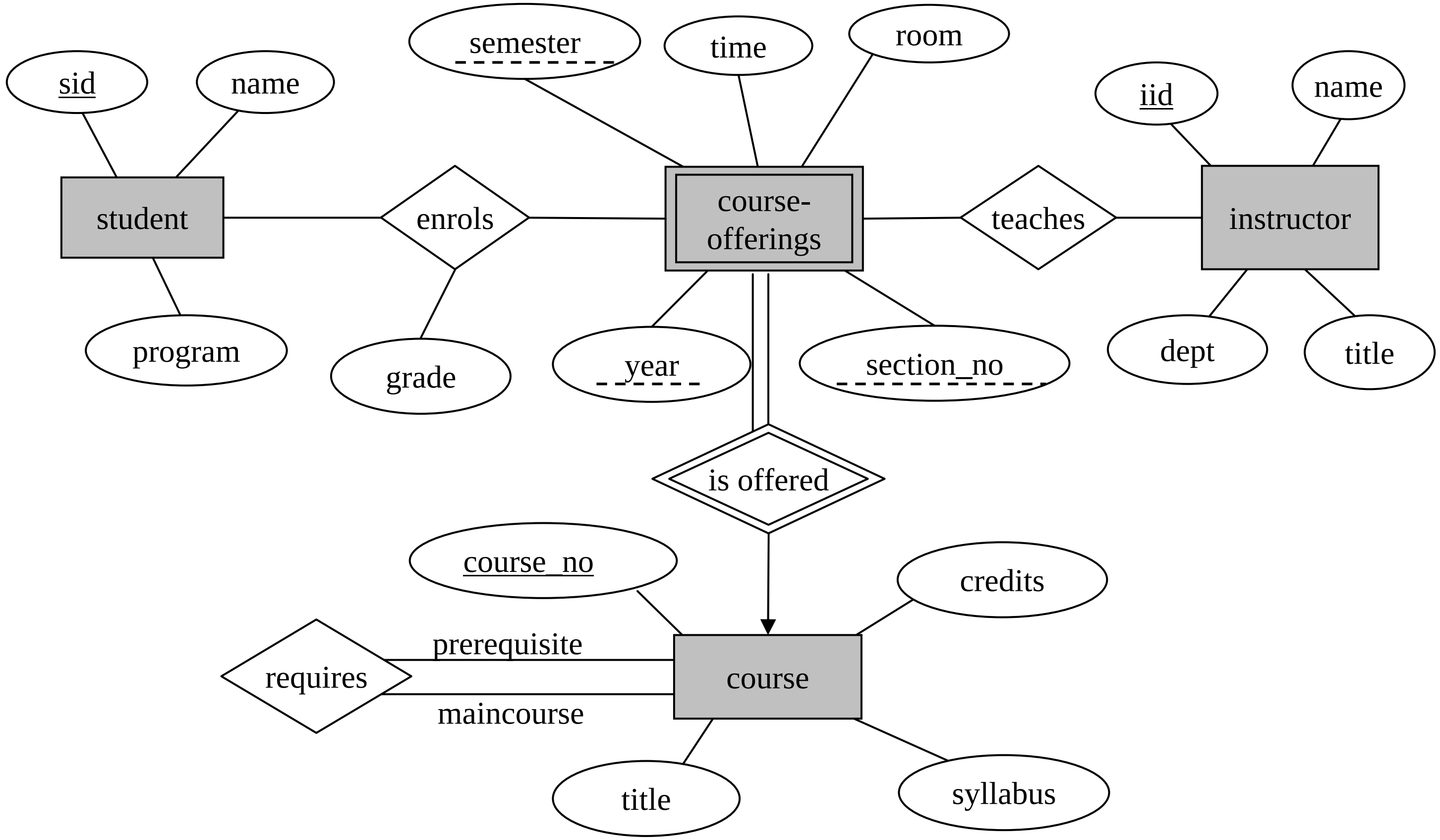
Relationship with the Weak Entity



E-R diagram for a university.

Table of Relationship with the Weak Entity

- There is no extra table for the relationship between a weak entity and its “owner/strong” entity.
- The relationship is already presented in the table for the weak entity.
  - *course-offerings (course_no, section_no, year, semester, time, room)*



Database Normalization

CIS 1902321

Database Normalization

- **Definition**
  - Optimizing table structures!
  - Removing duplicate data entries.

  - Accomplished by thoroughly investigating the various data types and their relationships with one another.

  - Follows a series of normalization “forms” or states.

Why Normalize?

- Improved speed.
- More efficient use of space.
- Increased data integrity.

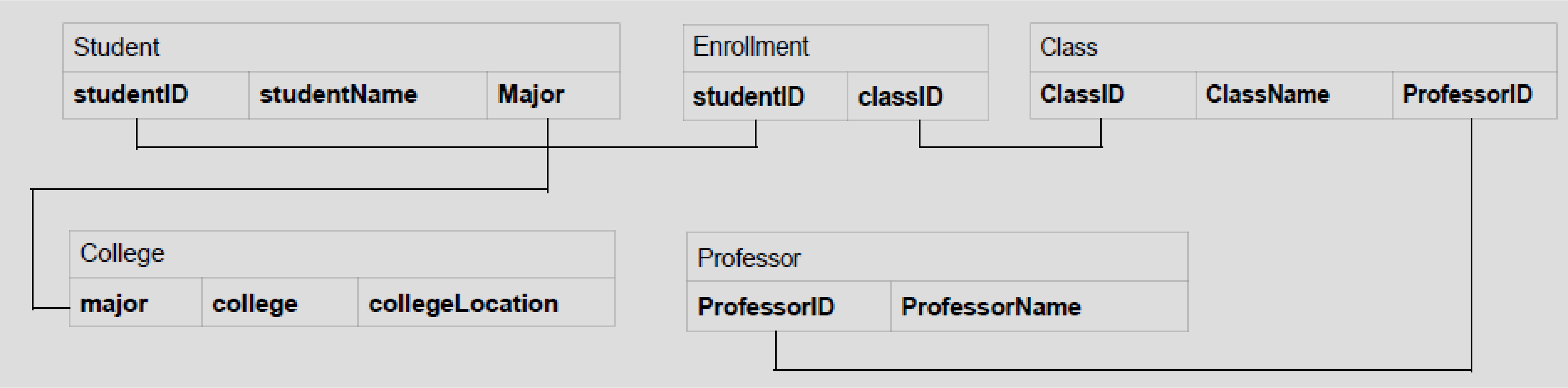
Example!

- Old Design

How ?

Student_id	StudentName	Major	college	collegeLocation	classID	className	ProfessorID	Professor Name
999-40-9876	Ahmad	Math	Science	UJ	Math101	Discrete Math	111-2231	XYZ
999-40-9876	Ahmad	Math	Science	UJ	Skills101	Computer Skills	111-2124	YZR
999-40-9000	Salem	CIS	KASIT	UJ	CIS322	DB	111-2222	URT

- New design (After Normalization)



Student

studentID	studentName	Major

Enrollment

studentID	classID

Class

ClassID	ClassName	ProfessorID

College

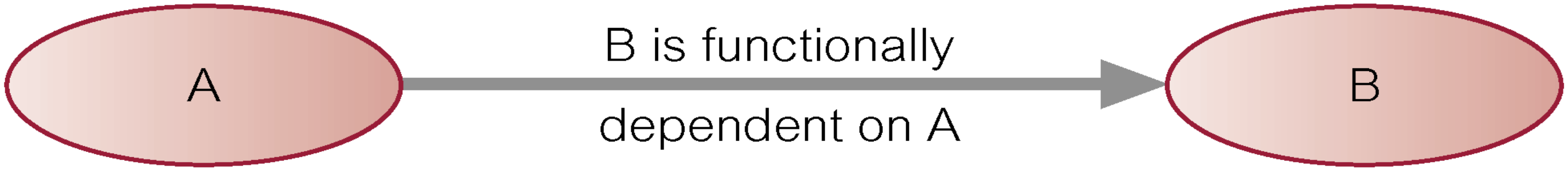
major	college	collegeLocation

Professor

ProfessorID	ProfessorName

Functional Dependency

Definition: *“relationship between or among attributes”*



- Attribute B is functionally dependent on A.

Functional Dependency: An Example!

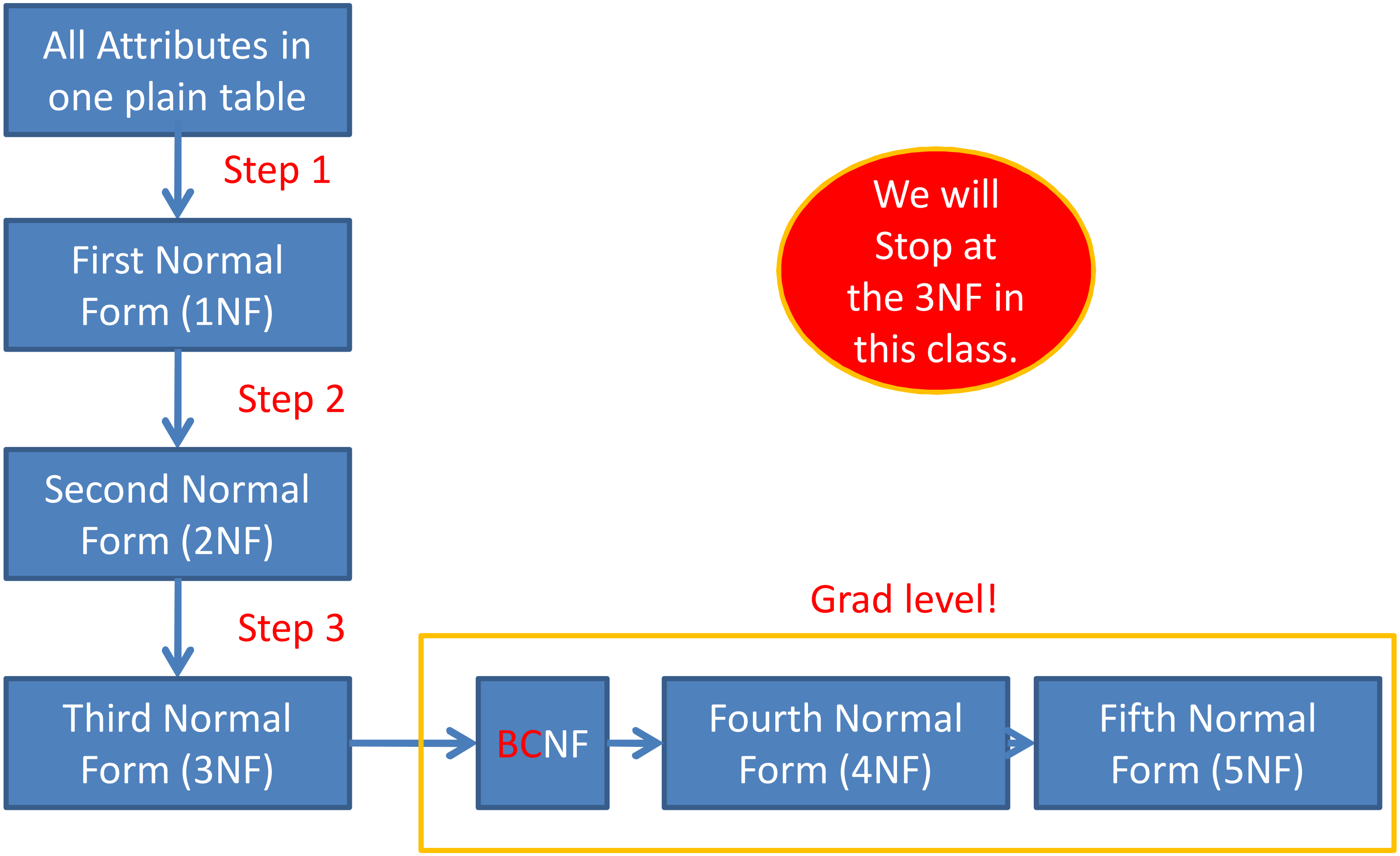
StudentID → StudentName

classID → className

(StudentID, classID) → Grade

StudentID	StudentName	ClassID	ClassName	Grade
St_1	Ahmad Salem	1902322	Database Design	A
St_1	Ahmad Salem	1902321	Data Security	B+
St_2	Osama Othman	1902322	Database Design	C+
St_3	Ameer Asem	1902322	Database Design	A

Normalization Steps



All Attributes in one plain table

Step 1

First Normal Form (1NF)

Step 2

Second Normal Form (2NF)

Step 3

Third Normal Form (3NF)

We will Stop at the 3NF in this class.

Grad level!

BCNF

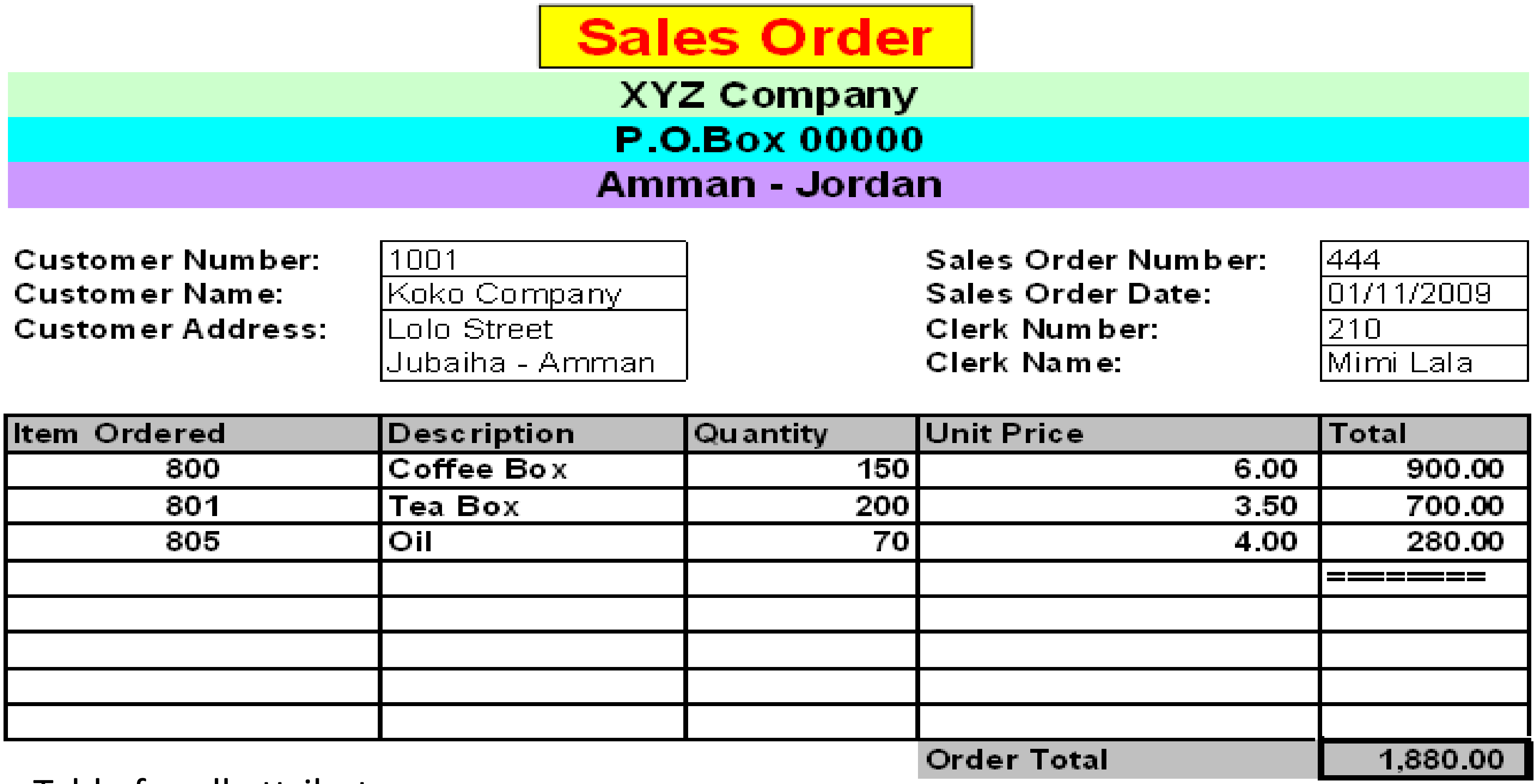
Fourth Normal Form (4NF)

Fifth Normal Form (5NF)

Normalization Steps

- Step1:
  - Elimination of repeated *groups of* data by creating separate tables of related data.
- Step 2:
  - Removal of partial dependencies.
- Step 3:
  - Removal of Transitive dependencies.

Case Study 1



Sales Order

XYZ Company

P.O.Box 00000

Amman - Jordan

Customer Number:	1001	Sales Order Number:	444
Customer Name:	Koko Company	Sales Order Date:	01/11/2009
Customer Address:	Lolo Street	Clerk Number:	210
	Jubaiha - Amman	Clerk Name:	Mimi Lala

Item Ordered	Description	Quantity	Unit Price	Total
800	Coffee Box	150	6.00	900.00
801	Tea Box	200	3.50	700.00
805	Oil	70	4.00	280.00
				========
			Order Total	1,880.00

- Table for all attributes

SalesOrderNo	Date	CustNo	CustName	CustAdd	ClerkNo	ClerkName	ItemNo	Desc	Qty	UnitPrice
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala	800, 801, 805	Coffee Box, Tea Box, Oil	150, 200, 70	6, 3.5, 4

Step 1: To 1NF

SalesOrderNo	Date	CustomerNo	CustomerName	CustomerAdd	ClerkNo	ClerkName	ItemNo	Description	Qty	Unit Price
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala	800, 801, 805	Coffee Box, Tea Box, Oil	150, 200, 70	6, 3.5, 4

Elimination of repeated *groups of* data by creating separate tables of related data.

Create a new table as follows to remove repeating groups:

SalesOrderNo, ItemNo, Description, Qty, UnitPrice

Original Table is left with:

SalesOrderNo, Date, CustomerNo, CustomerName, CustomerAdd, ClerkNo, ClerkName

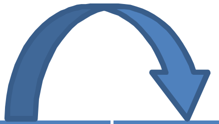
These two tables are a database in first normal form (1NF)

Step 2: To 2NF

Removal of partial dependencies.

- Partial Dependency: is a functional dependency where an attribute is functionally dependent on only part of the primary key (for composite primary keys).
- Create a Table with the functionally dependent data and the part of the key on which it depends.

SalesOrderNo	Date	CustomerNo	CustomerName	CustomerAdd	ClerkNo	ClerkName
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala



Partial Dependency

SalesOrderNo	ItemNo	Description	Qty	UnitPrice
444	800	Coffee Box	6	150
444	801	Tea Box	3.5	200
444	805	Oil	4	70

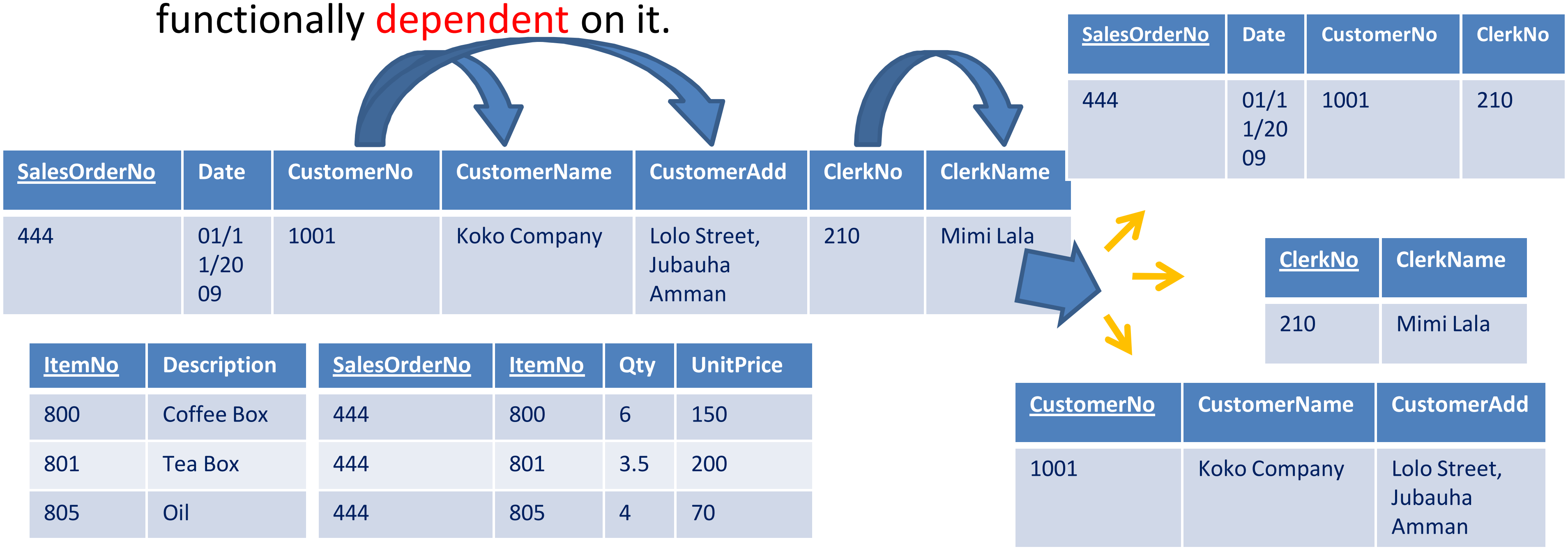
ItemNo	Description
800	Coffee Box
801	Tea Box
805	Oil

SalesOrderNo	ItemNo	Qty	UnitPrice
444	800	6	150
444	801	3.5	200
444	805	4	70

Step 3: To 3NF

Removal of Transitive dependencies.

- Transitive Dependency: is a functional dependency where an attribute is functionally dependent on an attribute other than the primary key. i.e., Its value is only indirectly determined by the primary key.
- Create a separate table containing the attribute and the fields that are functionally dependent on it.



SalesOrderNo	Date	CustomerNo	CustomerName	CustomerAdd	ClerkNo	ClerkName
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala

SalesOrderNo	Date	CustomerNo	ClerkNo
444	01/11/2009	1001	210

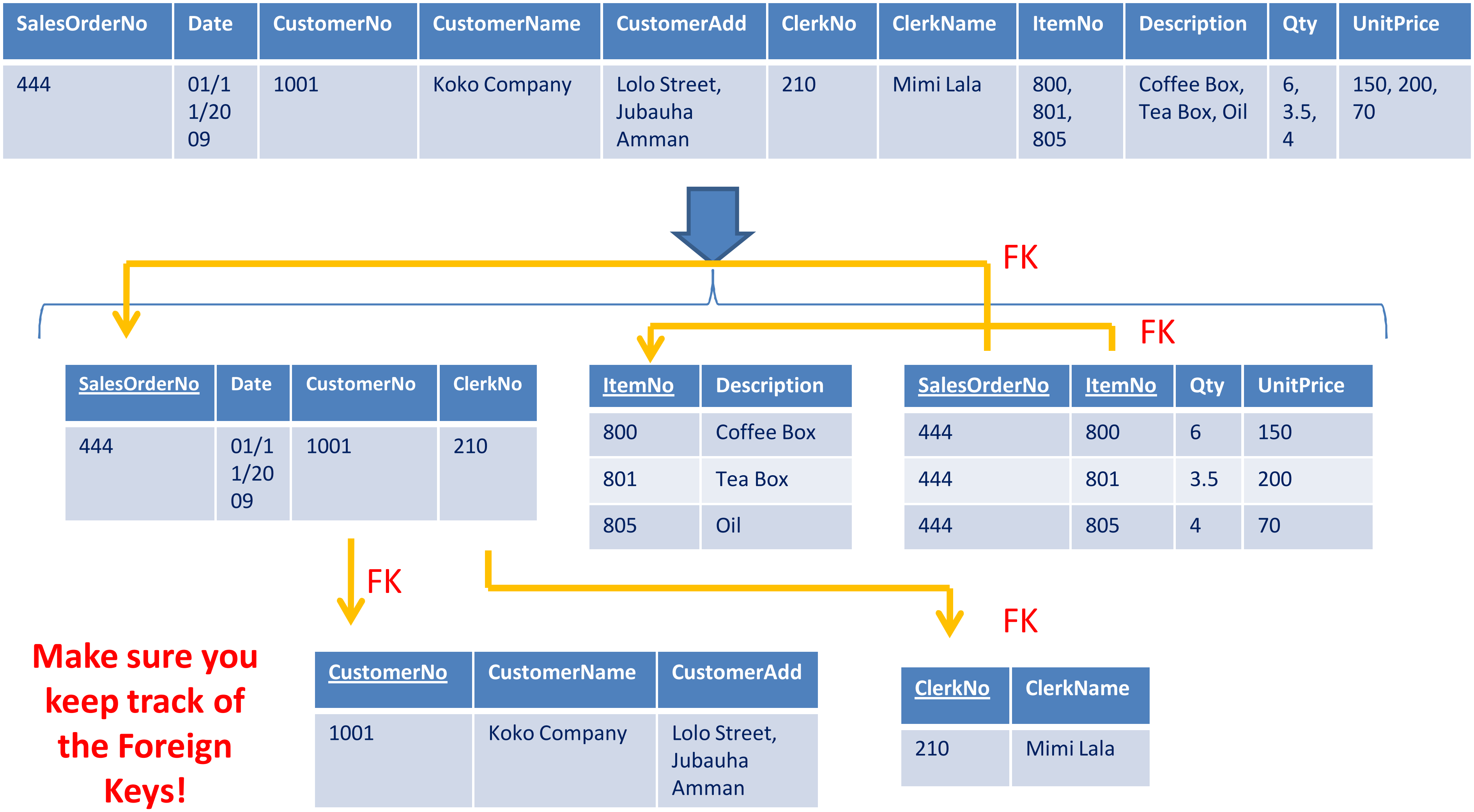
ClerkNo	ClerkName
210	Mimi Lala

CustomerNo	CustomerName	CustomerAdd
1001	Koko Company	Lolo Street, Jubauha Amman

ItemNo	Description
800	Coffee Box
801	Tea Box
805	Oil

SalesOrderNo	ItemNo	Qty	UnitPrice
444	800	6	150
444	801	3.5	200
444	805	4	70

Completed DB in 3NF



SalesOrderNo	Date	CustomerNo	CustomerName	CustomerAdd	ClerkNo	ClerkName	ItemNo	Description	Qty	UnitPrice
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala	800, 801, 805	Coffee Box, Tea Box, Oil	6, 3.5, 4	150, 200, 70

SalesOrderNo	Date	CustomerNo	ClerkNo
444	01/11/2009	1001	210

ItemNo	Description
800	Coffee Box
801	Tea Box
805	Oil

SalesOrderNo	ItemNo	Qty	UnitPrice
444	800	6	150
444	801	3.5	200
444	805	4	70

CustomerNo	CustomerName	CustomerAdd
1001	Koko Company	Lolo Street, Jubauha Amman

ClerkNo	ClerkName
210	Mimi Lala

Make sure you keep track of the Foreign Keys!

Anomalies

- Insertion Anomaly – A record about an entity cannot be inserted into the table without first inserting information about another entity. e.g., Cannot enter a customer without a sales order.
- Deletion Anomaly – A record cannot be deleted without deleting a record about a related entity. e.g., delete a sales order without deleting all of the customer's information.
- Update Anomaly – Cannot update information without changing information in many places. e.g., To update customer information, it must be updated for each sales order the customer has placed

SalesOrderNo	Date	CustomerNo	CustomerName	CustomerAdd	ClerkNo	ClerkName
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala
847	01/11/2010	1001	Koko Company	Lolo Street, Jubauha Amman	320	Yimi Kala

3NF Step

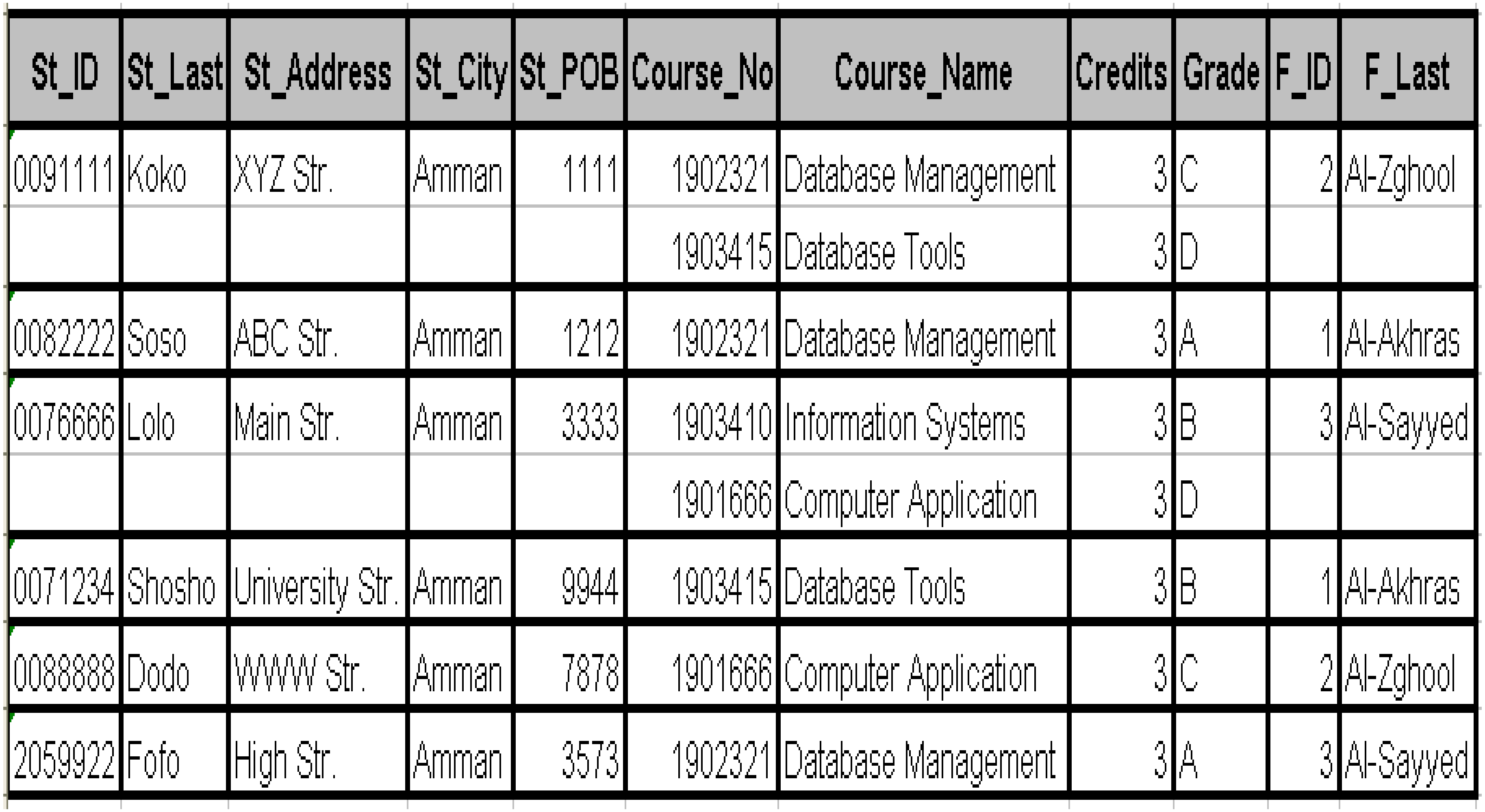
SalesOrderNo	Date	CustomerNo	ClerkNo
444	01/11/2009	1001	210
847	01/11/2010	1001	320

ClerkNo	ClerkName
210	Mimi Lala
320	Yimi Kala

CustomerNo	CustomerName	CustomerAdd
1001	Koko Company	Lolo Street, Jubauha Amman

Case Study 2

Student



St_ID	St_Last	St_Address	St_City	St_POB	Course_No	Course_Name	Credits	Grade	F_ID	F_Last
0091111	Koko	XYZ Str.	Amman	1111	1902321	Database Management	3	C	2	Al-Zghool
					1903415	Database Tools	3	D		
0082222	Soso	ABC Str.	Amman	1212	1902321	Database Management	3	A	1	Al-Akhras
0076666	Lolo	Main Str.	Amman	3333	1903410	Information Systems	3	B	3	Al-Sayyed
					1901666	Computer Application	3	D		
0071234	Shosho	University Str.	Amman	9944	1903415	Database Tools	3	B	1	Al-Akhras
0088888	Dodo	WWW Str.	Amman	7878	1901666	Computer Application	3	C	2	Al-Zghool
2059922	Fofo	High Str.	Amman	3573	1902321	Database Management	3	A	3	Al-Sayyed

Exercise Solution

Baseline Table contains:

St_ID, St_Last, St_Address, St_City, St_POB, Course_No, Course_Name, Credits, Grade, F_ID, F_Last

First Normal Form

In First Normal Form:

(1) Primary keys are set

(2) No repeating groups

Produced Tables:

StInfo: St_ID, St_Last, St_Address, St_City, St_POB

StCourses: St_ID, Course_No, Course_Name, Credits, Grade, F_ID, F_Last

Second Normal Form

In Second Normal Form:

(*) No partial dependency …

Produced Tables:

StInfo: St_ID, St_Last, St_Address, St_City, St_POB

StCourses: Course_No, Course_Name, Credits

StCourses: St_ID, Course_No, Grade, F_ID, F_Last

Third Normal Form

In Third Normal Form:

(*) No transitive dependency …

Produced Tables:

StInfo: St_ID, St_Last, St_Address, St_City, St_POB

StCourses: Course_No, Course_Name, Credits

StCourses: F_ID, F_Last

StCourses: St_ID, Course_No, Grade, F_ID

Case Study 3

PORJ_NUM	Proj_Name	EMP_Num	Emp_Name	Job_class	Charge_hours	Hours_worked
123	UJ-Proj	17	XYZ	Engineer	JD 100	35
		90	UHY	Designer	JD 150	21
		50	OKI	Admin	JD 400	39
389	KASIT-Proj	90	UHY	Designer	JD 150	17
		65	PLU	Worker	JD 75	40
		83	RGT	Worker	JD 75	40
		24	KUH	Engineer	JD 100	30

Candidate key is the Proj_NUM

Case 3: To 1NF

PORJ_NUM	Proj_Name	EMP_Num	Emp_Name	Job_class	Charge_hours	Hours_worked
123	UJ-Proj	17	XYZ	Engineer	JD 100	35
		90	UHY	Designer	JD 150	21
		50	OKI	Admin	JD 400	39
389	KASIT-Proj	90	UHY	Designer	JD 150	17
		65	PLU	Worker	JD 75	40
		83	RGT	Worker	JD 75	40
		24	KUH	Engineer	JD 100	30

Elimination of repeated Groups

Table 1:

PORJ_NUM	Proj_Name

Table 2:

PORJ_NUM	EMP_Num	Emp_Name	Job_class	Charge_hours	Hours_worked

Case 3: To 2NF

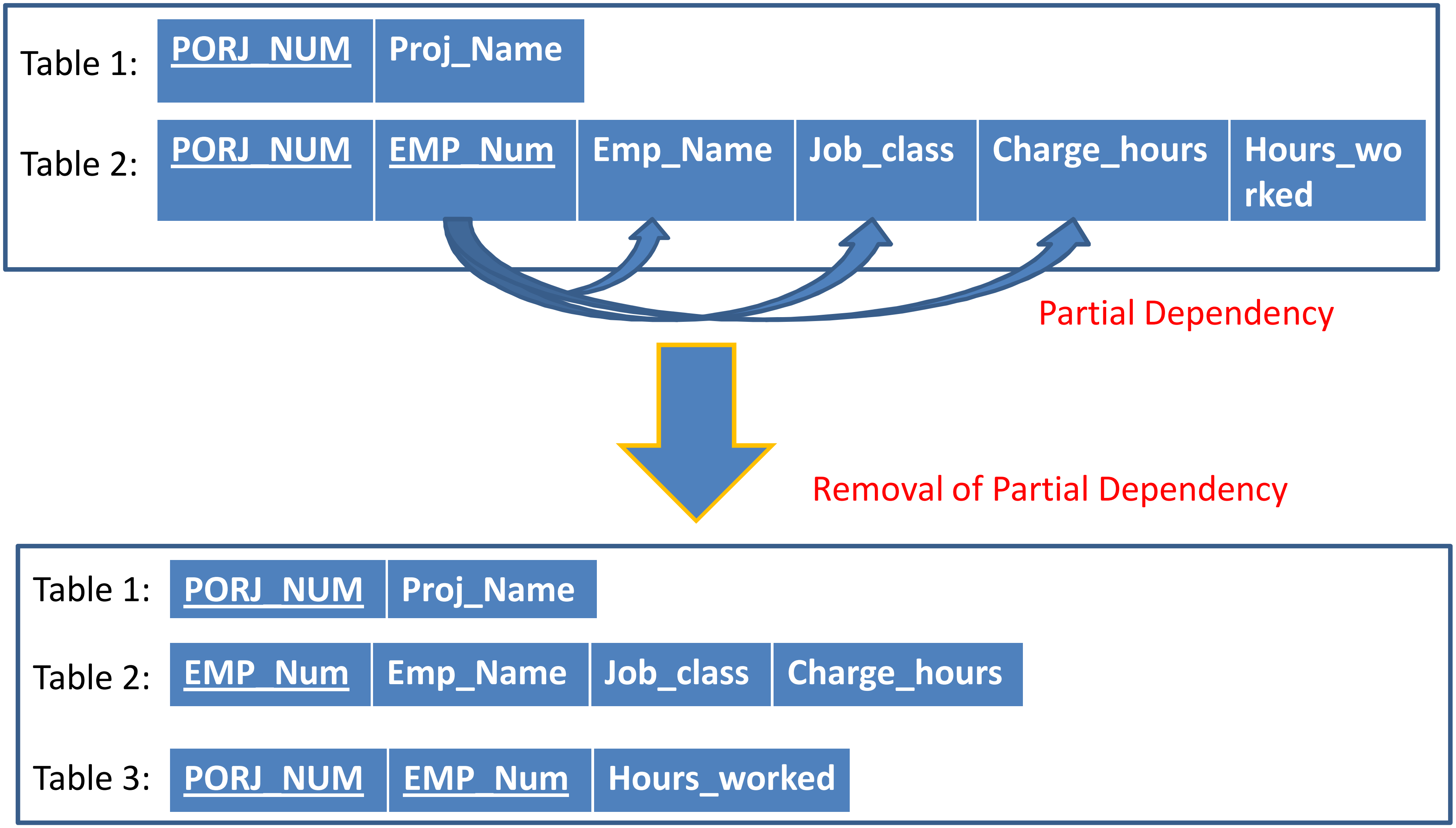


Table 1:

PORJ_NUM	Proj_Name

Table 2:

PORJ_NUM	EMP_Num	Emp_Name	Job_class	Charge_hours	Hours_worked

Partial Dependency

Removal of Partial Dependency

Table 1:

PORJ_NUM	Proj_Name

Table 2:

EMP_Num	Emp_Name	Job_class	Charge_hours

Table 3:

PORJ_NUM	EMP_Num	Hours_worked

Case 3: To 3NF

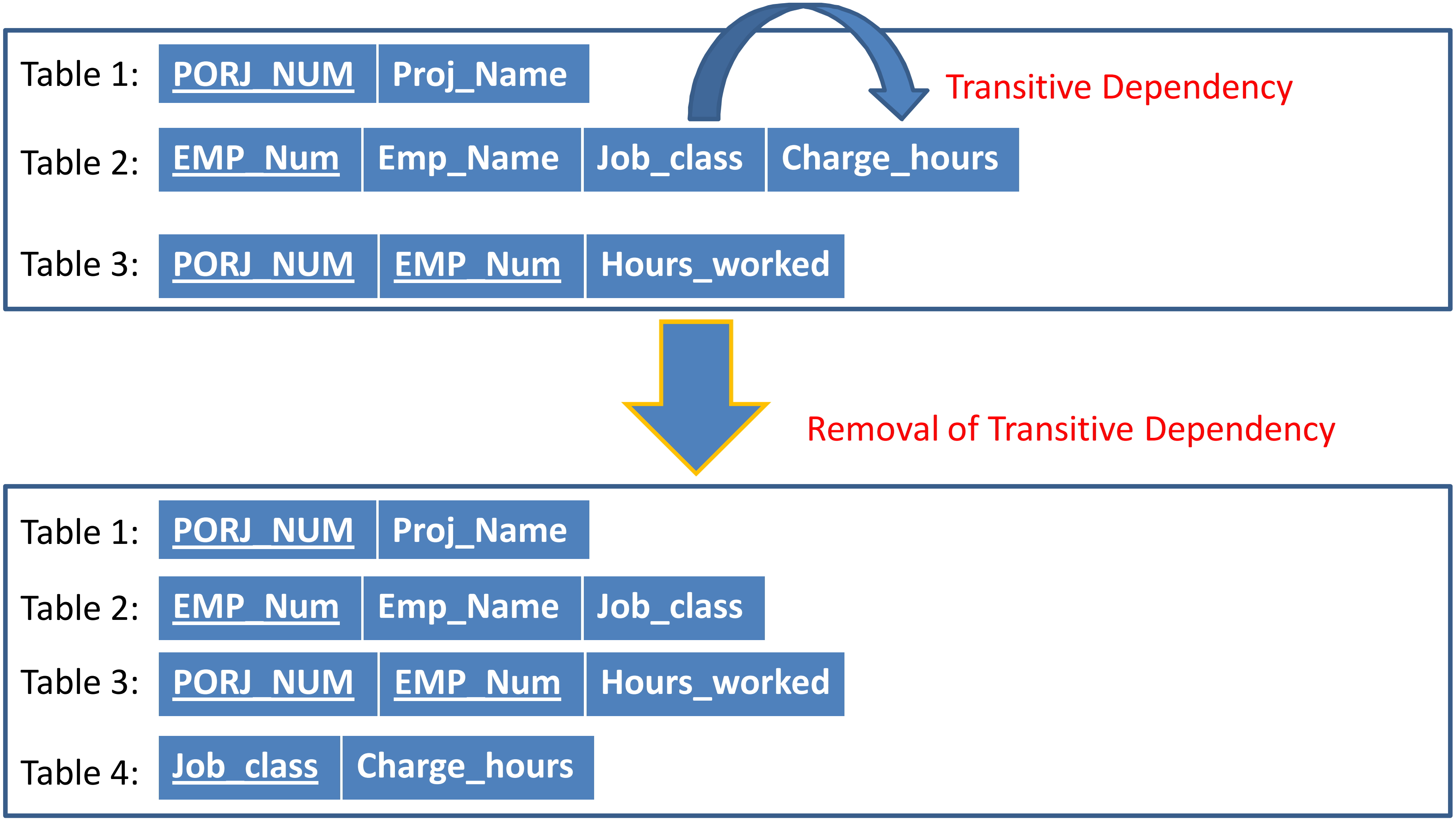


Table 1: PORJ_NUM | Proj_Name

Table 2: EMP_Num | Emp_Name | Job_class | Charge_hours

Table 3: PORJ_NUM | EMP_Num | Hours_worked

Transitive Dependency

Removal of Transitive Dependency

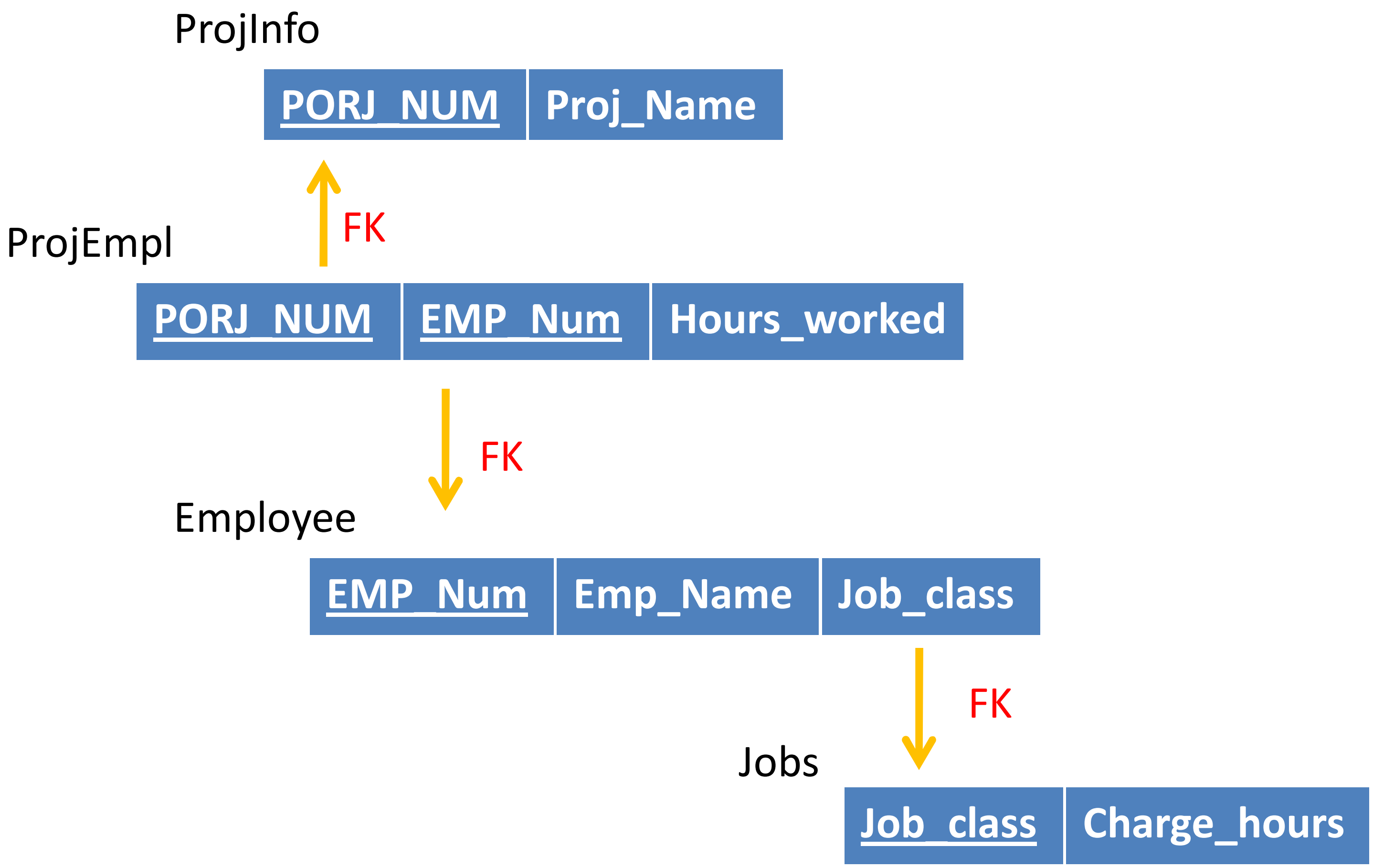
Table 1: PORJ_NUM | Proj_Name

Table 2: EMP_Num | Emp_Name | Job_class

Table 3: PORJ_NUM | EMP_Num | Hours_worked

Table 4: Job_class | Charge_hours

Case 3: DB in 3NF



ProjInfo

PORJ_NUM	Proj_Name

FK

ProjEmpl

PORJ_NUM	EMP_Num	Hours_worked

FK

Employee

EMP_Num	Emp_Name	Job_class

FK

Jobs

Job_class	Charge_hours

Case Study 4

- Perform Normalization for the Example in Slide # 4.

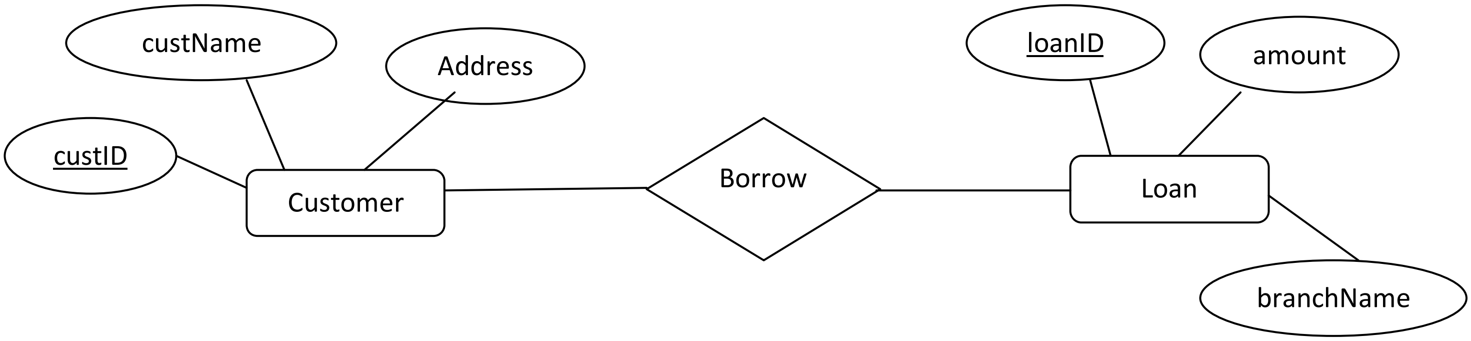
- Plain Table is

Student_id	StudentName	Major	college	collegeLocation	classID	className	ProfessorID	Professor Name
999-40-9876	Ahmad	Math	Science	UJ	Math101	Discrete Math	111-2231	XYZ
999-40-9876	Ahmad	Math	Science	UJ	Skills101	Computer Skills	111-2124	YZR
999-40-9000	Salem	CIS	KASIT	UJ	CIS322	DB	111-2222	URT

- Solution on Slide # 4

Sample Database applied in class (slides of CH 3) for testing purposes (only!) by Dr. Raja Alomari

For a quick lightweight sql server. Install Mysql server from www.mysql.com



<u>Schema reduction:</u>

1- Table 1: customer(**<u>custID</u>**, custName, Address)
2- Table 2: loan(<u>loanID</u>, amount, branchName)
3- Table 3: borrow(<u>custID, loanID</u>)
   NOTE: <u>1. Foreign Key:</u> borrow.custID **TO** customer.custID
   <u>2. Foreign Key:</u> borrow.loanID **TO** loan.loanID

<u>SQL statements to create the database (DDL):</u>

Create table customer(custID char(10), custName char(50) Not Null, address char(100), primary key(custID)) ;

Create table loan(loanID char(6), amount float Not Null, branchName char(50) Not null, primary key(loanID)) ;

Create table borrow (custID char(10), loanID char(6), primary key(custID, loanID), foreign key(custID) references customer(custID) on delete cascade, foreign key(loanID) references loan (loanID) on delete cascade) ;

<u>SQL statements to insert some data:</u>

1. Customer Table

Insert into customer values(‘C0001’,’Ahmad’,’Amman POBox 23134’);

Insert into customer values(‘C0002’,’Ahmad2’,’Amman POBox 342’);

Insert into customer values(‘C0003’,’Ahmad3’,’Amman POBox 342’);

Insert into customer values(‘C0004’,’Ahmad4’,’Amman POBox 342’);

2. Loan table

```
Insert into loan values(‘L0001’,20000,’JU Branch’);

Insert into loan values(‘L0002’,10000,’JU Branch’);

Insert into loan values(‘L0003’,1000,’JU Branch’);

Insert into loan values(‘L0004’,5000,’Aqaba Branch’);

Insert into loan values(‘L0005’,3000,’Aqaba Branch’);
```

3. Borrow: the transactions.

```
Insert into borrow  values(‘C0001’,’L0001’);

Insert into borrow  values(‘C0002’,’L0001’);

Insert into borrow  values(‘C0002’,’L0003’);
```

Sample queries:

```
Select * from customer;

Select * from loan;

Select customerName from customer, borrow where customer.custID = borrow.custID;

(Select * from customer) union (select * from borrow);

Select * from customer, loan;

Select distinct custName from customer, loan, borrow where customer.custID = borrow.custID and
borrow.loanID = loan.loanID and amount >10000;
```

Good SQL link:

http://www.pantz.org/software/mysql/mysqlcommands.html

This image cannot currently be displayed.

Chapter 3: Introduction to SQL

Database System Concepts, 6th Ed.

Chapter 3: Introduction to SQL

- Overview of the SQL Query Language
- Data Definition
- Basic Query Structure
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86, SQL-89, SQL-92
  - SQL:1999, SQL:2003, SQL:2008
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.

Data Definition Language

The SQL **data-definition language (DDL)** allows the specification of information about relations, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- And as we will see later, also other information such as
  - The set of indices to be maintained for each relations.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.

Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length *n.*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n.*
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.
- More are covered in Chapter 4.

Create Table Construct

- An SQL relation is defined using the **create table** command:

  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - r is the name of the relation
  - each A_i is an attribute name in the schema of relation r
  - D_i is the data type of values in the domain of attribute A_i

- Example:

  create table *instructor* (
  ID **char**(5),
  name **varchar**(20) **not null,**
  dept_name **varchar**(20),
  salary **numeric**(8,2))

- **insert into** *instructor* **values** ('10211', 'Smith', 'Biology', 66000);
- **insert into** *instructor* **values** ('10211', null, 'Biology', 66000);

Integrity Constraints in Create Table

- **not null**
- **primary key** (A_1, ..., A_n)
- **foreign key** (A_m, ..., A_n) **references** *r*

Example: Declare *ID* as the primary key for *instructor*

.

```
create table instructor (
    ID          char(5),
    name        varchar(20) not null,
    dept_name   varchar(20),
    salary      numeric(8,2),
    primary key (ID),
    foreign key (dept_name) references department)
```

primary key declaration on an attribute automatically ensures **not null**

And a Few More Relation Definitions

-
  ```
  create table student (
          ID              varchar(5),
          name            varchar(20) not null,
          dept_name       varchar(20),
          tot_cred        numeric(3,0),
          primary key (ID),
          foreign key (dept_name) references department) );
  ```

-
  ```
  create table takes (
          ID              varchar(5),
          course_id       varchar(8),
          sec_id          varchar(8),
          semester        varchar(6),
          year            numeric(4,0),
          grade           varchar(2),
          primary key (ID, course_id, sec_id, semester, year),
          foreign key (ID) references  student,
          foreign key (course_id, sec_id, semester, year) references section );
  ```

  - Note: *sec_id* can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester

And more still

-
  ```
  create table course (
          course_id       varchar(8) primary key,
          title           varchar(50),
          dept_name       varchar(20),
          credits         numeric(2,0),
          foreign key (dept_name) references department) );
  ```
  - Primary key declaration can be combined with attribute declaration as shown above

Drop and Alter Table Constructs

- **drop table** *student*
  - Deletes the table and its contents
- **delete from** *student*
  - Deletes all contents of table, but retains table
- **alter table**
  - **alter table** *r* **add** *A D*
    - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
    - All tuples in the relation are assigned *null* as the value for the new attribute.
  - **alter table** *r* **drop** *A*
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases

Basic Query Structure

- The SQL **data-manipulation language (DML)** provides the ability to query information, and insert, delete and update tuples
- A typical SQL query has the form:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.
- The result of an SQL query is a relation.

The select Clause

- The **select** clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:

  select *name*
  from *instructor*

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g. *Name* ≡ *NAME* ≡ *name*
  - Some people use upper case wherever we use bold font.

The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select**.**
- Find the names of all departments with instructor, and remove duplicates

```
select distinct dept_name
from instructor
```

- The keyword **all** specifies that duplicates not be removed.

```
select all dept_name
from instructor
```

The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

  select *
  from *instructor*

- The **select** clause can contain arithmetic expressions involving the operation, +, –, ∗, and /, and operating on constants or attributes of tuples.
- The query:

  select *ID, name, salary/12*
  from *instructor*

  would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept with salary > 80000

```
select name
from instructor
where dept_name = ‘Comp. Sci.'  and salary > 80000
```

- Comparison results can be combined using the logical connectives **and, or,** and **not.**
- Comparisons can be applied to results of arithmetic expressions.

The from Clause

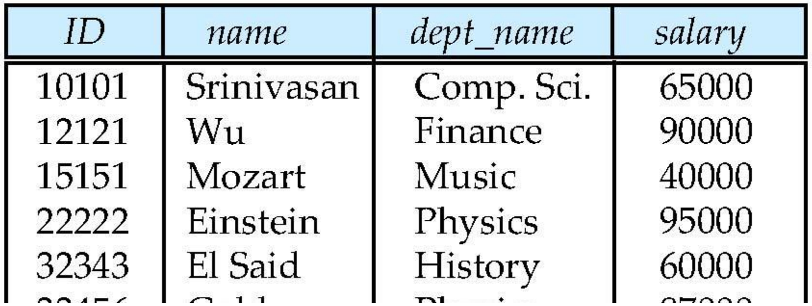
- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

  select *
  from *instructor, teaches*

  - generates every possible instructor – teaches pair, with all attributes from both relations
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra)

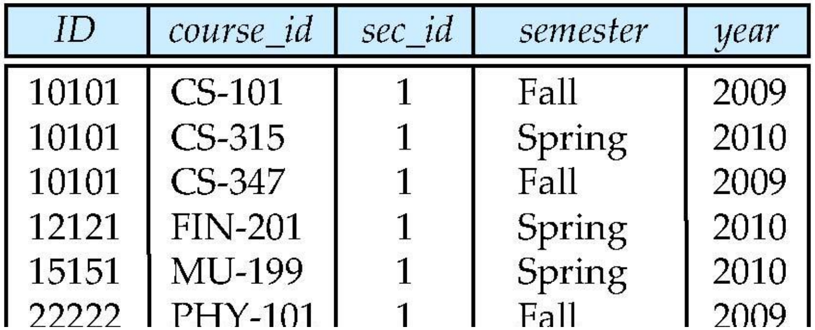
Cartesian Product: *instructor X teaches*

instructor

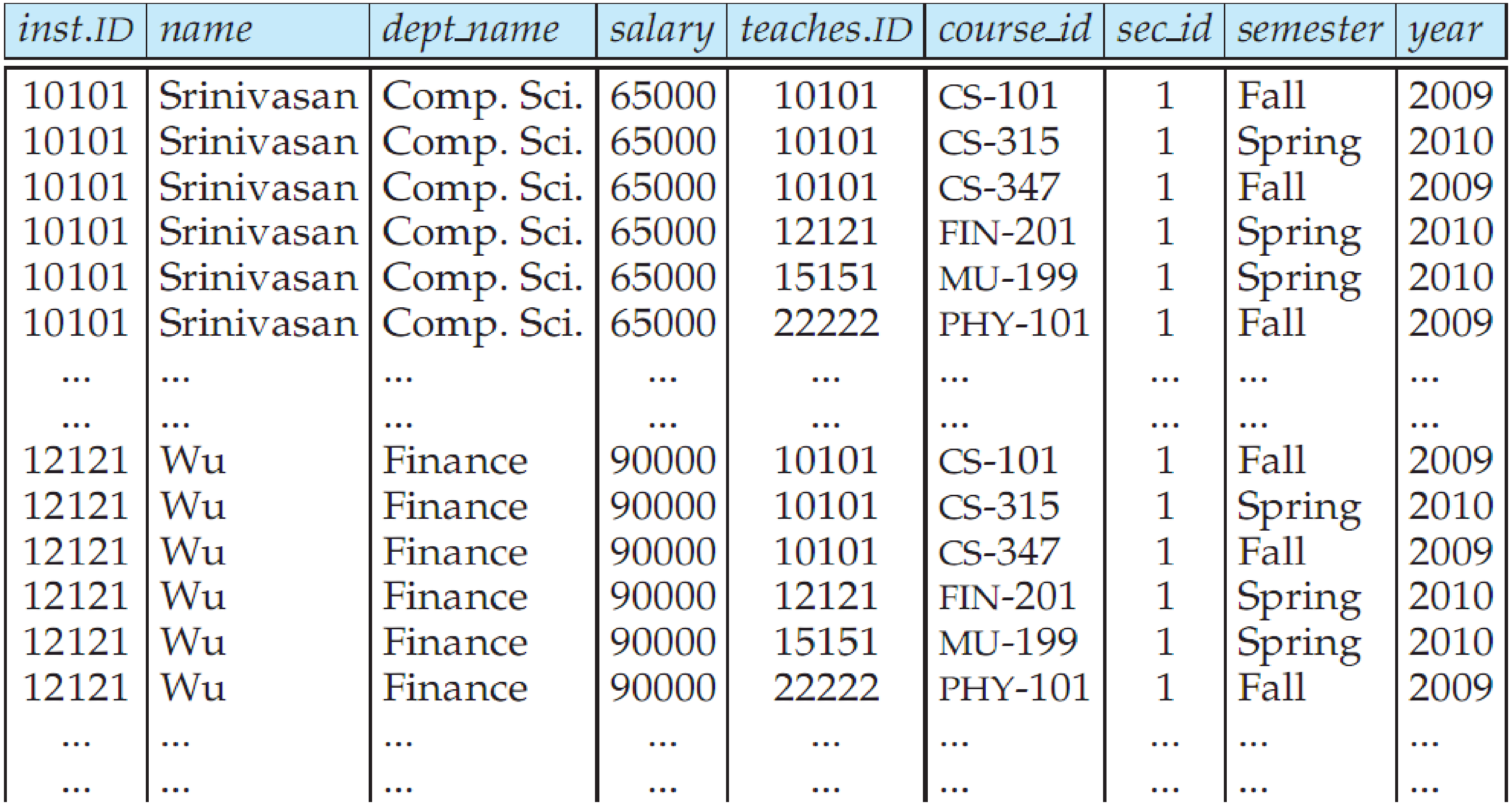


ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

teaches



ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009



inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

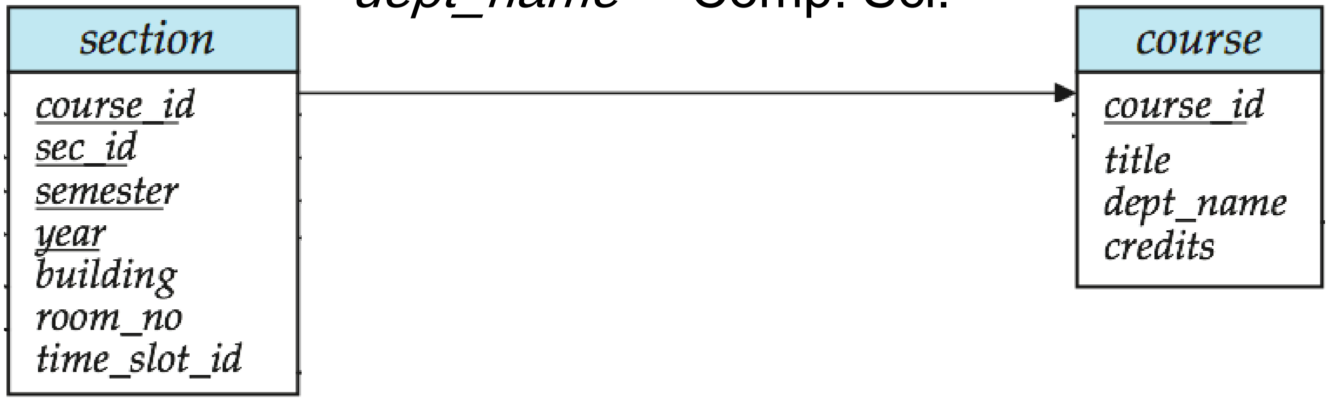
Joins

- For all instructors who have taught some course, find their names and the course ID of the courses they taught.

```
select name, course_id
 from instructor, teaches
 where   instructor.ID = teaches.ID
```

- Find the course ID, semester, year and title of each course offered by the Comp. Sci. department

```
select section.course_id, semester, year, title
 from section, course
 where   section.course_id = course.course_id  and
          dept_name = ‘Comp. Sci.'
```

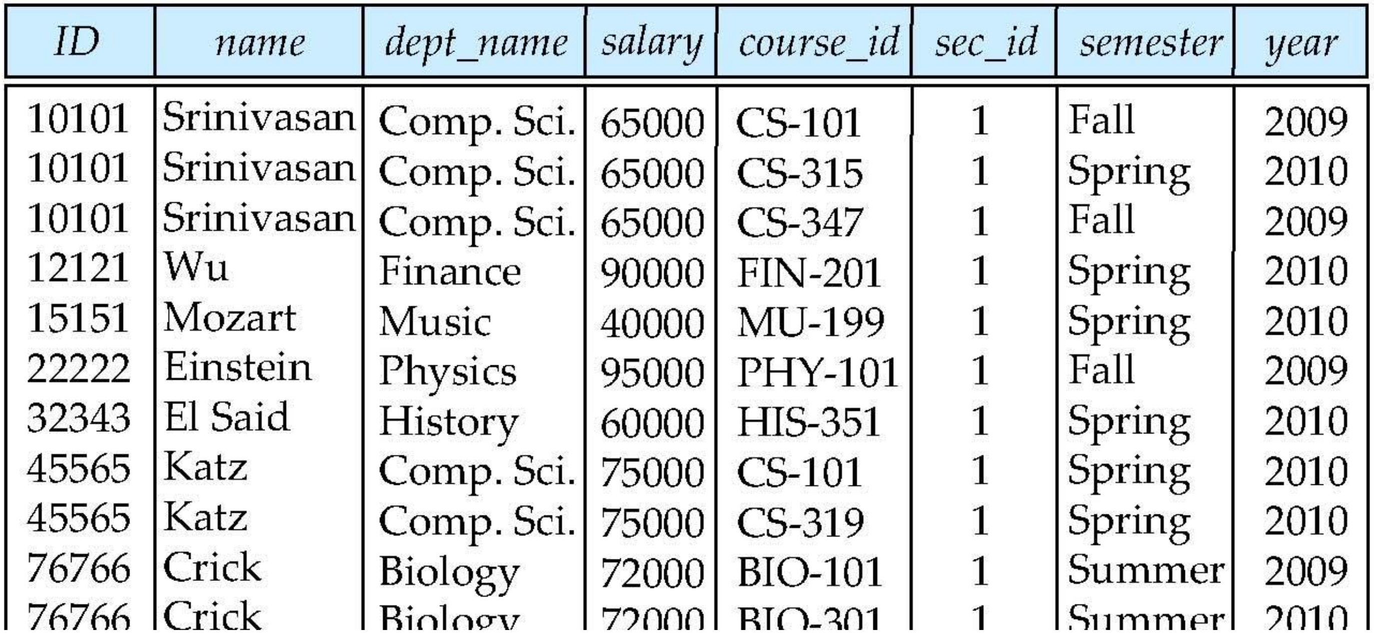


Try Writing Some Queries in SQL

- Suggest queries to be written…..

Natural Join

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column
- **select** *
  from *instructor* **natural join** *teaches*;



ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010

Natural Join Example

- List the names of instructors along with the course ID of the courses that they taught.

  -
    ```
    select name, course_id
    from instructor, teaches
    where instructor.ID = teaches.ID;
    ```

  -
    ```
    select name, course_id
    from instructor natural join teaches;
    ```

Natural Join (Cont.)

- Danger in natural join: beware of unrelated attributes with same name which get equated incorrectly
- List the names of instructors along with the the titles of courses that they teach
  - Incorrect version (makes course.dept_name = instructor.dept_name)
    - **select** *name*, *title*
      from *instructor* **natural join** *teaches* **natural join** *course*;
  - Correct version
    - **select** *name*, *title*
      from *instructor* **natural join** *teaches*, *course*
      where *teaches*.*course_id* = *course*.*course_id*;
  - Another correct version
    - **select** *name*, *title*
      from (*instructor* **natural join** *teaches*)
      join *course* **using**(*course_id*);

The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

  old-name **as** *new-name*

- E.g.

  -
    ```
    select ID, name, salary/12 as monthly_salary
    from instructor
    ```

- Find the names of all instructors who have a higher salary than some instructor in ‘Comp. Sci’.

  -
    ```
    select distinct T. name
    from instructor as T, instructor as S
    where T.salary > S.salary and S.dept_name = ‘Comp. Sci.’
    ```

- Keyword **as** is optional and may be omitted

  instructor **as** *T* ≡ *instructor T*

  - Keyword **as** must be omitted in Oracle

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator "like" uses patterns that are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.
- Find the names of all instructors whose name includes the substring "dar".

```
 select name
from instructor
where name like '%dar%'
```

- Match the string "100 %"

```
like '100 \%'  escape  '\'
```

String Operations (Cont.)

- Patters are case sensitive.
- Pattern matching examples:
  - ‘Intro%’ matches any string beginning with “Intro”.
  - ‘%Comp%’ matches any string containing “Comp” as a substring.
  - ‘_ _ _’ matches any string of exactly three characters.
  - ‘_ _ _ %’ matches any string of at least three characters.

- SQL supports a variety of string operations such as
  - concatenation (using “||”)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

  select distinct *name*
  from *instructor*
  order by *name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name* **desc**
- Can sort on multiple attributes
  - Example: **order by** *dept_name, name*

Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)

  -
    ```
    select name
      from instructor
      where salary between 90000 and 100000
    ```
- Tuple comparison

  -
    ```
    select name, course_id
    from instructor, teaches
    where (instructor.ID, dept_name) = (teaches.ID, ’Biology’);
    ```

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2:

1. $\boldsymbol{\sigma_\theta(r_1)}$**:** If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections σ_{θ},, then there are c_1 copies of t_1 in σ_θ (r_1).
2. $\boldsymbol{\Pi_A(r)}$**:** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.
3. $\boldsymbol{r_1}$ **x** $\boldsymbol{r_2}$ **:** If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are c_1 x c_2 copies of the tuple t_1. t_2 in r_1 x r_2

Duplicates (Cont.)

- Example: Suppose multiset relations r_1 (A, B) and r_2 (C) are as follows:

  r_1 = {(1, a) (2,a)} r_2 = {(2), (3), (3)}

- Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1)$ x r_2 would be

  {(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)}

- SQL duplicate semantics:

  select A_1, A_2, ..., A_n
  from r_1, r_2, ..., r_m
  where P

  is equivalent to the *multiset* version of the expression:

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$$

Set Operations

- Find courses that ran in Fall 2009 or in Spring 2010

```
(select course_id from section where sem = 'Fall' and year = 2009)
 union
(select course_id from section where sem = 'Spring' and year = 2010)
```

- Find courses that ran in Fall 2009 and in Spring 2010

```
(select course_id from section where sem = 'Fall' and year = 2009)
 intersect
(select course_id from section where sem = 'Spring' and year = 2010)
```

- Find courses that ran in Fall 2009 but not in Spring 2010

```
(select course_id from section where sem = 'Fall' and year = 2009)
 except
(select course_id from section where sem = 'Spring' and year = 2010)
```

Set Operations

- Set operations **union, intersect,** and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

  Suppose a tuple occurs m times in r and n times in s, then, it occurs:
  - $m + n$ times in r **union all** s
  - min(m,n) times in r **intersect all** s
  - max(0, $m - n$) times in r **except all** s

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null*.*

    select *name*
    from *instructor*
    where *salary* **is null**

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
  - Example*: 5 < null   or   null <> null   or   null = null*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*)   = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown) = unknown*
  - AND: *(true* **and** *unknown) = unknown,*
    (false **and** *unknown) = false,*
    (unknown **and** *unknown) = unknown*
  - NOT*: (***not** *unknown) = unknown*
  - "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

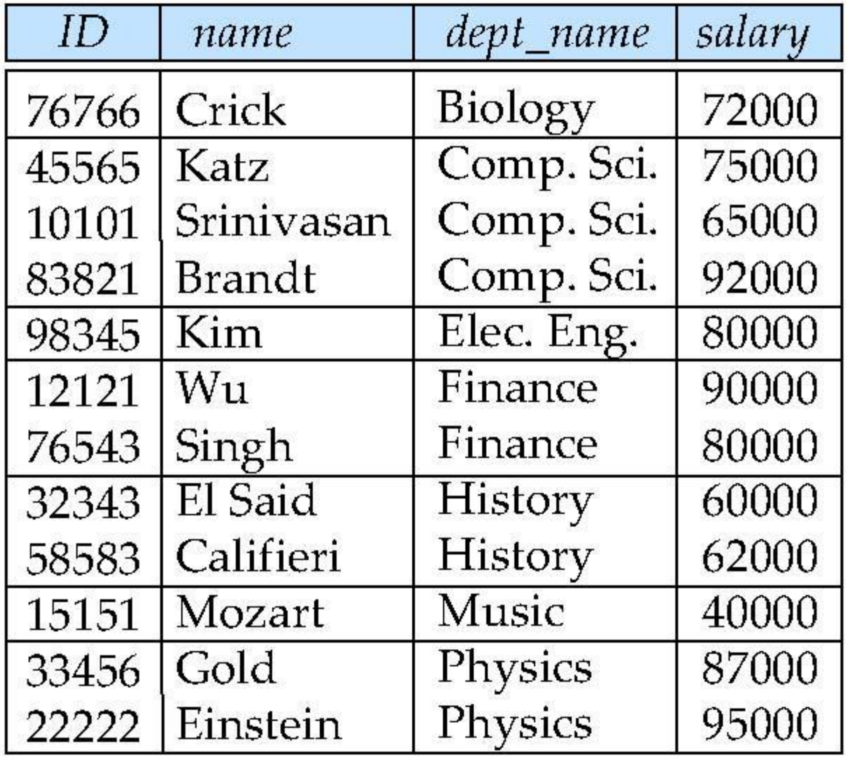
avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate Functions (Cont.)

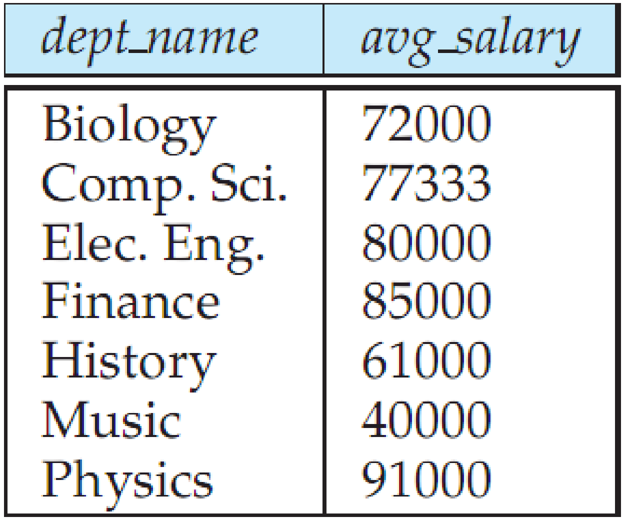
- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
from *instructor*
where *dept_name*= ’Comp. Sci.’;
- Find the total number of instructors who teach a course in the Spring 2010 semester
  - **select count** (**distinct** *ID*)
from *teaches*
where *semester* = ’Spring’ **and** *year* = 2010
- Find the number of tuples in the *course* relation
  - **select count** (*)
from *course*;

Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;
  - Note: departments with no instructor will not appear in result



ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000



dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list
  - /* erroneous query */
    select *dept_name*, *ID*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;

Aggregate Functions – Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

```
select dept_name, avg (salary)
from instructor
group by dept_name
having avg (salary) > 42000;
```

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Null Values and Aggregates

- Total all salaries

  select sum (*salary*)
  from *instructor*

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Example Query

- Find courses offered in Fall 2009 and in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
           course_id in (select course_id
                         from section
                         where semester = 'Spring' and year= 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
           course_id  not in (select course_id
                              from section
                              where semester = 'Spring' and year= 2010);
```

Example Query

- Find the total number of (distinct) studentswho have taken course sections taught by the instructor with *ID* 10101

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in
                          (select course_id, sec_id, semester, year
                      from teaches
                      where teaches.ID= 10101);
```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept_name = ’Biology’;
```

- Same query using > **some** clause

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept_name = ’Biology’);
```

Definition of Some Clause

- F <comp> **some** $r \Leftrightarrow \exists\ t \in r$ such that (F <comp> t)
  Where <comp> can be: $<,\ \leq,\ >,\ =,\ \neq$

$(5 < \textbf{some} \begin{array}{|c|}\hline 0\\ \hline 5\\ \hline 6\\ \hline\end{array}) = \text{true}$ (read: 5 < some tuple in the relation)

$(5 < \textbf{some} \begin{array}{|c|}\hline 0\\ \hline 5\\ \hline\end{array}) = \text{false}$

$(5 = \textbf{some} \begin{array}{|c|}\hline 0\\ \hline 5\\ \hline\end{array}) = \text{true}$

$(5 \neq \textbf{some} \begin{array}{|c|}\hline 0\\ \hline 5\\ \hline\end{array}) = \text{true (since } 0 \neq 5)$

(= **some**) $\equiv$ **in**

However, ($\neq$ **some**) $\not\equiv$ **not in**

Example Query

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept_name = ’Biology’);
```

Definition of all Clause

- F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> *t)*

$$(5 < \textbf{all} \ \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}\ ) = \text{false}$$

$$(5 < \textbf{all} \ \begin{array}{|c|}\hline 6 \\ \hline 10 \\ \hline \end{array}\ ) = \text{true}$$

$$(5 = \textbf{all} \ \begin{array}{|c|}\hline 4 \\ \hline 5 \\ \hline \end{array}\ ) = \text{false}$$

$$(5 \neq \textbf{all} \ \begin{array}{|c|}\hline 4 \\ \hline 6 \\ \hline \end{array}\ ) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$$

($\neq$ **all**) $\equiv$ **not in**

However, (= **all**) $\not\equiv$ **in**

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Not Exists

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology')
                  except
                   (select T.course_id
                    from takes as T
                    where S.ID = T.ID));
```

- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- *Note:* Cannot write this query using = **all** and its variants

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
  - (Evaluates to “true” on an empty set)
- Find all courses that were offered at most once in 2009

```
select T.course_id
 from course as T
 where unique (select R.course_id
                    from section as R
                    where T.course_id= R.course_id
                              and R.year = 2009);
```

Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000.

```
select dept_name, avg_salary
 from (select dept_name, avg (salary) as avg_salary
           from instructor
           group by dept_name)
 where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
select dept_name, avg_salary
 from (select dept_name, avg (salary)
           from instructor
           group by dept_name)
           as dept_avg (dept_name,  avg_salary)
  where avg_salary > 42000;
```

With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as
    (select max(budget)
      from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- E.g.

```
select dept_name,
      (select count(*)
         from instructor
        where department.dept_name = instructor.dept_name)
      as num_instructors
 from department;
```

- E.g.

```
select name
from instructor
where  salary * 10 >
      (select budget  from department
        where department.dept_name = instructor.dept_name)
```

- Runtime error if subquery returns more than one result tuple

Modification of the Database

- Deletion of tuples from a given relation
- Insertion of new tuples into a given relation
- Updating values in some tuples in a given relation

Modification of the Database – Deletion

- Delete all instructors

```
delete from instructor
```

- Delete all instructors from the Finance department

```
delete from instructor
where dept_name= ’Finance’;
```

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

```
delete from instructor
  where dept_name in (select dept_name
                      from department
                      where building = ’Watson’);
```

Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

delete from *instructor*
where *salary*< (**select avg** (*salary*) **from** *instructor*);

- Problem: as we delete tuples from deposit, the average salary changes
- Solution used in SQL:
  1. First, compute **avg** salary and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – Insertion

- Add a new tuple to *course*

```
insert into course
        values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- or equivalently

```
insert into course (course_id, title, dept_name, credits)
        values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- Add a new tuple to *student* with *tot_creds* set to null

```
insert into student
        values ('3003', 'Green', 'Finance', null);
```

Insertion (Cont.)

- Add all instructors to the *student* relation with tot_creds set to 0

```
insert into student
     select ID, name, dept_name, 0
      from   instructor
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like

```
insert into table1 select * from table1
```

would cause problems, if *table1* did not have any primary key defined.

Modification of the Database – Updates

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others receive a 5% raise
  - Write two **update** statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 100000;
 update instructor
     set salary = salary * 1.05
     where salary <= 100000;
```

  - The order is important
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement

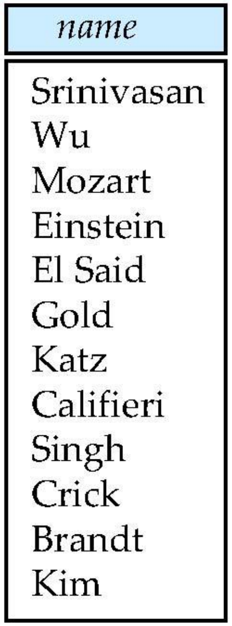
```
update instructor
     set salary = case
                        when salary <= 100000 then salary * 1.05
                        else salary * 1.03
                       end
```

This image cannot currently be displayed.

End of Chapter 3

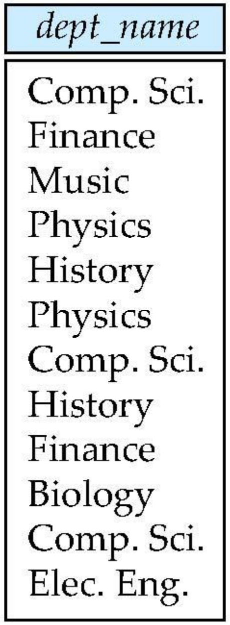
Database System Concepts, 6th Ed.

Figure 3.02



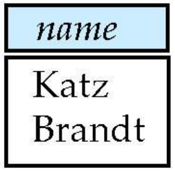
name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Figure 3.03



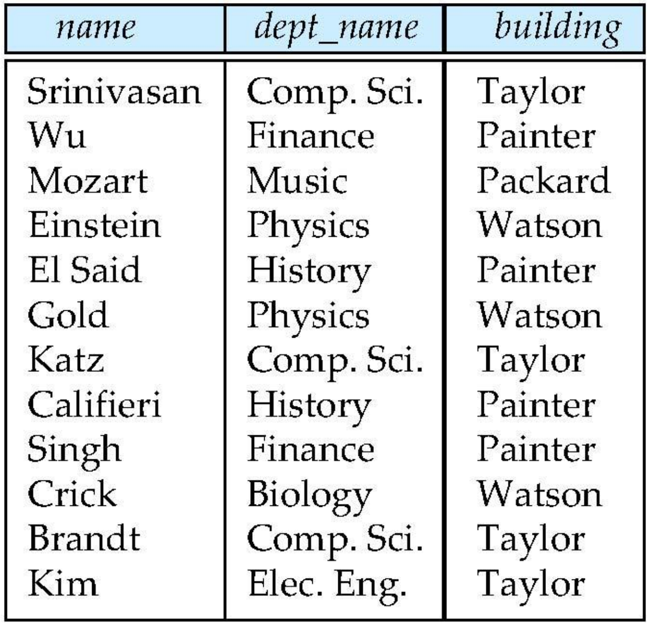
dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Figure 3.04



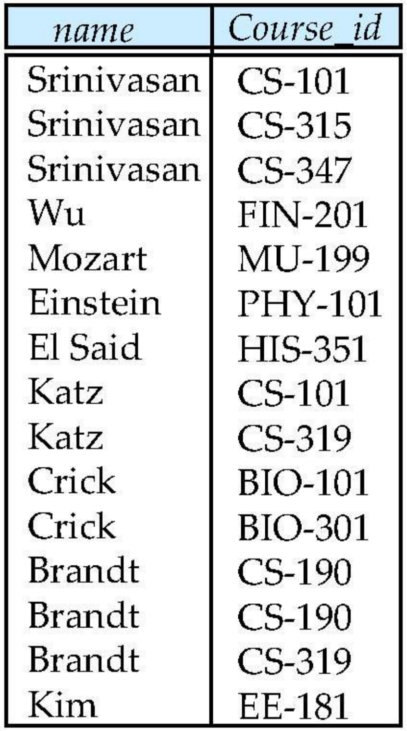
name
Katz
Brandt

Figure 3.05



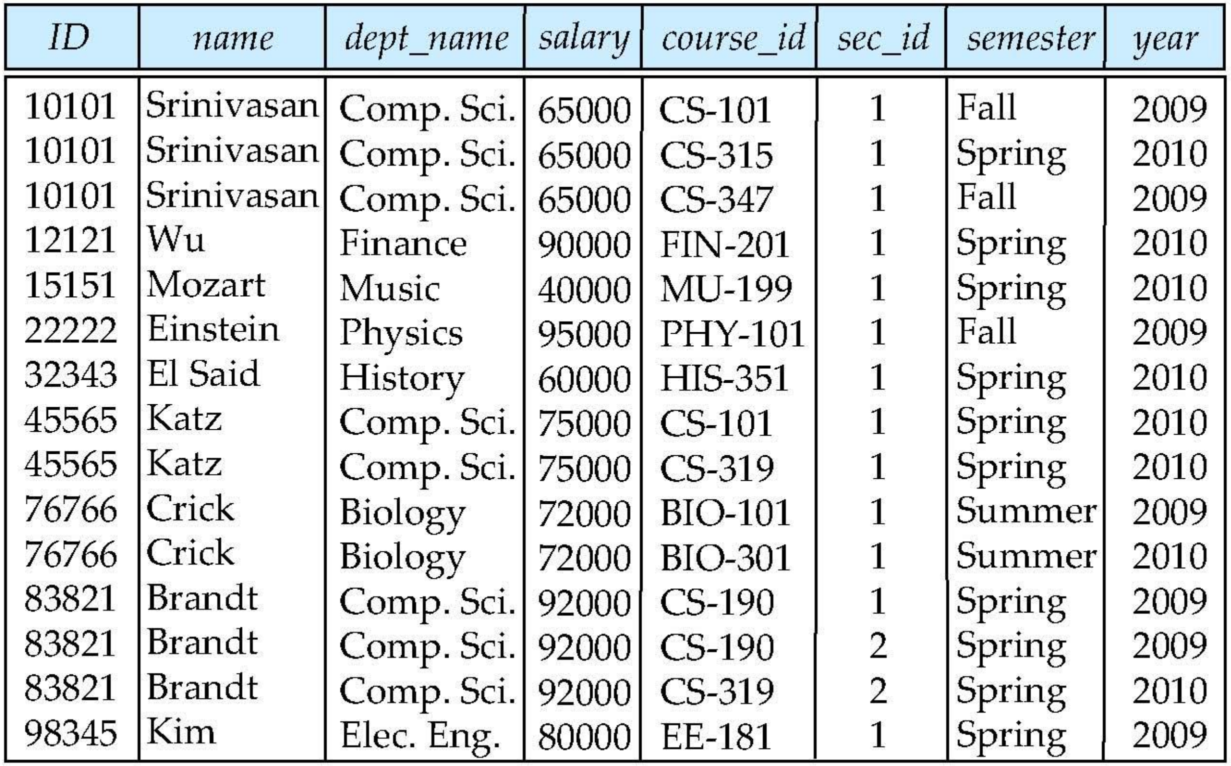
name	*dept_name*	*building*
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Califieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

Figure 3.07



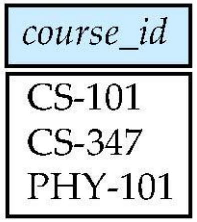
name	*Course_id*
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

Figure 3.08



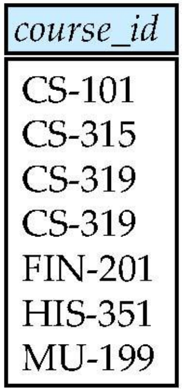
ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Figure 3.09



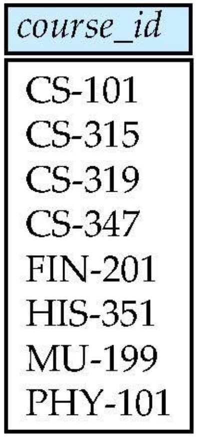
course_id
CS-101
CS-347
PHY-101

Figure 3.10



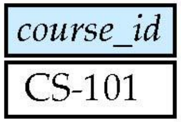
course_id
CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

Figure 3.11



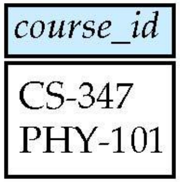
course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Figure 3.12



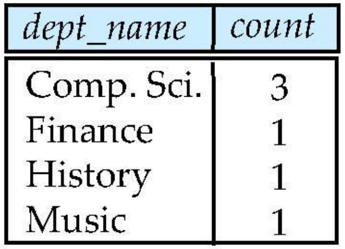
course_id
CS-101

Figure 3.13



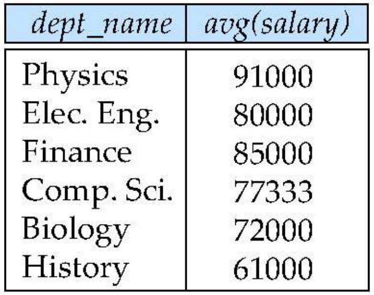
course_id
CS-347
PHY-101

Figure 3.16



dept_name	*count*
Comp. Sci.	3
Finance	1
History	1
Music	1

Figure 3.17



dept_name	*avg(salary)*
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Database Normalization

CIS 1902321

Database Normalization

- **Definition**
  - Optimizing table structures!
  - Removing duplicate data entries.

  - Accomplished by thoroughly investigating the various data types and their relationships with one another.

  - Follows a series of normalization “forms” or states.

Why Normalize?

- Improved speed.
- More efficient use of space.
- Increased data integrity.

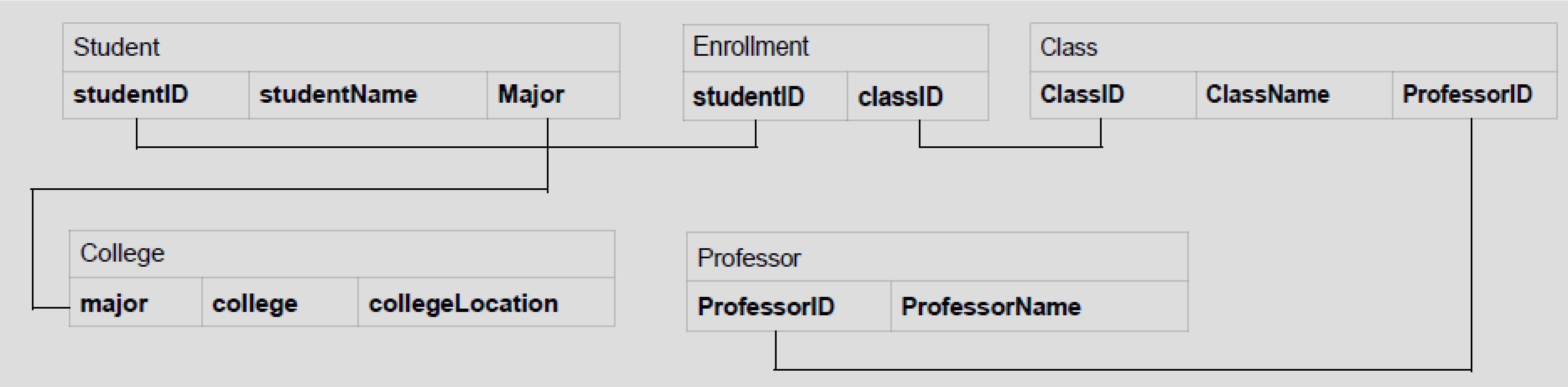
Example!

- Old Design

How ?

Student_id	StudentName	Major	college	collegeLocation	classID	className	ProfessorID	Professor Name
999-40-9876	Ahmad	Math	Science	UJ	Math101	Discrete Math	111-2231	XYZ
999-40-9876	Ahmad	Math	Science	UJ	Skills101	Computer Skills	111-2124	YZR
999-40-9000	Salem	CIS	KASIT	UJ	CIS322	DB	111-2222	URT

- New design (After Normalization)



Student

studentID	studentName	Major

Enrollment

studentID	classID

Class

ClassID	ClassName	ProfessorID

College

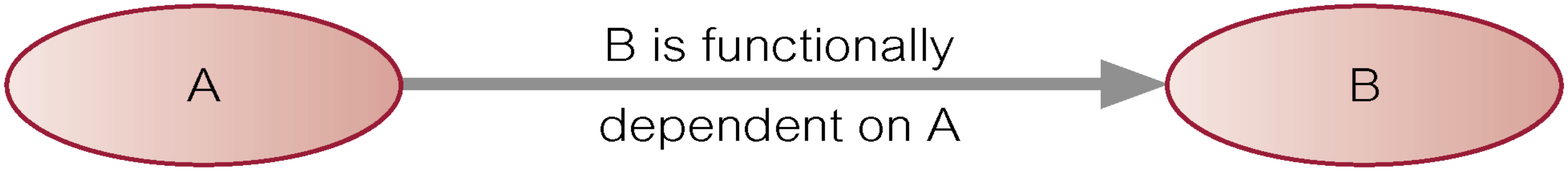
major	college	collegeLocation

Professor

ProfessorID	ProfessorName

Functional Dependency

Definition: *"relationship between or among attributes"*



- Attribute B is functionally dependent on A.

Functional Dependency: An Example!

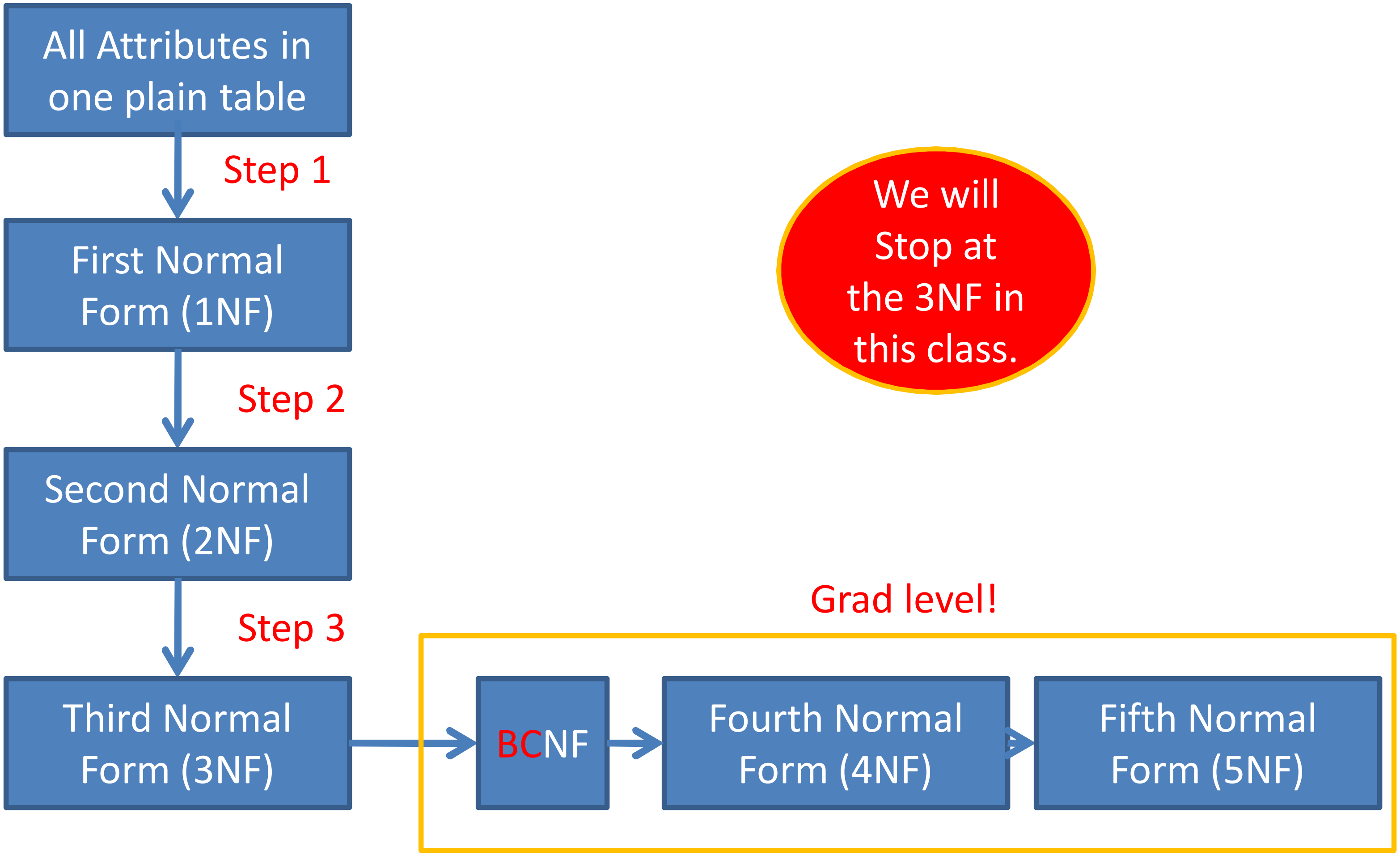
StudentID → StudentName

classID → className

(StudentID, classID) → Grade

StudentID	StudentName	ClassID	ClassName	Grade
St_1	Ahmad Salem	1902322	Database Design	A
St_1	Ahmad Salem	1902321	Data Security	B+
St_2	Osama Othman	1902322	Database Design	C+
St_3	Ameer Asem	1902322	Database Design	A

Normalization Steps



All Attributes in one plain table

Step 1

First Normal Form (1NF)

Step 2

Second Normal Form (2NF)

Step 3

Third Normal Form (3NF)

We will Stop at the 3NF in this class.

Grad level!

BCNF

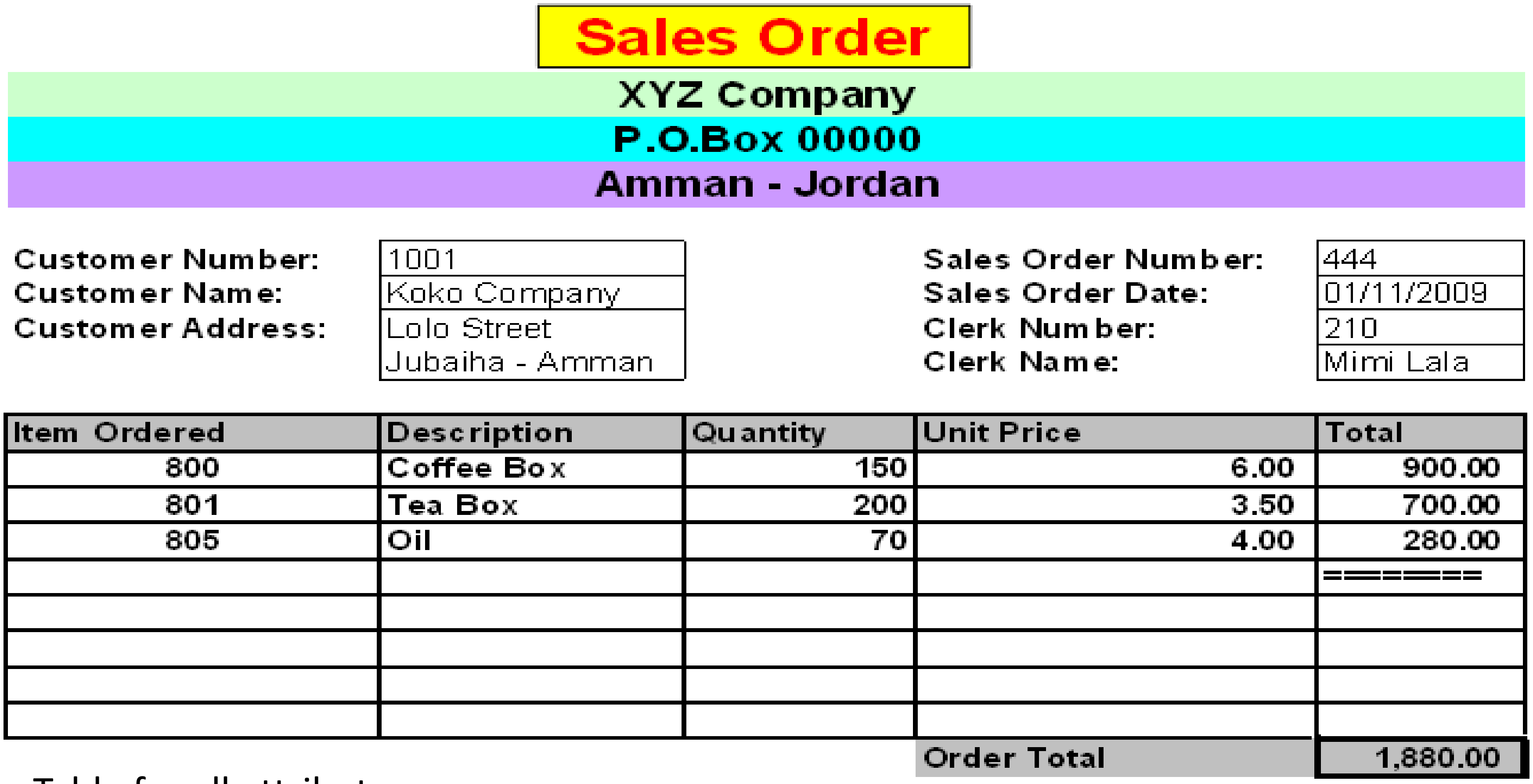
Fourth Normal Form (4NF)

Fifth Normal Form (5NF)

Normalization Steps

- Step1:
  - Elimination of repeated *groups of* data by creating separate tables of related data.
- Step 2:
  - Removal of partial dependencies.
- Step 3:
  - Removal of Transitive dependencies.

Case Study 1



Sales Order

XYZ Company

P.O.Box 00000

Amman - Jordan

Customer Number:	1001	Sales Order Number:	444
Customer Name:	Koko Company	Sales Order Date:	01/11/2009
Customer Address:	Lolo Street	Clerk Number:	210
	Jubaiha - Amman	Clerk Name:	Mimi Lala

Item Ordered	Description	Quantity	Unit Price	Total
800	Coffee Box	150	6.00	900.00
801	Tea Box	200	3.50	700.00
805	Oil	70	4.00	280.00
				========
			Order Total	1,880.00

- Table for all attributes

SalesOrderNo	Date	CustNo	CustName	CustAdd	ClerkNo	ClerkName	ItemNo	Desc	Qty	UnitPrice
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala	800, 801, 805	Coffee Box, Tea Box, Oil	150, 200, 70	6, 3.5, 4

Step 1: To 1NF

SalesOrderNo	Date	CustomerNo	CustomerName	CustomerAdd	ClerkNo	ClerkName	ItemNo	Description	Qty	Unit Price
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala	800, 801, 805	Coffee Box, Tea Box, Oil	150, 200, 70	6, 3.5, 4

Elimination of repeated *groups of* data by creating separate tables of related data.

Create a new table as follows to remove repeating groups:

SalesOrderNo, ItemNo, Description, Qty, UnitPrice

Original Table is left with:

SalesOrderNo, Date, CustomerNo, CustomerName, CustomerAdd, ClerkNo, ClerkName

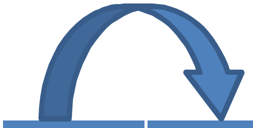
These two tables are a database in first normal form (1NF)

Step 2: To 2NF

Removal of partial dependencies.

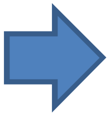
- Partial Dependency: is a functional dependency where an attribute is functionally dependent on only part of the primary key (for composite primary keys).
- Create a Table with the functionally dependent data and the part of the key on which it depends.

SalesOrderNo	Date	CustomerNo	CustomerName	CustomerAdd	ClerkNo	ClerkName
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala



Partial Dependency

SalesOrderNo	ItemNo	Description	Qty	UnitPrice
444	800	Coffee Box	6	150
444	801	Tea Box	3.5	200
444	805	Oil	4	70



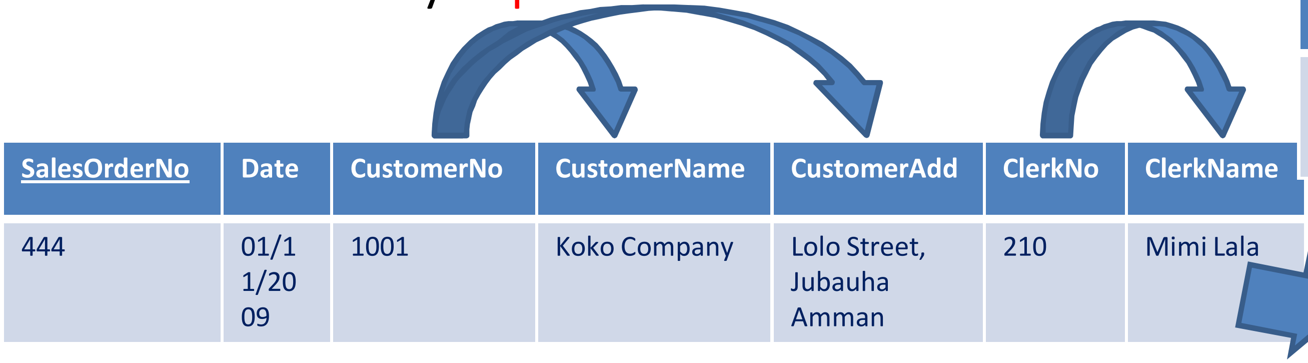
ItemNo	Description
800	Coffee Box
801	Tea Box
805	Oil

SalesOrderNo	ItemNo	Qty	UnitPrice
444	800	6	150
444	801	3.5	200
444	805	4	70

Step 3: To 3NF

Removal of Transitive dependencies.

- Transitive Dependency: is a functional dependency where an attribute is functionally dependent on an attribute other than the primary key. i.e., Its value is only indirectly determined by the primary key.
- Create a separate table containing the attribute and the fields that are functionally dependent on it.



SalesOrderNo	Date	CustomerNo	CustomerName	CustomerAdd	ClerkNo	ClerkName
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala

SalesOrderNo	Date	CustomerNo	ClerkNo
444	01/11/2009	1001	210

ClerkNo	ClerkName
210	Mimi Lala

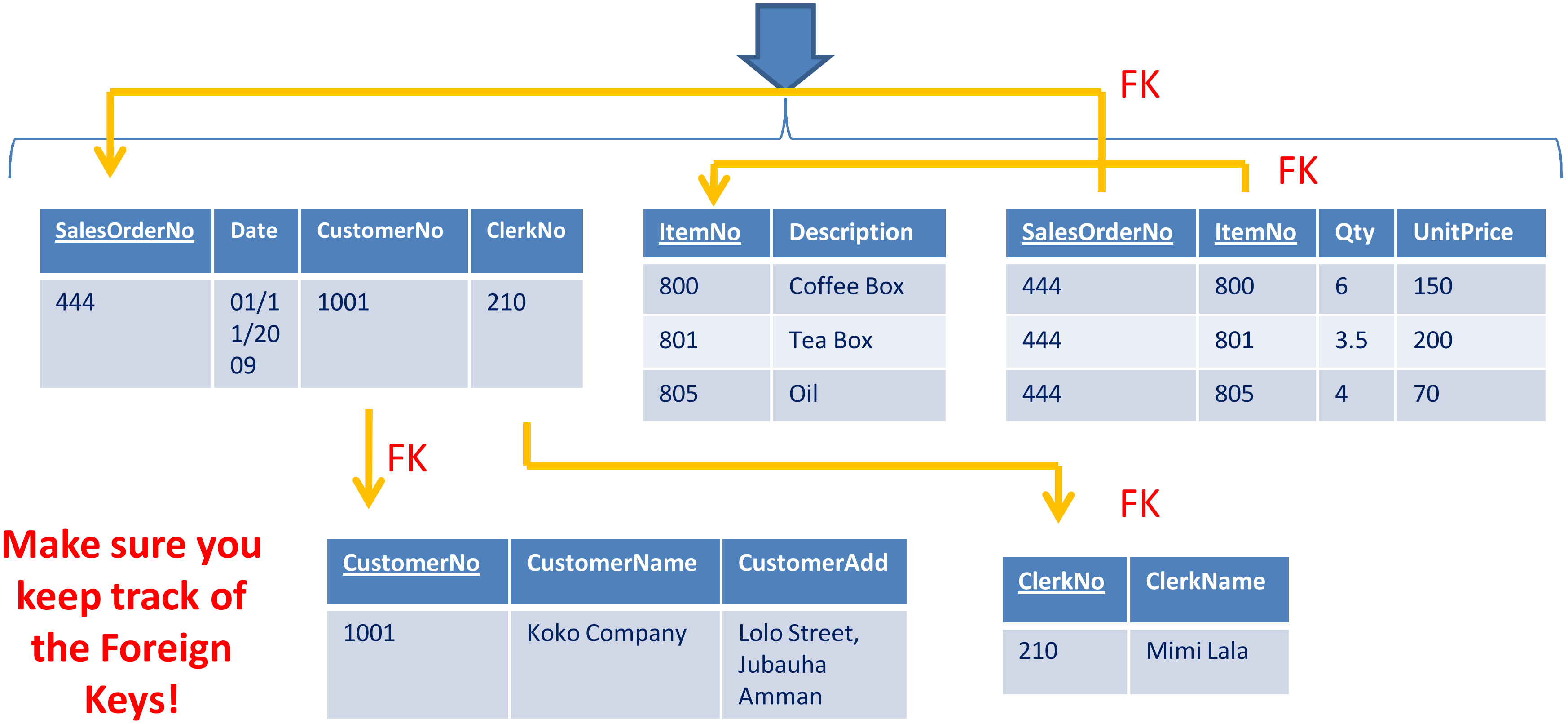
CustomerNo	CustomerName	CustomerAdd
1001	Koko Company	Lolo Street, Jubauha Amman

ItemNo	Description
800	Coffee Box
801	Tea Box
805	Oil

SalesOrderNo	ItemNo	Qty	UnitPrice
444	800	6	150
444	801	3.5	200
444	805	4	70

Completed DB in 3NF

SalesOrderNo	Date	CustomerNo	CustomerName	CustomerAdd	ClerkNo	ClerkName	ItemNo	Description	Qty	UnitPrice
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala	800, 801, 805	Coffee Box, Tea Box, Oil	6, 3.5, 4	150, 200, 70



SalesOrderNo	Date	CustomerNo	ClerkNo
444	01/11/2009	1001	210

ItemNo	Description
800	Coffee Box
801	Tea Box
805	Oil

SalesOrderNo	ItemNo	Qty	UnitPrice
444	800	6	150
444	801	3.5	200
444	805	4	70

CustomerNo	CustomerName	CustomerAdd
1001	Koko Company	Lolo Street, Jubauha Amman

ClerkNo	ClerkName
210	Mimi Lala

Make sure you keep track of the Foreign Keys!

Anomalies

- Insertion Anomaly – A record about an entity cannot be inserted into the table without first inserting information about another entity. e.g., Cannot enter a customer without a sales order.
- Deletion Anomaly – A record cannot be deleted without deleting a record about a related entity. e.g., delete a sales order without deleting all of the customer's information.
- Update Anomaly – Cannot update information without changing information in many places. e.g., To update customer information, it must be updated for each sales order the customer has placed

SalesOrderNo	Date	CustomerNo	CustomerName	CustomerAdd	ClerkNo	ClerkName
444	01/11/2009	1001	Koko Company	Lolo Street, Jubauha Amman	210	Mimi Lala
847	01/11/2010	1001	Koko Company	Lolo Street, Jubauha Amman	320	Yimi Kala

3NF Step

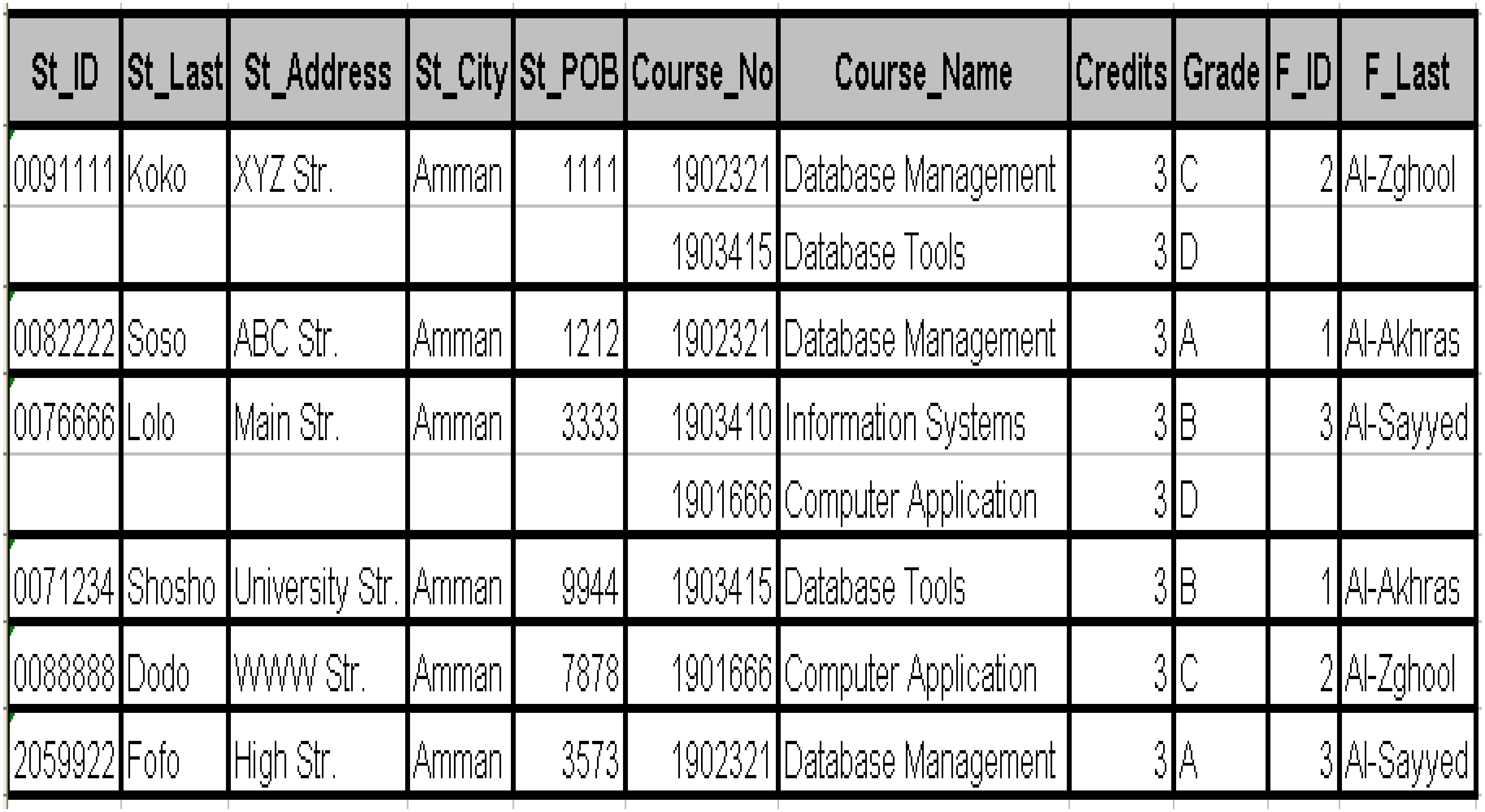
SalesOrderNo	Date	CustomerNo	ClerkNo
444	01/11/2009	1001	210
847	01/11/2010	1001	320

ClerkNo	ClerkName
210	Mimi Lala
320	Yimi Kala

CustomerNo	CustomerName	CustomerAdd
1001	Koko Company	Lolo Street, Jubauha Amman

Case Study 2

Student



St_ID	St_Last	St_Address	St_City	St_POB	Course_No	Course_Name	Credits	Grade	F_ID	F_Last
0091111	Koko	XYZ Str.	Amman	1111	1902321	Database Management	3	C	2	Al-Zghool
					1903415	Database Tools	3	D		
0082222	Soso	ABC Str.	Amman	1212	1902321	Database Management	3	A	1	Al-Akhras
0076666	Lolo	Main Str.	Amman	3333	1903410	Information Systems	3	B	3	Al-Sayyed
					1901666	Computer Application	3	D		
0071234	Shosho	University Str.	Amman	9944	1903415	Database Tools	3	B	1	Al-Akhras
0088888	Dodo	WWW Str.	Amman	7878	1901666	Computer Application	3	C	2	Al-Zghool
2059922	Fofo	High Str.	Amman	3573	1902321	Database Management	3	A	3	Al-Sayyed

Exercise Solution

Baseline Table contains:

St_ID, St_Last, St_Address, St_City, St_POB, Course_No, Course_Name, Credits, Grade, F_ID, F_Last

First Normal Form

In First Normal Form:

(1) Primary keys are set

(2) No repeating groups

Produced Tables:

StInfo: St_ID, St_Last, St_Address, St_City, St_POB

StCourses: St_ID, Course_No, Course_Name, Credits, Grade, F_ID, F_Last

Second Normal Form

In Second Normal Form:

(*) No partial dependency …

Produced Tables:

StInfo: St_ID, St_Last, St_Address, St_City, St_POB

StCourses: Course_No, Course_Name, Credits

StCourses: St_ID, Course_No, Grade, F_ID, F_Last

Third Normal Form

In Third Normal Form:

(*) No transitive dependency …

Produced Tables:

StInfo: St_ID, St_Last, St_Address, St_City, St_POB

StCourses: Course_No, Course_Name, Credits

StCourses: F_ID, F_Last

StCourses: St_ID, Course_No, Grade, F_ID

Case Study 3

PORJ_NUM	Proj_Name	EMP_Num	Emp_Name	Job_class	Charge_hours	Hours_worked
123	UJ-Proj	17	XYZ	Engineer	JD 100	35
		90	UHY	Designer	JD 150	21
		50	OKI	Admin	JD 400	39
389	KASIT-Proj	90	UHY	Designer	JD 150	17
		65	PLU	Worker	JD 75	40
		83	RGT	Worker	JD 75	40
		24	KUH	Engineer	JD 100	30

Candidate key is the Proj_NUM

Case 3: To 1NF

PORJ_NUM	Proj_Name	EMP_Num	Emp_Name	Job_class	Charge_hours	Hours_worked
123	UJ-Proj	17	XYZ	Engineer	JD 100	35
		90	UHY	Designer	JD 150	21
		50	OKI	Admin	JD 400	39
389	KASIT-Proj	90	UHY	Designer	JD 150	17
		65	PLU	Worker	JD 75	40
		83	RGT	Worker	JD 75	40
		24	KUH	Engineer	JD 100	30

Elimination of repeated Groups

Table 1:

PORJ_NUM	Proj_Name

Table 2:

PORJ_NUM	EMP_Num	Emp_Name	Job_class	Charge_hours	Hours_worked

Case 3: To 2NF

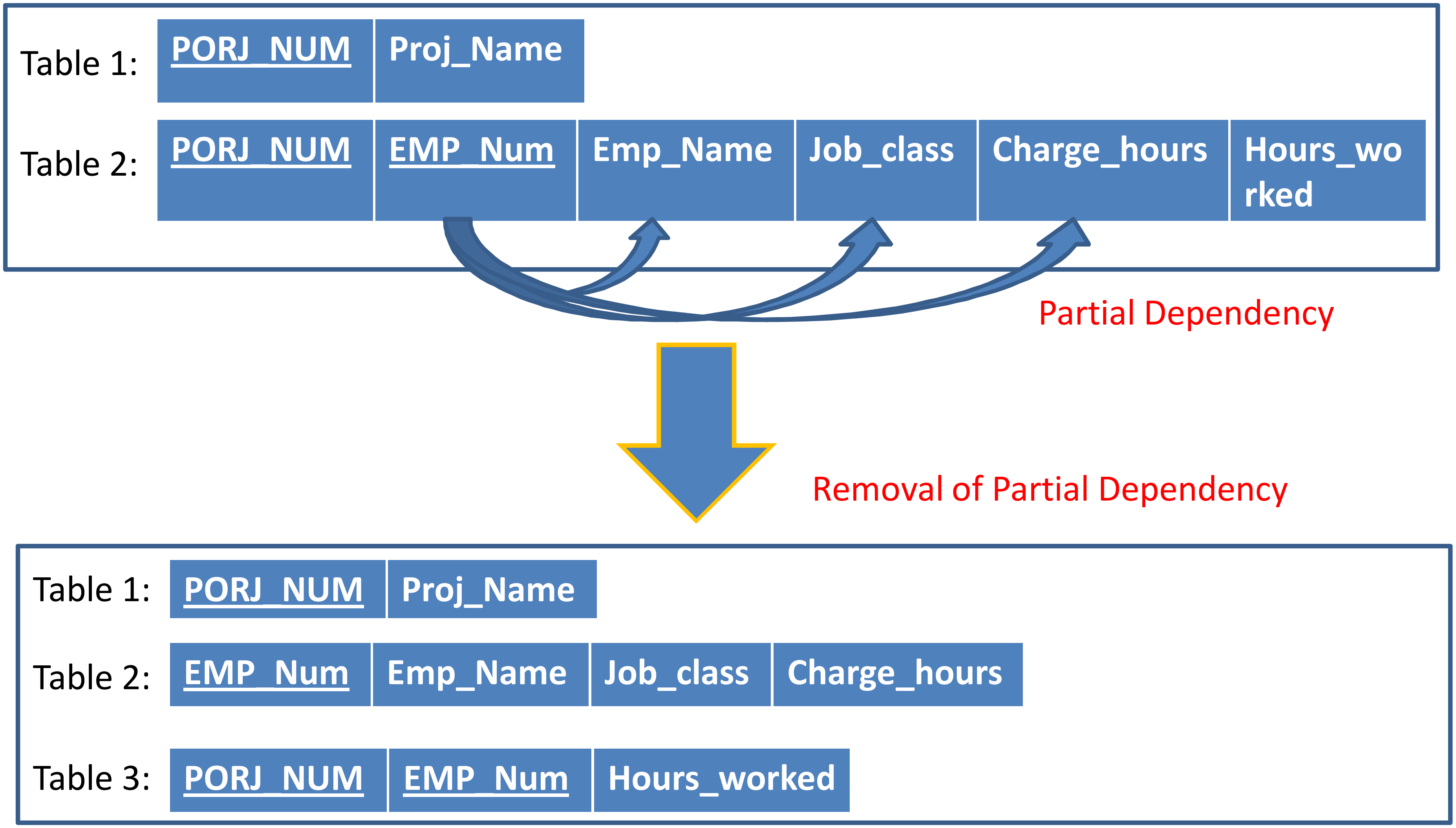


Table 1:

PORJ_NUM	Proj_Name

Table 2:

PORJ_NUM	EMP_Num	Emp_Name	Job_class	Charge_hours	Hours_worked

Table 1:

PORJ_NUM	Proj_Name

Table 2:

EMP_Num	Emp_Name	Job_class	Charge_hours

Table 3:

PORJ_NUM	EMP_Num	Hours_worked

Case 3: To 3NF

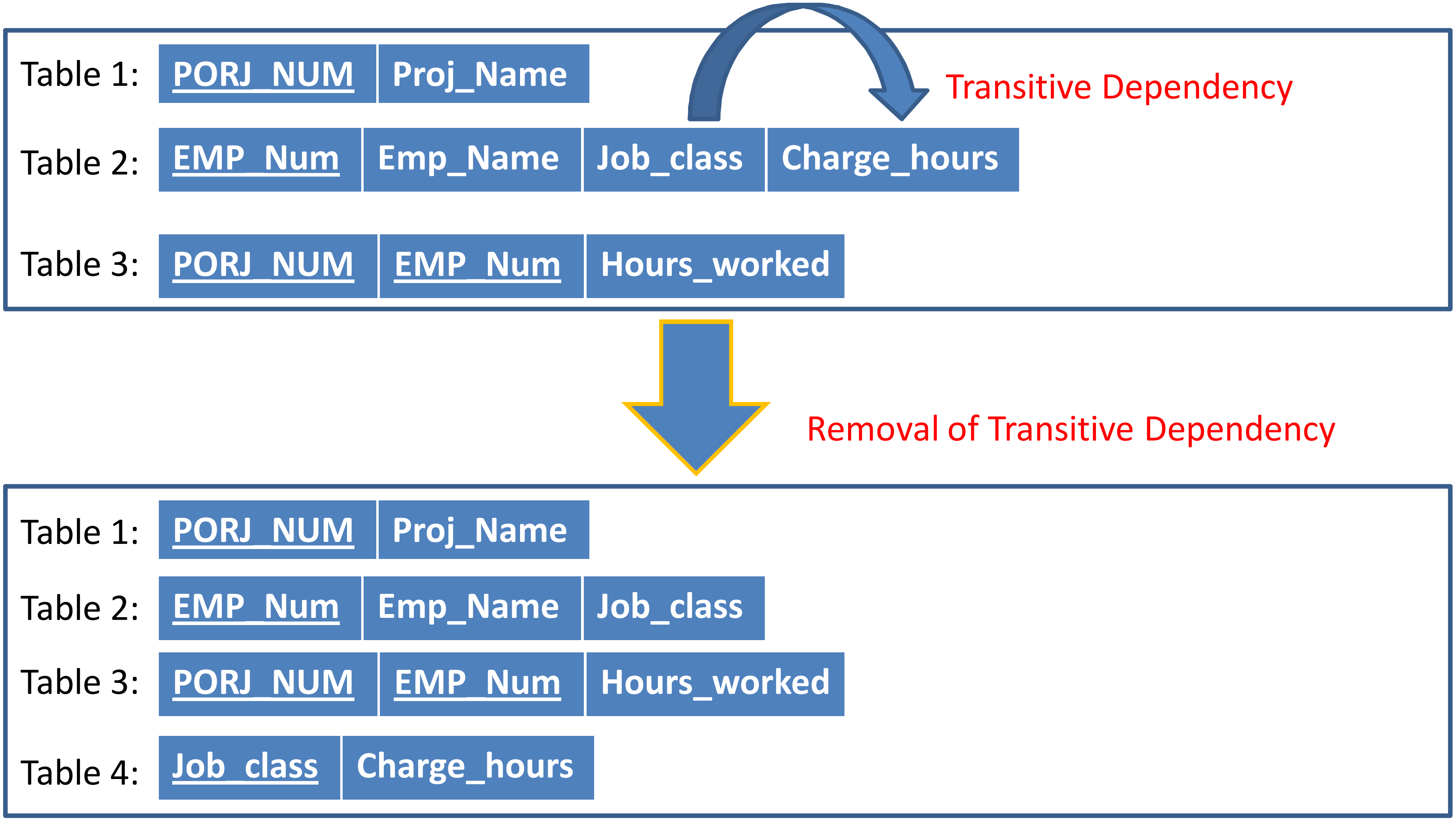


Table 1: PORJ_NUM | Proj_Name

Table 2: EMP_Num | Emp_Name | Job_class | Charge_hours

Table 3: PORJ_NUM | EMP_Num | Hours_worked

Transitive Dependency

Removal of Transitive Dependency

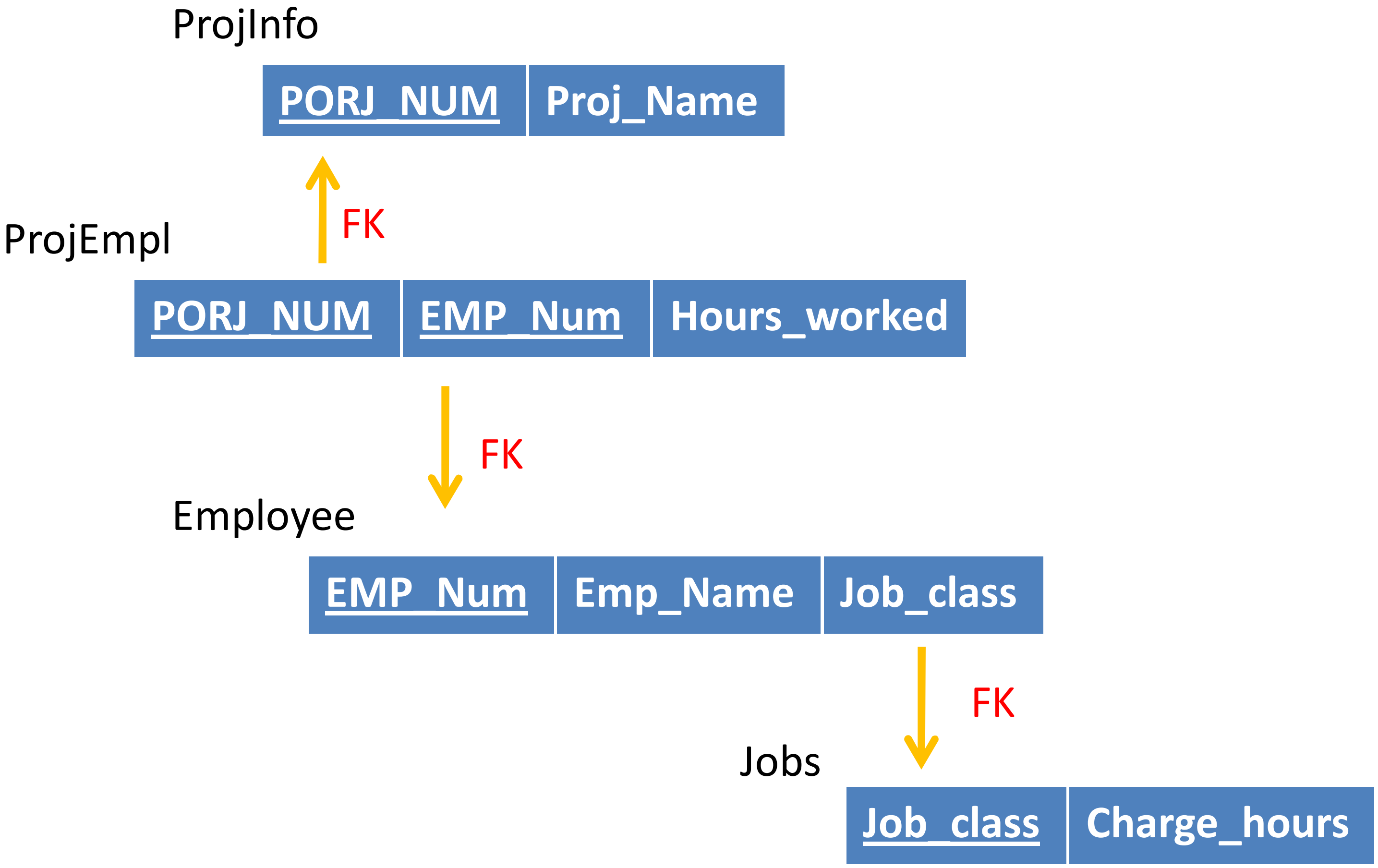
Table 1: PORJ_NUM | Proj_Name

Table 2: EMP_Num | Emp_Name | Job_class

Table 3: PORJ_NUM | EMP_Num | Hours_worked

Table 4: Job_class | Charge_hours

Case 3: DB in 3NF



ProjInfo

PORJ_NUM	Proj_Name

FK

ProjEmpl

PORJ_NUM	EMP_Num	Hours_worked

FK

Employee

EMP_Num	Emp_Name	Job_class

FK

Jobs

Job_class	Charge_hours

Case Study 4

- Perform Normalization for the Example in Slide # 4.

- Plain Table is

Student_id	StudentName	Major	college	collegeLocation	classID	className	ProfessorID	Professor Name
999-40-9876	Ahmad	Math	Science	UJ	Math101	Discrete Math	111-2231	XYZ
999-40-9876	Ahmad	Math	Science	UJ	Skills101	Computer Skills	111-2124	YZR
999-40-9000	Salem	CIS	KASIT	UJ	CIS322	DB	111-2222	URT

- Solution on Slide # 4

Chapter 4: Intermediate SQL

Database System Concepts, 6th Ed.

Chapter 4: Intermediate SQL

- Join Operations
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Authorization

Slide updated: Dr. Alomari

Join Operation

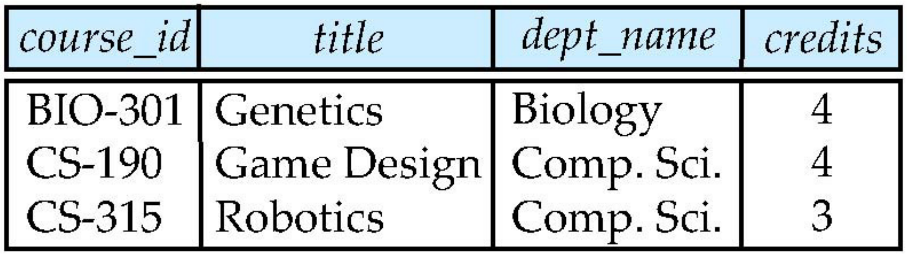
1. **Natural Join operation**
   - A join operation between two tables based on the two common (same name) columns between the two tables.
2. **Inner Join operation**
   - Specifies a join between two tables with an explicit join clause.
3. **Natural Outer Join Operation:**
   A. **Natural Left outer join:** a join operation for two tables with all records from left (first) table not exist in right (second) table.

   B. **Natural Right outer join:** a join operation for two tables with all records from right (second) table not exist in left (first) table.
4. **Cross Join operation**
   - A join operation that produces the Cartesian product of two tables.

Slide added: Dr. Alomari

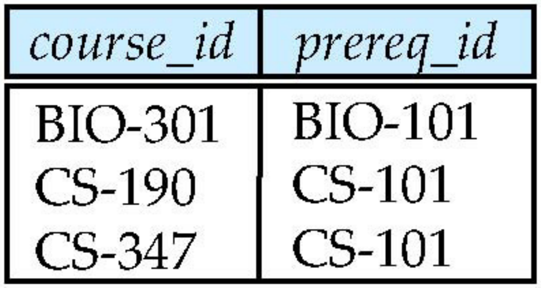
Join operations – Example

- Relation *course*



course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*



course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that

  prereq information is missing for CS-315 and

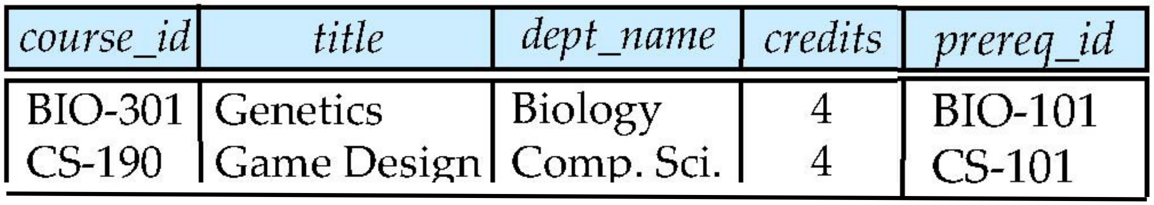
  course information is missing for CS-437

Note that the prereq table results from a unary many-to-many relation. The prereq table should not allow any record entry course_id or prereq_id that is not in the course table (foreign key constraint). However, this is just as an example to show the various join types.

Slide updated: Dr. Alomari:

1- Natural Join

Select * from Course natural join Prereq;

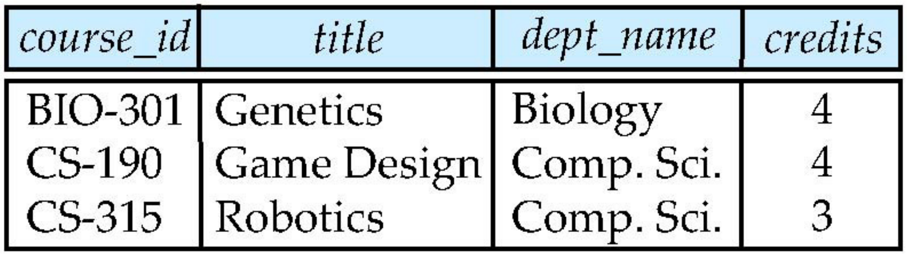


course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101

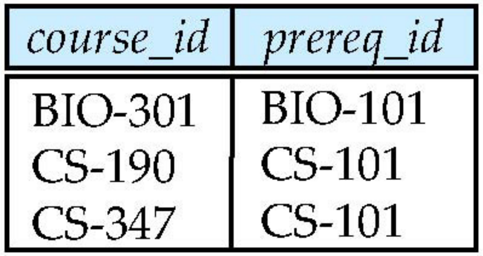
Note that the course_id (common attribute) is not repeated.

Slide added: Dr. Alomari

2- Inner Join

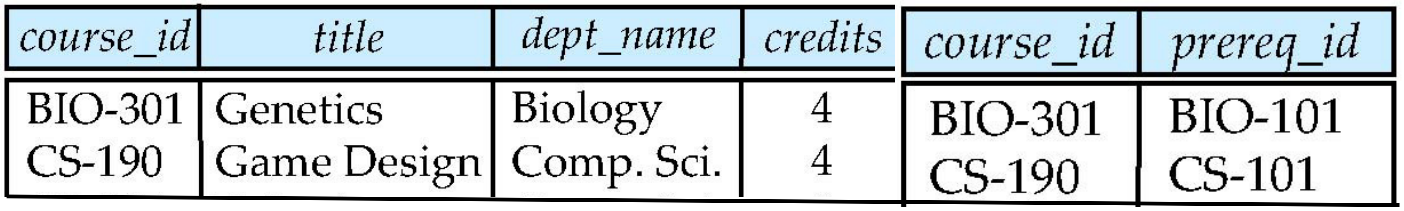


course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3



course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Select * from Course inner join Prereq on Course.course_id = Prereq.course_id;



course_id	*title*	*dept_name*	*credits*	*course_id*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-301	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-190	CS-101

Note: the repetition of the course_id.
Natural join does not have this repetition .

Slide added: Dr. Alomari

Note about Inner Join

- Note that: if you do not specify the matching attributes (columns) between the two tables in the Inner join, the results will be the Cartesian product of the two tables.

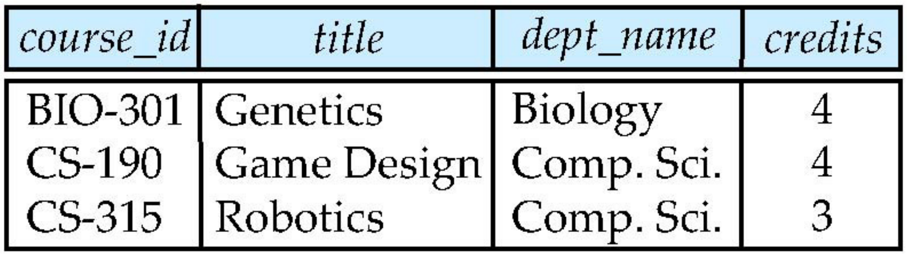
```
Select * from Course inner join Prereq;
```

Which will be the same as:

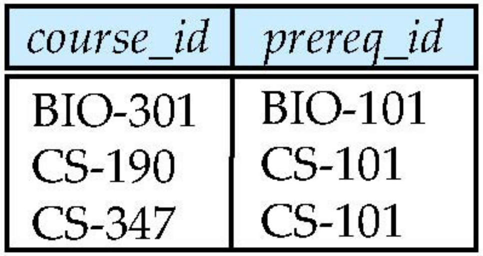
```
Select * from Course, Prereq;
```

Slide added: Dr. Alomari

3-A Natural Left Outer Join

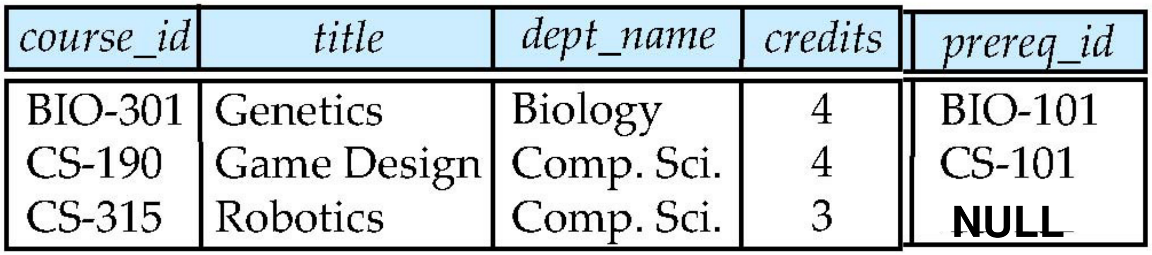


course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3



course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Select * from Course natural left outer join Prereq;



course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	**NULL**

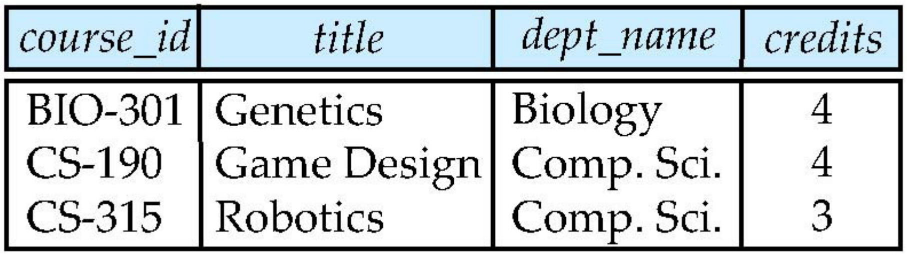
Displays:
1- All common records.
2- All records in left (first) table and not in right (second) table.
*** Adds NULL for missing values.**

Note 1: The column course_id is NOT repeated.

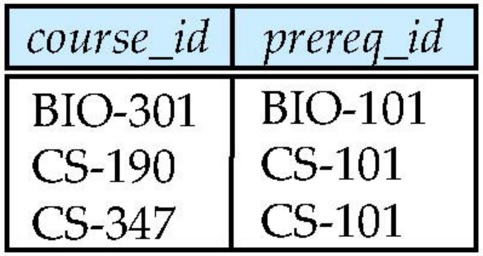
Note 2: note the added Null value for missing data.

Slide added: Dr. Alomari

3-B Natural Right Outer Join

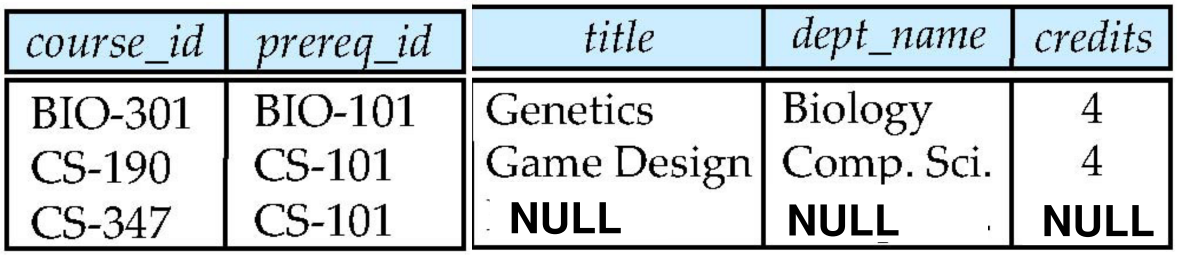


course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3



course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

```
Select * from Course natural right outer join Prereq;
```



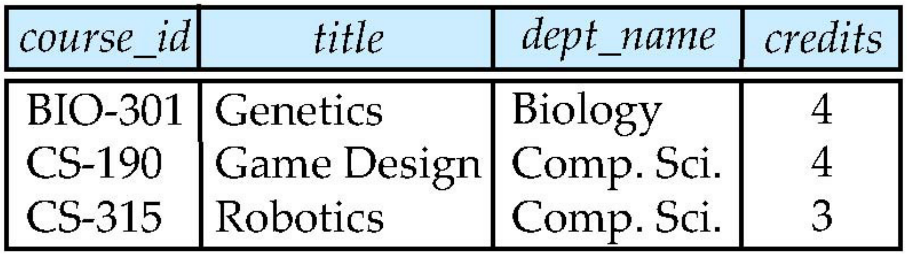
course_id	prereq_id	title	dept_name	credits
BIO-301	BIO-101	Genetics	Biology	4
CS-190	CS-101	Game Design	Comp. Sci.	4
CS-347	CS-101	NULL	NULL	NULL

Displays:
1- All common records.
2- All records in right(second) table and not in left (first) table.
*** Adds NULL for missing values.**

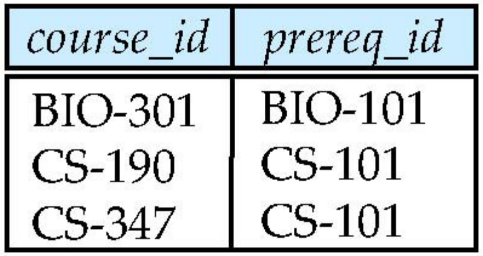
Note 1: The column course_id is NOT repeated.
Note 2: note the added Null value for missing data.
Note 3: The column of right table are first.

Slide added: Dr. Alomari

4- Cross Join

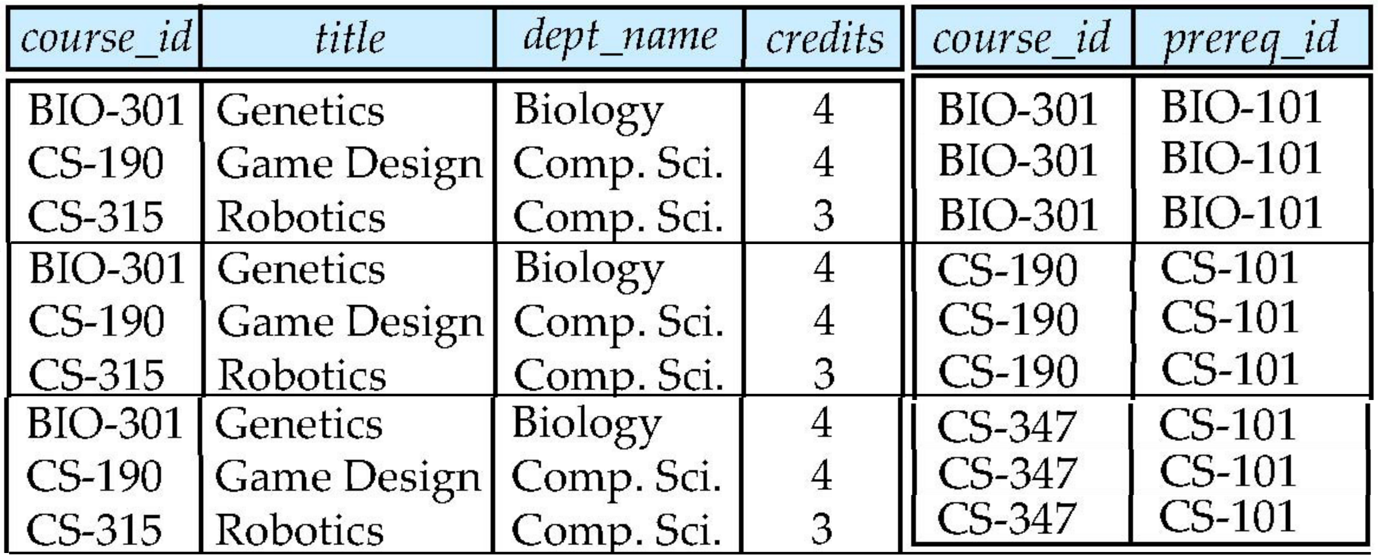


course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3



course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Select * from Course cross join Prereq;



course_id	title	dept_name	credits	course_id	prereq_id
BIO-301	Genetics	Biology	4	BIO-301	BIO-101
CS-190	Game Design	Comp. Sci.	4	BIO-301	BIO-101
CS-315	Robotics	Comp. Sci.	3	BIO-301	BIO-101
BIO-301	Genetics	Biology	4	CS-190	CS-101
CS-190	Game Design	Comp. Sci.	4	CS-190	CS-101
CS-315	Robotics	Comp. Sci.	3	CS-190	CS-101
BIO-301	Genetics	Biology	4	CS-347	CS-101
CS-190	Game Design	Comp. Sci.	4	CS-347	CS-101
CS-315	Robotics	Comp. Sci.	3	CS-347	CS-101

Note: This is Cartesian product

Slide added: Dr. Alomari

Note about Cartesian Product

- There are many select statements that produce the same output, for example, to obtain the Cartesian product:

```
Select * from Course cross join Prereq;
```

```
Select * from Course inner join Prereq;
```

```
Select * from Course join Prereq;
```

```
Select * from Course, Prereq;
```

Slide added: Dr. Alomari

Note 2: Cross join v.s. Inner join on a common attribute

These three statements are equivalent:

```
Select * from Course cross join Prereq on Course.course_id = Prereq.course_id;
```

```
Select * from Course inner join Prereq on Course.course_id = Prereq.course_id;
```

```
Select * from Course, Prereq where Course.course_id = Prereq.course_id;
```

However, this statement does not repeat the common attribute (course_id):

```
Select * from Course natural join Prereq;
```

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

Example Views

- A view of instructors without their salary

```
create view faculty as
   select ID, name, dept_name
   from instructor
```

- Find all instructors in the Biology department

```
select name
from faculty
where dept_name = ‘Biology’
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as
     select dept_name, sum (salary)
     from instructor
    group by dept_name;
```

Views Defined Using Other Views

-
  ```
  create view physics_fall_2009 as
     select course.course_id, sec_id, building, room_number
     from course, section
     where course.course_id = section.course_id
              and course.dept_name = ’Physics’
              and section.semester = ’Fall’
              and section.year = ’2009’;
  ```

-
  ```
  create view physics_fall_2009_watson as
     select course_id, room_number
     from physics_fall_2009
     where building= ’Watson’;
  ```

View Expansion

- Expand use of a view in a query/another view

```
create view physics_fall_2009_watson as
(select course_id, room_number
from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = ’Physics’
            and section.semester = ’Fall’
            and section.year = ’2009’)
where building= ’Watson’;
```

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be *recursive* if it depends on itself.

View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

```
repeat
   Find any view relation $v_i$ in $e_1$
   Replace the view relation $v_i$ by the expression defining $v_i$
until no more view relations are present in $e_1$
```

- As long as the view definitions are not recursive, this loop will terminate

Transactions

- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions
- Transactions begin implicitly
  - Ended by **commit work** or **rollback work**
- But default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (e.g. using API)
  - In SQL:1999, can use: **begin atomic** …. **end**
    - Not supported on most databases

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

Integrity Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate

Not Null and Unique Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**
- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes *A*1, *A*2, … *A*m
    form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

The check clause

- **check** (P)

  where P is a predicate

  Example: ensure that semester is one of fall, winter, spring or summer:

```
create table section (
    course_id varchar (8),
    sec_id varchar (8),
    semester varchar (6),
    year numeric (4,0),
    building varchar (15),
    room_number varchar (7),
    time slot id varchar (4),
    primary key (course_id, sec_id, semester, year),
    check (semester in (’Fall’, ’Winter’, ’Spring’, ’Summer’))
);
```

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If "Biology" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "Biology".
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

Cascading Actions in Referential Integrity

-
  ```
  create table course (
      course_id   char(5) primary key,
      title       varchar(20),
      dept_name varchar(20) references department
  )
  ```

-
  ```
  create table course (
      …
      dept_name varchar(20),
      foreign key (dept_name) references department
                on delete cascade
                on update cascade,
      . . .
  )
  ```
- alternative actions to cascade: **set null**, **set default**

Integrity Constraint Violation During Transactions

- E.g.

```
create table person (
    ID  char(10),
    name char(40),
    mother char(10),
    father  char(10),
    primary key ID,
    foreign key father references person,
    foreign key mother references  person)
```

- How to insert a tuple without causing constraint violation ?
  - insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking (next slide)

Complex Check Clauses

- **check** (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))
  - why not use a foreign key here?
- Every section has at least one instructor teaching the section.
  - how to write this?
- Unfortunately: subquery in check clause not supported by pretty much any database
  - Alternative: triggers (later)
- **create assertion** <assertion-name> **check** <predicate>;
  - Also not supported by anyone

Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** ‘2005-7-27’
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** ‘09:00:30’ **time** ‘09:00:30.75’
- **timestamp**: date plus time of day
  - Example: **timestamp** ‘2005-7-27 09:00:30.75’
- **interval:** period of time
  - Example: interval ‘1’ day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Index Creation

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))
- **create index** *studentID_index* **on** *student*(*ID*)
- Indices are data structures used to speed up access to records with specified values for index attributes
  - e.g. **select** *
    from *student*
    where *ID* = ‘12345’

  can be executed by using the index to find the required record, without looking at all records of *student*

  More on indices in Chapter 11

User-Defined Types

- **create type** construct in SQL creates user-defined type

```
create type Dollars as numeric (12,2) final
```

-
  ```
  create table department
  (dept_name varchar (20),
  building varchar (15),
  budget Dollars);
  ```

Domains

- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- **create domain** *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** (’Bachelors’, ’Masters’, ’Doctorate’));

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of character data
  - When a query returns a large object, a pointer is returned rather than the large object itself.

Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database schema

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  grant <privilege list>

  on <relation name or view name> **to** <user list>

- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:

    grant select on *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list>

  on <relation name or view name> **from** <user list>

- Example:

  revoke select on *branch* **from** U_1, U_2, U_3

- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Roles

- **create role** instructor;
- **grant** *instructor* **to Amit;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = ’Geology’);
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
  - creator of view did not have some permissions on *instructor?*

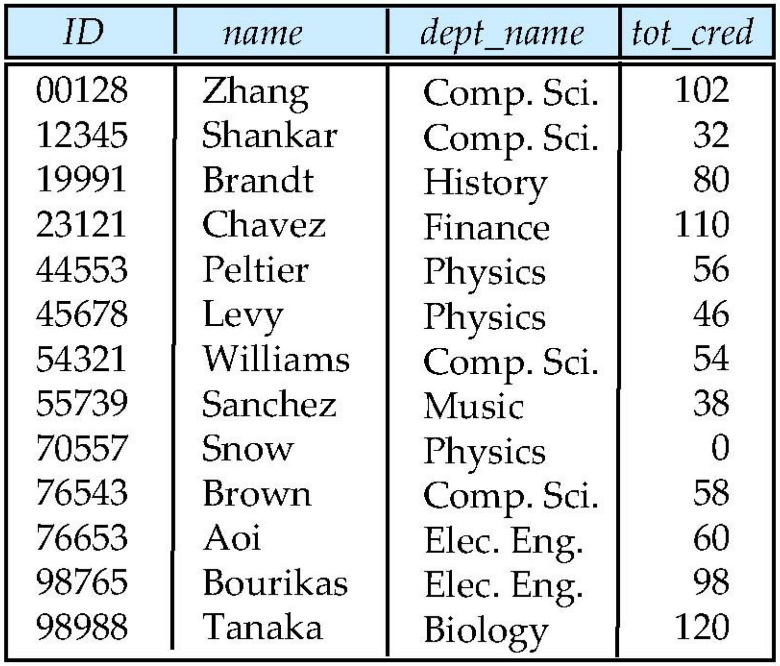
Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
- Etc. read Section 4.6 for more details we have omitted here.

End of Chapter 4

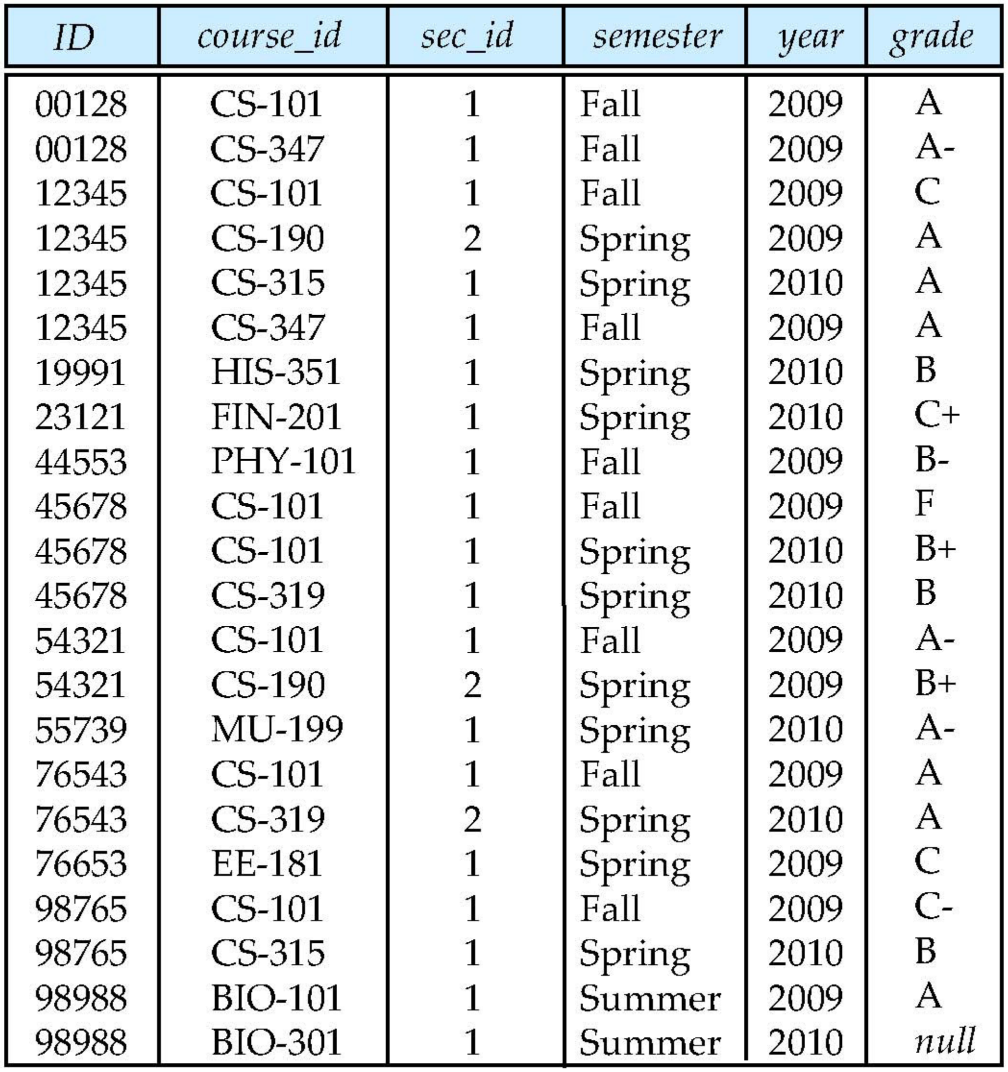
Database System Concepts, 6th Ed.

Figure 4.01



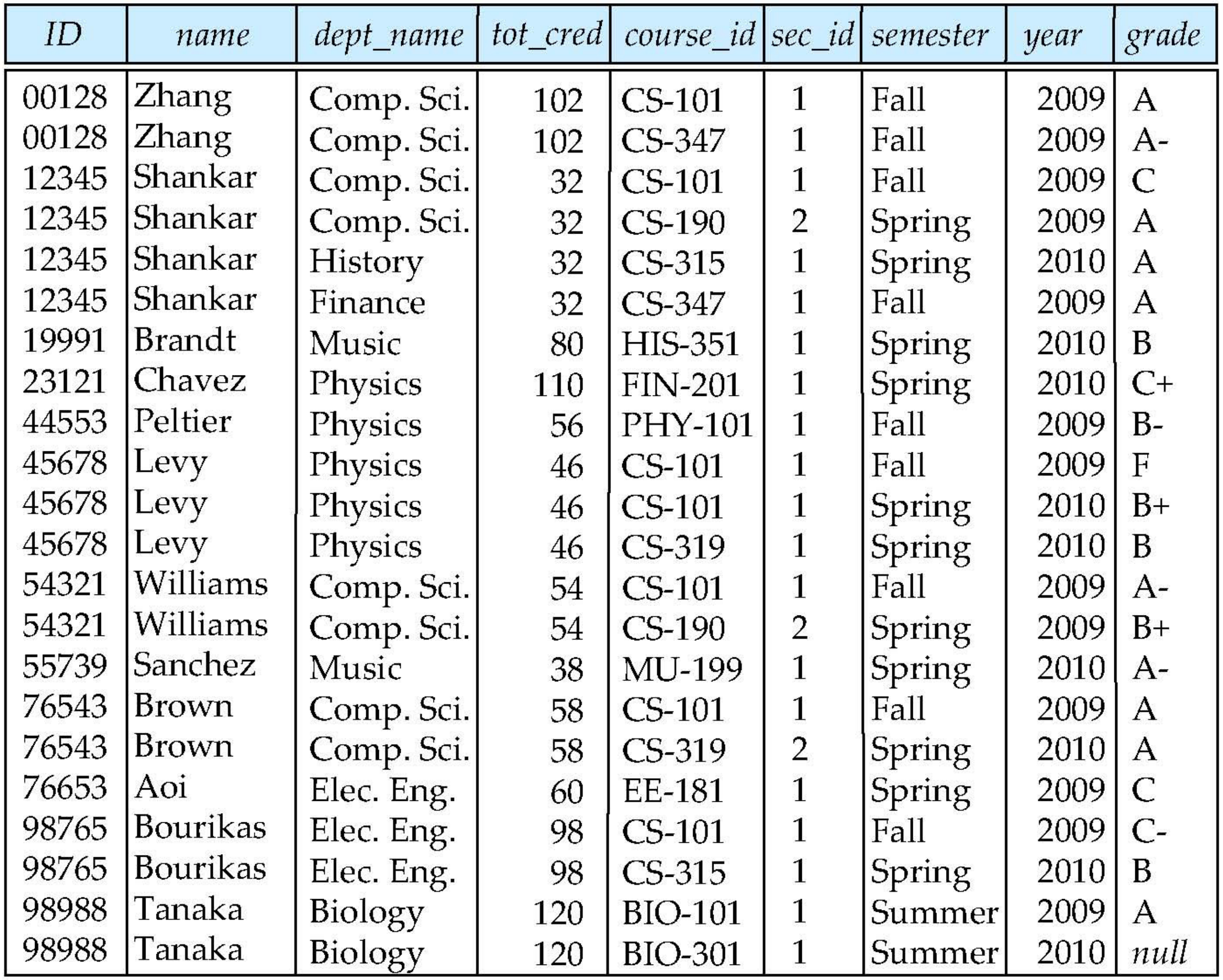
ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Figure 4.02



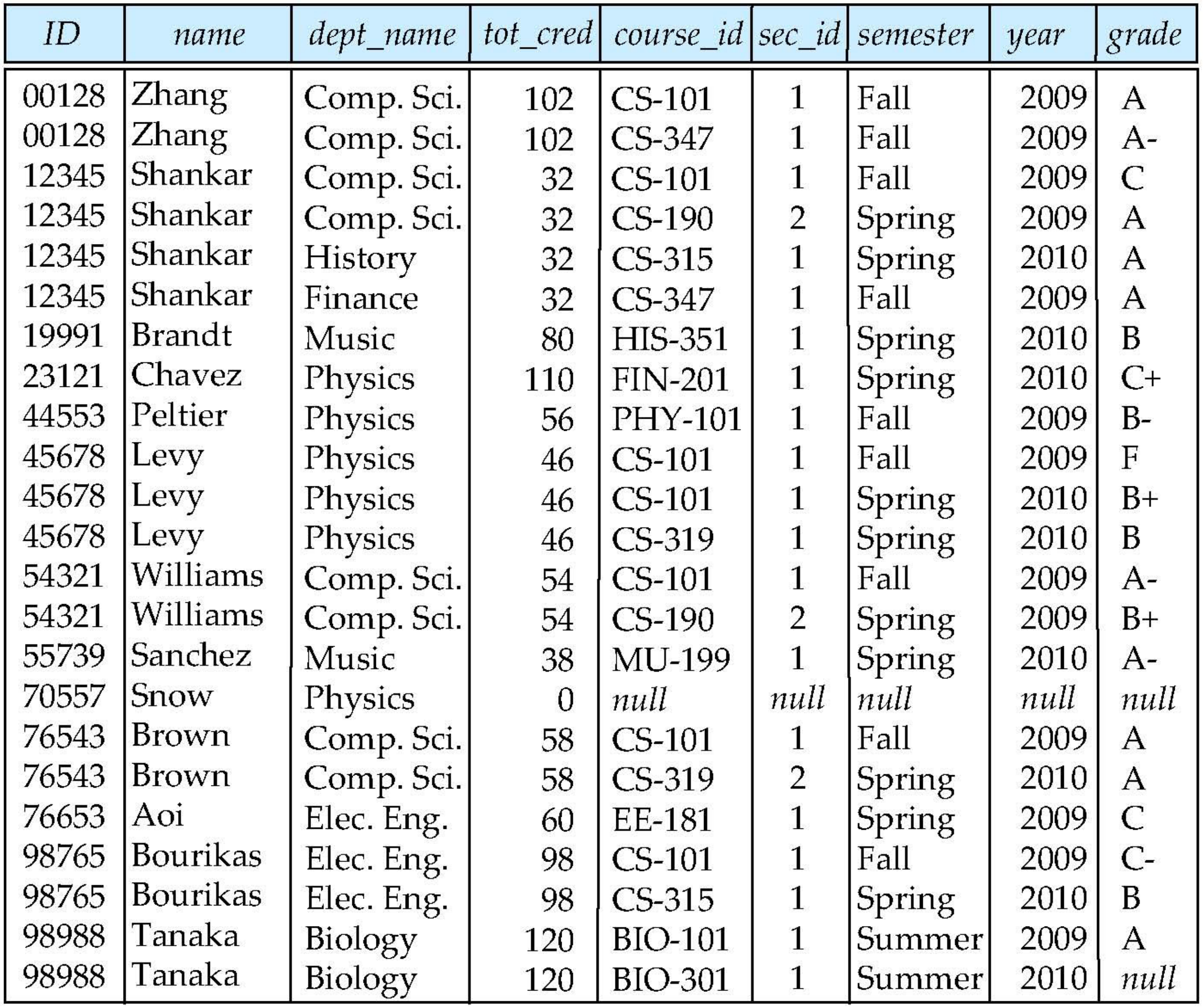
ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-101	1	Fall	2009	C
12345	CS-190	2	Spring	2009	A
12345	CS-315	1	Spring	2010	A
12345	CS-347	1	Fall	2009	A
19991	HIS-351	1	Spring	2010	B
23121	FIN-201	1	Spring	2010	C+
44553	PHY-101	1	Fall	2009	B-
45678	CS-101	1	Fall	2009	F
45678	CS-101	1	Spring	2010	B+
45678	CS-319	1	Spring	2010	B
54321	CS-101	1	Fall	2009	A-
54321	CS-190	2	Spring	2009	B+
55739	MU-199	1	Spring	2010	A-
76543	CS-101	1	Fall	2009	A
76543	CS-319	2	Spring	2010	A
76653	EE-181	1	Spring	2009	C
98765	CS-101	1	Fall	2009	C-
98765	CS-315	1	Spring	2010	B
98988	BIO-101	1	Summer	2009	A
98988	BIO-301	1	Summer	2010	*null*

Figure 4.03



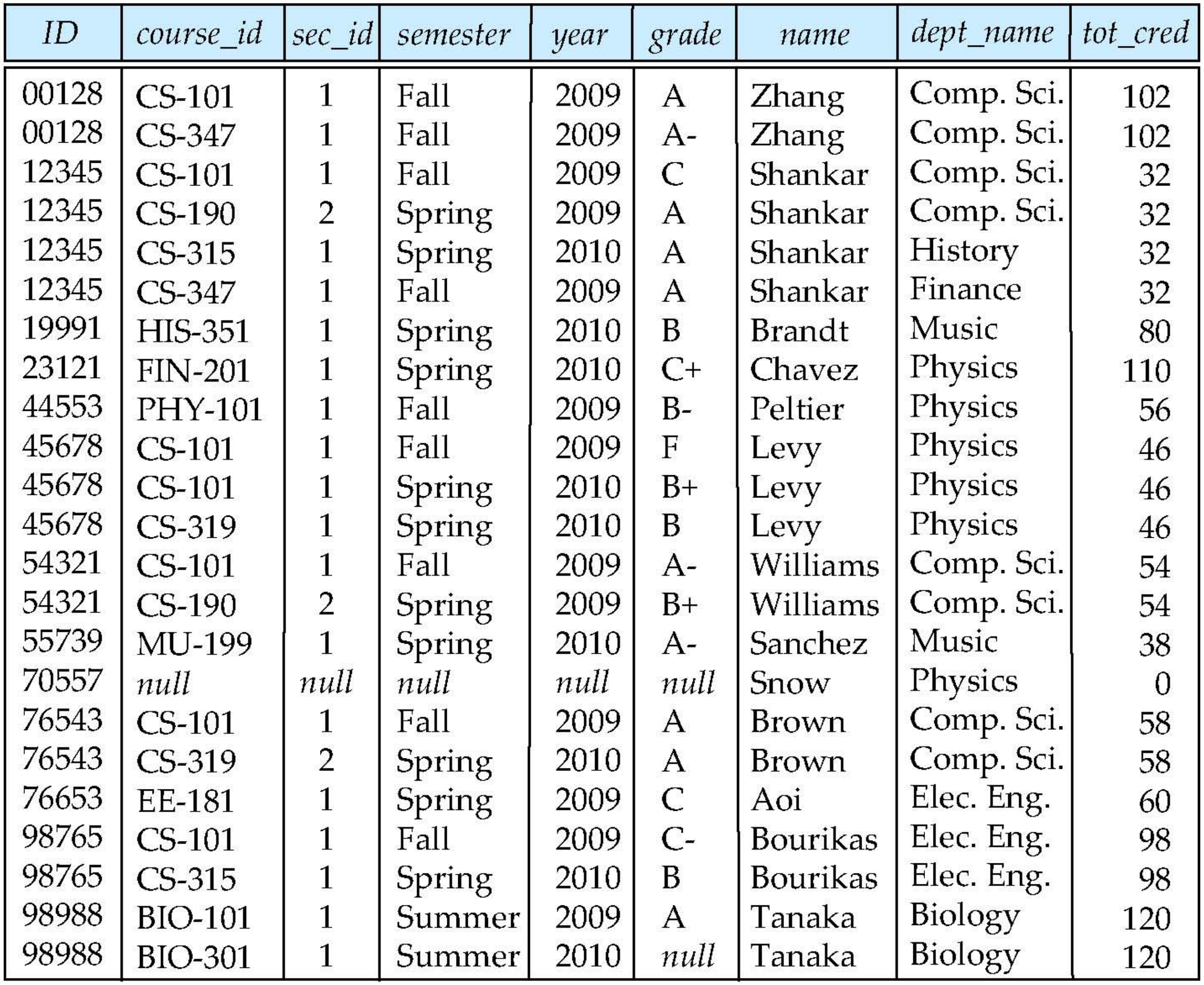
ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	History	32	CS-315	1	Spring	2010	A
12345	Shankar	Finance	32	CS-347	1	Fall	2009	A
19991	Brandt	Music	80	HIS-351	1	Spring	2010	B
23121	Chavez	Physics	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	*null*

Figure 4.04



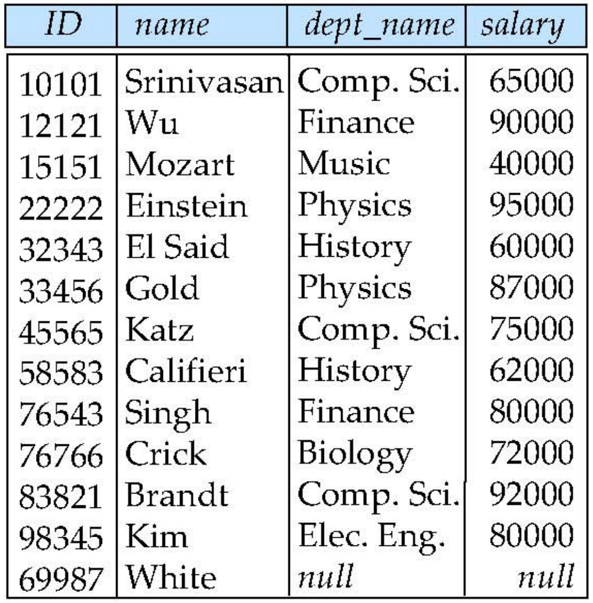
ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2009	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2009	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2009	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2009	A
12345	Shankar	History	32	CS-315	1	Spring	2010	A
12345	Shankar	Finance	32	CS-347	1	Fall	2009	A
19991	Brandt	Music	80	HIS-351	1	Spring	2010	B
23121	Chavez	Physics	110	FIN-201	1	Spring	2010	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2009	B-
45678	Levy	Physics	46	CS-101	1	Fall	2009	F
45678	Levy	Physics	46	CS-101	1	Spring	2010	B+
45678	Levy	Physics	46	CS-319	1	Spring	2010	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2009	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2009	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2010	A-
70557	Snow	Physics	0	*null*	*null*	*null*	*null*	*null*
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2009	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2010	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2009	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2009	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2010	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2009	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2010	*null*

Figure 4.05



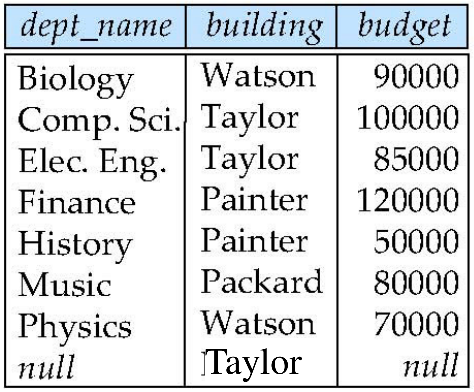
ID	*course_id*	*sec_id*	*semester*	*year*	*grade*	*name*	*dept_name*	*tot_cred*
00128	CS-101	1	Fall	2009	A	Zhang	Comp. Sci.	102
00128	CS-347	1	Fall	2009	A-	Zhang	Comp. Sci.	102
12345	CS-101	1	Fall	2009	C	Shankar	Comp. Sci.	32
12345	CS-190	2	Spring	2009	A	Shankar	Comp. Sci.	32
12345	CS-315	1	Spring	2010	A	Shankar	History	32
12345	CS-347	1	Fall	2009	A	Shankar	Finance	32
19991	HIS-351	1	Spring	2010	B	Brandt	Music	80
23121	FIN-201	1	Spring	2010	C+	Chavez	Physics	110
44553	PHY-101	1	Fall	2009	B-	Peltier	Physics	56
45678	CS-101	1	Fall	2009	F	Levy	Physics	46
45678	CS-101	1	Spring	2010	B+	Levy	Physics	46
45678	CS-319	1	Spring	2010	B	Levy	Physics	46
54321	CS-101	1	Fall	2009	A-	Williams	Comp. Sci.	54
54321	CS-190	2	Spring	2009	B+	Williams	Comp. Sci.	54
55739	MU-199	1	Spring	2010	A-	Sanchez	Music	38
70557	*null*	*null*	*null*	*null*	*null*	Snow	Physics	0
76543	CS-101	1	Fall	2009	A	Brown	Comp. Sci.	58
76543	CS-319	2	Spring	2010	A	Brown	Comp. Sci.	58
76653	EE-181	1	Spring	2009	C	Aoi	Elec. Eng.	60
98765	CS-101	1	Fall	2009	C-	Bourikas	Elec. Eng.	98
98765	CS-315	1	Spring	2010	B	Bourikas	Elec. Eng.	98
98988	BIO-101	1	Summer	2009	A	Tanaka	Biology	120
98988	BIO-301	1	Summer	2010	*null*	Tanaka	Biology	120

Figure 4.07



ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	*null*	*null*

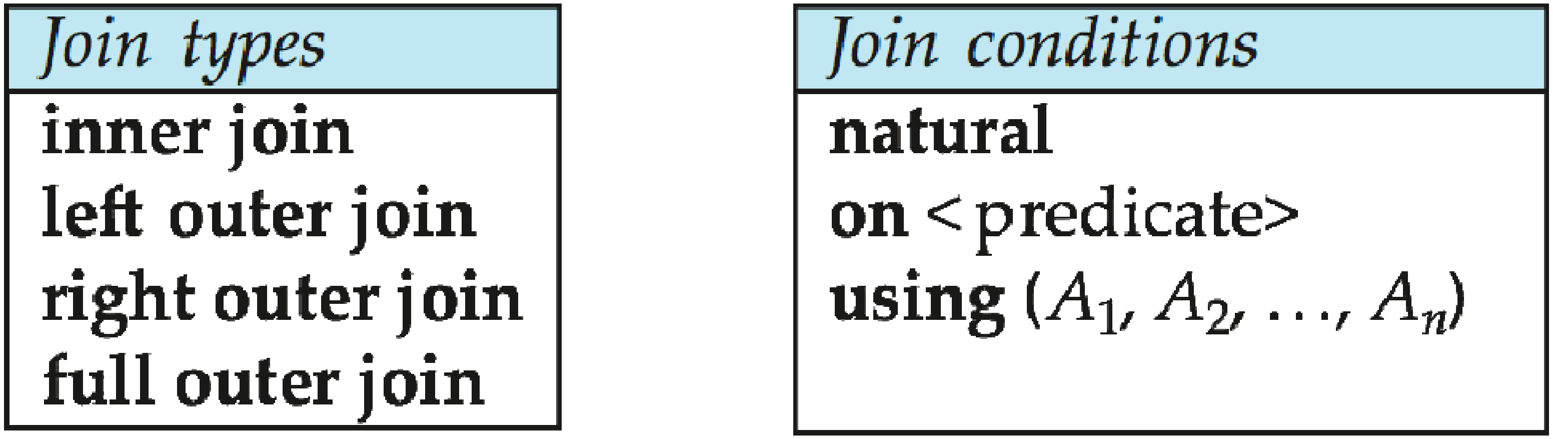
instructor



dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
null	Taylor	*null*

department

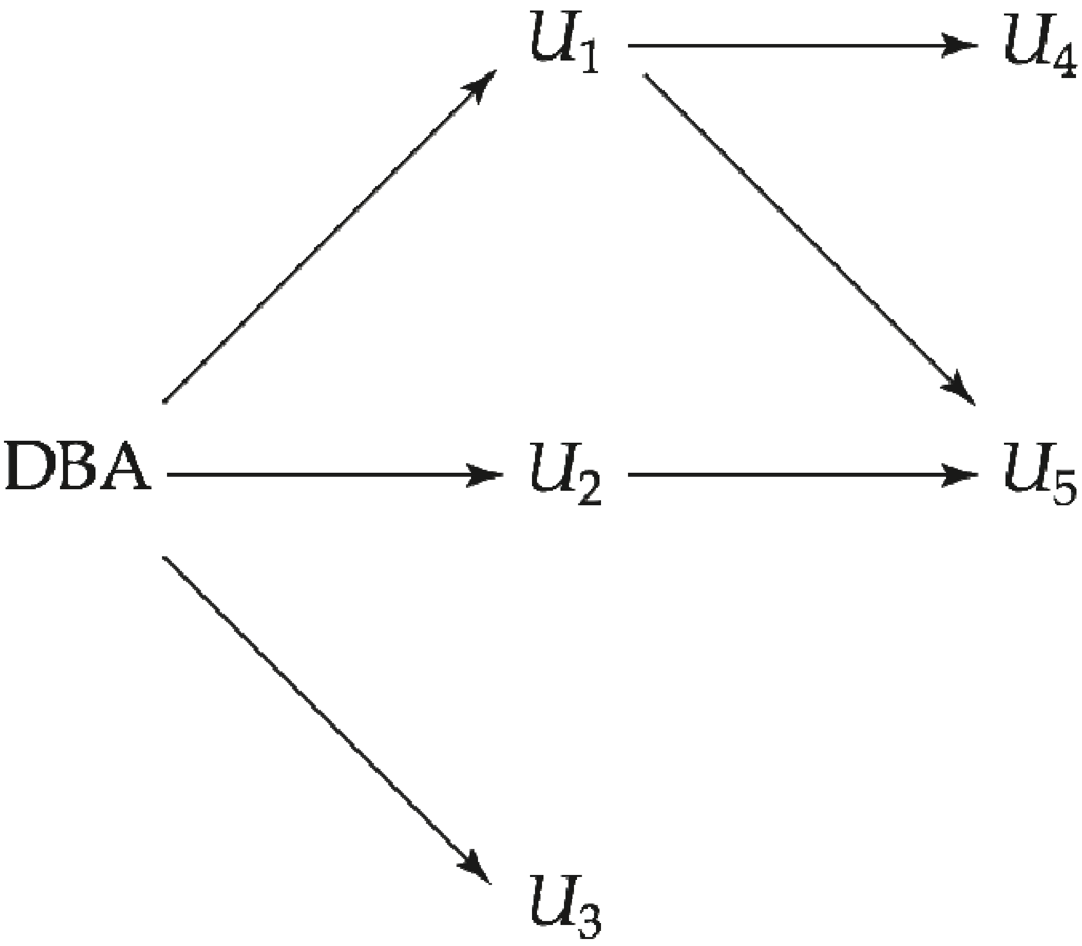
Figure 4.06



Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on < predicate>
using $(A_1, A_2, \ldots, A_n)$

Figure 4.03



ER-D Mapping to a Relational Database

Summary with examples

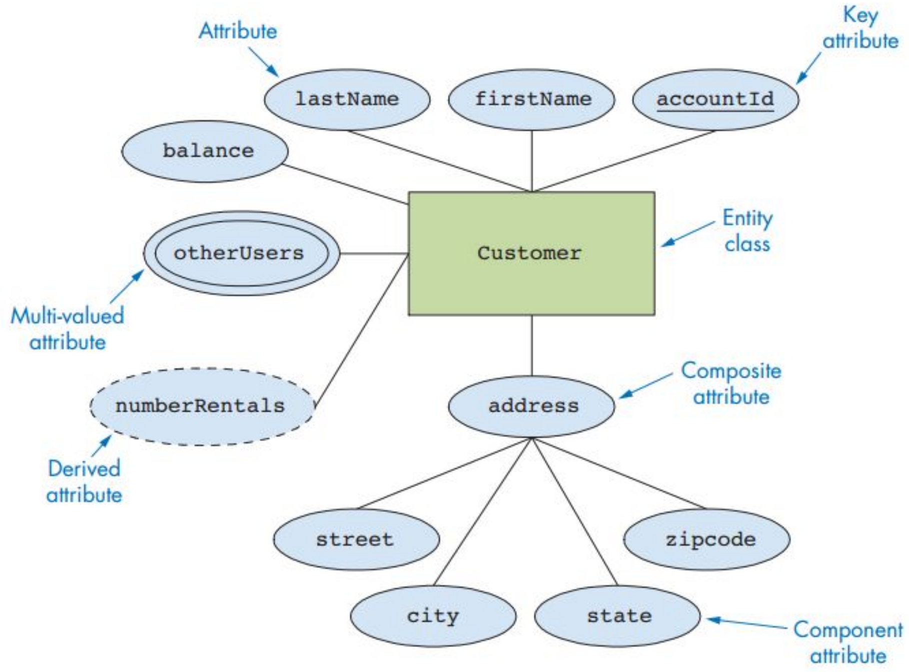
Goal

- We use ER-D Modelling to understand the informational needs of a system. Once the model is satisfactory, we then implement our design in a relational database. In order to implement a database we must decide on the relations, their attributes and primary keys.

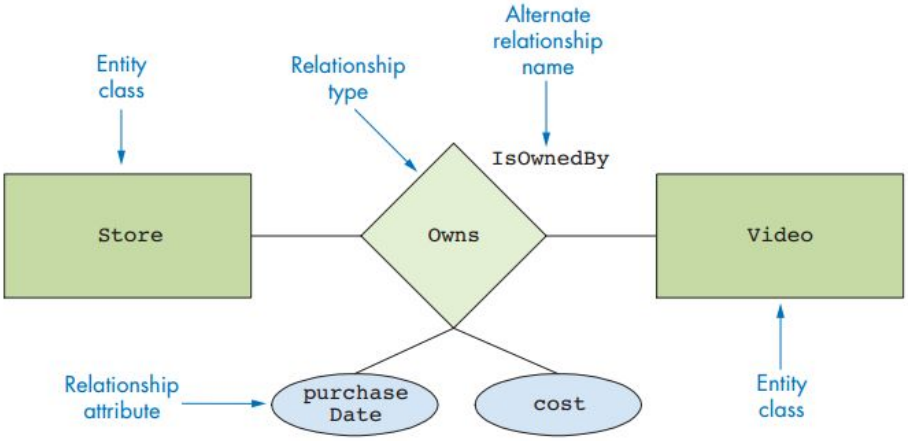
- This section contains some simple rules that we use to **map** an ER-D to a relational database.

Revision

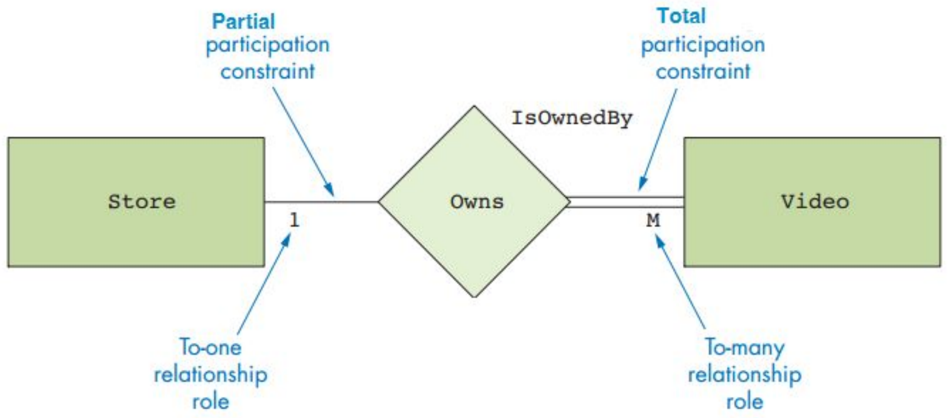
An entity


Attribute
Key attribute
lastName
firstName
accountId
balance
otherUsers
Customer
Entity class
Multi-valued attribute
numberRentals
address
Composite attribute
Derived attribute
street
zipcode
city
state
Component attribute

A relationship



A relationship with participation and cardinality symbols


Partial participation constraint
Total participation constraint
IsOwnedBy
Store
Owns
Video
1
M
To-one relationship role
To-many relationship role

ER-D Mapping

Basics for this process are detailed in this section

Entity

- Each **entity** turns into a **table**. It is important to distinguish entity types as being either strong or weak.

  - **Strong Entity Types**

    Strong, or regular, entity types are mapped to their own table. The attributes are included and the PK is chosen to be one of these attributes.

  - **Weak Entity Types**

    Weak entity types are mapped to their own table. The PK attributes of any table created from the related identifying entity types are included as FK attributes. **The PK of the table for the weak entity** type is the combination of the PKs of tables (from the related identifying entity types) and the discriminator of the weak entity.

Attributes

- All **attributes**, with the exception of derived and composite attributes, turns into a **column** in the table as follows:
  - Note: You choose to include derived attributes if their presence will improve performance

  - Simple: is a column in the table representing the entity
  - Multivalued: Each multi-valued attribute is implemented using a new **table**. This table will include the primary key of the original entity type and the multi-valued attribute. The primary key of this relation is the primary key of the original entity set and the multi-valued attribute.
  - Composite: Composite attributes are not represented as columns in the table. However the simple attributes comprising the composite attribute do appear in the table.

Relationships

- In order to understand relationships mapping we need to understand the concept of the Keys first.

Keys and constraints

- Relational modeling uses primary keys and foreign keys to maintain relationships
- **Primary keys** are typically the (unique) identifier noted on the conceptual model.
- **Foreign keys** are the PK of another entity to which an entity has a relationship.
- **Composite primary keys** are keys that are made of more than one attribute such as the PKs used for:
  - Weak entity
  - Conjunction table (M:M relationship mapping)
- Entity **integrity** constraints: A PK attribute must not be null.
- **Referential integrity** constraints: Matching of primary and foreign keys

Relationships (con’t)

- The implementation of **relationships** involve the use of **foreign keys**.
- It is important to consider both the degree and the cardinality of a relationship.
  - One-To-One
  - One- To-Many
  - Many-To-Many

Relationships (con’t)
One-To-One

- **Binary One-To-One**

In general, with a one-to-one relationship, you have a choice regarding where to implement the relationship. You may choose to place a foreign key in one of the two table. Also, include any attributes defined for the relationship in the same table as the foreign key.

Relationships (con't) One-To-Many

- **One-To-Many**

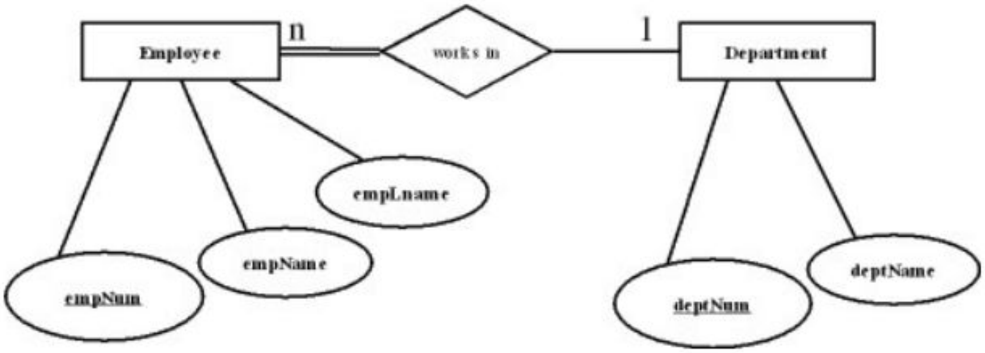
With a one-to-many relationship you must place a foreign key (and any attributes defined for the relationship) in the table corresponding to the many side of the relationship.

Relationships (con't) Many-To-Many

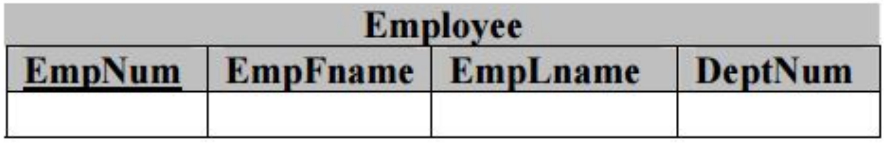
- **Many-To-Many**

A many-to-many relationship must be implemented with a separate table **(conjunction table)**. This new relation will have a composite primary key comprising the primary keys of the tables implementing the participating entity types plus any discriminator attribute. Also include as attributes, in this table, any attributes defined for the relationship.

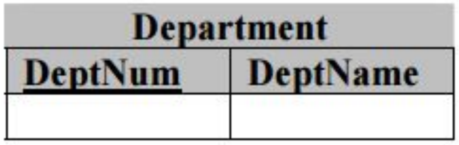
Example 1: ER-D



Example 1: Tables after Mapping



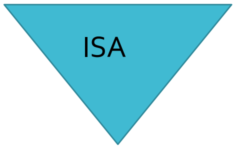
Employee			
EmpNum	EmpFname	EmpLname	DeptNum

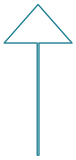


Department	
DeptNum	DeptName

Specialization in ER-D

- Specialization is a top-down design process; we designate sub-groupings within an entity set that are distinctive from other entities in the set.
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labeled ISA or a hollow arrow-head.



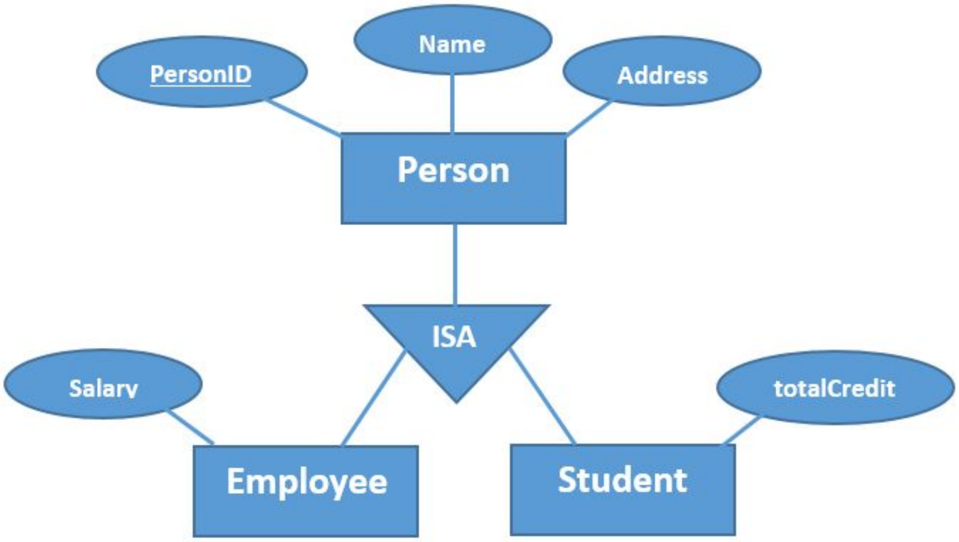


Examples of specialization

The following are example of specialization in possible entity sets and some attributes that represent these distinctive entity sets. We can call them Superclass and Subclass.

- A person *(superclass)* can be
  - Employee has salary *(subclass)*
  - Student has credit hours *(subclass)*
- Student can be
  - Graduate has office number
  - Undergraduate has residential college
- University employee can be
  - Instructor has rank
  - Secretary has hours per week and a relation with Employee

Person specialization example



Specialization (con’t)

- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.
- How entity sets are mapped into tables depends on the following constraints:
  - Overlapping or Disjoint
  - Completeness constraints (Total or Partial)

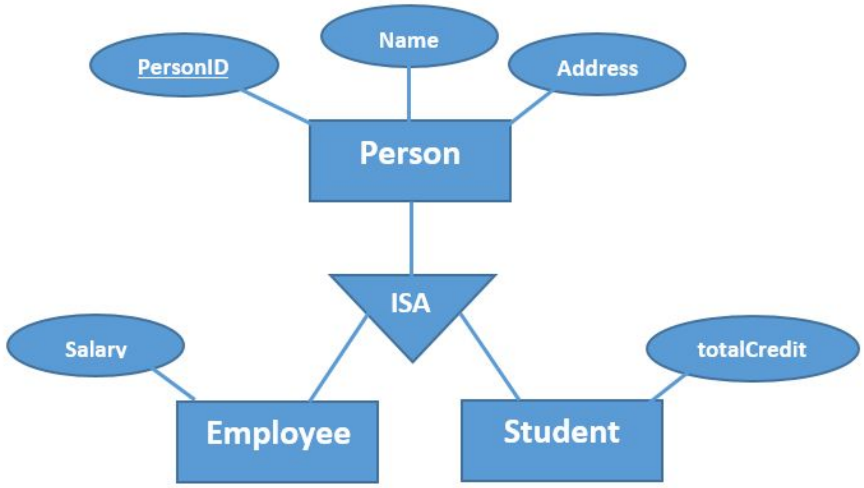
Overlapping and Disjoint

- This relates to whether or not entities may belong to more than one lower-level entity set. The lower-level entity sets may be one of the following:
  - **Disjoint** constraint requires that an entity belong to no more than one lower-level entity set. Ex. (An entity in the Student table can be either a graduate student or an undergraduate student, but cannot be both)
  - **Overlapping:** the same entity may belong to more than one lower-level entity set. Ex. (In a university that allows its employees to continue their studies, the person can be both an employee and a student)

Completeness constraints

- It specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the specialization. This constraint may be one of the following:
  - **Total specialization**: each higher-level entity must belong to a lower-level entity set.
  - **Partial specialization:** some higher-level entities may not belong to any lower-level entity set.
- The default is partial specialization and the total participation is depicted on the ER-D as *double lines* ||

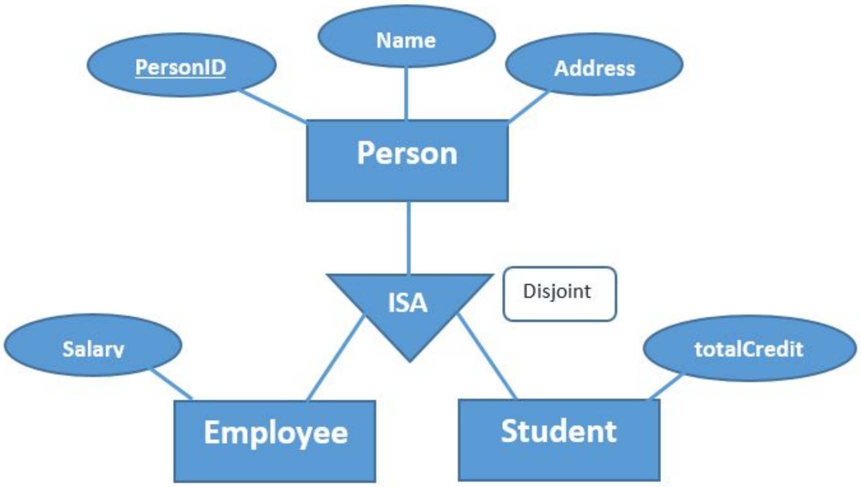
How to map specialization to schema



The obvious solution is to have 3 entities and follow the attribute inheritance feature of specialization.

But!!! Partial, Disjoint (3 tables)

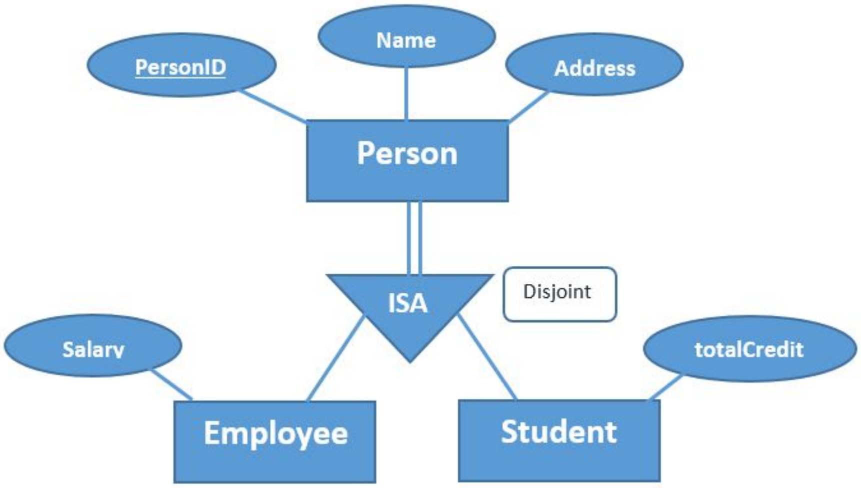
As it is partial, we have to use person table. As it is disjoint, there is no redundancy as no employee is student and no student is employee.



Person=(<u>PersonID</u>, Name, Address)
Employee=(<u>PersonID</u>, Name, Address, Salary)
Student=(<u>PersonID</u>, Name, Address, totalCredit)

And Total, Disjoint (2 tables)

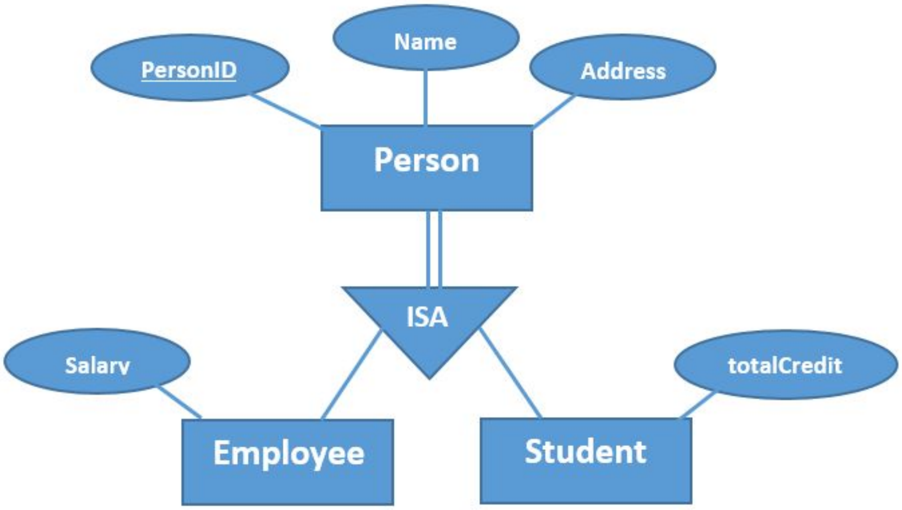
As it is total, no need to create person table As it is disjoint, there is no redundancy as no employee is customer and no customer is employee



Employee=(PersonID, Name, Address, Salary)
Student=(PersonID, Name, Address, totalCredit)

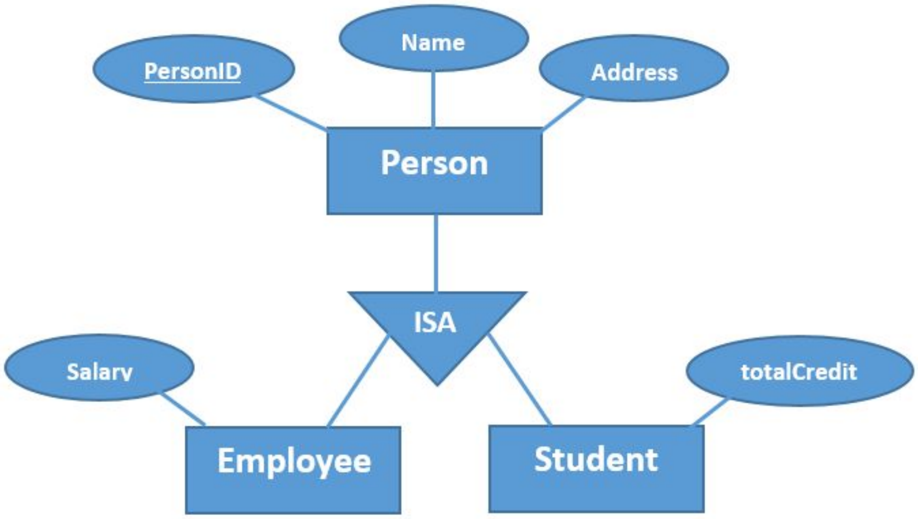
And Total, overlapping (3 tables)

Although it is total, but we still need to create person table because otherwise, there will be some redundancy.



Person = (<u>Person_id</u>, name, address)
employee = (<u>Person_id</u>, salary)
Student = (<u>Person_id</u>, totalCredit)

And Partial, overlapping (3 tables)

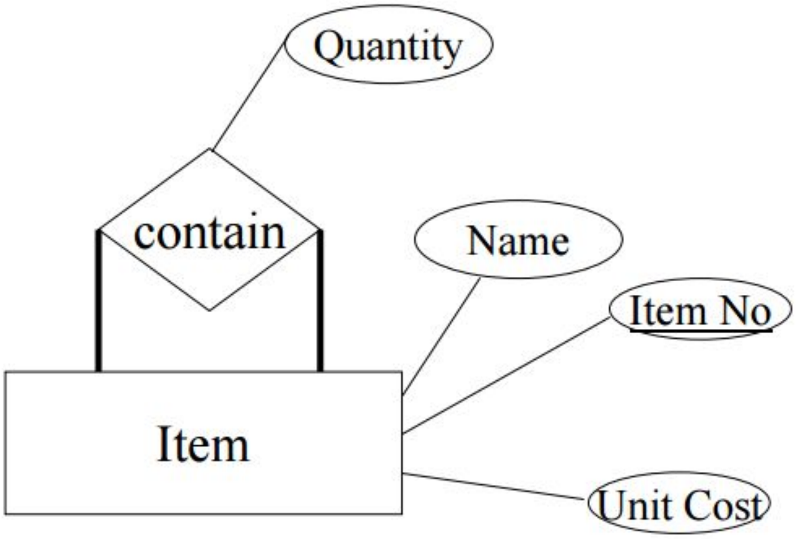


Person = (<u>Person_id</u>, name, address)
employee = (<u>Person_id</u>, salary)
Student= (<u>Person_id</u>, totalCredit)

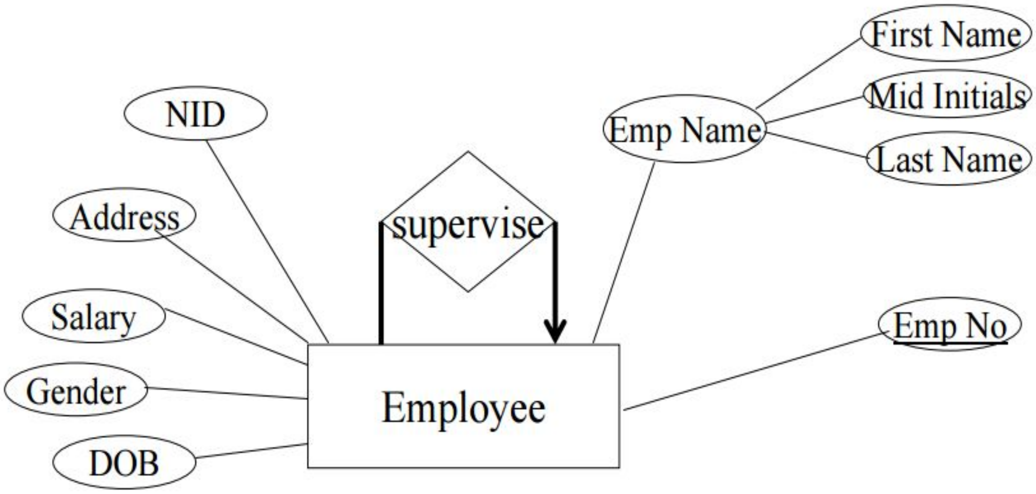
Summary for Specialization

- Disjoint: Inherit all the attributes. The mapping of the super class as a table depends on the completeness constraint.
- Overlapping: Do not inherit all, only the PK to form a relation with the super class. Doing so we avoid redundancy. The super class is always mapped into a table.

Exercise 1

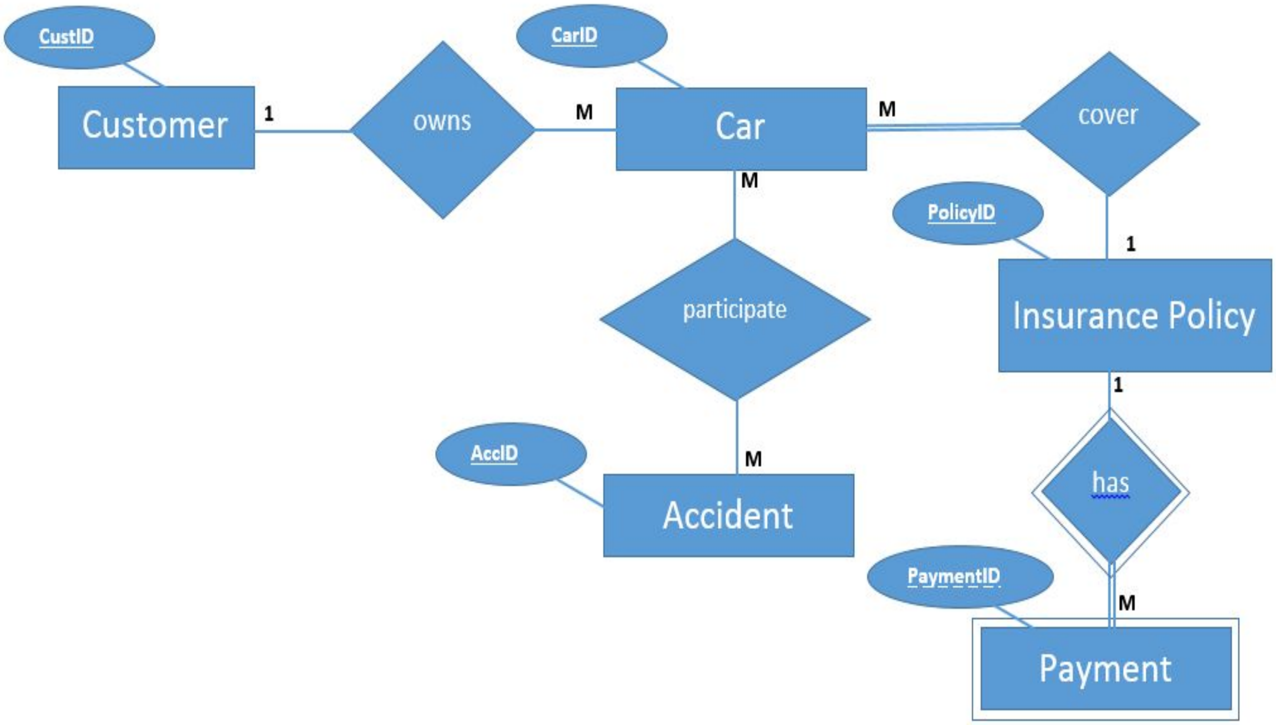


Exercise 2


First Name
Mid Initials
NID
Emp Name
Last Name
Address
supervise
Salary
Emp No
Gender
Employee
DOB

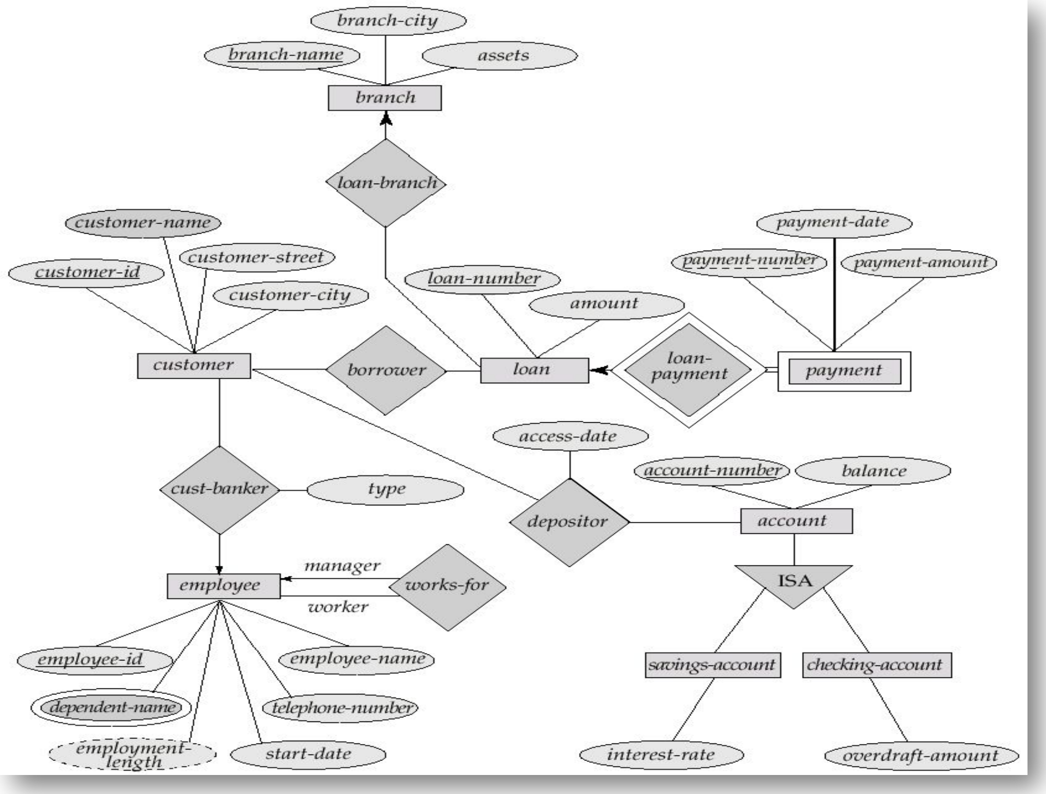
Exercise 3:

Map the Entities and their PK and FK if needed.

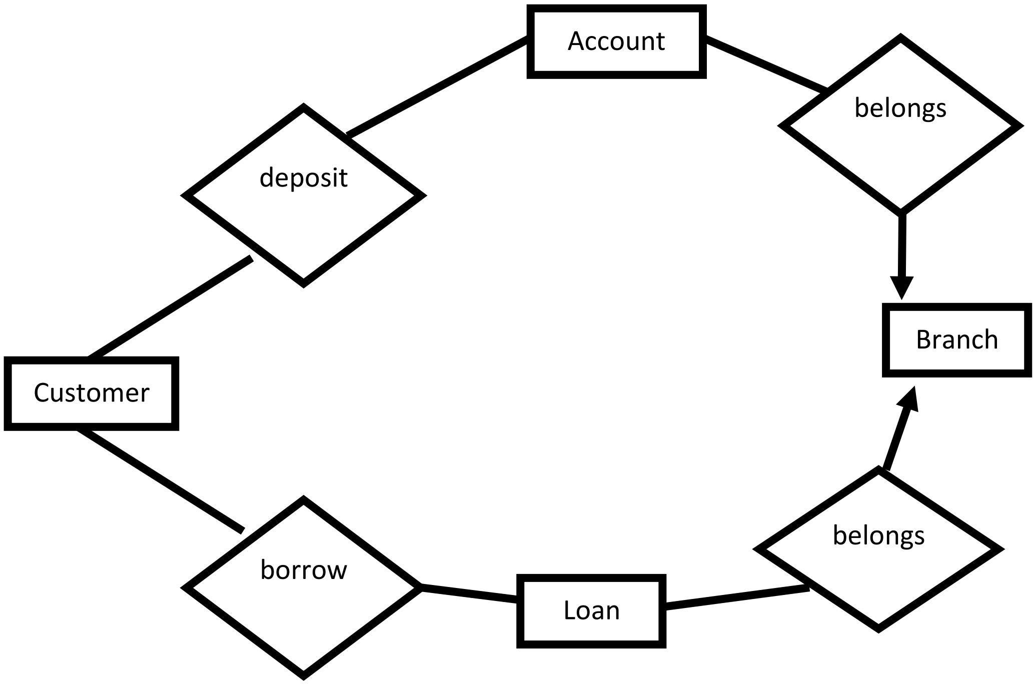


Exercise 4:

Map the entities, their attributes, PKs, and FKs if needed.



ERM



Schema from slide 55 of CH# 3 (Modified)

branch(BID, branch_name, branch_city, assets)	
customer(CID, customer_name, customer_street, customer_city)	
loan(LID, BID, amount)	Foreign key (BID) to branch(BID)
borrow(CID,LID)	Foreign key (CID) to Customer(CID) Foreign key (LID) to Loan (LID)
account(AID, BID, balance)	Foreign key (BID) to branch(BID)
deposit(CID, AID)	Foreign key (CID) to Customer(CID) Foreign key (AID) to account (AID)

SQL statements:

```
>> Create table customer(CID char(6), CName char(50), CStreet char(50), CCity char(50), primary key(CID));
>> Create table branch (BID char(4), BName char(50), BCity char(50), primary key(BID));
>> Create table loan (LID char(4), BID char(4), amount float, primary key(LID), foreign key(BID) references branch(BID));
>> Create table borrow (LID char(4), CID char(6), primary key(LID, CID), foreign key(LID) references loan(LID), foreign key(CID) references customer(CID) );
>> Create table account (AID char(4), BID char(4), balance float, primary key(AID), foreign key(BID) references branch (BID));
>> Create table deposit (AID char(4), CID char(6), primary key(AID, CID), foreign key(AID) references account (AID), foreign key(CID) references customer (CID));
```

Insert of data:

```
>> insert into customer values(‘C00001’, ‘Ahmad1’, ‘Amman street’,’Amman’);
>> insert into customer(CID, CName, CCity) values(‘C00002’, ‘Ahmad2’,’Amman’);
>> insert into customer(CID, CName, CCity) values(‘C00003’, ‘Ahmad3’,’Amman’);
```

Simple query:

```
>> select * from customer;
>> select CID, CCity from customer;
>> select CName, CCity from customer, borrow where customer.CID = borrow.CID;
>> select CName, balance from customer, deposit, account where customer.CID = borrow.CID and deposit.AID = account.AID;
>> select CName, balance from customer, deposit, account where customer.CID = borrow.CID and deposit.AID = account.AID and balance >= 10000;
```