CMSC424: Database Design

Module: Design: E/R Models and Normalization

Design Process

Instructor: Amol Deshpande
amol@umd.edu

Design Process; E/R Basics

- Book Chapters (6th Edition)
  - Sections 7.1

- Key Topics
  - Steps in application and database design process
  - Two approaches to doing database design

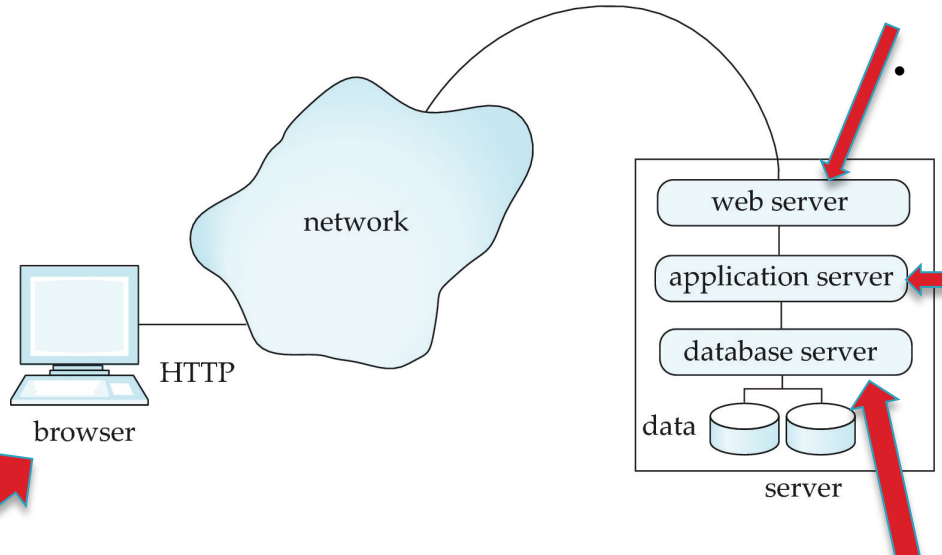
Design Process

- To create an end-to-end database-backed application, we must:
  - Design the database schema for hosting the data
  - Design the application programs for accessing and updating the data
  - Design security schemes to control access to the data

- Typically an iterative process, involving many decision points and stakeholders
  - *computing environments, where to deploy, how to host, languages to use, data model, database systems, application frameworks, etc. etc.*

- Need clear understanding of user requirements
  - Followed by conceptual designs → functional requirements → physical designs → implementation
  - Need to keep revisiting earlier decisions as requirements evolve

What runs where?



- Flask, Django, Tomcat, Node.js, and others
- Accept requests from the client and pass to the application server
- Pass application server response back to the client
- Support HTTP and HTTPS connections

- Encapsulates business logic
- Needs to support different user flows
- Needs to handle all of the rendering and visualization
- Ruby-on-rails, Django, Flask, Angular, React, PHP, and many others

1. Web Browser (Firefox, Chrome, Safari, Edge)
2. HTML to render webpages
3. Javascript for “client-side scripting” (running code in your browser without contacting the server)
4. Flash (not supported much – too much security risk)
5. Java “applets” – less common today

- PostgreSQL, Oracle, SQL Server, Amazon RDS (Relational Databases)
- MongoDB (Document/JSON databases)
- SQLite --- not typically for production environments
- Pretty much any database can be used…

“Database” Design

- Goal: design the logical database schema
  - Try to avoid redundancy
    - Can lead to inconsistencies and require manual intervention
    - Makes it harder to program against the database
      - Need additional code/processes to update everywhere
      - Harder to make schema changes and migrate data
  - Ensure faithfulness to the requirements
    - Need to make sure it supports the use cases and the application requirements
    - Capturing all the data properly
      - Any data properties not captured cannot be stored in the database
    - Capture the constraints accurately
      - e.g., don’t want to set `s_id` as the primary key for `advisor(s_id, i_id)` if we expect multiple advisors for a student
- Need a systematic way to do this for large schemas

"Database" Design

- Approach 1: Using a logical data model like the Entity-Relationship Model
  - Easier for humans to work with and visualize
  - Abstracts away the details, and allows focusing on the important issues
  - Richer than relational model, but allows easy conversion to relational for implementation
  - Harder to keep up to date – requires a lot of discipline
- Approach 2: Normalization Theory
  - Helps formalize the key design pitfalls and how to avoid them
- The two approaches are complementary and important to know both of them

Schema ”Evolution” and Challenges

- Initial application schema nicely designed and normalized
- But as business requirements changes,
  - Schemas need to be modified
  - Data needs to be ”migrated” from old schema to new schema
- Ideally the new schema is also normalized and properly designed
- However…
  - More changes to schema → More changes to applications running on top
  - Incremental schema changes often preferred by developers
  - Result: After a few iterations, the schema is not properly normalized any more
- No good solutions to date
  - Using “views” can help, but also requires discipline
  - Things we discuss here provide the foundations needed…

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Basics of E/R Models

Instructor: Amol Deshpande
amol@umd.edu

Basics of E/R Modeling

- Book Chapters (6th Edition)
  - Sections 7.2, 7.3.1, 7.3.3, 7.5.1-7.5.5

- Key Topics
  - Basics
  - Different types of attributes
  - Cardinalities of relationships
  - How to identify ”keys” for relationships

Entity-Relationship Model

- Two key concepts
  - *Entities*:
    - An object that *exists* and is *distinguishable* from other objects
      - Examples: Bob Smith, BofA, CMSC424
    - Have *attributes* (people have names and addresses)
    - Form *entity sets* with other entities of the same type that share the same properties
      - Set of all people, set of all classes
    - Entity sets may overlap
      - Customers and Employees

Entity-Relationship Model

- Two key concepts
  - *Relationships*:
    - Relate 2 or more entities
      - E.g. Bob Smith *has account at* College Park Branch
    - Form *relationship sets* with other relationships of the same type that share the same properties
      - Customers *have accounts at* Branches
    - Can have attributes:
      - *has account at* may have an attribute *start-date*
    - Can involve more than 2 entities
      - Employee *works at* Branch *at* Job

Entities and relationships

Two Entity Sets

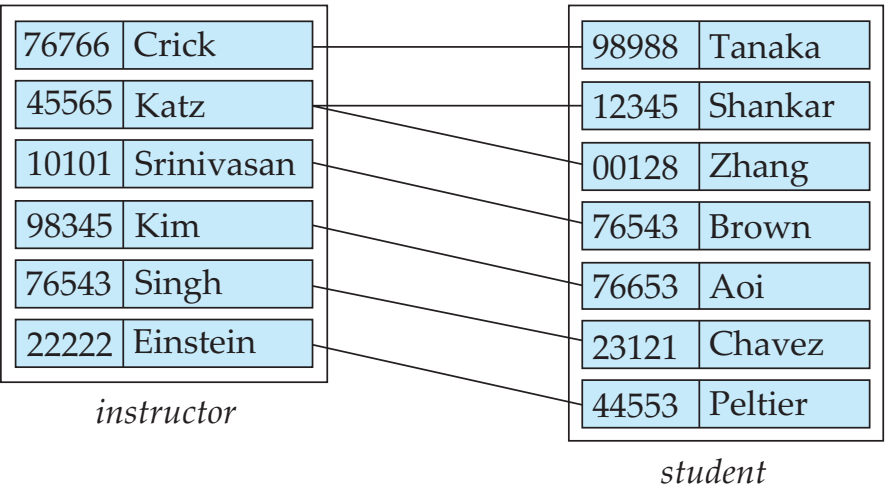
76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

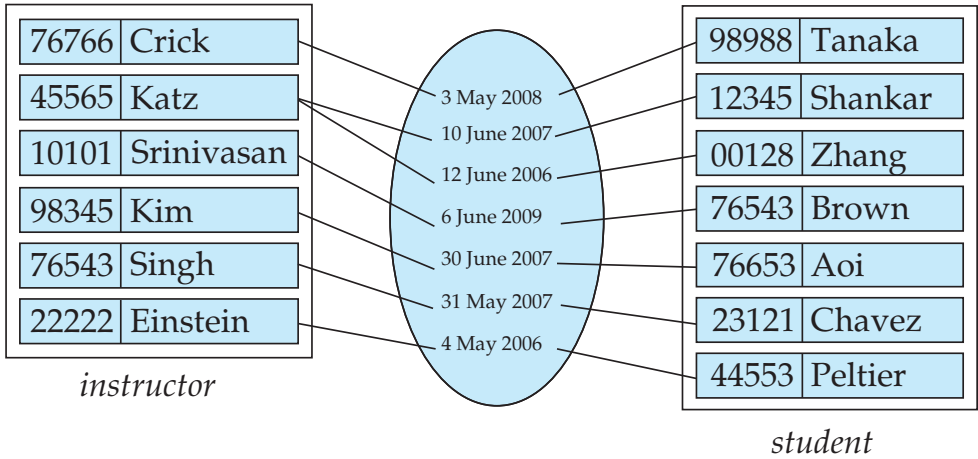
instructor

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

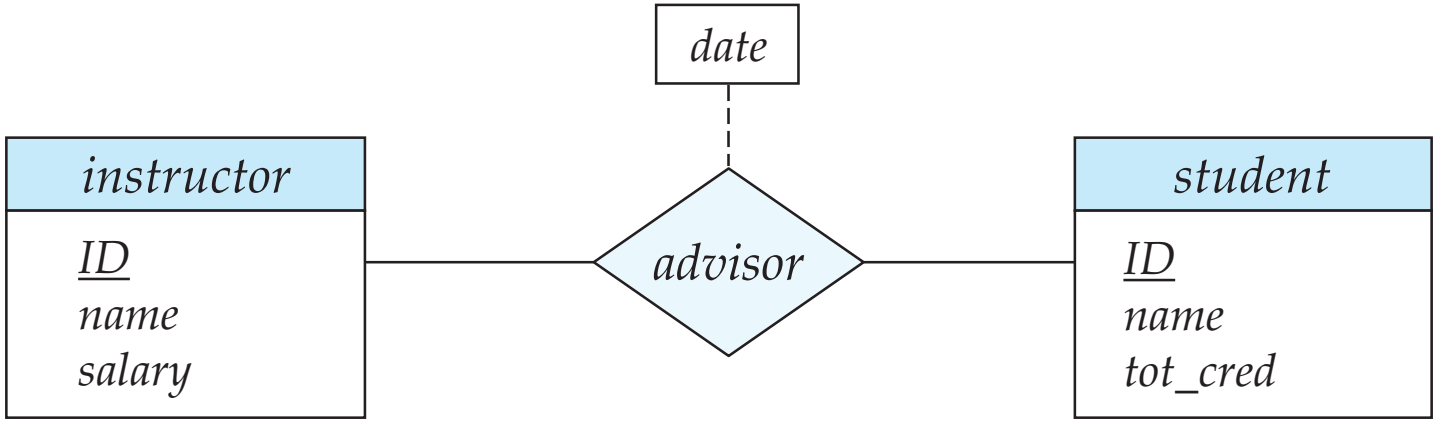
student

Advisor Relationship, with and without attributes

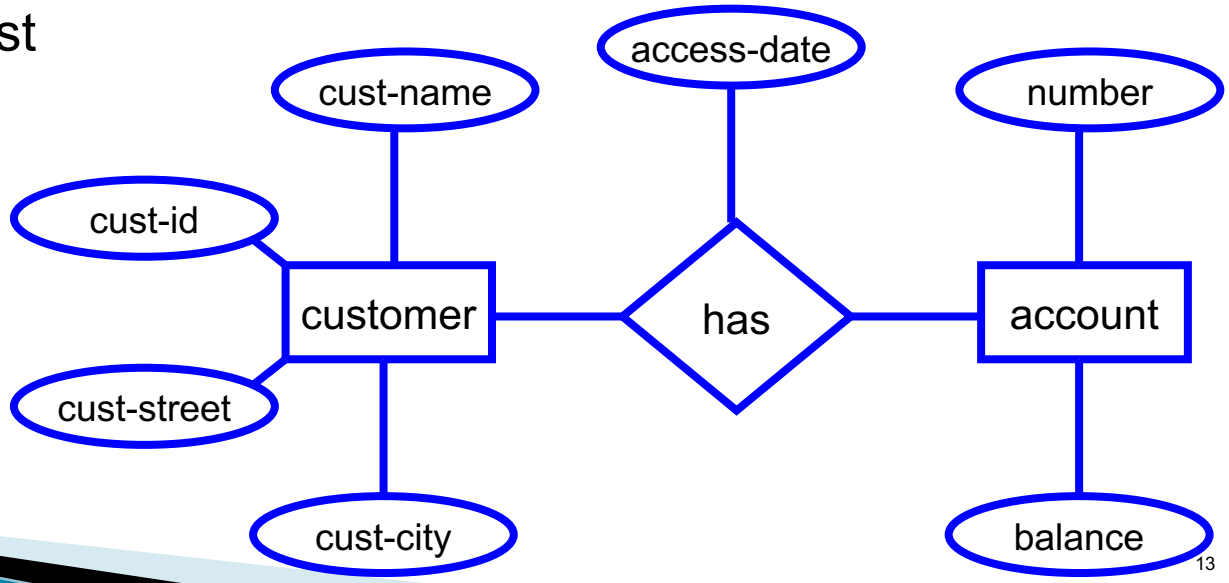




ER Diagram



Alternative representation, used in the book in the past

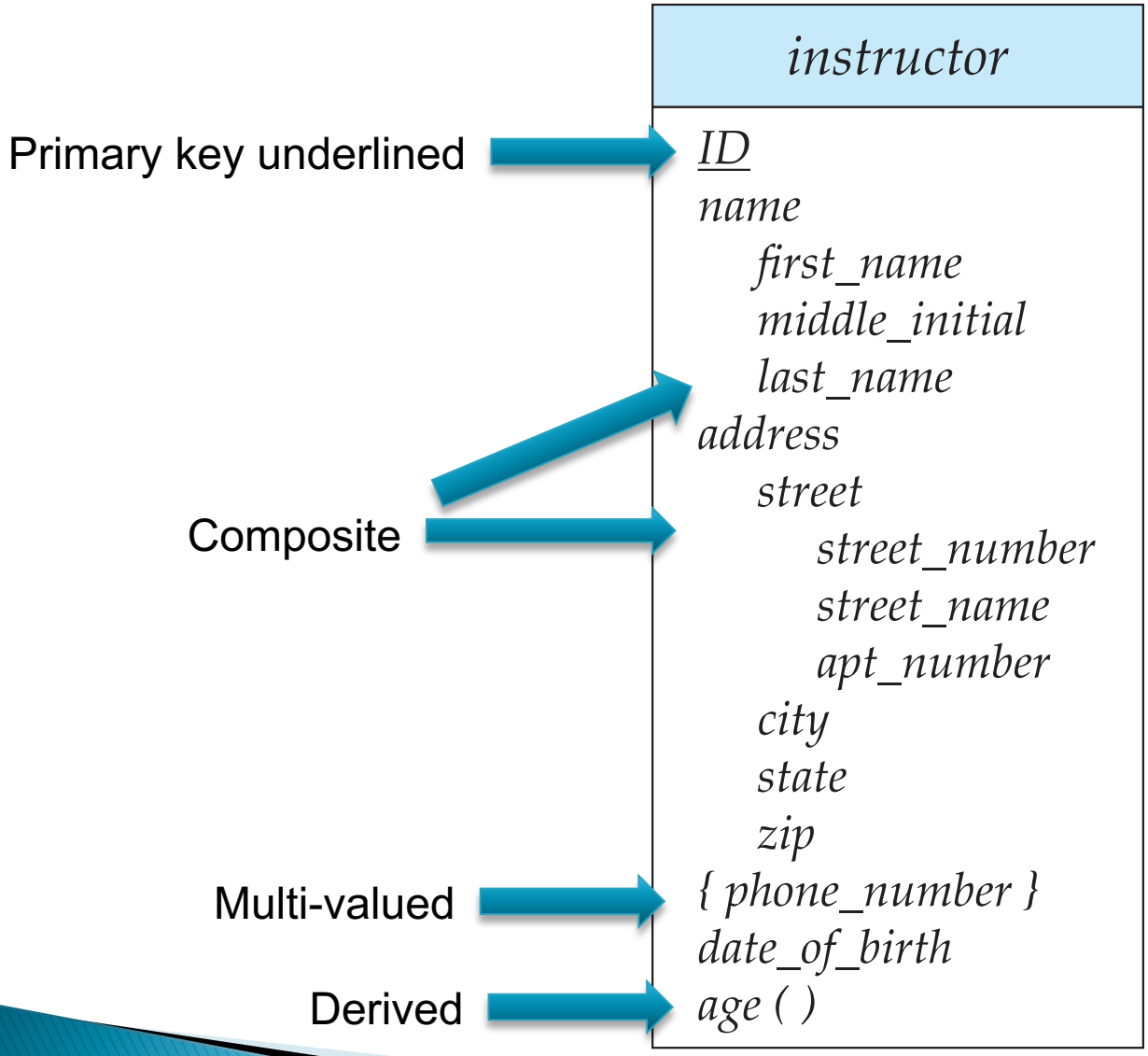


Both notations used commonly

Types of Attributes

- Simple vs Composite
  - Single value per attribute ?
- Single-valued vs Multi-valued
  - E.g. Phone numbers are multi-valued
- Derived
  - If date-of-birth is present, age can be derived
  - Can help in avoiding redundancy, enforcing constraints etc…

Types of Attributes

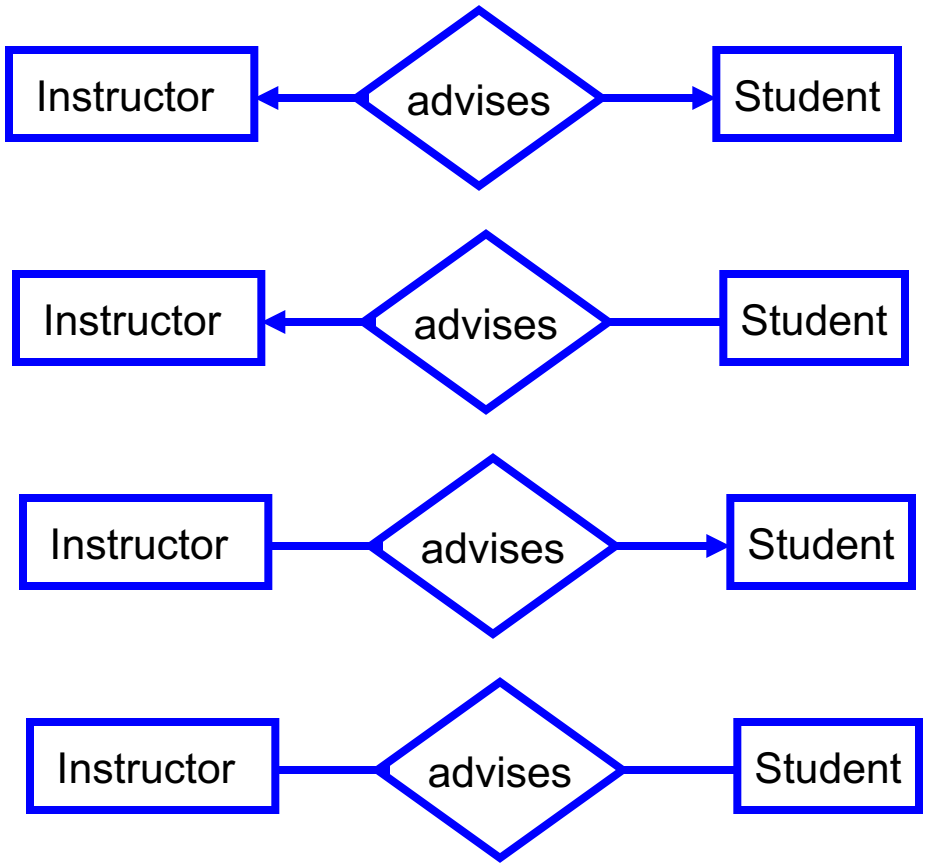


Relationship Cardinalities

- We may know:
  - One customer can only open one account
  - OR
  - One customer can open multiple accounts
- Representing this is important
- Why ?
  - Better manipulation of data
    - If former, can store the account info in the customer table
  - Can enforce such a constraint
    - Application logic will have to do it; NOT GOOD
  - Remember: If not represented in conceptual model, the domain knowledge may be lost

Mapping Cardinalities

- One-to-One
- One-to-Many
- Many-to-One
- Many-to-Many



Mapping Cardinalities

- Express the number of entities to which another entity can be associated via a relationship set
- Most useful in describing binary relationship sets
- N-ary relationships ?
  - More complicated
  - Details in the book

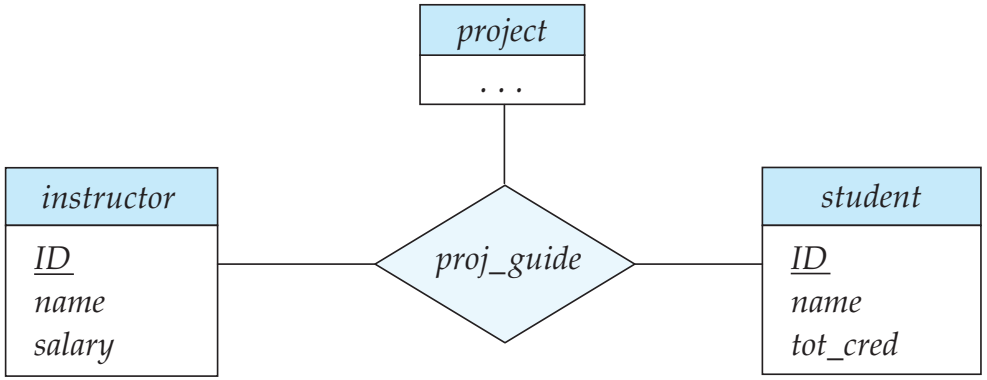


Figure 7.13 E-R diagram with a ternary relationship.

Relationship Set Keys

- What attributes are needed to represent a relationship completely and uniquely ?
  - Union of primary keys of the entities involved, and relationship attributes

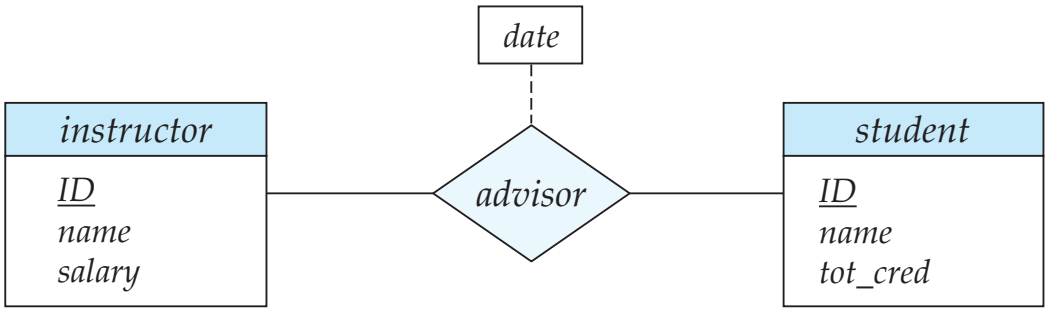


Figure 7.8 E-R diagram with an attribute attached to a relationship set.

  - {instructor.ID, date, student.ID} describes a relationship completely

Relationship Set Keys

- Is *{student_id, date, instructor_id}* a candidate key *?*
  - No. Attribute *date* can be removed from this set without losing key-ness
  - In fact, union of primary keys of associated entities is always a superkey

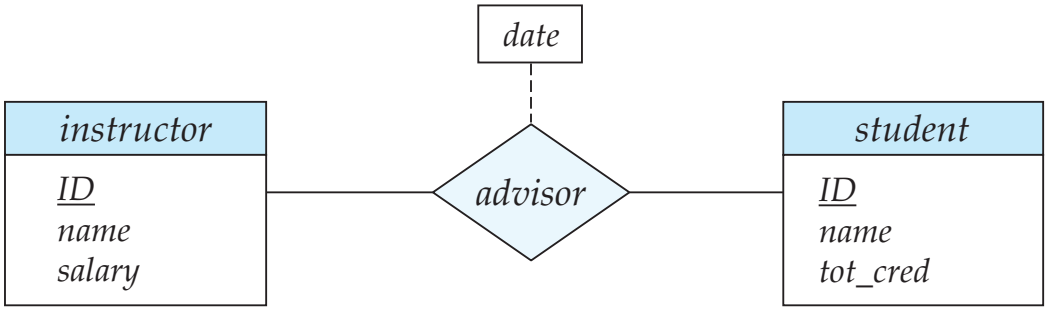


Figure 7.8 E-R diagram with an attribute attached to a relationship set.

Relationship Set Keys

- Is {student_id, instructor_id} a candidate key ?
  - Depends

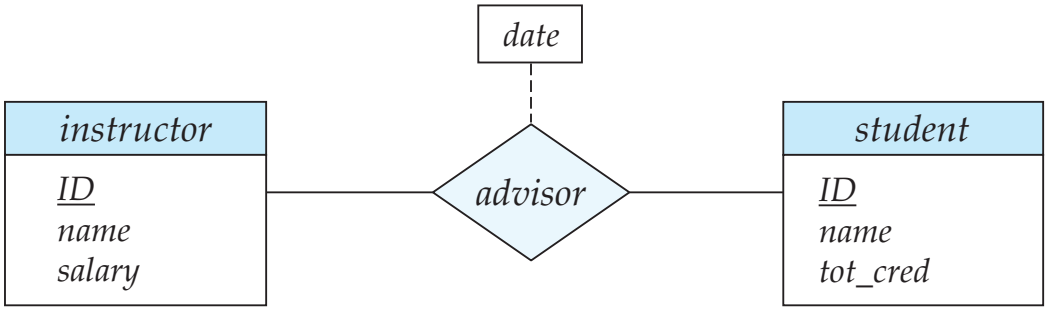


Figure 7.8 E-R diagram with an attribute attached to a relationship set.

Relationship Set Keys

- Is {student_id, instructor_id} a candidate key ?
  - Depends

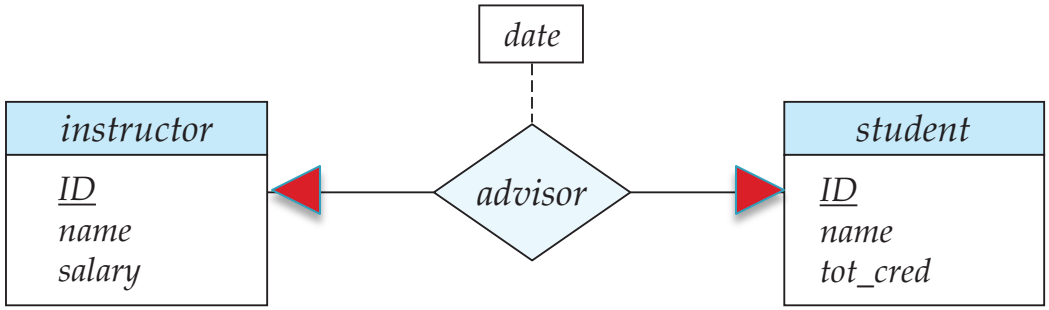


Figure 7.8 E-R diagram with an attribute attached to a relationship set.

- If one-to-one relationship, either *{instructor_id}* or *{student_id}* sufficient
  - Since a given *instructor* can only have one *advisee*, an instructor entity can only participate in one relationship
  - Ditto *student*

Relationship Set Keys

- Is {student_id, instructor_id} a candidate key ?
  - Depends

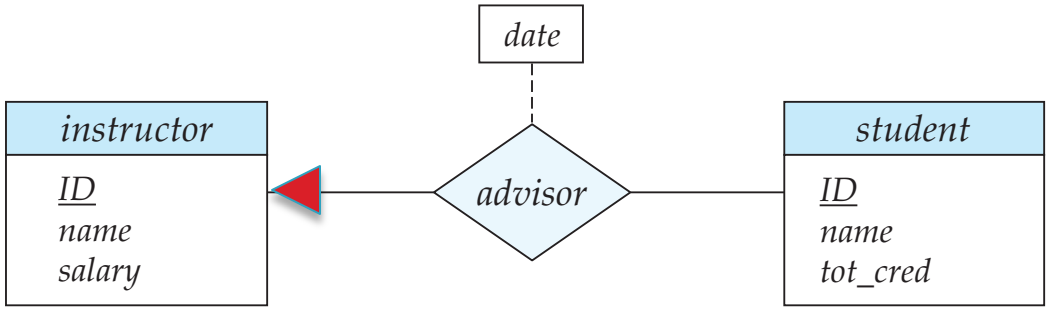


Figure 7.8 E-R diagram with an attribute attached to a relationship set.

- If one-to-many relationship (as shown), *{student_id}* is a candidate key
  - A given instructor can have many advisees, but at most one advisor per student allowed

Relationship Set Keys

- General rule for binary relationships
  - one-to-one: primary key of either entity set
  - one-to-many: primary key of the entity set on the many side
  - many-to-many: union of primary keys of the associate entity sets
- n-ary relationships
  - More complicated rules

CMSC424: Database Design

Module: Design: E/R Models and Normalization

More E/R Constructs

Instructor: Amol Deshpande
amol@umd.edu

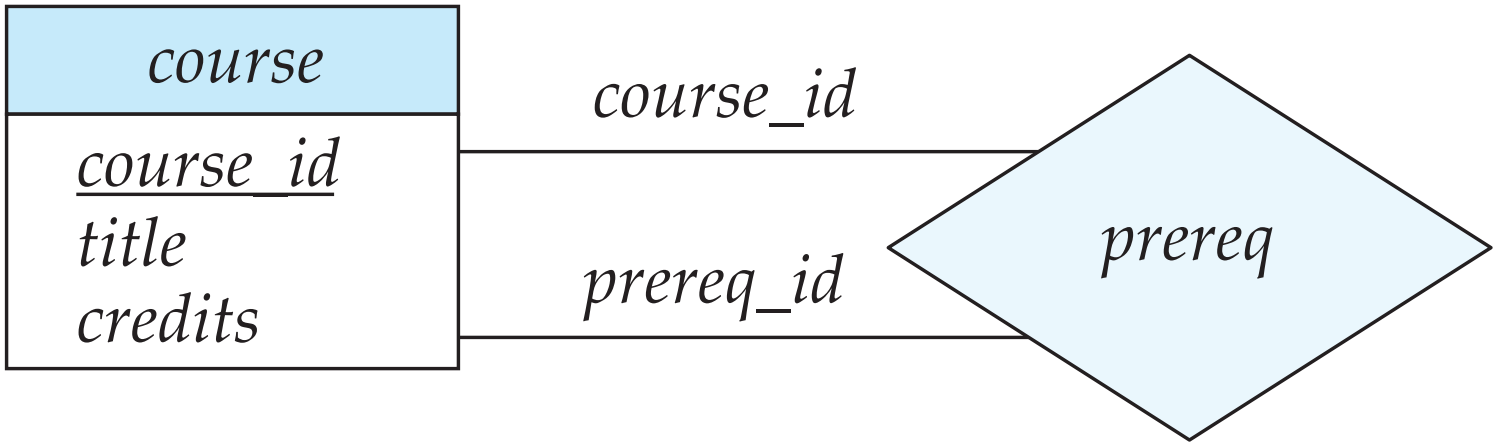
More E/R Constructs

- Book Chapters (6th Edition)
  - Sections 7.5.4, 7.5.6, 7.8

- Key Topics
  - Recursive Relationships and Roles
  - Weak Entity Sets
  - Specialization/Generalization
  - Aggregation

Recursive Relationships

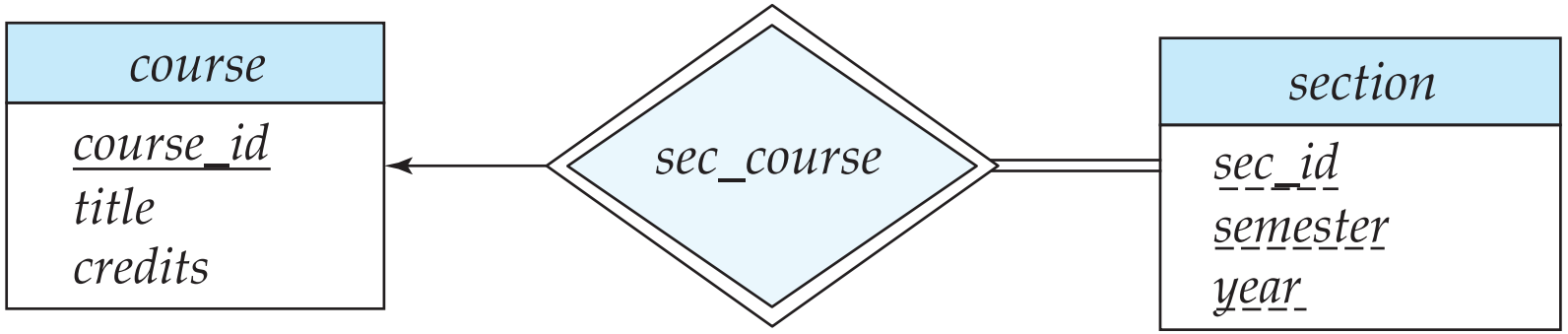
- Sometimes a relationship associates an entity set to itself
- Need “roles” to distinguish



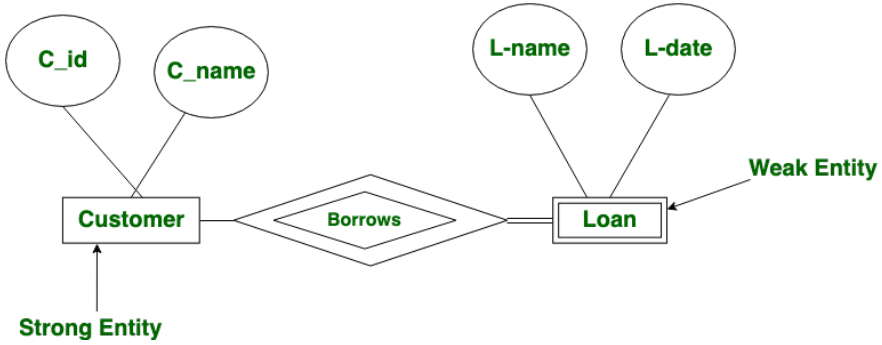
course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Weak Entity Sets

- An entity set without enough attributes to have a primary key
  - E.g. Section Entity
- Still need to be able to distinguish between weak entities
  - Called “discriminator attributes”: dashed underline

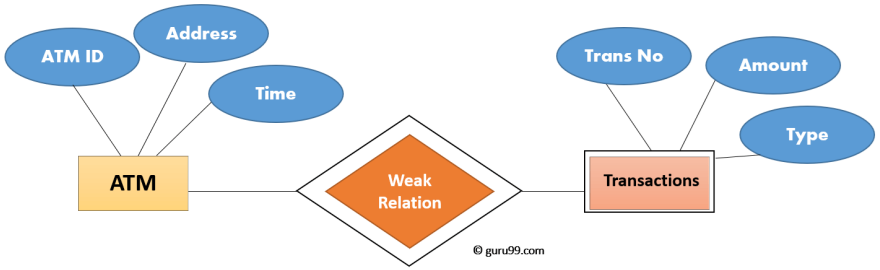


Examples of Weak Entity Sets

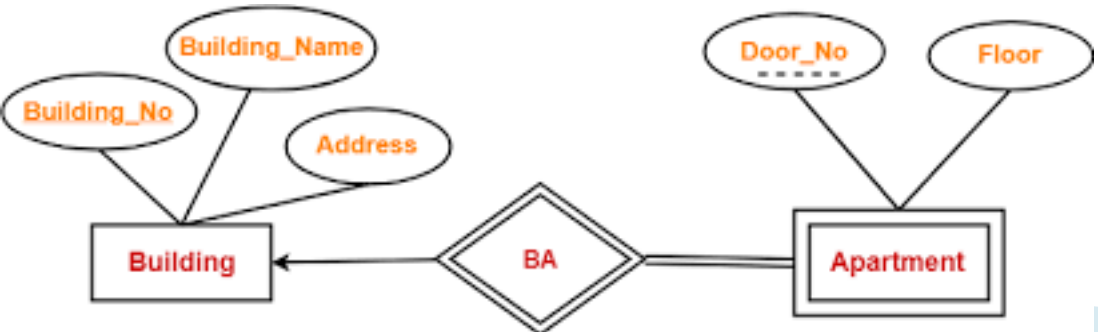


Loan may or may not have an extra unique identifier

If transaction numbers are per ATM (i.e., first transaction from that ATM gets number 1, etc.), then Transactions is a weak entity

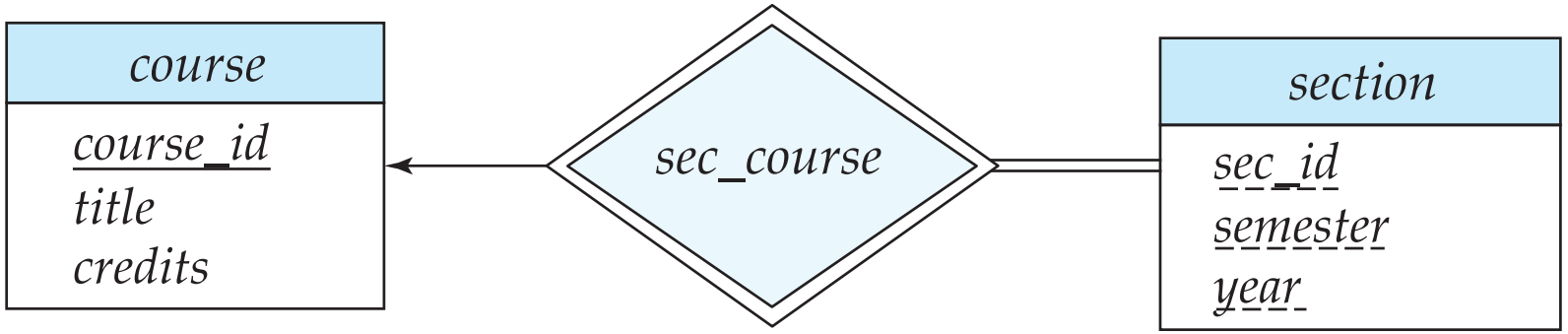


Apartments don't have a unique identifer (across all buildings) without the building information



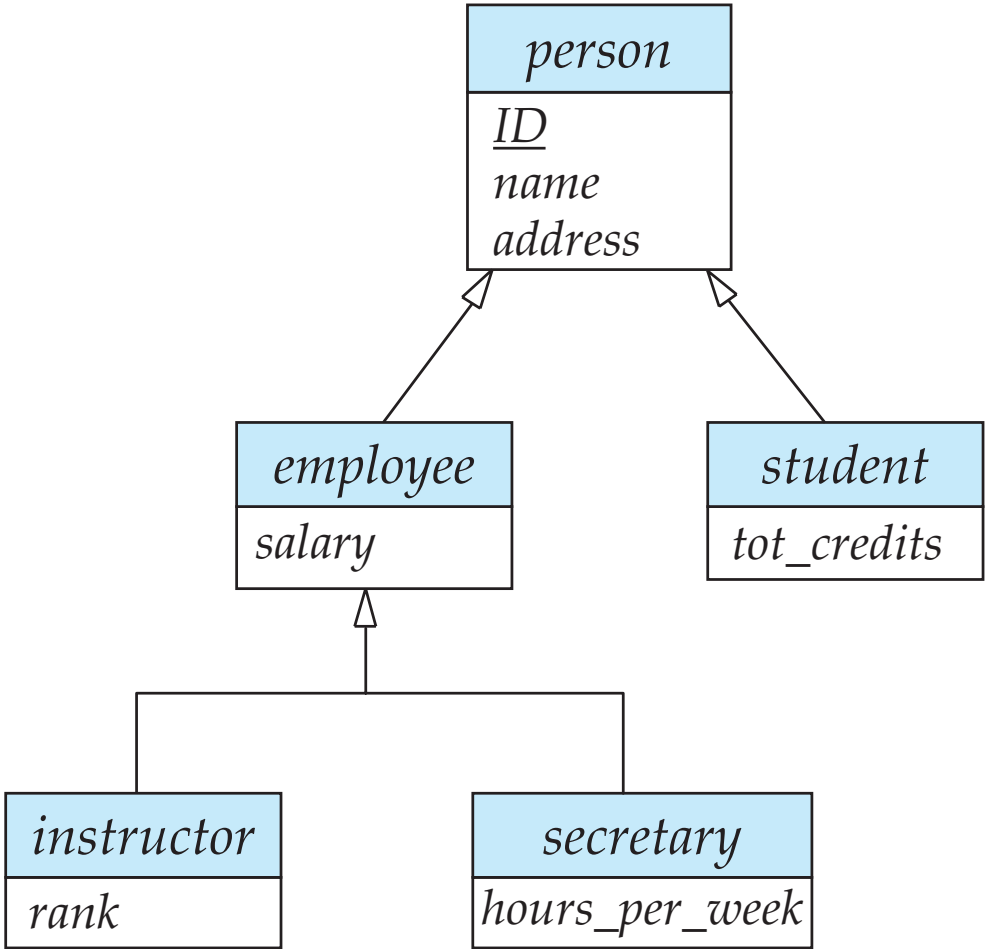
Participation Constraints

- Allow specifying full participation from an entity set in a relationship
  - i.e., every entity from that entity set ”must” participate in at least one relationship
  - Most common for Weak Entity Sets, but useful otherwise as well



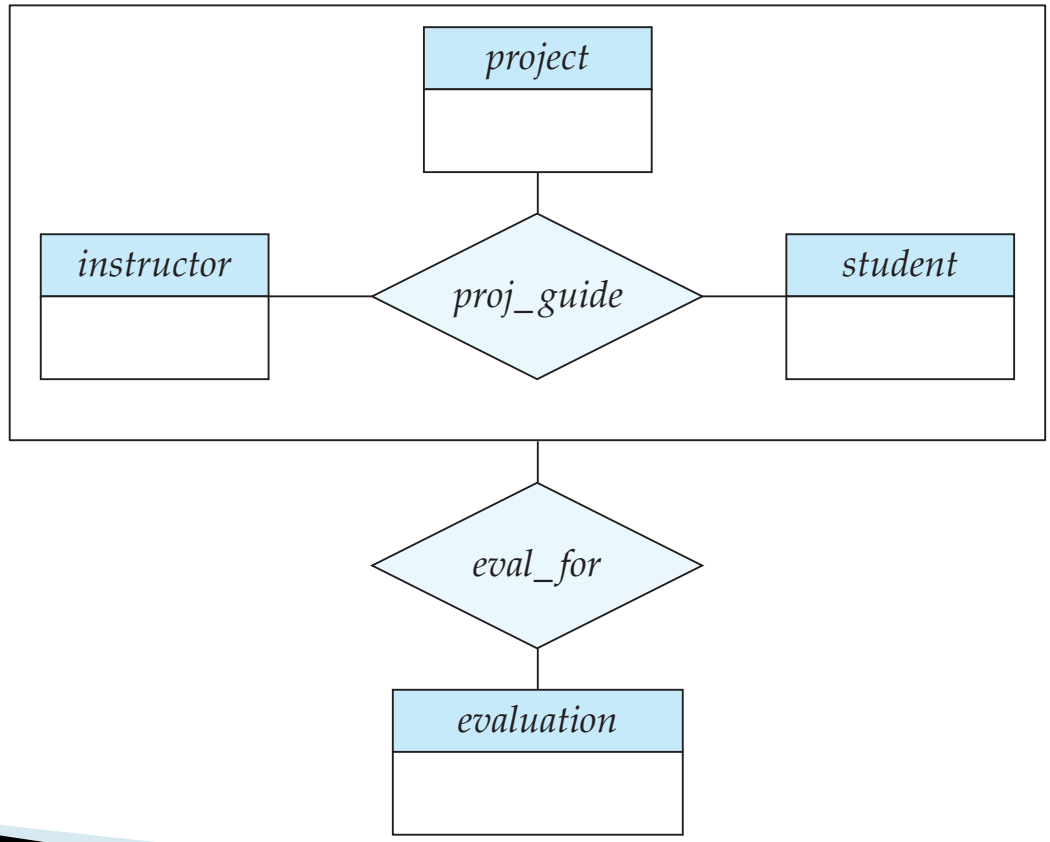
Specialization/Generalization

Similar to object-oriented programming: allows inheritance etc.



Aggregation

- No relationships allowed between relationships
- Suppose we want to record evaluations of a student by a guide on a project



CMSC424: Database Design

Module: Design: E/R Models and Normalization

Converting to Relational

Instructor: Amol Deshpande
amol@umd.edu

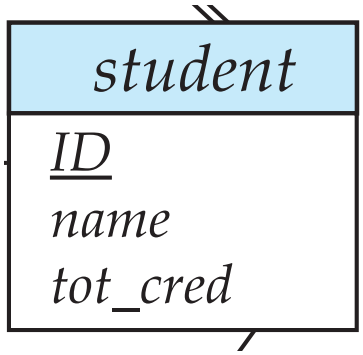
Converting E/R Models to Relations

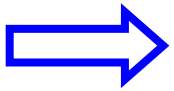
- Book Chapters (6th Edition)
  - Sections 7.6, 7.8.6

- Key Topics
  - Creating Relational Schema from an E/R Model
  - Mapping Entities and Relationships to Relations
  - Weak Entity Sets to Relations
  - Other E/R Constructs

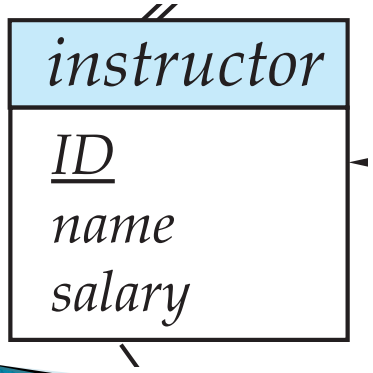
E/R Diagrams → Relations

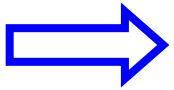
- Convert entity sets into a relational schema with the same set of attributes





Student (ID, name, tot_cred)

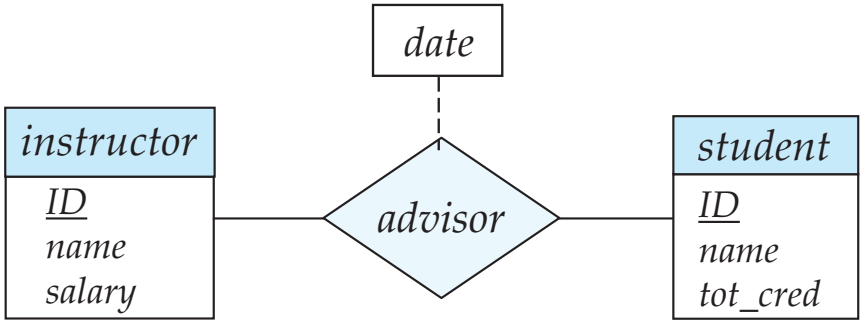




Instructor(ID, name, salary)

E/R Diagrams → Relations

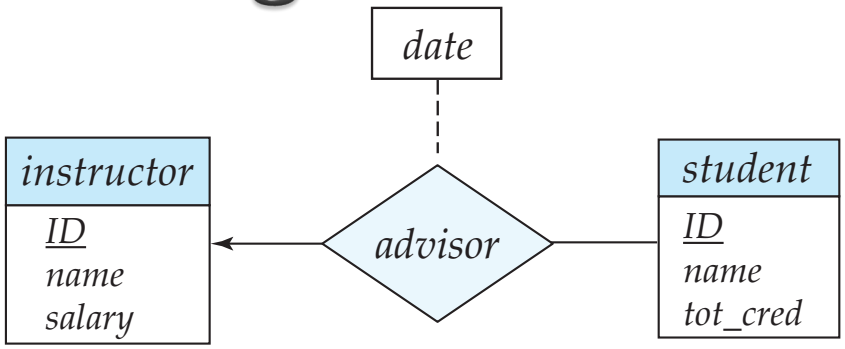
- Convert relationship sets *also* into a relational schema
- Remember: A relationship is completely described by primary keys of associated entities and its own attributes



We can do better for many-to-one or one-to-one

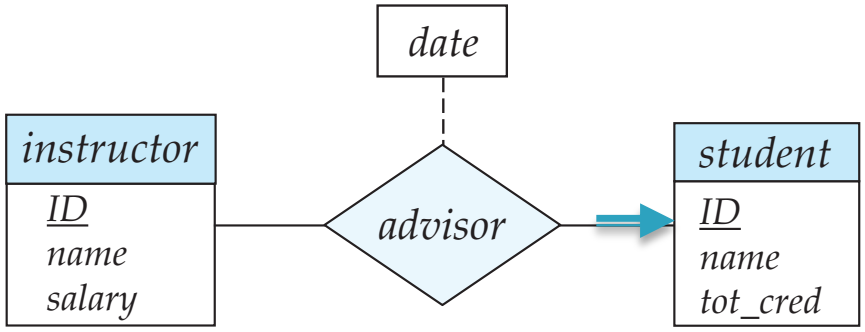
Advisor (student_ID, instructor_ID, date)

E/R Diagrams → Relations



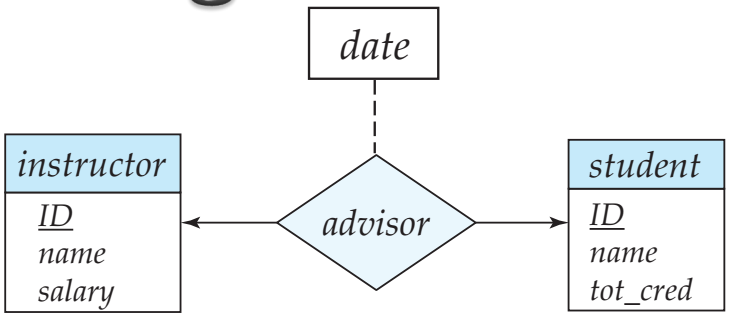
Foreign key into Instructor relation

Fold into Student:
Student(ID, name, tot_credits, advisor_ID, date)



Fold into Instructor:
Instructor(ID, name, salary, advisee_ID, date)

E/R Diagrams → Relations



Fold into Student:
Student(ID, name, tot_credits, advisor_ID)

OR

Fold into Instructor:
Instructor(ID, name, salary, advisee_ID)

Weak Entity Sets

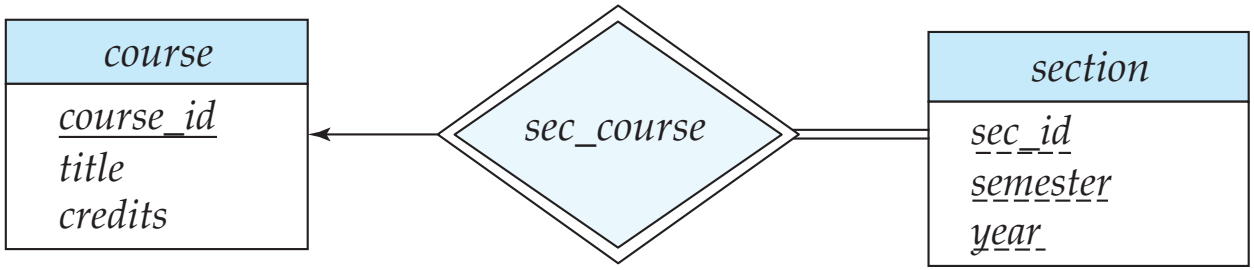


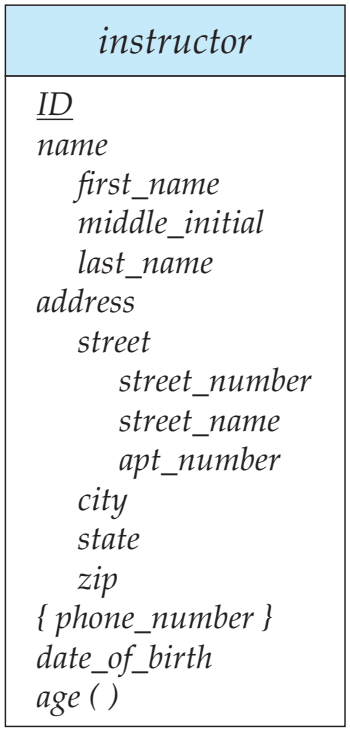
Figure 7.14 E-R diagram with a weak entity set.

Need to copy the primary key from the strong entity set:

Section(course_id, sec_id, semester, year)

Primary key for section = Primary key for course + Discriminator Attributes

Multi-valued Attributes



instructor (*ID*, *first_name*, *middle_name*, *last_name*, *street_number*, *street_name*, *apt_number*, *city*, *state*, *zip_code*, *date_of_birth*)

BUT

Phone_number needs to be split out into a separate table

Instructor_Phone(Instructor_ID, phone_number)

Specialization and Generalization

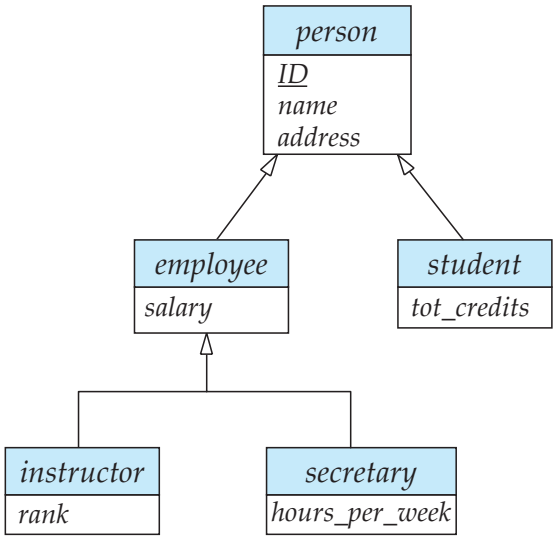


Figure 7.21 Specialization and generalization.

A few different ways to handle it

1. Common table for common information and separate tables for additional information

   person (<u>*ID*</u>, *name*, *street*, *city*)
   employee (<u>*ID*</u>, *salary*)
   student (<u>*ID*</u>, *tot_cred*)

2. Separate tables altogether – good idea if an employee can't be a student also – querying becomes harder (have to do unions for queries across all "persons")

   employee (<u>*ID*</u>, *name*, *street*, *city*, *salary*)
   student (<u>*ID*</u>, *name*, *street*, *city*, *tot_cred*)

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Design Issues;
Alternate Notations

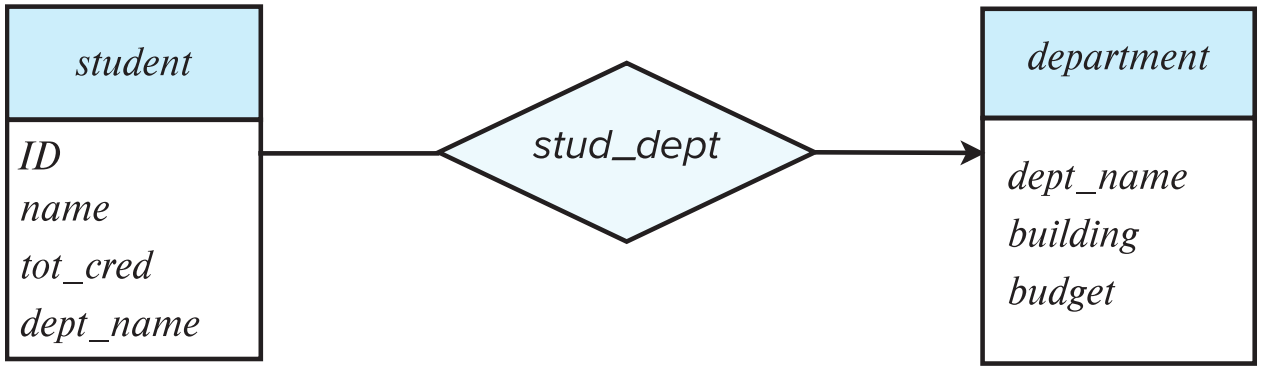
Instructor: Amol Deshpande
amol@umd.edu

Design Issues; Alternate Notations

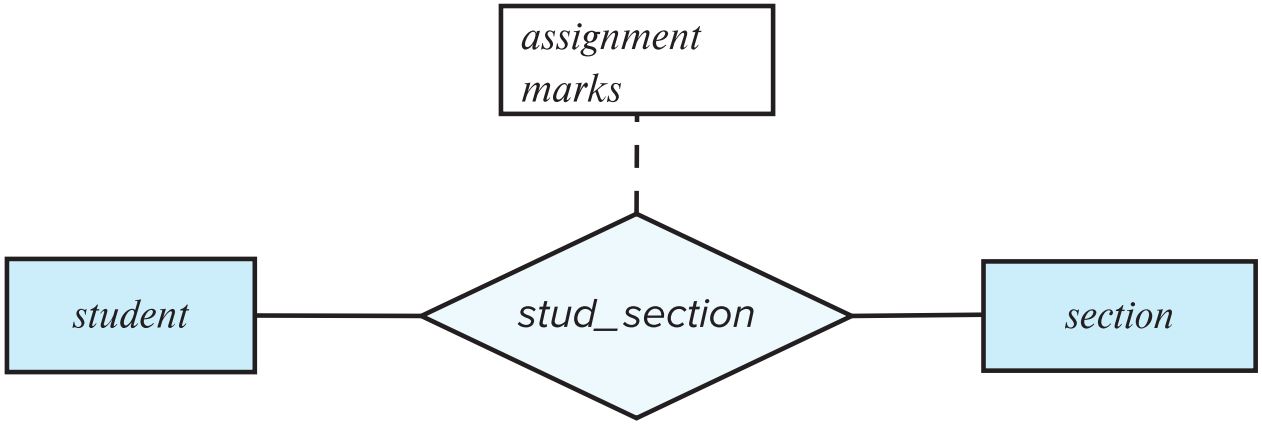
- Book Chapters (6th Edition)
  - Sections 7.7, 7.9 (briefly)

- Key Topics
  - Some Common Mistakes
  - Choosing between different ways to do the same thing
  - Alternate notations commonly used (including UML)
  - Recap

Some Common Mistakes

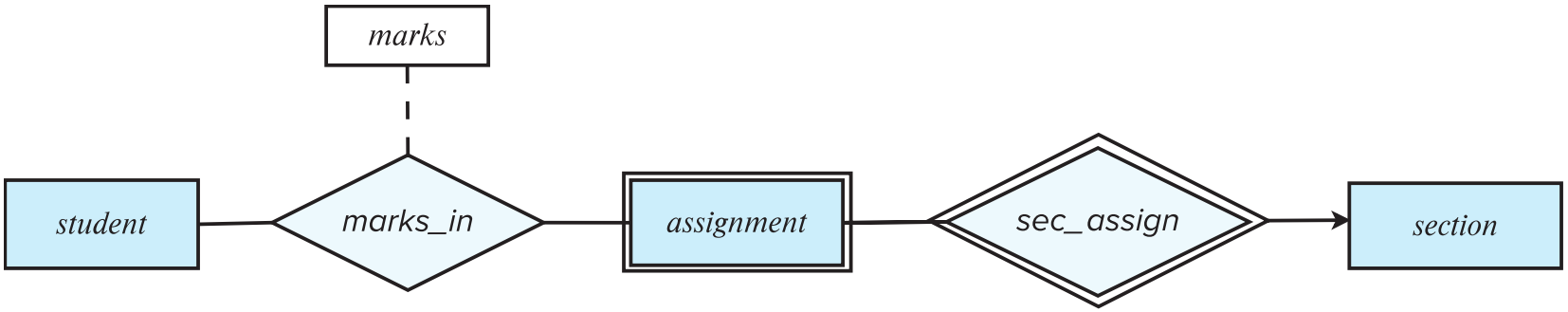


(a) Incorrect use of attribute

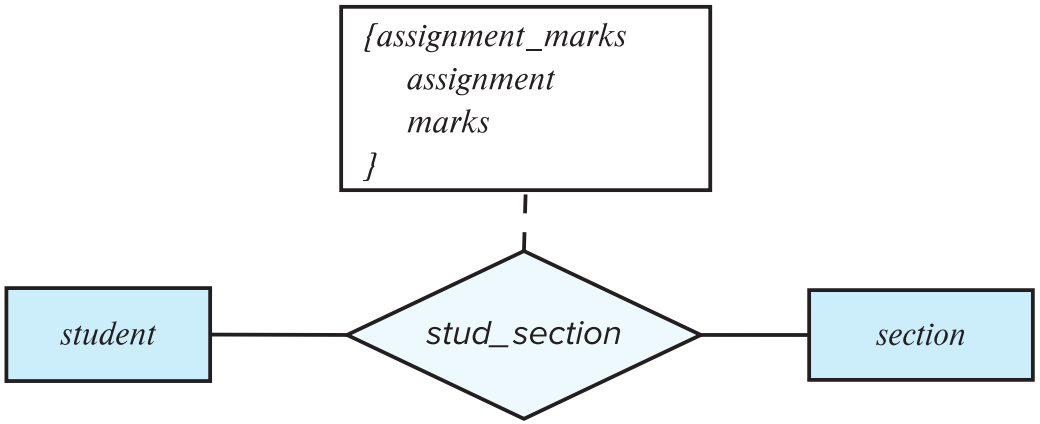


(b) Erroneous use of relationship attributes

Some Common Mistakes



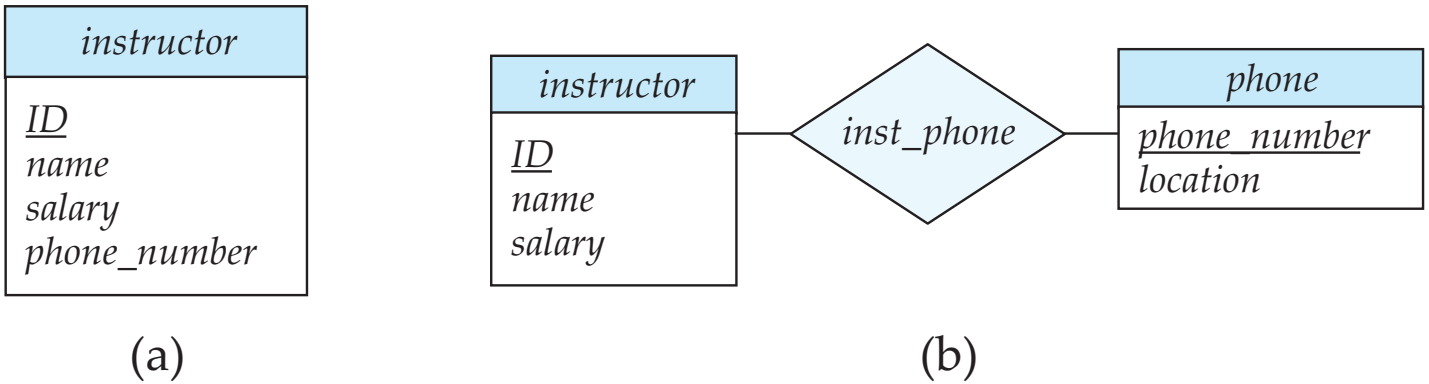
(c) Correct alternative to erroneous E-R diagram (b)



(d) Correct alternative to erroneous E-R diagram (b)

Design Issues

- Entity sets vs attributes
  - Depends on the semantics of the application
  - Consider *telephone*



Design Issues

- Entity sets vs Relationsihp sets
  - Consider *takes*

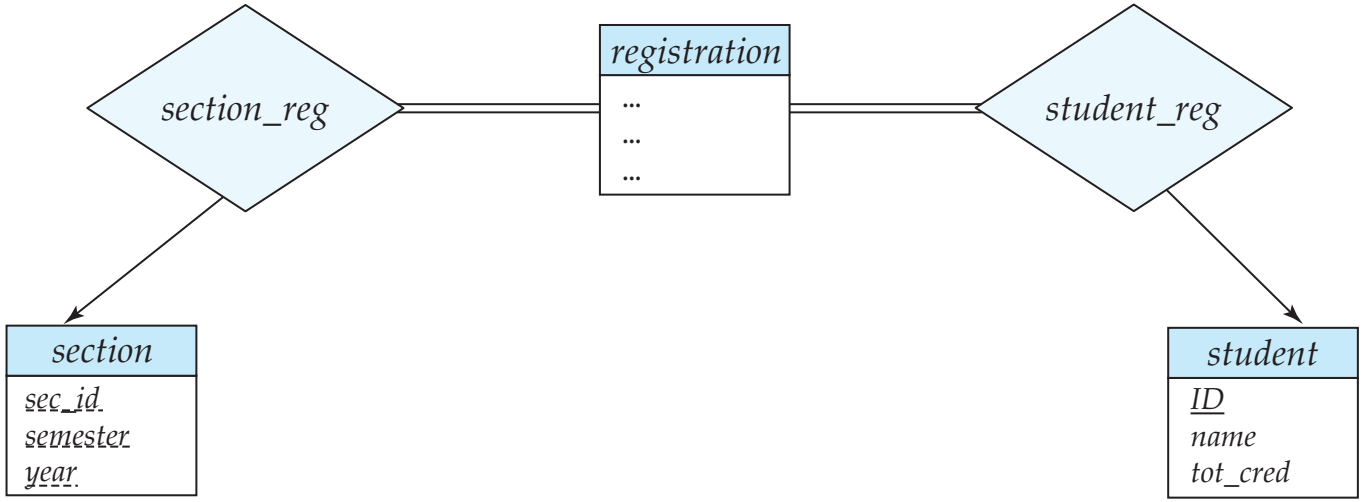


Figure 7.18 Replacement of *takes* by *registration* and two relationship sets

Design Issues

- N-ary vs binary relationships
  - Possible to avoid n-ary relationships, but there are some cases where it is advantageous to use them

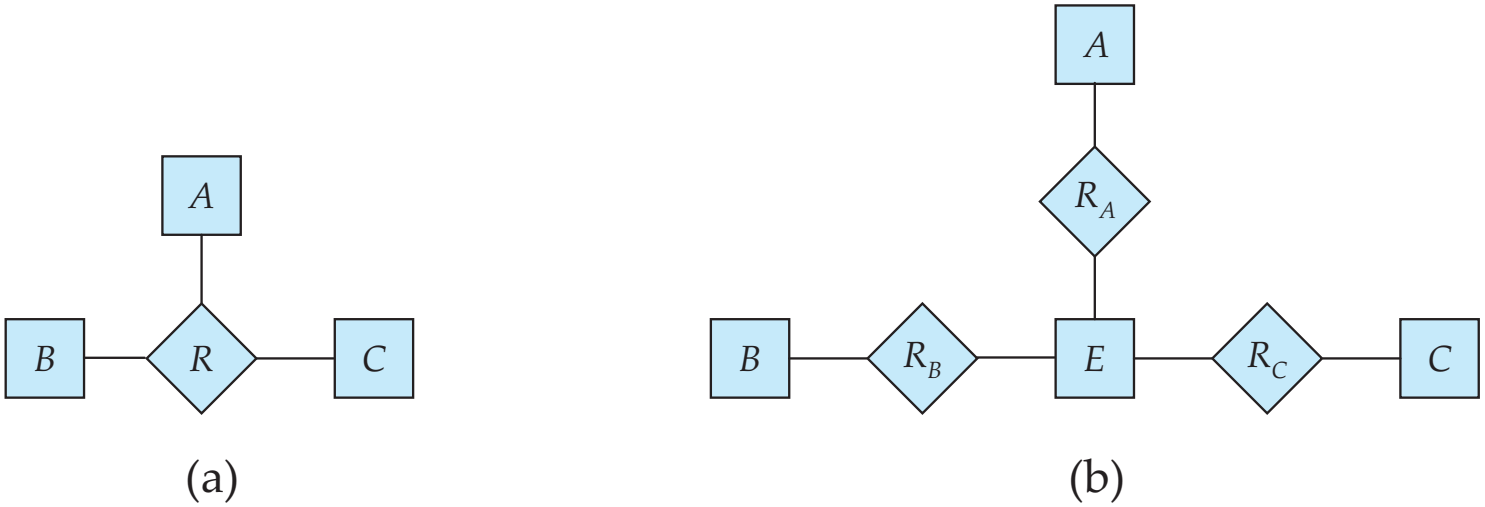


Figure 7.19 Ternary relationship versus three binary relationships.

Alternate Notations

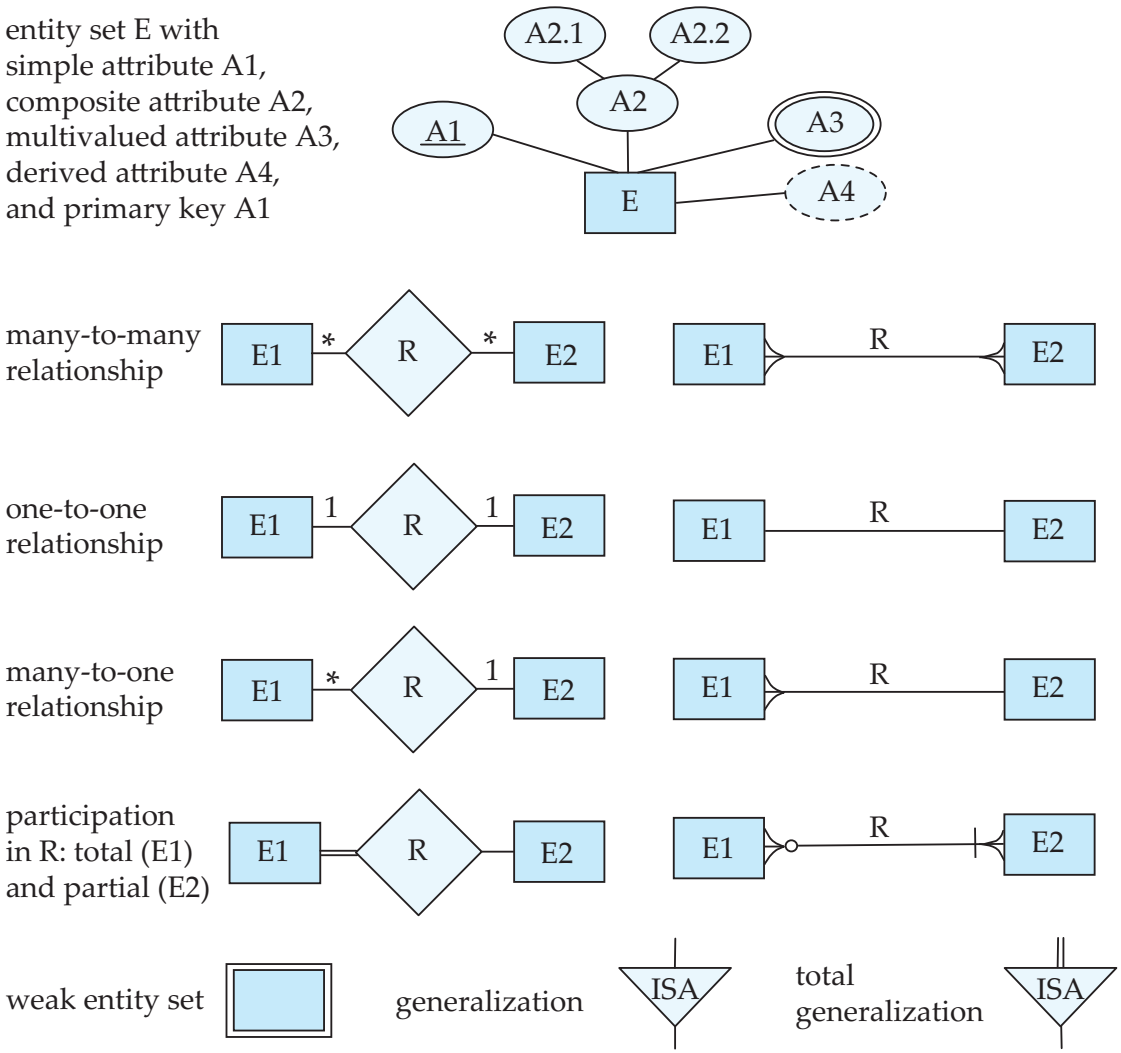
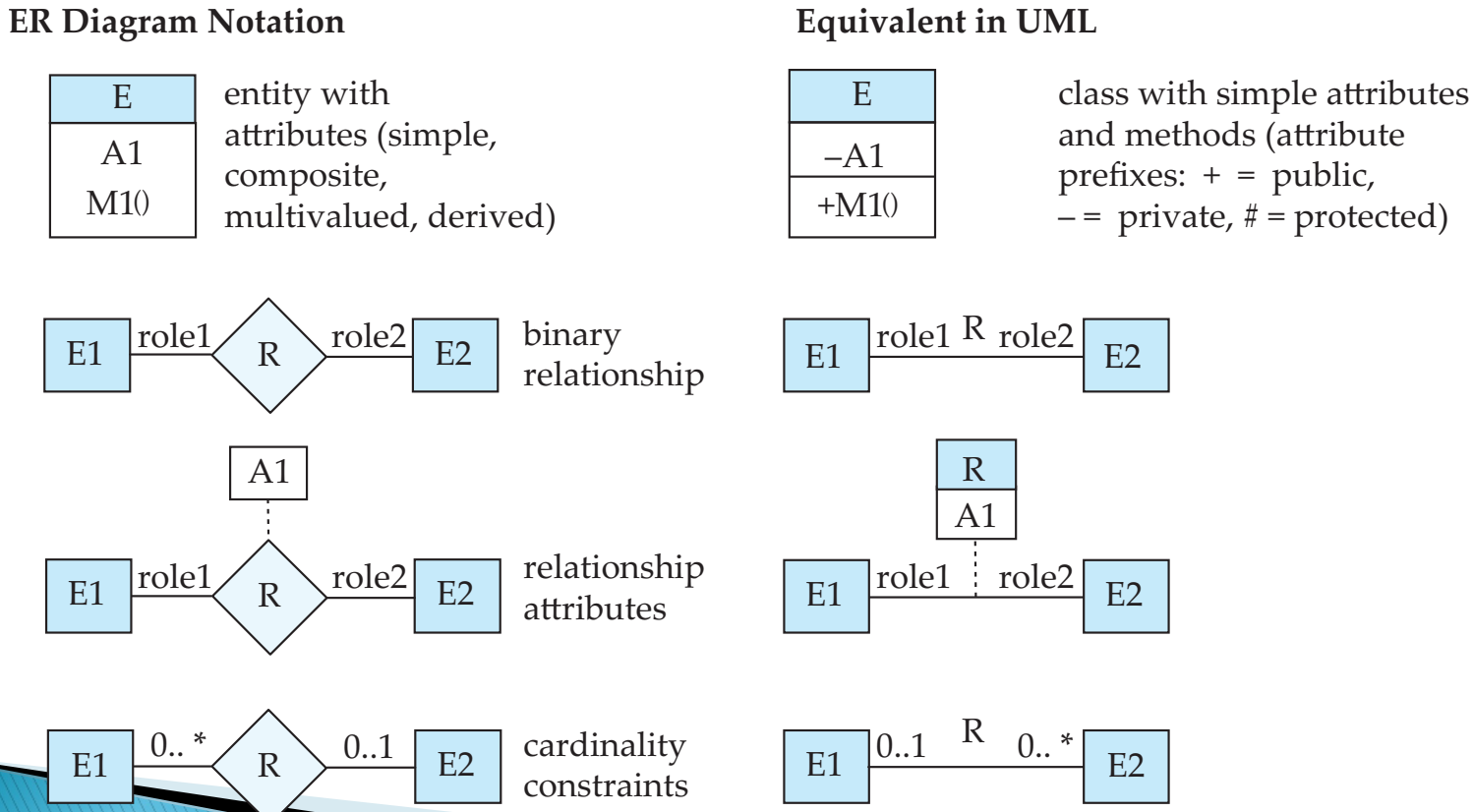


Figure 7.25 Alternative E-R notations.

Unified Modeling Language (UML)

- More comprehensive – covers use cases, flow of tasks between components, implementation diagrams, etc., in addition to data representation



Thoughts…

- Nothing about actual data
  - How is it stored ?
- No talk about the query languages
  - How do we access the data ?
- Semantic vs Syntactic Data Models
  - Remember: E/R Model is used for conceptual modeling
  - Many conceptual models have the same properties
- They are much more about representing the knowledge than about database storage/querying

Thoughts…

- Basic design principles
  - Faithful
    - Must make sense
  - Satisfies the application requirements
  - Models the requisite domain knowledge
    - If not modeled, lost afterwards
  - Avoid redundancy
    - Potential for inconsistencies
  - Go for simplicity
- Typically an iterative process that goes back and forth

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Normalization: Basics

Instructor: Amol Deshpande
amol@umd.edu

Relational Database Design

- Where did we come up with the schema that we used ?
  - E.g. why not store the student course titles with their names ?
- If from an E-R diagram, then:
  - Did we make the right decisions with the E-R diagram ?
- Goals:
  - Formal definition of what it means to be a "good" schema.
  - How to achieve it.
- More abstract and formal than most other topics we will study

Normalization

- Book Chapters (6th Edition)
  - Section 8.1, 8.2

- Key Topics
  - What makes a ”good” schema
  - Problems with small schemas
  - Problems with large schemas
  - Atomic domains and First Normal Form

Simplified University Database Schema

Student(*student_id*, name, tot_cred)

Student_Dept(*student_id, dept_name*)

Department(*dept_name*, building, budget)

Course(*course_id*, title, dept_name, credits)

Takes(*course_id, student_id, semester, year*)

Changed to:

Student_Dept(*student_id, dept_name*, name, tot_cred, building, budget)

<Student, Student_Dept, and Department Merged Together>

Course(*course_id*, title, dept_name, credits)

Takes(*course_id, student_id*, semester, year)

Is this a good schema ???

Student_Dept(*student_id, dept_name*, name, tot_cred, building, budget)

student_id	dept_name	name	tot_cred	building	budget
s1	Comp. Sci.	John	30	Iribe Center	10 M
s2	Comp. Sci.	Alice	20	Iribe Center	10 M
s2	Math	Alice	20	Kirwan Hall	10 M
s3	Comp. Sci.	Mike	30	Iribe Center	10 M
s3	Math	Mike	30	Kirwan Hall	10 M

Issues:

1. Redundancy ➔ higher storage, inconsistencies ("anomalies")

   update anomalies, insertion anamolies

2. Need nulls

   Unable to represent some information without using nulls

   How to store depts w/o students, or vice versa ?

   Can't have NULLs in primary keys

Student_Dept(student_ids, *dept_name*, names, tot_creds, building, budget)

student_ids	dept_name	names	tot_creds	building	budget
{s1, s2, s3}	Comp. Sci.	{John, Alice, Mike}	{30, 20, 30}	Iribe Center	10 M
{s2, s3}	Math	{Alice, Mike}	{20, 30}	Kirwan Hall	10 M

Issues:

3. Avoid sets

- Hard to represent
- Hard to query
- In this case, too many issues

Smaller schemas always good ????

Split Course(course_id, title, dept_name, credits) into:

Course1 (course_id, title, dept_name) Course2(course_id, credits)???

course_id	title	dept_name
c1	"Intro to.."	Comp. Sci.
c2	"Discrete Structures"	Comp. Sci.
c3	"Database Design"	Comp. Sci.

course_id	credits
c1	3
c2	3
c3	3

This process is also called *"decomposition"*

Issues:

4. Requires more joins (w/o any obvious benefits)
5. Hard to check for some dependencies

What if the "credits" depend on the "dept_name" (e.g., all CS courses must be 3 credits)?

No easy way to ensure that constraint (w/o a join)

Smaller schemas always good ????

Decompose Takes(course_id, student_id, semester, year) into:

course_id	student_id	semester	year
c1	s1	Fall	2020
c1	s2	Spring	2020
c2	s1	Spring	2020

Takes1(course_id, semester, year)

course_id	semester	year
c1	Fall	2020
c1	Spring	2020
c2	Spring	2020

Takes2(course_id, student_id)

course_id	student_id
c1	s1
c1	s2
c2	s1

Issues:

6. “joining” them back (on course_id) results in more tuples than what we started with

(c1, s1, Spring 2020) & (c1, s2, Fall 2020)

This is a “lossy” decomposition

We lost some constraints/information

The previous example was a “lossless” decomposition.

Desiderata

- No sets
- Correct and faithful to the original design
  - Must avoid lossy decompositions
- As little redundancy as possible
  - To avoid potential anomalies
- No “inability to represent information”
  - Nulls shouldn’t be required to store information
- Dependency preservation
  - Should be possible to check for constraints

Not always possible.

We sometimes relax these for:

simpler schemas, and *fewer joins during queries.*

Overall Approach

1. We will encode and list all our knowledge about the schema
   - e.g., Functional dependencies (FDs)

     SSN → name (means: SSN "implies" length)
     - If two tuples have the same "SSN", they must have the same "name"

       movietitle → length ???? Not true.
     - But, (movietitle, movieYear) → length --- True.
2. We will define a set of rules that the schema must follow to be considered good
   - "Normal forms": 1NF, 2NF, 3NF, BCNF, 4NF, …
   - A normal form specifies constraints on the schemas and FDs
3. If not in a "normal form", we modify the schema

Atomic Domains and 1st Normal Form

- A domain is called "atomic" if the elements can be considered indivisible
  - i.e., not composite or sets
  - Somewhat subjective and depends on how it is being used

- What about CMSC424?
  - A natural split into "CMSC" and "424".
  - Technically not atomic since programs/analysis often split it
  - Often treated as atomic, but better to keep as separate columns

- As long as all attributes are atomic → 1st Normal Form

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Functional Dependencies

Instructor: Amol Deshpande
amol@cs.umd.edu

Functional Dependencies

- Book Chapters (6th Edition)
  - Section 8.3.1

- Key Topics
  - Definition of a FD
  - Examples
  - Holding on an instance vs on all “legal” instances
  - FDs and Redundancies

Functional Dependencies

- On a relational schema: R(A, B, C, …)

  A → B (A "implies" B)

  means that if two tuples have the same value for A, they have the same value for B

- *A way to reason about duplication in a relational schema*

FDs: Example 1

student_id	dept_name	name	tot_cred	building	budget
s1	Comp. Sci.	John	30	Iribe Center	10 M
s2	Comp. Sci.	Alice	20	Iribe Center	10 M
s2	Math	Alice	20	Kirwan Hall	10 M
s3	Comp. Sci.	Mike	30	Iribe Center	10 M
s3	Math	Mike	30	Kirwan Hall	10 M

student_id → name

student_id → name, tot_cred

dept_name → building

dept_name → building, budget

FDs: Example 2

State Name	State Code	State Population	County Name	County Population	Senator Name	Senator Elected	Senator Born	Senator Affiliation
Alabama	AL	4779736	Autauga	54571	Jeff Sessions	1997	1946	'R'
Alabama	AL	4779736	Baldwin	182265	Jeff Sessions	1997	1946	'R'
Alabama	AL	4779736	Barbour	27457	Jeff Sessions	1997	1946	'R'
Alabama	AL	4779736	Autauga	54571	Richard Shelby	1987	1934	'R'
Alabama	AL	4779736	Baldwin	182265	Richard Shelby	1987	1934	'R'
Alabama	AL	4779736	Barbour	27457	Richard Shelby	1987	1934	'R'

State Name → State Code
State Code → State Name
Senator Name → Senator Born

FDs: Example 3

Course ID	Course Name	Dept Name	Credits	Section ID	Semester	Year	Building	Room No.	Capacity	Time Slot ID

Functional dependencies

course_id → title, dept_name, credits
building, room_number → capacity
course_id, section_id, semester, year → building, room_number, time_slot_id

Functional Dependencies

- Let *R* be a relation schema and

  $$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The *functional dependency*

  $$\alpha \rightarrow \beta$$

  holds on *R* iff for any *legal* relations *r*(R), whenever two tuples t_1 and t_2 of *r* have same values for α, they have same values for β.

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

- Example:

A	B
1	4
1	5
3	7

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

Functional Dependencies

Difference between holding on an *instance* and holding on *all legal relation*

student_id	dept_name	name	tot_cred	building	budget
s1	Comp. Sci.	John	30	Iribe Center	10 M
s2	Comp. Sci.	Alice	20	Iribe Center	10 M
s2	Math	Alice	20	Kirwan Hall	10 M
s3	Comp. Sci.	Mike	30	Iribe Center	10 M
s3	Math	Mike	30	Kirwan Hall	10 M

Name → Tot_Cred *holds on this instance*

Is this a true functional dependency ? ***No****.*

Two students with the same name can have the different credits.

Can't draw conclusions based on a *single instance*

Need to use domain knowledge to decide which FDs hold

FDs and Redundancy

- Consider a table: R(A, B, C):
  - With FDs: B → C, and A → BC
  - So "A" is a Key, but "B" is not
- So: there is a FD whose left hand side is not a key
  - **Leads to redundancy**

Since B is not unique, it may be duplicated
Every time B is duplicated, so is C

Not a problem with A → BC
A can never be duplicated

A	B	C
a1	b1	c1
a2	b1	c1
a3	b1	c1
a4	b2	c2
a5	b2	c2
a6	b3	c3
a7	b4	c1

Not a duplication → Two different tuples just happen to have the same value for C

FDs and Redundancy

- Better to split it up

A	B
a1	b1
a2	b1
a3	b1
a4	b2
a5	b2
a6	b3
a7	b4

B	C
b1	c1
b2	c2
b3	c3
b4	c1

Not a duplication → Two different tuples just happen to have the same value for C

Functional Dependencies

- Functional dependencies and *keys*
  - A *key* constraint is a specific form of a FD.
  - E.g. if *A* is a superkey for *R,* then:

    $$A \rightarrow R$$

  - Similarly for *candidate keys and primary keys.*

- Deriving FDs
  - A set of FDs may imply other FDs
  - *e.g. If A → B, and B → C, then clearly A → C*
  - *We will see a formal method for inferring this later*

Definitions

1. A relation instance *r satisfies* a set of functional dependencies, *F*, if the FDs hold on the relation

2. F holds on a relation schema *R* if no legal (allowable) relation instance of *R* violates it

3. A functional dependency, *A → B,* is called *trivial* if:
   - B is a subset of A
   - e.g. Movieyear, length → length

4. Given a set of functional dependencies, *F,* its *closure,* F^+, is all the FDs that are implied by FDs in *F.*

CMSC424: Database Design

Module: Design: E/R Models and Normalization

FDs: Armstrong Axioms, etc.

Instructor: Amol Deshpande
amol@umd.edu

Working with Functional Dependencies

- Book Chapters (6th Edition)
  - Section 8.4.1, 8.4.2, 8.4.3
- Key Topics
  - Closure of an attribute and attribute set
  - Armstrong Axioms
  - Extraneous Attributes
  - Canonical Cover
- Sufficient to get a high-level idea of these – don't need to understand the entire theory to follow rest of this

1. Closure

- Given a set of functional dependencies, *F,* its *closure, F^+ ,* is all FDs that are implied by FDs in *F.*
  - *e.g. If A → B, and B → C, then clearly A → C*

- We can find F+ by applying Armstrong's Axioms:
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
  - if $\alpha \rightarrow \beta$, then $\gamma\ \alpha \rightarrow \gamma\ \beta$ **(augmentation)**
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**

- These rules are
  - sound (generate only functional dependencies that actually hold)
  - complete (generate all functional dependencies that hold)

Additional rules

- If $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$, then $\alpha \rightarrow \beta\gamma$ **(union)**
- If $\alpha \rightarrow \beta\gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ **(decomposition)**
- If $\alpha \rightarrow \beta$ and $\gamma\ \beta \rightarrow \delta$, then $\alpha\ \gamma \rightarrow \delta$ **(pseudotransitivity)**

- The above rules can be inferred from Armstrong's axioms.

Example

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- Some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,
      and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$,
      and then transitivity

2. Closure of an attribute set

- Given a set of attributes *A* and a set of FDs *F, closure of A under F* is the set of all attributes implied by *A*

- In other words, the largest *B* such that: $A \rightarrow B$

- Redefining *super keys:*
  - *The closure of a super key is the entire relation schema*

- Redefining *candidate keys:*
  1. It is a super key
  2. No subset of it is a super key

Computing the closure for *A*

- Simple algorithm

- 1. Start with *B* = *A*.
- 2. Go over all functional dependencies, $\beta \rightarrow \gamma$, in F^+
- 3. If $\beta \subseteq B$, *then*

  Add γ to *B*
- 4. Repeat till *B* changes

Example

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$

- $(AG)^+$?
  - *1.* result = AG
  - 2.result = ABCG (A $\rightarrow$ C and A $\rightarrow$ B)
  - 3.result = ABCGH (CG $\rightarrow$ H and CG $\subseteq$ AGBC)
  - 4.result = ABCGHI (CG $\rightarrow$ I and CG $\subseteq$ AGBCH

- Is (AG) a candidate key ?

  1. It is a super key.

  2. (A+) = ABCH, (G+) = G.

  YES.

Uses of attribute set closures

- Determining *superkeys and candidate keys*
- Determining if $A \rightarrow B$ is a valid FD
  - Check if A+ contains B
- Can be used to compute *F+*

3. Extraneous Attributes

- Consider *F,* and a functional dependency, *A* → *B.*

- "Extraneous": Are there any attributes in *A or B* that can be safely removed ?

  Without changing the constraints implied by F

- Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
  - C is extraneous in AB → CD since AB → C can be inferred even after deleting C
  - ie., given: A → C, and AB → D, we can use Armstrong Axioms to infer *AB* → *CD*

4. Canonical Cover

- A *canonical cover* for *F* is a set of dependencies F_c such that
  - F logically implies all dependencies in F_c, and
  - F_c logically implies all dependencies in F, and
  - No functional dependency in F_c contains an extraneous attribute, and
  - Each left side of functional dependency in F_c is unique
- In some (vague) sense, it is a *minimal* version of *F*
- Read up algorithms to compute F_c

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Decompositions

Instructor: Amol Deshpande
amol@cs.umd.edu

Lossless and Lossy Decompositions

- Book Chapters (6th Edition)
  - Section 8.4.4

- Key Topics
  - How to decompose a schema in a lossless manner
  - Dependency preserving decompositions

Decompositions

- Splitting a relational schema *R* into two relations *R1, R2,* typically for normalization

- e.g., R(A, B, C, D, E) can be decomposed into:
  - R1(A, B, C), R2(D, E)
  - R1(A, B, C, D), R2(D, E)
  - …

- When is this okay to do?
  - The two resulting relations must be equivalent to the original relation… always

- Otherwise, it is a “lossy” decomposition, and not allowed

Loss-less Decompositions

- Definition: A decomposition of *R* into *(R1, R2)* is called *lossless* if, for all legal instances of *r(R):*

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

- In other words, projecting on *R1 and R2,* and *joining back,* results in the relation you started with

- Rule: A decomposition of *R* into *(R1, R2)* is *lossless,* iff:

$$R1 \cap R2 \rightarrow R1 \quad \textit{or} \quad R1 \cap R2 \rightarrow R2$$

in *F+.*

- Why? The join attributes then form a key for one of the relations
  - Each tuple from the other relation joins with exactly one from that relation

Loss-less Decompositions

- Example: R(A, B, C), FDs: A → B
  - Decomposition into R1(A, B) and R2(A, C) is lossless
    - (R1 ∩ R2 =) A → (R1 =) AB
  - Decomposition into R1(A, B) and R2(B, C) is not lossless

A	B	C
a1	b1	c1
a1	b1	c2
a2	b1	c3
a3	b1	c4

A	B
a1	b1
a2	b1
a3	b1

⋈

B	C
b1	c1
b1	c2
b1	c3
b1	c4

=

A	B	C
a1	b1	c1
a1	b1	c2
a1	b1	c3
a1	b1	c4
a2	b1	c1
a2	b1	c2
a2	b1	c3
a2	b1	c4
a3	b1	c1
a3	b1	c2
a3	b1	c3
a3	b1	c4

Dependency-preserving Decompositions

Is it easy to check if the dependencies in *F* hold ?

Okay as long as the dependencies can be checked in the same table.

Consider *R = (A, B, C)*, and *F ={A → B, B → C}*

1. Decompose into R1 = (A, B), and R2 = (A, C)

Lossless ? Yes.

But, makes it hard to check for *B → C*

The data is in multiple tables.

2. On the other hand, *R1 = (A, B), and R2 = (B, C),*

is both lossless and dependency-preserving

Really ? What about *A → C ?*

If we can check *A → B*, and *B → C, A → C* is implied.

Dependency-preserving Decompositions

- Definition:
  - Consider decomposition of *R into R1, …, Rn.*
  - Let F_i be the set of dependencies F^+ that include only attributes in R_i.

- The decomposition is dependency preserving, if

$$(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$$

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Boyce-Codd Normal Form

Instructor: Amol Deshpande
amol@umd.edu

Boyce Codd Normal Form

- Book Chapters (6th Edition)
  - Section 8.3.2

- Key Topics
  - Definition
  - How BCNF helps avoid redundancy
  - How to decompose a schema into BCNF

Approach

1. We will encode and list all our knowledge about the schema
   - Functional dependencies (FDs)
   - Also:
     - Multi-valued dependencies (briefly discuss later)
     - Join dependencies etc…
2. We will define a set of rules that the schema must follow to be considered good
   - "Normal forms": 1NF, 2NF, 3NF, BCNF, 4NF, …
   - A normal form specifies constraints on the schemas and FDs
3. If not in a "normal form", we modify the schema

BCNF: Boyce-Codd Normal Form

- A relation schema *R* is "in BCNF" if:
  - Every functional dependency *A* → *B* that holds on it is *EITHER*:

    1. Trivial *OR*
    2. *A* is a *superkey* of *R*

- *Why is BCNF good ?*
  - Guarantees that there can be no redundancy because of a functional dependency
  - Consider a relation *r(A, B, C, D)* with functional dependency *A* → *B* and two tuples: (a1, b1, c1, d1), and (a1, b1, c2, d2)
    - *b1* is repeated because of the functional dependency
    - BUT this relation is not in BCNF
      - *A* → *B* is neither trivial nor is *A* a superkey for the relation

BCNF and Redundancy

- *Why does redundancy arise ?*
  - Given a FD, A → B, if A is repeated (B – A) has to be repeated
  1. If rule 1 is satisfied, (B – A) is empty, so not a problem.
  2. If rule 2 is satisfied, then A can't be repeated, so this doesn't happen either

- Hence no redundancy because of FDs
  - Redundancy may exist because of other types of dependencies
    - Higher normal forms used for that (specifically, 4NF)
  - Data may naturally have duplicated/redundant data
    - We can't control that unless a FD or some other dependency is defined

BCNF

- What if the schema is not in BCNF ?
  - *Decompose (split) the schema into two pieces.*

- From the previous example: split the schema into:
  - *r1(A, B), r2(A, C, D)*
  - The first schema is in BCNF, the second one may not be (and may require further decomposition)
  - No repetition now: *r1* contains *(a1, b1)*, but *b1* will not be repeated

- Careful: you want the decomposition to be lossless
  - *No information should be lost*
    - The above decomposition is lossless

Achieving BCNF Schemas

For all dependencies *A* → *B* in *F+,* check if *A* is a superkey

By using attribute closure

If not, then

Choose a dependency in F+ that breaks the BCNF rules, say A → B

Create R1 = A B

Create R2 = A (R – B – A)

Note that: R1 ∩ R2 = A and A → AB (= R1), so this is lossless decomposition

Repeat for *R1, and R2*

By defining F1+ to be all dependencies in F that contain only attributes in R1

Similarly F2+

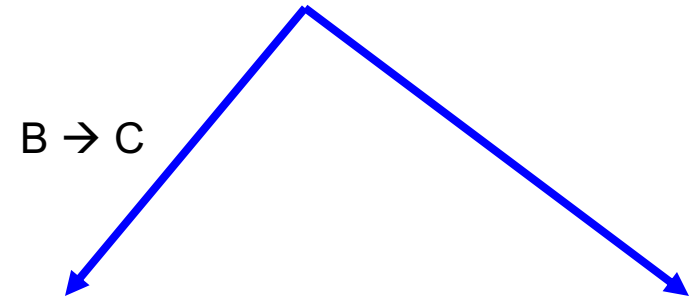
Example 1

R = (A, B, C)

F = {A → B, B → C}

Candidate keys = {A}

BCNF = No. B → C violates.



R1 = (B, C)

F1 = {B → C}

Candidate keys = {B}

BCNF = true

R2 = (A, B)

F2 = {A → B}

Candidate keys = {A}

BCNF = true

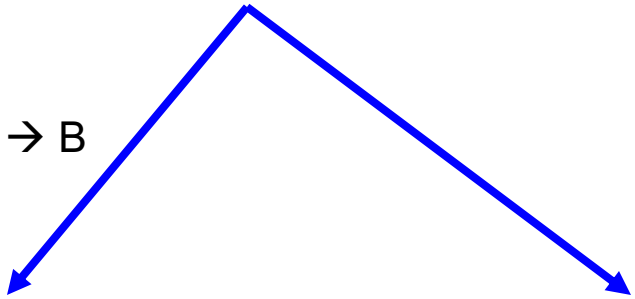
Example 2-1

R = (A, B, C, D, E)

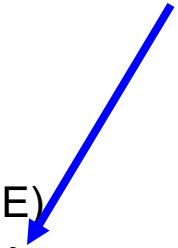
F = {A → B, BC → D}

Candidate keys = {ACE}

BCNF = Violated by {A → B, BC → D} etc…



From A → B and BC → D by pseudo-transitivity



R1 = (A, B)

F1 = {A → B}

Candidate keys = {A}

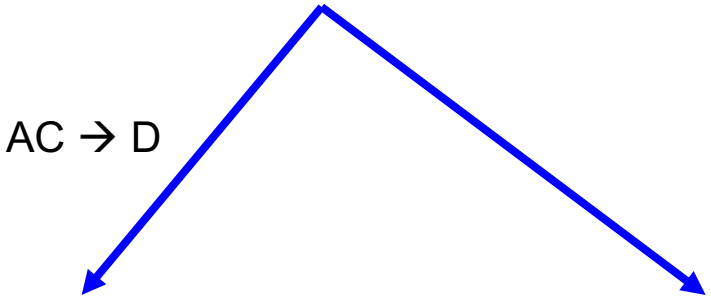
BCNF = true

R2 = (A, C, D, E)

F2 = {AC → D}

Candidate keys = {ACE}

BCNF = false (AC → D)



R3 = (A, C, D)

F3 = {AC → D}

Candidate keys = {AC}

BCNF = true

R4 = (A, C, E)

F4 = {} [[only trivial]]

Candidate keys = {ACE}

BCNF = true

Dependency preservation ???

We can check:

A → B (R1), AC → D (R3),

but we lost BC → D

So this is not a dependency -preserving decomposition

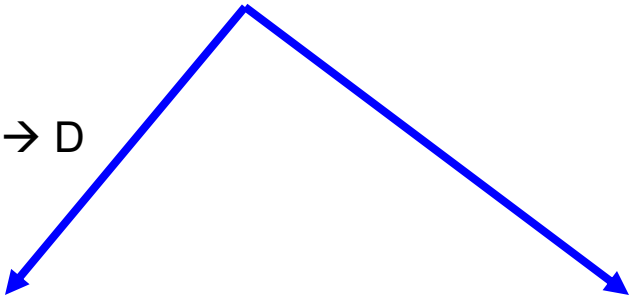
Example 2-2

R = (A, B, C, D, E)

F = {A → B, BC → D}

Candidate keys = {ACE}

BCNF = Violated by {A → B, BC → D} etc…



R1 = (B, C, D)

F1 = {BC → D}

Candidate keys = {BC}

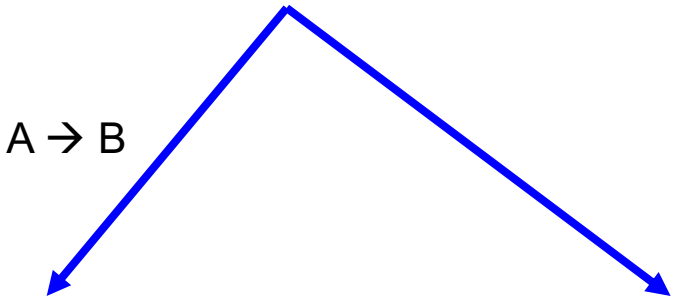
BCNF = true

R2 = (B, C, A, E)

F2 = {A → B}

Candidate keys = {ACE}

BCNF = false (A → B)



R3 = (A, B)

F3 = {A → B}

Candidate keys = {A}

BCNF = true

R4 = (A, C, E)

F4 = {} [[only trivial]]

Candidate keys = {ACE}

BCNF = true

Dependency preservation ???

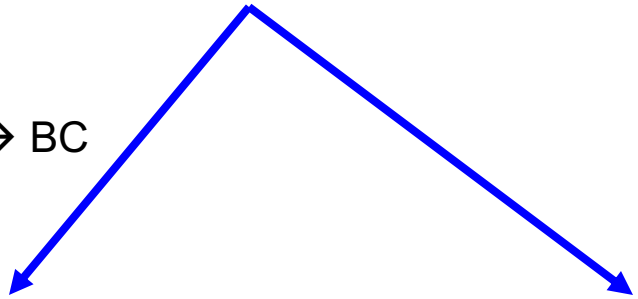
We can check:

BC → D (R1), A → B (R3),

Dependency-preserving decomposition

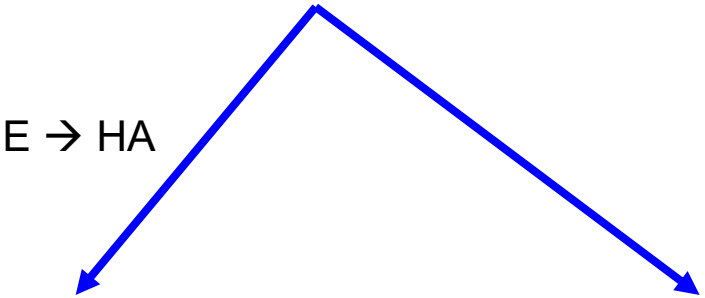
Example 3

R = (A, B, C, D, E, H)
F = {A → BC, E → HA}
Candidate keys = {DE}
BCNF = Violated by {A → BC} etc…



R1 = (A, B, C)
F1 = {A → BC}
Candidate keys = {A}
BCNF = true

R2 = (A, D, E, H)
F2 = {E → HA}
Candidate keys = {DE}
BCNF = false (E → HA)



R3 = (E, H, A)
F3 = {E → HA}
Candidate keys = {E}
BCNF = true

R4 = (ED)
F4 = {} [[only trivial]]
Candidate keys = {DE}
BCNF = true

Dependency preservation ???
We can check:
A → BC (R1), E → HA (R3),
Dependency-preserving decomposition

CMSC424: Database Design

Module: Design: E/R Models and Normalization

3NF, 4NF, and Other Issues

Instructor: Amol Deshpande
amol@umd.edu

3rd and 4th Normal Forms

- Book Chapters (6th Edition)
  - Section 8.3.4, 8.3.5, 8.5.2, 8.6 (at a high level)

- Key Topics
  - BCNF can't always preserve dependencies
  - How 3NF fixes that
  - BCNF causes redundancy because of "multi-valued dependencies"
  - How 4NF fixes that

Issue 1: BCNF may not preserve dependencies

- $R = (J, K, L\}$
- $F = \{JK \rightarrow L, L \rightarrow K\}$

- Two candidate keys = *JK* and *JL*

- *R* is not in BCNF

- Any decomposition of *R* will fail to preserve

  $JK \rightarrow L$

- This implies that testing for $JK \rightarrow L$ requires a join

Issue 1: BCNF may not preserve dependencies

- Not always possible to find a dependency-preserving decomposition that is in BCNF.
- PTIME to determine if there exists a dependency-preserving decomposition in BCNF
  - in size of F
- NP-Hard to find one if it exists
- Better results exist if F satisfies certain properties

3NF (3rd Normal Form)

- Definition: *Prime* attributes

  An attribute that is contained in a candidate key for R

- Example 1:
  - R = (A, B, C, D, E, H}, F = {A → BC, E → HA},
  - Candidate keys = {ED}
  - Prime attributes: D, E

- Example 2:
  - R = (J, K, L), F = {JK → L, L → K},
  - Candidate keys = {JL, JK}
  - Prime attributes: J, K, L

- Observation/Intuition:

  1. A *key* has no redundancy (is not repeated in a relation)
  2. A *prime attribute* has limited redundancy

3NF (3rd Normal Form)

- Given a relation schema *R,* and a set of functional dependencies *F,* if every FD, *A* → *B*, is either:
  1. Trivial, or
  2. *A* is a *superkey* of *R, or*
  3. *All attributes in (B – A) are prime*

  Then, *R* is in *3NF (3rd Normal Form)*

- *Why is 3NF good ?*

3NF and Redundancy

- *Why does redundancy arise ?*
  - Given a FD, A → B, if A is repeated (B – A) has to be repeated
  1. If rule 1 is satisfied, (B – A) is empty, so not a problem.
  2. If rule 2 is satisfied, then A can't be repeated, so this doesn't happen either
  3. If not, rule 3 says (B – A) must contain only *prime attributes*

     This limits the redundancy somewhat.

- So 3NF relaxes BCNF somewhat by allowing for some (hopefully limited) redundancy
- Why ?
  - *There always exists a dependency-preserving lossless decomposition in 3NF.*

Decomposing into 3NF

- A *synthesis* algorithm
- Start with the canonical cover, and construct the 3NF schema directly
- Homework assignment.

3NF Real Example

- Consider a table:
  - dept_advisor(s_id, i_id, dept_name)
  - … with a constraint/domain knowledge that: a student has one advisor per department

- FDs:
  - i_id --> dept_name
  - s_id, dept_name → i_id

- Not in BCNF (i_id is not a key), but in 3NF

Issue 2: BCNF and redundancy

MovieTitle	MovieYear	StarName	Address
Star wars	1977	Harrison Ford	Address 1, LA
Star wars	1977	Harrison Ford	Address 2, FL
Indiana Jones	198x	Harrison Ford	Address 1, LA
Indiana Jones	198x	Harrison Ford	Address 2, FL
Witness	19xx	Harrison Ford	Address 1, LA
Witness	19xx	Harrison Ford	Address 2, FL
…	…	…	…

Lot of redundancy

FDs ? No non-trivial FDs.

So the schema is trivially in BCNF (and 3NF)

What went wrong ?

Multi-valued Dependencies

- The redundancy is because of *multi-valued dependencies*
- *Denoted:*

  starname $\rightarrow\!\!\rightarrow$ *address*

  starname $\rightarrow\!\!\rightarrow$ *movietitle, movieyear*

- Should not happen if the schema is constructed from an E/R diagram

- Functional dependencies are a special case of multi-valued dependencies

Another Example

- Consider a ”person” table that stores the names of children and phone numbers

$$(ID, child\_name, phone\_number)$$

- The two pieces of information are independent

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

4NF

- Similar to BCNF, except with MVDs instead of FDs.

- Given a relation schema *R,* and a set of multi-valued dependencies *F,* if every MVD, $A \rightarrow\rightarrow B$, is either:

    1. Trivial, or
    2. *A* is a *superkey* of *R*

  Then, *R* is in *4NF (4th Normal Form)*

- 4NF → BCNF → 3NF → 2NF → 1NF:
  - If a schema is in 4NF, it is in BCNF.
  - If a schema is in BCNF, it is in 3NF.
- Other way round is untrue.

Comparing the normal forms

	3NF	BCNF	4NF
Eliminates redundancy because of FD's	Mostly	Yes	Yes
Eliminates redundancy because of MVD's	No	No	Yes
Preserves FDs	Yes.	Maybe	Maybe
Preserves MVDs	Maybe	Maybe	Maybe

4NF is typically desired and achieved.

A good E/R diagram won't generate non-4NF relations at all

Choice between 3NF and BCNF is up to the designer

CMSC424: Database Design

Module: Design: E/R Models and Normalization

Recap and Other Issues

Instructor: Amol Deshpande
amol@umd.edu

Recap and Other Issues

- Book Chapters (6th Edition)
  - Section 8.8

- Key Topics
  - Database design process
  - Denormalization
  - Other normal forms
  - Recap

Database design process

- Three ways to come up with a schema

1. Using E/R diagram
   - If good, then little normalization is needed
   - Tends to generate 4NF designs
2. A universal relation *R* that contains all attributes.
   - Called universal relation approach
   - Note that MVDs will be needed in this case
3. An *ad hoc* schema that is then normalized
   - MVDs may be needed in this case

Recap

- What about 1st and 2nd normal forms ?
- 1NF:
  - Essentially says that no set-valued attributes allowed
  - Formally, a domain is called *atomic* if the elements of the domain are considered indivisible
  - A schema is in 1NF if the domains of all attributes are atomic
  - We assumed 1NF throughout the discussion
    - Non 1NF is just not a good idea
- 2NF:
  - Mainly historic interest
  - See Exercise 7.15 in the book

Recap

- We would like our relation schemas to:
  - Not allow potential redundancy because of FDs or MVDs
  - Be *dependency-preserving:*
    - Make it easy to check for dependencies
    - Since they are a form of integrity constraints
- Functional Dependencies/Multi-valued Dependencies
  - Domain knowledge about the data properties
- Normal forms
  - Defines the rules that schemas must follow
  - 4NF is preferred, but 3NF is sometimes used instead

Recap

- Denormalization
  - After doing the normalization, we may have too many tables
  - We may *denormalize* for performance reasons
    - Too many tables → too many joins during queries
  - A better option is to use *views* instead
    - So if a specific set of tables is joined often, create a view on the join

- More advanced normal forms
  - project-join normal form (PJNF or 5NF)
  - domain-key normal form
  - Rarely used in practice