
Concordia University
Computer Science and Software Engineering
COMP353: Databases
Section CD
Summer 2023

Instructor: **Khaled Jababo (jababo@encs.concordia.ca)**
Lectures: **TuTh 2:45 PM – 5:30 PM @ MB-2.210**
Office hour: **Monday 5:45 - 6:45 PM @ ER-1175**

Introduction to Databases and SQL

What is a Database?

- A database is a collection of data that exists over a long period of time (*Persistent storage*)
- This collection should be logically coherent and have some inherent meaning, typically about an enterprise → it may not be a random pile of data

Examples of Databases

- List of names, addresses, and phone numbers of your friends
- Information about employees, departments, salaries, managers, etc. in a COMPANY
- Information about students, courses, grades, professors, etc. in a UNIVERSITY
- Information about books, users, etc. in a LIBRARY

Database Management System (DBMS)

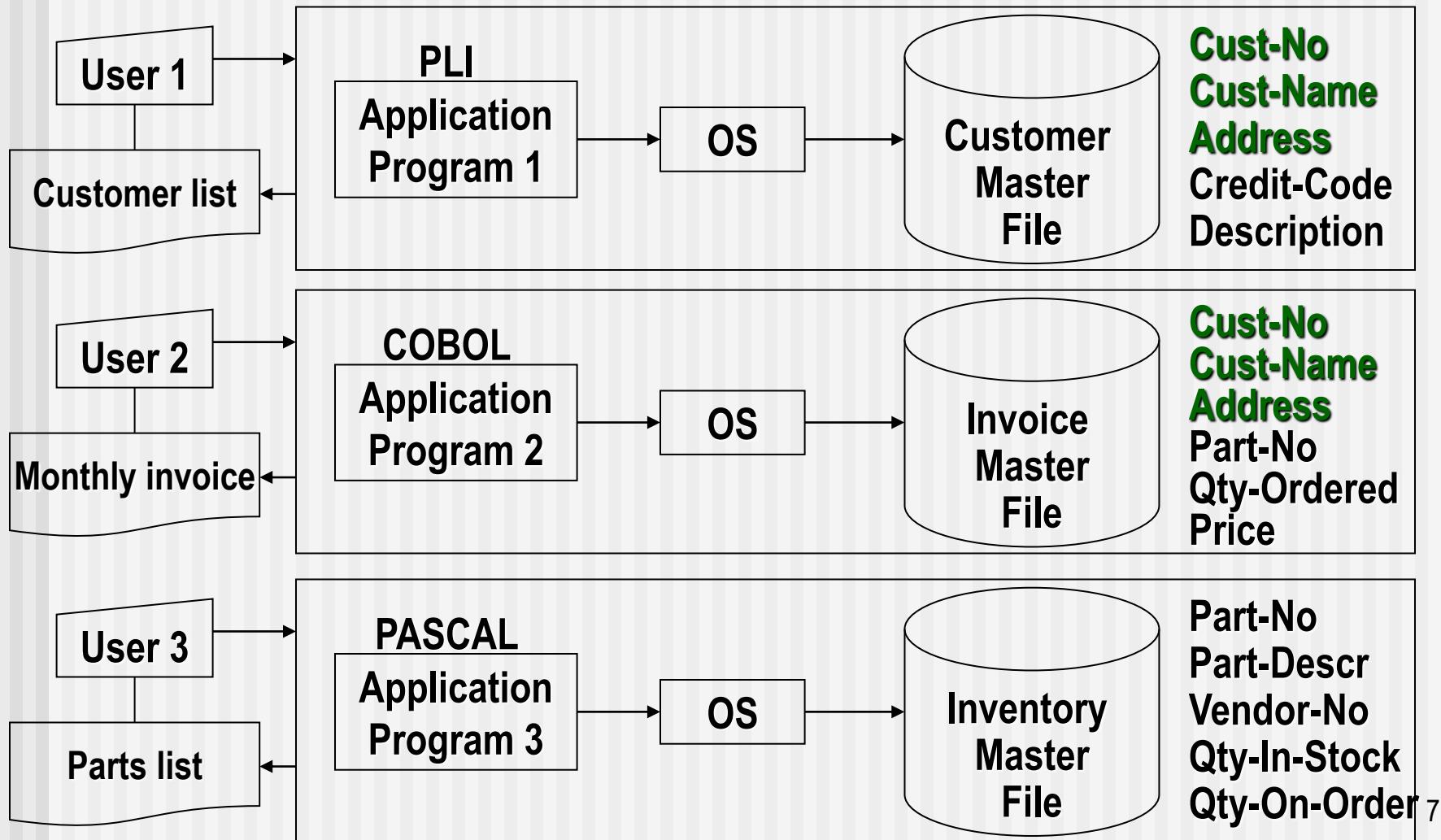
- A DBMS is a complex software package developed to *store* and “*manage*” databases
- Note the distinction between DB, DBS, and DBMS:

Database system = **Database + DBMS**

What does a DBMS provide?

- Supports *convenient, efficient, and secure access and manipulation of large amounts of data*
- *(high-level) Programming interface*: Gives users the ability to create, query, and modify the data
- *Persistent storage*: Supports the storage of data over a *long period of time*
- *Transaction management and recovery*: Controls access to shared data from multiple, simultaneous users with properties Atomicity, Consistency, Isolation, Durability (ACID)

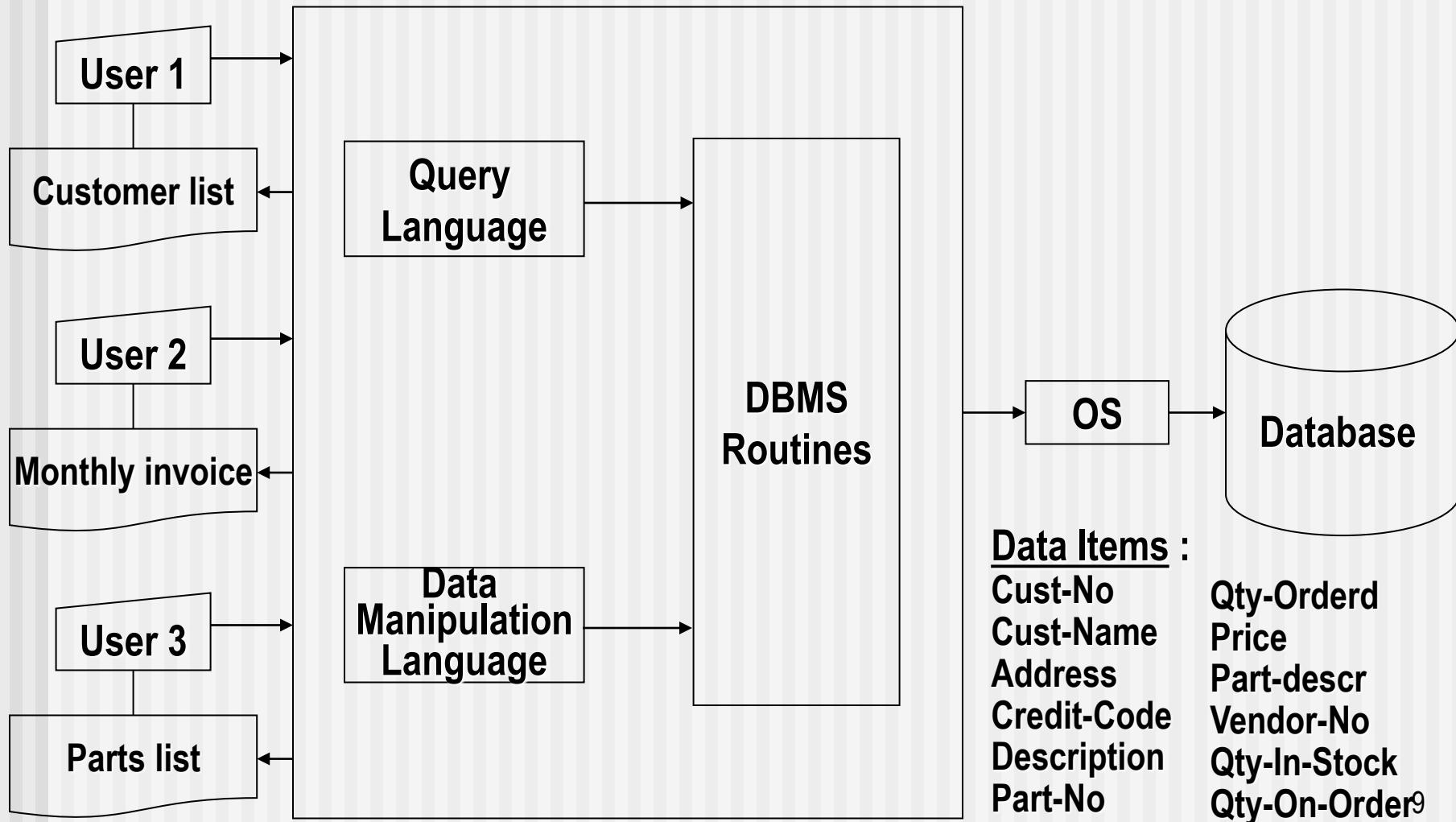
File Processing Systems (FPS)



Disadvantages of FPS

- Redundancy of data: Identical data are distributed over various files – a major source of problems
 - Waste of storage space: When the same field is stored in several files, the required storage space is needlessly high → **high storage cost**
 - Multiple updates: One field may be updated in one file but not in others → **inconsistency** and lack of data integrity and hence potential conflicting reports
 - Multiple programming languages: Dealing with several programming languages which are often not user friendly → **high system maintenance cost**

Database Systems



Advantages of Databases

- Minimize data redundancy and avoid inconsistency
 - They provide:
- Concurrent access to shared data
- Centralized control over data management
- Security and authorization
- Integrity and reliability
- Data abstraction and independence

Aspects of Database Studies

- Modeling and design of databases ✓
- Database programming ✓
- DBMS implementation

The first two aspects are studied in COMP 353

The third one is studied in COMP 451

What is this course about?

- A database is a “collection of data.” This data is managed by a DBMS
- Databases are essential today to support *commercial, engineering, and scientific applications.*
- They are at the core of many scientific investigations.
- Their power comes from a rich body of knowledge and technology developed over several decades
- In this course, we study fundamental concepts, techniques, and tools for *database design* and *programming*.
- In **COMP451**, we study details of DB *implementation*

A quick test!

- Which one of the following is the main source of the problems in file processing systems, addressed by databases?
 - A.** Waste of storage space.
 - B.** Update anomalies, which result in lack of data integrity.
 - C.** Data redundancy.
 - D.** Data inconsistency.

Data Modeling and Database Design

An Overview

Types of Data Models

- A Data Model is a collection of concepts, describing
 - data and relationships among data
 - data semantics and data constraints
- Entity-Relationship (ER) Model ✓
- Relational Model ✓
- Object-Oriented Data Model (ODL) ✓
- Logical Data Model (Datalog) ✓
- Earlier “record” based Data Models
 - Network
 - Hierarchical

Relational Model

- Data is organized in relations (tables)

The user should/need not be concerned with the underlying storage data structure.

- Relational database schema:

 - Set of table names – $D = \{R_1, \dots, R_n\}$

 - Set of attributes for each table – $R_i = \{A_1, \dots, A_k\}$

- Examples of tables:

 - **Account**= {accNum, branchNam, amount, customerId}

 - **Movie**= {title, year, director, studio}

Relational Model

- Most widely used model
 - Vendors: Oracle, IBM, Informix, Microsoft, Sybase, etc.
- Competitor: object-oriented model
 - ObjectStore, Postgres, etc.
- Another approach: *object-relational model*

Objectives of Database Systems

- A DB system should be **simple**, so that many users with little skills could interact with the system **conveniently**
- It should be **complex**, so that many (complex) queries and transactions could be handled/processed **efficiently**

*But these objectives are contradictory!
So how to achieve both?*

Three Views of Data

External view



User 1

Employee name
Employee address

SIN
Annual salary

User 2

Conceptual view



Database Administrator (DBA)

Employee name: string
SIN: dec, key
Employee address: string
Employee health card No: string, unique
Annual salary: float

Internal view

Employee name: string length 25 offset 0
SIN: 9 dec offset 25 unique
Employee health card No: string length 10 offset 34 unique
Employee address: string length 51 offset 44
Annual salary: 9,2 dec offset 95

Three Views / Levels of Data

- **Internal (physical) level**

A block of consecutive bytes actually holding the data

- **Conceptual (logical) level**

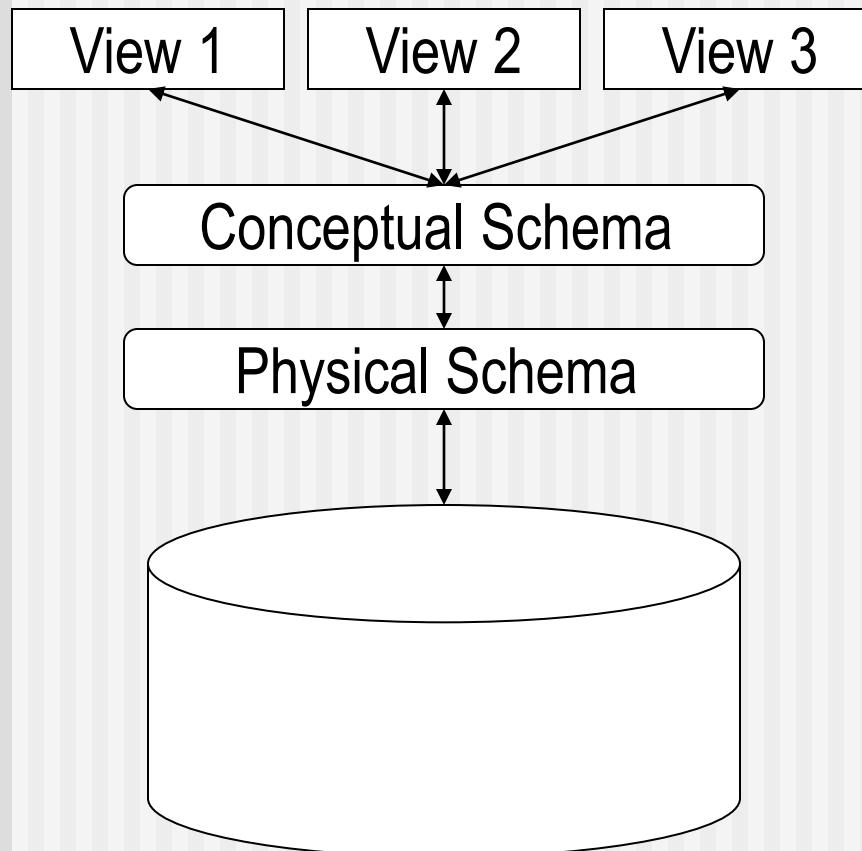
```
type emp = record
    SIN : integer;
    name : string;
    address : string;
    salary : real;
    healthCard : string;
end
```

- **External (logical) level**

View 1 : (emp.name, emp.address)

View 2 : (emp.SIN, emp.salary)

Levels of Abstraction in DB's



- Views describe how users “see” the data
- Conceptual schema defines logical structure
- Physical schema describes the storage structure of data and the indexes

Abstraction is achieved through describing each level in terms of a schema using a particular data model

Schemas at different levels of abstraction

- View (or External users): are typically determined during requirements analysis (often defined as views over some of the concepts in the logical DB schema)
- Conceptual (or Logical) Schema: an outcome of a database design ([a main focus in this course](#))
- Physical Schema: storage and index structures associated with relations

Schemas and Instances

- A database *instance* is the current content of the DB
- A database *schema* is the structure of the data (relations/classes), described in some suitable data model
 - e.g. relation:
 - Students {sid, name, department, dob, address} rep. as a *set* or
 - Students (sid, name, department, dob, address) as a *tuple*

Students

<i>sid</i>	<i>name</i>	<i>department</i>	<i>dob</i>	<i>address</i>
1112223	John Smith	CS	12-01-82	22 Pine, #1203
2223334	Ali Brown	EE	31-08-73	2000 St. Marc
3334445	Sana Kordi	CS	23-11-79	1150 Guy

Data Independence

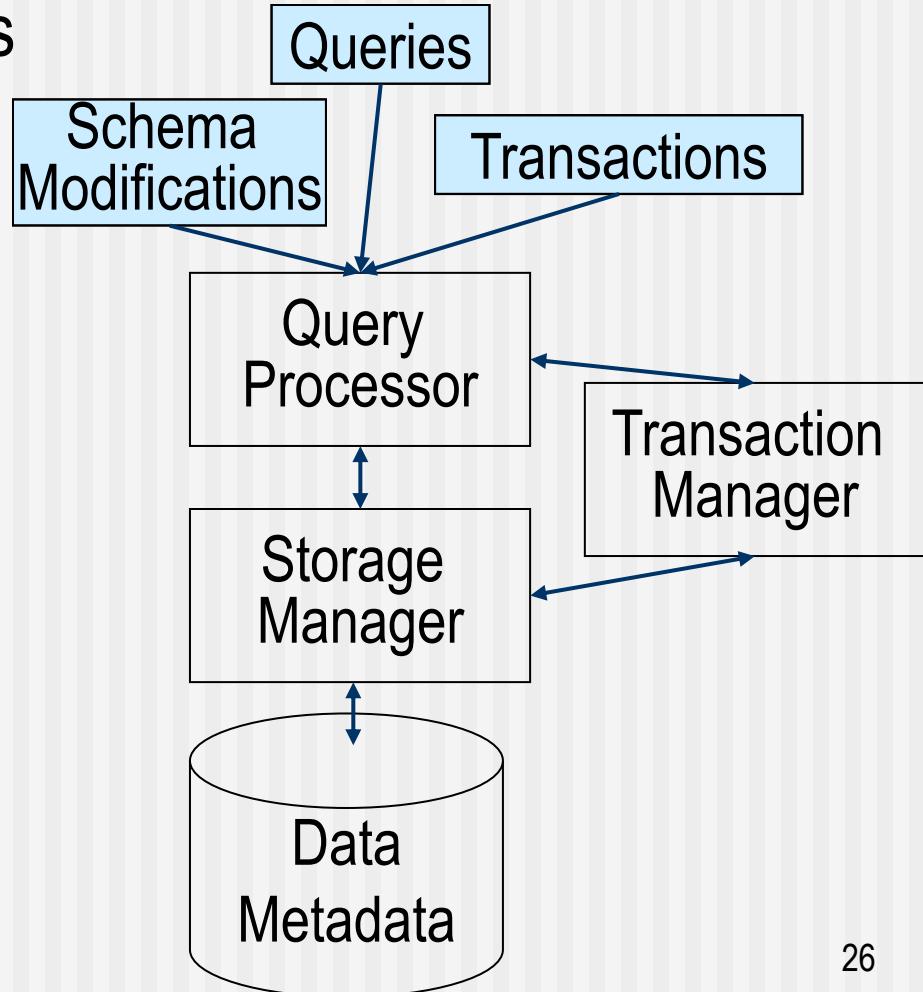
- Defn: the ability to modify definition of schema at one level with little or no affect on the schema (s) at a higher level
 - Achieved through the use of three levels of data abstraction
- Logical Data Independence
 - Ability to modify logical schema with little or no affect/change to rewrite the application programs
 - E.g., adding new fields to a record or changing the type of a field
- Physical Data Independence
 - Ability to modify physical schema with little or no impact on the conceptual schema or the application programs, i.e., *the possibility of having separate schemas at the physical and conceptual levels*
 - E.g., changing a file structure from sequential to direct access ²⁴

DBMS Implementation

Overview

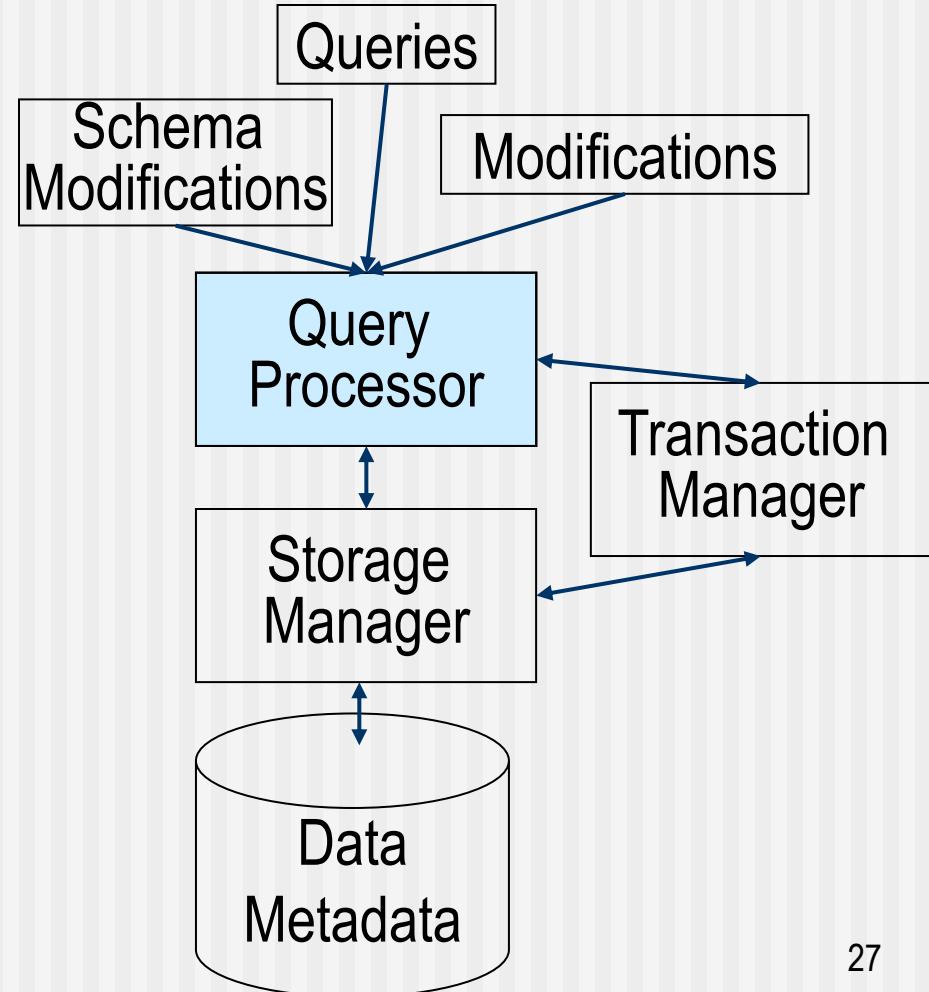
Architecture of a DBMS

- There are 3 types of inputs to DBMS:
 - Queries
 - Transactions, i.e., data Modifications
 - Schema Creations/Modifications



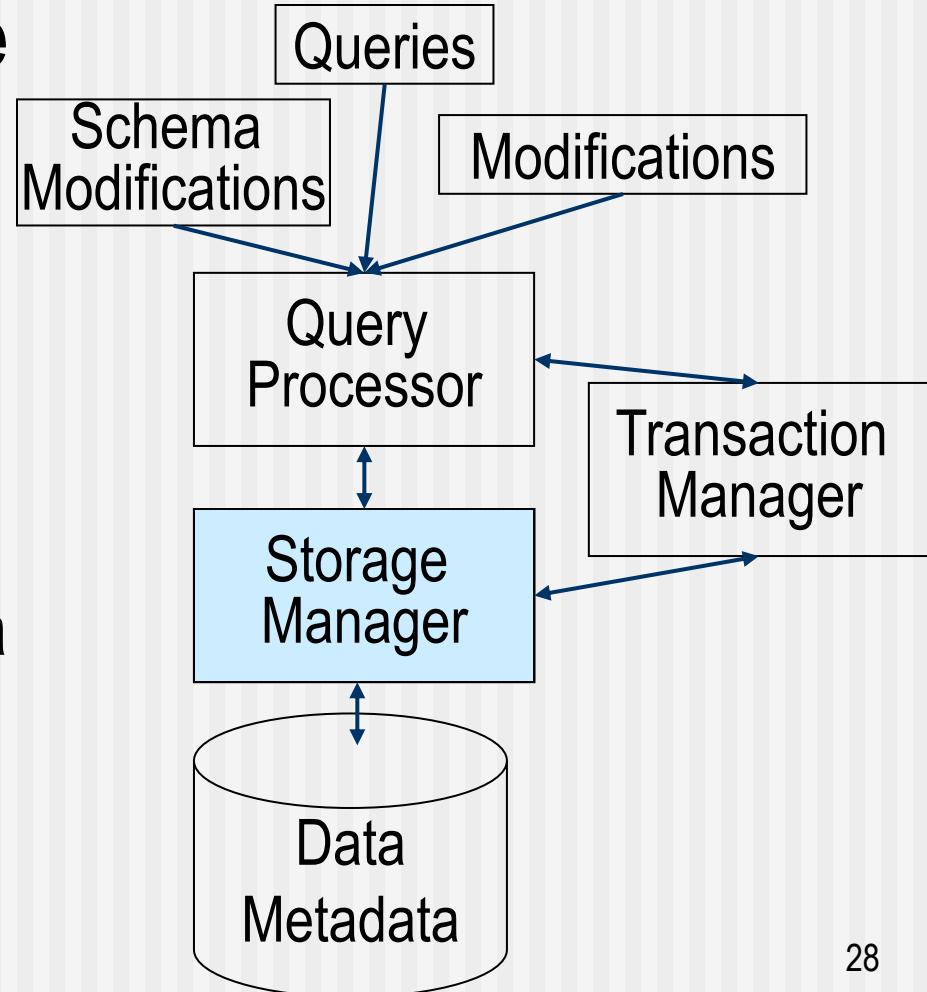
Architecture of a DBMS

- The **query processor** handles:
 - Queries
 - Modifications (of both data and schema)
- The job of the **query processor + query optimizer (QO)** is
 - To find the “best” plan to process the query
 - To issue commands to storage/buffer manager



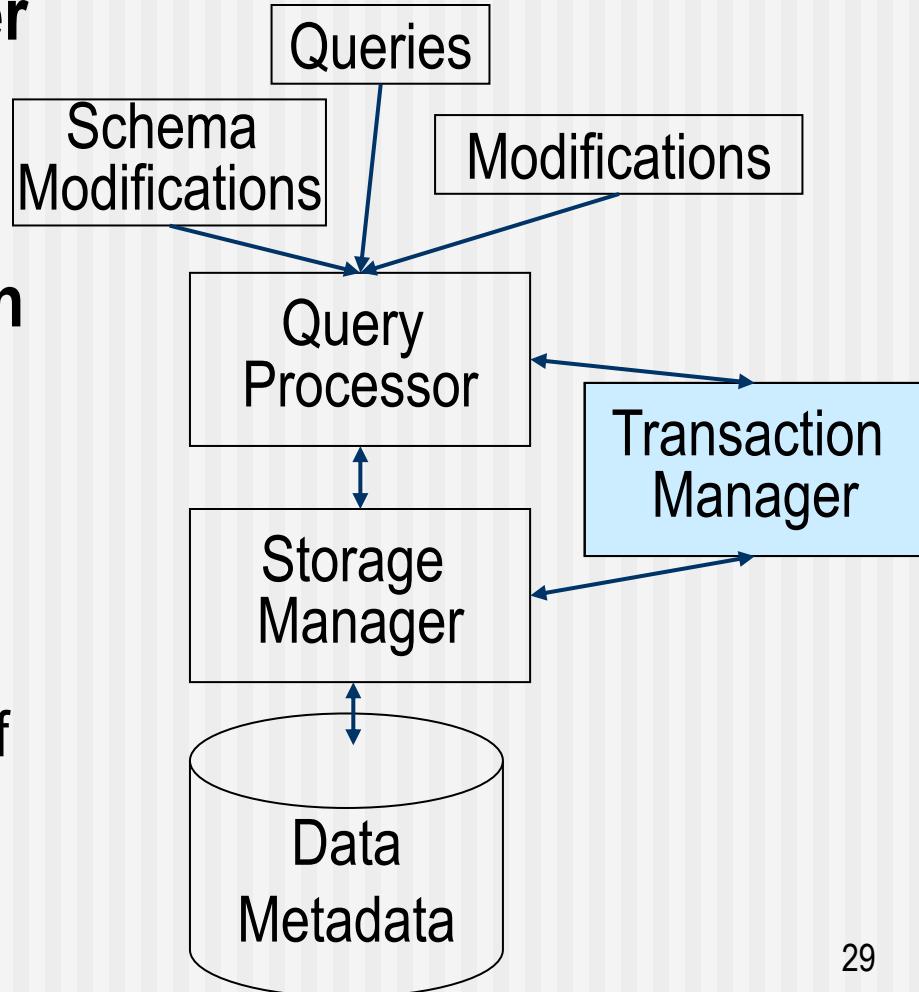
Architecture of a DBMS

- The job of the **storage manager** is
 - To obtain information requested **from** the data storage
 - To modify the information **to** the data storage when requested.



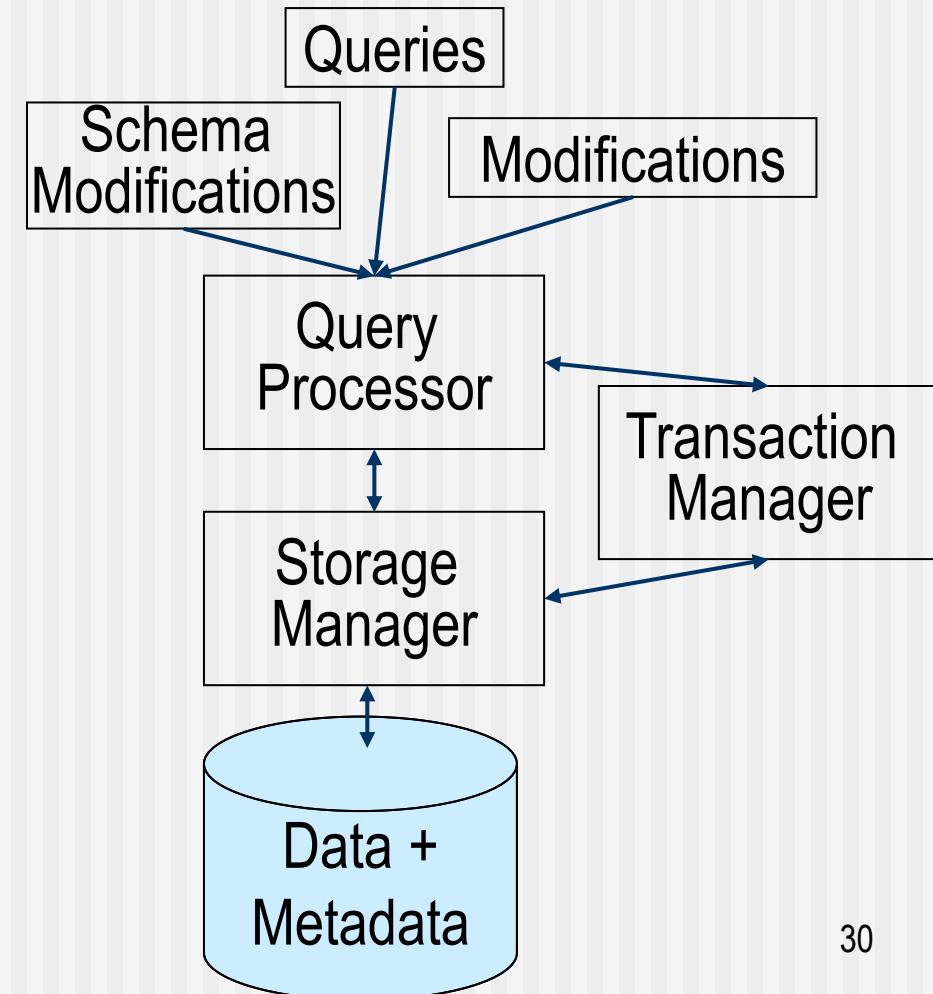
Architecture of a DBMS

- The **transaction manager** is responsible for the **consistency** of the data
- The job of the **transaction manager** is to ensure:
 - several queries running simultaneously do not “interfere” with each other
 - Integrity of the data even if there is a power failure (*Recovery system*)

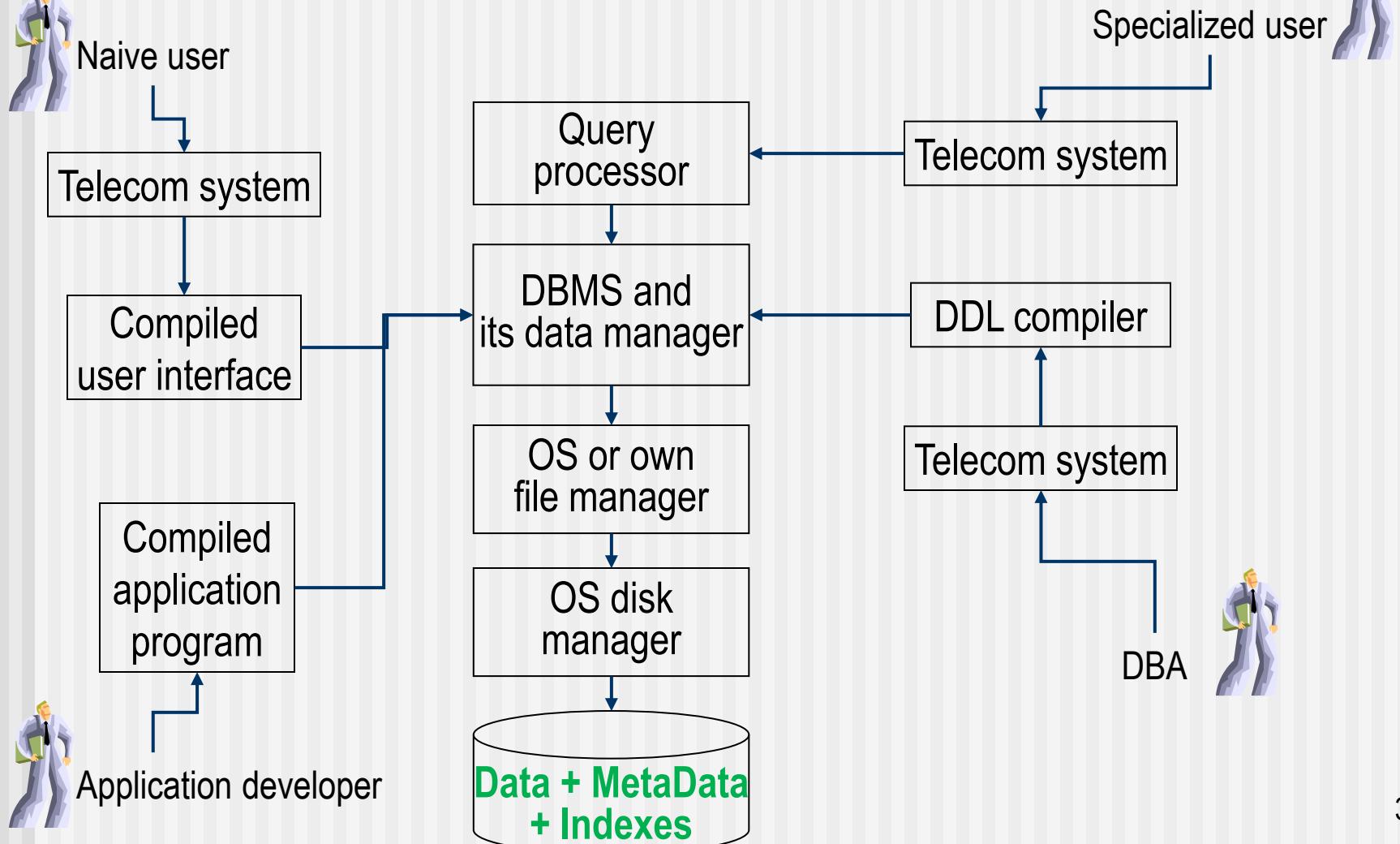


Architecture of a DBMS

- A representation of data and other relevant information on disk
- It contains:
 - Data
 - Metadata



Users of a Database System



Database Programming

Overview

Database Languages

- A Database Management System (DBMS) provides two types of languages, which may also be viewed as components of the DBMS language:
 - Data Definition Language (DDL)
 - Language (notation) for *defining* a database schema
 - It includes syntax for declaring tables, indexes, views, constraints, etc.)
 - Data Manipulation Language (DML)
 - Language for *accessing* and *manipulating* the data (organized/stored according to the appropriate data model)

Query Languages

- Commercial:
 - SQL ✓
- Theoretical/Abstract:
 - Relational Algebra ✓
 - Relational Calculus
 - Datalog ✓

SQL

- Developed originally at IBM in 1976
 - First standard: SQL-86
 - Second standard: SQL-92
 - Other standards: SQL-99, or SQL3, well over 1,000 page doc.
 - Currently SQL-2019 which supports MultiDim. Arrays
- De-facto standard of the relational database world; replaced all other DB languages
- The SQL query language components:
 - DDL
 - DML

Simple SQL Queries

- A SQL query has a form:

SELECT . . .

FROM . . .

WHERE . . . ;

- The **SELECT** clause indicates which attributes should appear in the output.
- The **FROM** gives the relation(s) the query refers to
- The **WHERE** clause is a Boolean expression indicating which tuples are of interest.
- A query result is a **bag**, in general
- A query result is **unnamed**.

Example SQL Query

- Relation schema:

Course (courseNumber, name, noOfCredits)

- Query:

Find all the courses stored in the database

- Query in SQL:

SELECT *

FROM Course;

Note: “ * “ means **all** attributes in the **relation(s)** involved.

Example SQL Query

- Relation schema:
Movie (title, year, length, filmType)
- Query:
Find the titles of all movies stored in the database
- Query in SQL:
**SELECT title
FROM Movie;**

Example SQL Query

- Relation schema:
Student (ID, firstName, lastName, address, GPA)
- Query:
Find the ID of every student whose GPA is more than 3
- Query in SQL:
**SELECT ID
FROM Student
WHERE GPA > 3;**

Example SQL Query

- Relation schema:
Student (ID, firstName, lastName, address, GPA)
- Query:
Find the ID and last name of every student with first name 'John',
who has a GPA > 3
- Query in SQL:
SELECT ID, lastName
FROM Student
WHERE firstName = 'John' **AND** GPA > 3;

WHERE clause

- The expressions that may follow **WHERE** are conditions
 - Standard comparison operators Θ includes { `=`, `<>`, `<`, `>`, `<=`, `>=` }
 - The values that may be compared include constants and attributes of the relation(s) mentioned in **FROM** clause
 - Simple expression
 - $A \text{ op } Value$
 - $A \text{ op } B$
where A, B are attributes and op is a comparison operator
- We may also apply the usual arithmetic operators, `+`, `-`, `*`, `/`, etc. to numeric values before comparing them
 - $(\text{year} - 1930) * (\text{year} - 1930) < 100$
- The result of a comparison is a Boolean value, **TRUE** or **FALSE**
- Boolean expressions can be combined by the logical operators **AND**, **OR**, and **NOT**

Example SQL Query

- Relation schema:

Movie (title, year, length, filmType)

- Query:

Find the titles of all color movies produced in 1990

- Query in SQL:

SELECT title

FROM Movie

WHERE filmType = 'color' **AND** year = 1990;

Example SQL Query

- Relation schema:
Movie (title, year, length, filmType)
- Query:
Find the titles of color movies that are either made after 1970 or are less than 90 minutes long
- Query in SQL:
SELECT title
FROM Movie
WHERE (year > 1970 **OR** length < 90) **AND** filmType = 'color';
- Note the precedence rules, when parentheses are absent:
AND takes precedence over **OR**, and
NOT takes precedence over **AND** and **OR**

Products and Joins

- SQL has a simple way to “couple” relations in one query
 - How? By “listing” the relevant relation(s) in the **FROM** clause
- All the relations in the **FROM** clause are coupled through **Cartesian product** (shown as \times in algebra notation)

Cartesian Product

■ From Set Theory:

- The **Cartesian Product** of two sets R and S is the set of **all** pairs (a, b) such that: $a \in R$ and $b \in S$.
- Denoted as $R \times S$
- Note:
 - In general, $R \times S \neq S \times R$

A quick test!

- Let $R(A_1, \dots, A_n)$ be a relation schema and r be any instance of R . Suppose r has m tuples. Which of the following is the number of ways in which r may be represented in the relational model?
 - A.** $m * n$
 - B.** 2^m
 - C.** $m! * n!$
 - D.** 2^n

Example

Instance R:

A	B
1	2
3	4

Instance S:

B	C	D
2	5	6
4	7	8
9	10	11

R x S:

A	R.B	S.B	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

Example

Instance of Student:

ID	firstName	lastName	GPA	Address
111	Joe	Smith	4.0	45 Pine av.
222	Sue	Brown	3.1	71 Main st.
333	Ann	Johns	3.7	39 Bay st.

Instance of Course:

courseNumber	name	noOfCredits
Comp352	Data structures	3
Comp353	Databases	4

SELECT * FROM Student, Course;

ID	firstName	lastName	GPA	Address	courseNumber	name	noOfCredits
111	Joe	Smith	4.0	45 Pine av.	Comp352	Data structures	3
111	Joe	Smith	4.0	45 Pine av.	Comp353	Databases	4
222	Sue	Brown	3.1	71 Main st.	Comp352	Data structures	3
222	Sue	Brown	3.1	71 Main st.	Comp353	Databases	4
333	Ann	Johns	3.7	39 Bay st.	Comp352	Data structures	3
333	Ann	Johns	3.7	39 Bay st.	Comp353	Databases	4

Example

Instance of Student:

ID	firstName	lastName	GPA	Address
111	Joe	Smith	4.0	45 Pine av.
222	Sue	Brown	3.1	71 Main st.
333	Ann	Johns	3.7	39 Bay st.

Instance of Course:

courseNumber	name	noOfCredits
Comp352	Data structures	3
Comp353	Databases	4

**SELECT ID, courseNumber
FROM Student, Course;**

ID	courseNumber
111	Comp352
111	Comp353
222	Comp352
222	Comp353
333	Comp352
333	Comp353

Example

- Relation schemas:
 - Student (ID, firstName, lastName, address, GPA)**
 - Ugrad (ID, major)**
- Query:
 - Find “all” information about every undergraduate student
- We try first by computing the Cartesian product (×)
 - SELECT * FROM Student, Ugrad;**

Example

Instance of Student:

ID	firstName	lastName	GPA	Address
111	Joe	Smith	4.0	45 Pine av.
222	Sue	Brown	3.1	71 Main st.
333	Ann	Johns	3.7	39 Bay st.

Instance of Ugrad:

ID	major
111	CS
333	EE

SELECT * FROM Student, Ugrad;

ID	firstName	lastName	GPA	Address	ID	major
111	Joe	Smith	4.0	45 Pine av.	111	CS
111	Joe	Smith	4.0	45 Pine av.	333	EE
222	Sue	Brown	3.1	71 Main st.	111	CS
222	Sue	Brown	3.1	71 Main st.	333	EE
333	Ann	Johns	3.7	39 Bay st.	111	CS
333	Ann	Johns	3.7	39 Bay st.	333	EE

Only the green tuples should be in the query result. How to pick them only?

Example

Instance of Student:

ID	firstName	lastName	GPA	Address
111	Joe	Smith	4.0	45 Pine av.
222	Sue	Brown	3.1	71 Main st.
333	Ann	Johns	3.7	39 Bay st.

Instance of Ugrad:

ID	major
111	CS
333	EE

```
SELECT *  
FROM Student, Ugrad  
WHERE Student.ID = Ugrad.ID;
```

ID	firstName	lastName	GPA	Address	ID	major
111	Joe	Smith	4.0	45 Pine av.	111	CS
333	Ann	Johns	3.7	39 Bay st.	333	EE

Join in SQL

- The above query is an example of **Join** operation
- There are different kinds of joins, which we will study!
- To join relations R_1, \dots, R_n in SQL:
 - List all these relations in the **FROM** clause
 - Express the conditions in the **WHERE** clause in order to get the “desired” **join**

Joining Relations

- Relation schemas:
 - Movie** (title, year, length, filmType)
 - Owns** (title, year, studioName)
- Query: Find **title**, **length**, and **studio name** of every movie
- Query in SQL:

```
SELECT Movie.title, Movie.length, Owns.studioName  
FROM Movie, Owns  
WHERE Movie.title = Owns.title AND Movie.year = Owns.year;
```

Question: Is **Owns** in **Owns.studioName** necessary?

Joining Relations

- Relation schemas:

- Movie** (title, year, length, filmType)
- Owns** (title, year, studioName)

- Query:

- Find the title and length of every movie produced by Disney studio.**

- Query in SQL:

- SELECT Movie.title, length**

- FROM Movie, Owns**

- WHERE Movie.title = Owns.title AND**

- Movie.year = Owns.year AND studioName = 'Disney';**

Joining Relations

- Relation schemas:

- Movie** (title, year, length, filmType)

- Owns** (title, year, studioName)

- StarsIn** (title, year, starName)

- Query:

- Find the title and length of Disney movies with JR as an actress.

- Query in SQL:

- ```
SELECT Movie.title, Movie.length
```

- ```
FROM Movie, Owns, StarsIn
```

- ```
WHERE Movie.title = Owns.title AND Movie.year = Owns.year
 AND Movie.title = StarsIn.title AND Movie.year = StarsIn.year
 AND studioName = 'Disney' AND starName = 'JR';
```

# Example

Movie

| title | year | length | filmType |
|-------|------|--------|----------|
| T1    | 1990 | 124    | color    |
| T2    | 1991 | 144    | color    |

Owns

| title | year | studioName |
|-------|------|------------|
| T1    | 1990 | Disney     |
| T2    | 1991 | MGM        |

StarsIn

| title | year | starName |
|-------|------|----------|
| T1    | 1990 | JR       |
| T2    | 1991 | JR       |

| title | length |
|-------|--------|
| T1    | 124    |

```
SELECT Movie.title, Movie.length
FROM Movie, Owns, StarsIn
WHERE Movie.title = Owns.title AND Movie.year =
 Owns.year AND Movie.title = StarsIn.title AND
 Movie.year = StarsIn.year AND studioName = 'Disney'
AND starName = 'JR';
```

# Aggregation in SQL

---

- SQL provides 5 operators that can be applied to a column of a relation in order to produce some kind of “summary”
- These operators are called ***aggregations***
- They are used in a **SELECT** clause and often applied to a scalar-valued attribute (column) or an expression in general.

# Aggregation Operators

---

## ■ SUM

- Returns the sum of values in the column

## ■ AVG

- Returns the average of values in the column

## ■ MIN

- Returns the least value in the column

## ■ MAX

- Returns the greatest value in the column

## ■ COUNT

- Returns the number of values in the column, including the duplicates, unless the keyword **DISTINCT** is used explicitly

# Example

---

- Relation schema:

**Exec**(name, address, cert#, netWorth)

- Query:

Find the average net worth of the movie executives

- Query in SQL:

**SELECT AVG(netWorth)**

**FROM Exec;**

- The sum of “all” values in the column **netWorth** divided by the number of these values
- In general, if a value **v** appears **n** times in the column, it contributes the value **n\*v** to computing the average

# Example

---

- Relation schema:  
**Exec** (name, address, cert#, netWorth)
- Query:  
How many movie executives are there in the Exec relation?
- Query in SQL:  
**SELECT COUNT(\*)  
FROM Exec;**
  - The use of \* as a parameter is unique to **COUNT**;  
Its use for other aggregation operations makes no sense.

# Example

---

- Relation schema:

**Exec** (name, address, cert#, netWorth)

- Query:

How many different names are there in the Exec relation?

- Query in SQL:

```
SELECT COUNT (DISTINCT name)
```

```
FROM Exec;
```

- In query processing time, the system first eliminates the duplicates from the column **name**, and then counts the number of present values

# Aggregation -- Grouping

---

- To answer a query, we may need to “group” the tuples according to the values of some other column(s)
- Example: Suppose we want to find:

Total length in minutes of movies produced by each studio:

`Movie(title, year, length, filmType, studioName, producerC#)`

- We must group the tuples in the *Movie* relation according to their *studio*, and then find the sum of the *lengths* within each group. The result displayed would look like:

studio    SUM(length)

Disney    12345

MGM    54321

...

...

# Aggregation - Grouping

---

- Relation schema:

**Movie**(title, year, length, filmType, studioName, producerC#)

- Query: What is the total length in minutes produced by each studio?

- Query formulated/expressed in SQL:

```
SELECT studioName, SUM(length)
```

```
FROM Movie
```

```
GROUP BY studioName;
```

- Whatever aggregation used in the **SELECT** clause will be applied only within groups
- Only those attributes mentioned in the **GROUP BY** clause may appear unaggregated in the **SELECT** clause
- Can we use GROUP BY without using aggregation?

# Aggregation -- Grouping

---

- Relation schema:

**Movie**(title, year, length, filmType, studioName, producerC#)

**Exec**(name, address, cert#, netWorth)

- Query:

For each producer (name), list the total length of the films produced

- Query in SQL:

```
SELECT Exec.name, SUM(Movie.length)
```

```
FROM Exec, Movie
```

```
WHERE Movie.producerC# = Exec.cert#
```

```
GROUP BY Exec.name;
```

# A rule about null values!

---

- Nulls are counted when grouping but ignored when aggregating.

Example: Consider the instance below of  $R(A,B)$ .

Which one of the following tuples will *not* be in the output?

**Select A, Sum(B)**  
**From R**  
**Group By A;**

- A.** (null, null)
- B.** (2,4)
- C.** (1,null)
- D.** (null,1)

| A    | B    |
|------|------|
| null | 1    |
| 2    | 1    |
| null | null |
| 3    | 2    |
| 2    | 3    |
| 1    | null |

# A rule about null values!

---

- The answer:

**Select A, Sum(B)**  
**From R**  
**Group By A;**

- ✓ (null, null)
- (2,4)
- (1,null)
- (null,1)

| A    | B    |
|------|------|
| null | 1    |
| 2    | 1    |
| null | null |
| 3    | 2    |
| 2    | 3    |
| 1    | null |

# Another test!

---

- Consider again the same instance of R(A,B) containing the tuples: (null,1), (2,1), (null, null), (3,2), (2,3), and (1,null). Which of the following tuples will be in the result of the query below?

**Select A, Sum(B)**

**From R**

**Where B<>2**

**Group By A;**

- A.** (null, 0)
- B.** (1,null)
- C.** (2,3)
- D.** (2,4)

# Answer!

---

- Consider an instance of R(A,B) with the tuples (null,1), (2,1), (null, null), (3,2), (2,3), and (1,null). Which one of the following tuples will be present in the result of the query below?

Select A, Sum(B)

From R

Where B<>2

Group By A;

- (null, 0) null 1
- (1,null) 2 1
- (2,3) 2 3
- ✓ (2,4)

# Aggregation – HAVING clause

---

- We might be interested in not every group but those which satisfy certain conditions
- For this, after a **GROUP BY** clause use a **HAVING** clause
- **HAVING** is followed by some conditions about the group
- We can *not* use a **HAVING** clause without **GROUP BY**

# Aggregation – HAVING clause

---

## ■ Relation schema:

**Movie** (title, year, length, filmType, studioName, producerC#)

**Exec**(name, address, cert#, netWorth)

## ■ Query:

For those producers who made at least one film prior to 1930, list the total length of the films produced

## ■ Query in SQL:

**SELECT** Exec.name, **SUM**(Movie.length)

**FROM** Exec, Movie

**WHERE** producerC# = cert#

**GROUP BY** Exec.name

**HAVING** **MIN**(Movie.year) < 1930;

# Aggregation – HAVING clause

---

- This query chooses the group based on the property of **each group**  
**SELECT Exec.name, SUM(Movie.length)**  
**FROM Exec, Movie**  
**WHERE producerC# = cert#**  
**GROUP BY Exec.name**  
**HAVING MIN(Movie.year) < 1930;**
- Consider the following query which chooses the movies based on the property of **each movie tuple**:

```
SELECT Exec.name, SUM(Movie.length)
FROM Exec, Movie
WHERE producerC# = cert# AND Movie.year < 1930
GROUP BY Exec.name;
```

# Order By

- The SQL statements/**queries** we looked at so far return an **unordered relation/bag**. *What if we want the result displayed in a certain order?*  
**Movie** (title, year, length, filmType, studioName, **producerC#**)

```
SELECT Exec.name, SUM(Movie.length)
FROM Exec, Movie
WHERE producerC# = cert#
GROUP BY Exec.name
HAVING MIN(Movie.year) < 1930
ORDER BY Exec.name ASC;
```

In general:

```
ORDER BY A ASC, B DESC, C ASC;
```

# Database Modifications

---

- **SQL & Database Modifications?**
  - We now look at SQL statements that do not return tuples, but rather *change the state (content) of the database*
- There are three types of such statements/**transactions**:
  - **Insert** tuples into a relation
  - **Delete** certain tuples from a relation
  - **Update** values of certain attributes of certain existing tuples

These types of operations that modify the database content are referred to as ***transactions***

# Insertion

---

- The insertion statement consists of:
  - The keyword **INSERT INTO**
  - The name of a relation  $R$
  - A parenthesized list of attributes of the relation  $R$
  - The keyword **VALUES**
  - A tuple expression, that is, a parenthesized list of concrete values, one for each attribute in the attribute list
- The form of an insert statement:

**INSERT INTO  $R(A_1, \dots, A_n)$  VALUES ( $v_1, \dots, v_n$ ) ;**

- This command inserts the tuple  $(v_1, \dots, v_n)$  to table  $R$ , where  $v_i$  is the value of attribute  $A_i$ , for  $i = 1, \dots, n$

# Insertion

---

- Relation schema:  
**StarsIn (title, year, starName)**
- Update the database:  
Add “Sydney Greenstreet” to the list of stars of *The Maltese Falcon*
- In SQL:

**INSERT INTO StarsIn (title,year, starName)**

**VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');**

***Another formulation of this query:***

**INSERT INTO StarsIn**

**VALUES('The Maltese Falcon', 1942, 'Sydney Greenstreet');**

# Insertion

---

- The previous insertion statement was “simple” in that it added **one** tuple only into a relation
- Instead of using *explicit* values for a tuple in insertion, we can request a **set** of tuples to be inserted. For this we define, in a *subquery*, the set of tuples from an existing relation
- This subquery replaces the keyword **VALUES** and the tuple expression in the **INSERT** statement

# Insertion

---

- Database schema:

**Studio**(name, address, presC#)

**Movie**(title, year, length, filmType, **studioName**, producerC#)

- Update the database:

Add to **Studio**, all **studio names** mentioned in the **Movie** relation

- **Note:** If the list of attributes in an “insert” statement does not include all the attributes of the relation, the tuple created will have the **default** value for each missing attribute
- Since there is no way to determine an **address** or a **presC#** for a studio tuple, **NULL** will be used for these attributes.

# Insertion

---

- Database schema:

**Studio**(name, address, presC#)

**Movie**(title, year, length, filmType, studioName, producerC#)

- Update the database:

Add to **Studio**, all **studio names** mentioned in the **Movie** relation

- In SQL:

**INSERT INTO Studio(name)**

**SELECT DISTINCT studioName**

**FROM Movie**

**WHERE studioName NOT IN (SELECT name  
FROM Studio);**

# Deletion

---

- A delete statement consists of :
  - The keyword **DELETE FROM**
  - The name of a relation  $R$
  - The keyword **WHERE**
  - A condition
- The syntax of the delete statement:  
**DELETE FROM  $R$  WHERE <condition>;**
  - The effect of executing this statement is that “every tuple” in relation  $R$  satisfying the condition will be deleted from  $R$
  - Note: unlike the INSERT, we MAY need a WHERE clause here

# Deletion

---

- Relation schema:  
**StarsIn(title, year, starName)**
- Update:

Delete the tuple that says:

Sydney Greenstreet was a star in *The Maltese Falcon*

- In SQL:  
**DELETE FROM StarIn  
WHERE title = 'The Maltese Falcon' AND  
starName = 'Sydney Greenstreet';**

# Deletion

---

- Relation schema:  
**Exec(name, address, cert#, netWorth)**
- Update:  
Delete every movie executive whose net worth is < \$10,000,000
- In SQL:  
**DELETE FROM Exec  
WHERE netWorth < 10,000,000;**

Anything wrong here?!

# Deletion

---

- Relation schema:

**Studio**(name, address, presC#)

**Movie**(title, year, length, filmType, studioName, producerC#)

- Update:

Delete from **Studio**, those studios not mentioned in **Movie**  
(i.e., we don't want to have non-productive studios!!)

- In SQL:

**DELETE FROM** Studio

**WHERE** name **NOT IN** (**SELECT** StudioName  
**FROM** Movie);

# Update

---

- Update statement consists of:
  - The keyword **UPDATE**
  - The name of a relation  $R$
  - The keyword **SET**
  - A list of formulas, each of which will assign a value to an attribute of  $R$
  - The keyword **WHERE**
  - A condition
- The syntax of the update statement:

**UPDATE**  $R$  **SET** <new-value assignments> **WHERE** <condition>;

# Update

---

- Database schema:

**Studio**(name, address, **presC#**)

**Exec**(name, address, cert#, netWorth)

- Update:

Modify table **Exec** by attaching the title ‘**Pres.**’ in front of the name of every movie executive who is also the president of some studio

- In SQL:

**UPDATE Exec**

**SET name = 'Pres. ' || name**              ← this line performs the update

**WHERE cert# IN (SELECT presC#**

**FROM Studio);**

# Defining Database Schema

---

- SQL includes two types of statements:
  - DML
  - DDL
- So far we looked at the DML part to specify or modify the relation/database instances.
- The DDL part allows us to define or modify the relation/database schemas.

# Defining Database Schema

---

- To create a table in SQL:

- **CREATE TABLE** *name* (*list of elements*);

- Principal elements are *attributes* and their *types*,  
but declarations of **key** and **constraints** may also appear

- Example:

```
CREATE TABLE Star (
 name CHAR(30),
 address VARCHAR(255),
 gender CHAR(1),
 birthdate DATE
);
```

# Defining Database Schema

---

- To delete a table from the database:

- **DROP TABLE** *name*;

- Example:

**DROP TABLE** Star;

# Data types

---

- **INT** or **INTEGER**
- **REAL** or **FLOAT**
- **DECIMAL**(n, d) -- **NUMERIC**(n, d)
  - **DECIMAL**(6, 2), e.g., 0123.45
- **CHAR(n)/BIT(B)** fixed length character/bit string
  - Unused part is padded with the "pad character", denoted as **\_**
- **VARCHAR(n) / BIT VARYING(n)** variable-length strings up to n characters
- Oracle also uses **VARCHAR2(n)**, which is truly varying length;  
Since **VARCHAR** uses fixed array with **end-marker**, it is not followed any longer in Oracle.

# Data types (cont'd)

---

## ■ SQL2 Syntax for:

- Time: 'hh:mm:ss[.ss...]'
- Date: 'yyyy-mm-dd' (m =0 or 1)

## ■ Example:

```
CREATE TABLE Days(d DATE);
INSERT INTO Days VALUES('2012-12-23');
```

- ❖ Note 1: In Oracle, the default format of date is 'dd-mon-yy', e.g.,  
`INSERT INTO Days VALUES('22-jan-18');`
- ❖ Note 2: The Oracle function `to_date` converts a specified format into default, e.g.,  
`INSERT INTO Days VALUES (to_date('2018-01-22', 'yyyy-mm-dd'));`

# Altering Relation Schemas

---

## ■ Adding Columns

- Add an attribute to an existing relation R:

**ALTER TABLE R ADD <column declaration>;**

## ■ Example: Add attribute phone to table Star

- **ALTER TABLE Star ADD phone CHAR(16);**

## ■ Removing Columns

- Remove an attribute from a relation R using DROP:

**ALTER TABLE R DROP COLUMN <column\_name>;**

## ■ Example: Remove column phone from Star

- **ALTER TABLE Star DROP COLUMN phone;**

**Note: Can't drop a column, if it is the only column**

# Note: Different Alter commands

---

- MySQL:

ALTER TABLE t ADD COLUMN bd DATE Constraints;

ALTER TABLE t DROP bd;

- SQL Server:

ALTER TABLE t ADD COLUMN bd DATE Cons...;

ALTER TABLE t DROP COLUMN bd;

- Oracle:

ALTER TABLE t ADD bd DATE Constraints;

ALTER TABLE t DROP COLUMN bd;

# Attribute Properties

---

- We can assert that the value of an attribute A to be:
  - **NOT NULL**
    - Then every tuple must have a “real” value (not null) for this attribute
  - **DEFAULT value**
    - Null is the default value for every attribute
    - However, we can consider/define any value we wish as the default for a column, when we create a table.

# Attribute Properties

---

```
CREATE TABLE Star (
 name CHAR(30),
 address VARCHAR(255),
 gender CHAR(1) DEFAULT '?',
 birthdate DATE NOT NULL);
```

- Example: Add an attribute with a default value:
    - **ALTER TABLE** Star **ADD** phone CHAR(16) **DEFAULT** 'unlisted';
  - **INSERT INTO** Star(name, birthdate) **VALUES** ('Sally', '0000-00-00')
- | name  | address | gender | birthdate  | phone    |
|-------|---------|--------|------------|----------|
| Sally | NULL    | ?      | 0000-00-00 | unlisted |
- **INSERT INTO** Star(name, phone) **VALUES** ('Sally', '333-2255');
    - this insertion op. fails since the value for **birthdate** is not given, since Null was disallowed by the user.

# Attribute Properties

---

To add default value after an attribute is defined:

- **ALTER TABLE** Star **ALTER phone SET DEFAULT 'no-phone';**
- In Oracle:  
**ALTER TABLE** Star **MODIFY phone DEFAULT 'no-phone';**

# Conceptual Database Design

---

**Entity/Relationship (E/R) Model**

# Database Design

---

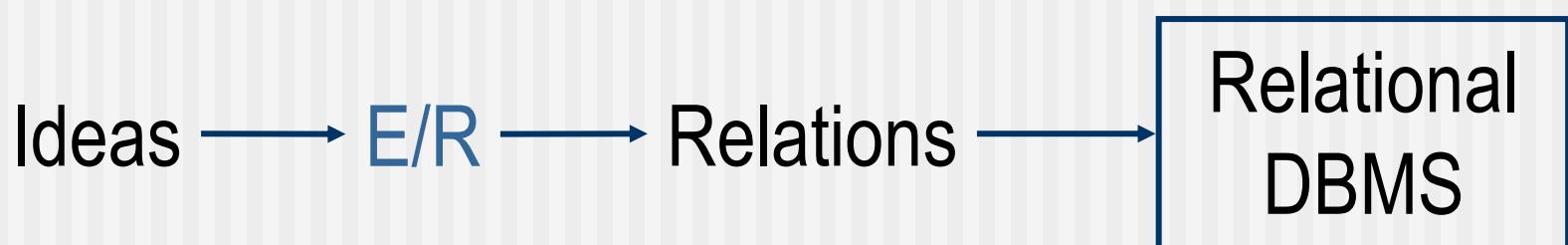
- Requirements collection and analysis
  - Determine what information the database must hold
  - Determine the relationships among the components of that information
- ➔ *Conceptual database design* using some **data model**
- A **data model** is a collection of concepts for describing:
  1. data and relationships among data
  2. data semantics and constraints

# Design Approaches & Notations

---

- Entity-Relationship Model (E/R)
- Object-Oriented (e.g., Object Definition Language -- ODL)
- Semi-structured data (e.g., XML)

## A Design Process



# Entity-Relationship (E/R) Model

---

- E/R model (Chen 1976) is a *graphical* language/notation to present a database model
- It grew out of modeling applications
- Widely used in conceptual database design
- No single standard for the E/R language/notation

# Entity-Relationship (E/R) Model

- Collection of abstraction / modeling primitives



Entity set



Relationship set



Attribute



Multiplicity of relationships



Referential integrity



Weak entity set



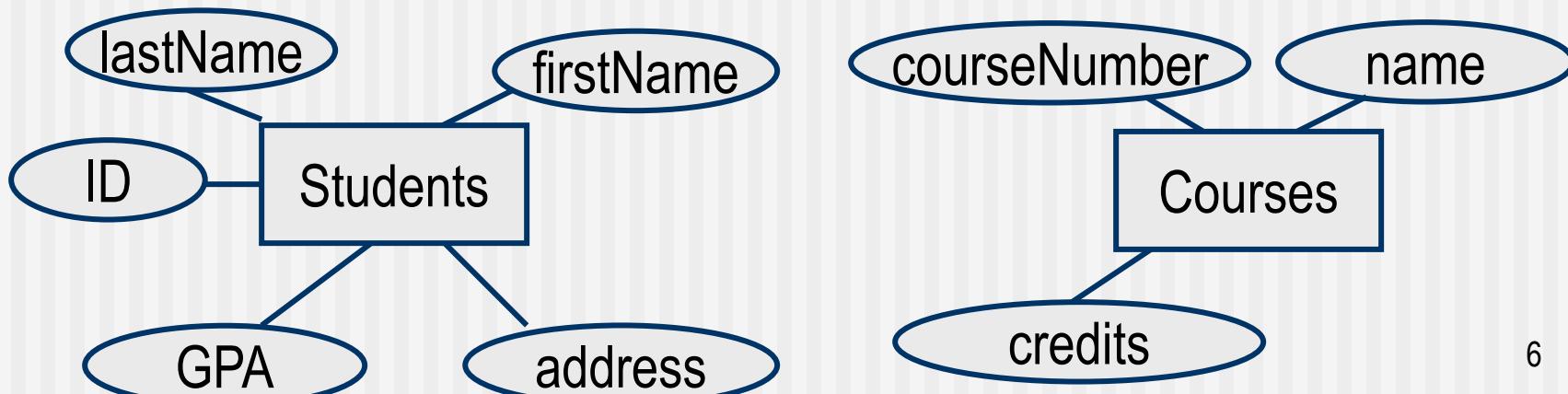
Weak relationship set



Inheritance

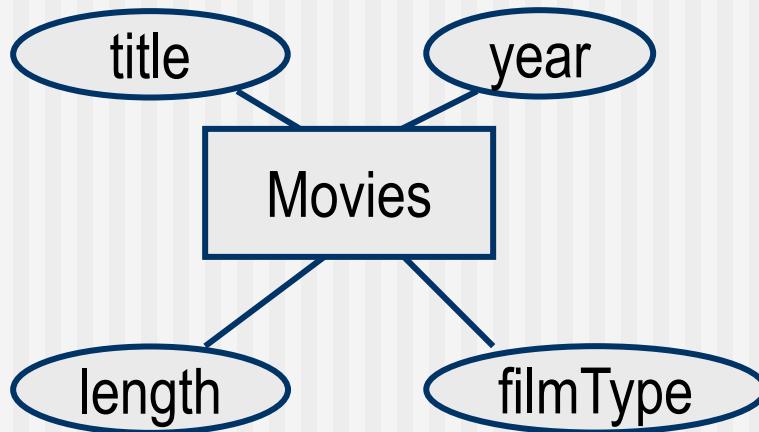
# Entities and entity sets

- **Entity** – A real world object that is “distinguishable” from other objects
- **Entity Set** -- A collection of similar entities
  - All entities in an entity set have the same set of **attributes**
- In ODL:
  - **Object** corresponds to entity
  - **Class** corresponds to entity set



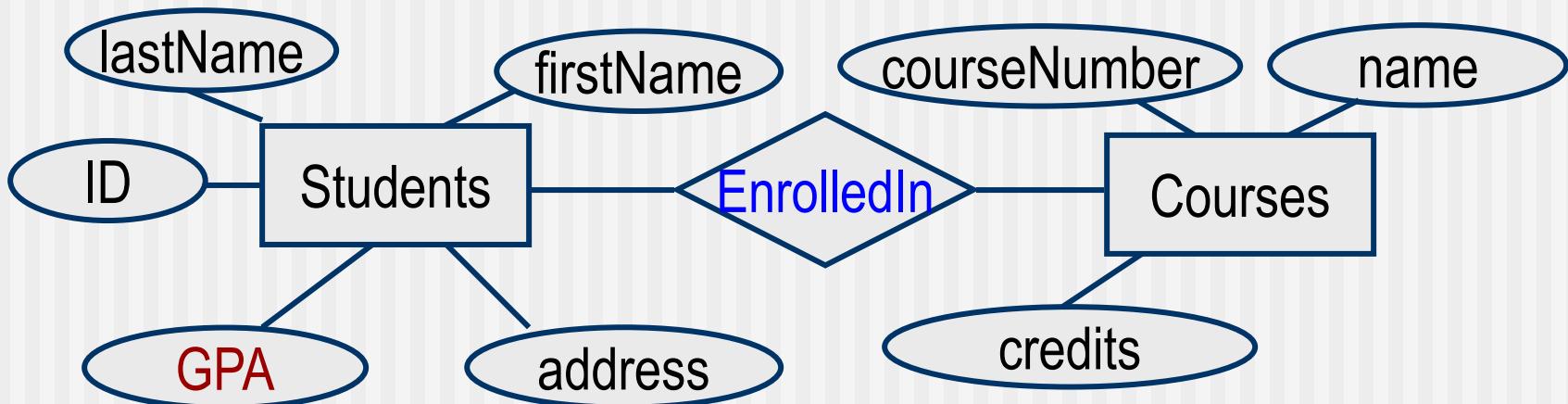
# Entities and entity sets

---



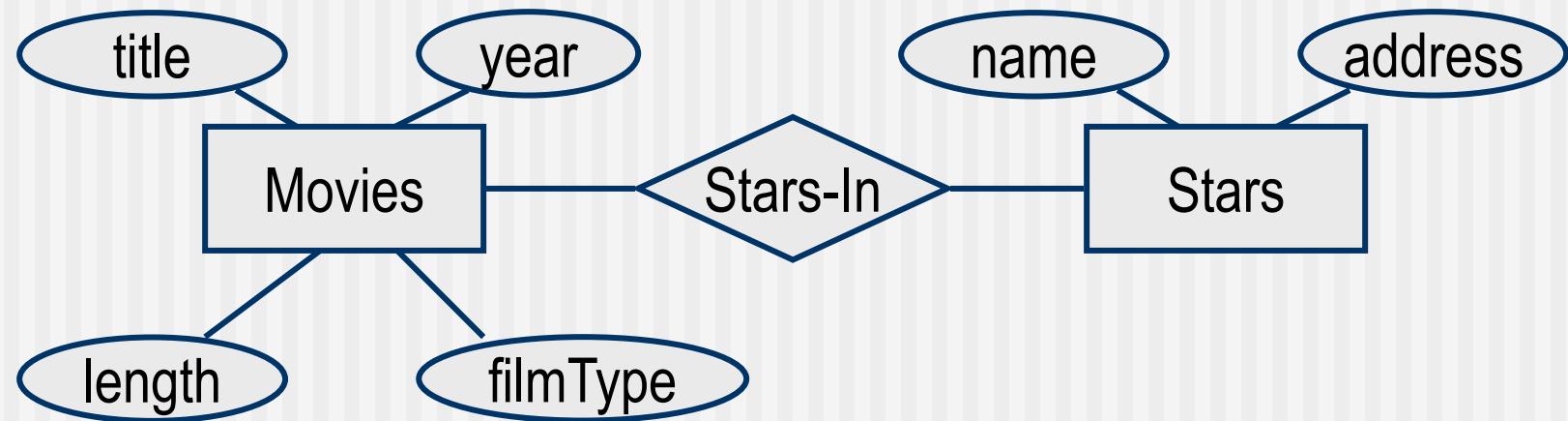
# Relationships and Relationship Sets

- **Relationships** are association among entities
- **Relationship set** is a set of relationships of the same type
  - If  $E_1, \dots, E_n$  are entity sets, a relationship  $R$  on these sets is defined as:  $R \subseteq E_1 \times \dots \times E_n$ 
    - In general, relationships are  $n$ -ary, where  $n \geq 2$
    - Many database relationships are binary



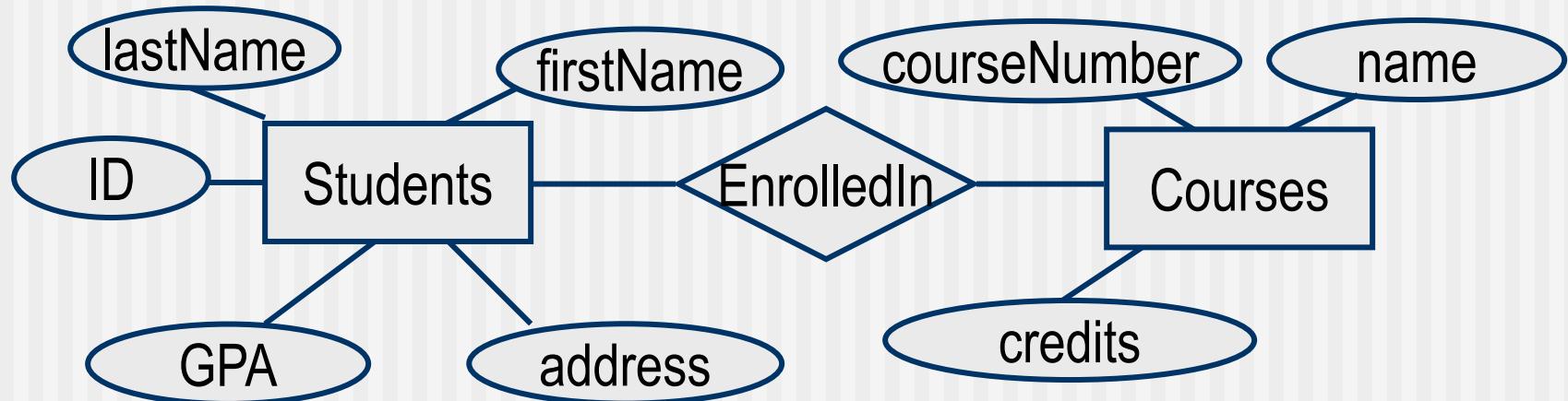
# Relationships and Relationship Sets

---



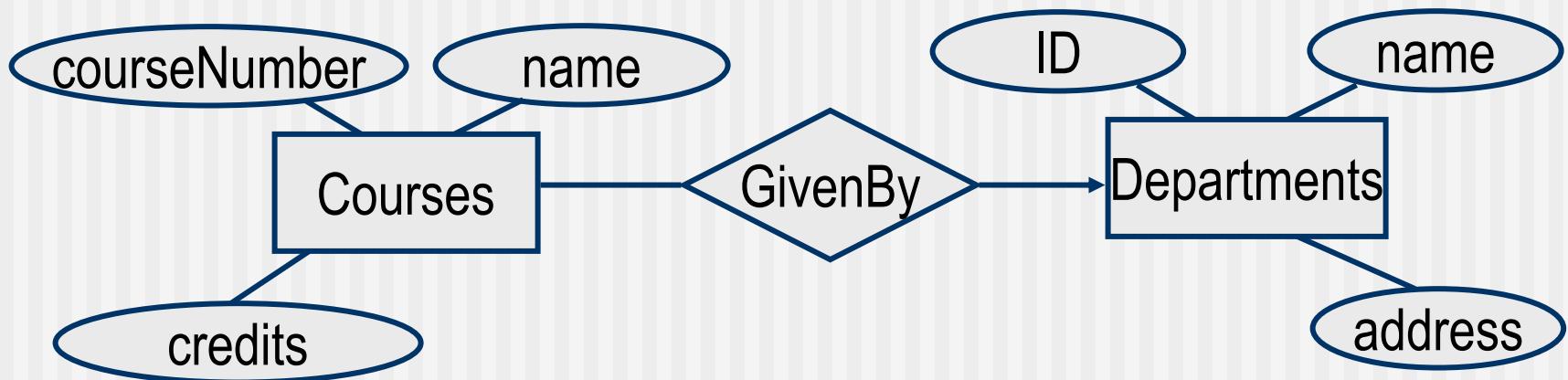
# Multiplicity of Binary Relationships

- **EnrolledIn** relationship between entity sets **Student** and **Course** is **many-many**



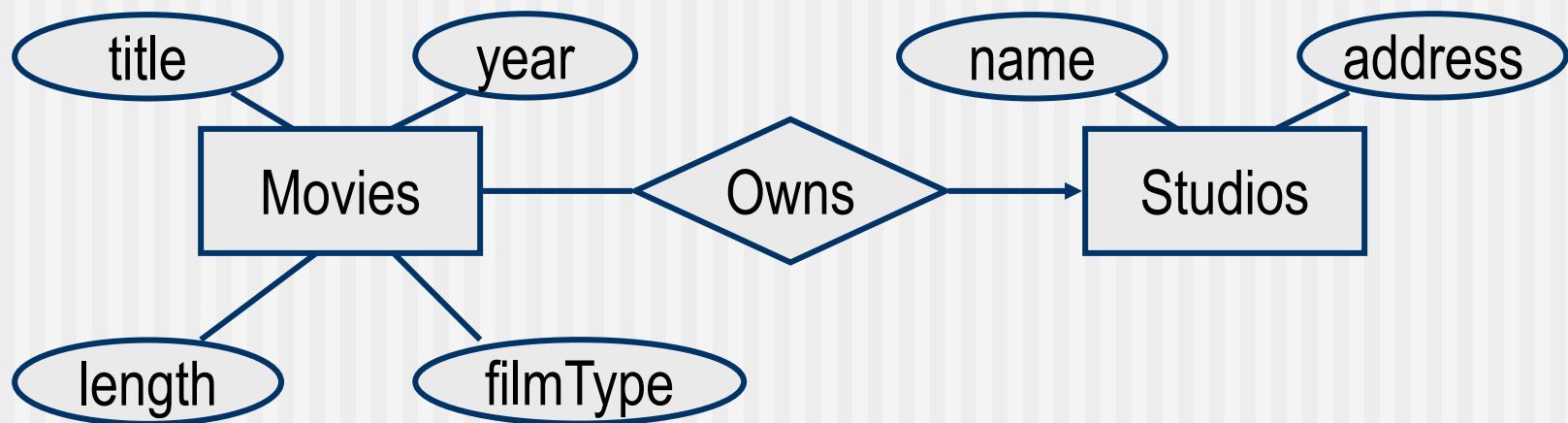
# Multiplicity of Binary Relationships

- In E/R diagrams, arrows can be used to indicate the multiplicity of a relationship



- If a relationship is **many-one** from entity set **Course** to **Department**, we place an **arrow entering Department**
- The arrow indicates that each entity in the entity set **Course** is related to **at most one** entity in the entity set **Department**
- In this case, we may also say that the relationship from **Department** to **Course** is one-many (also shown as 1-M).

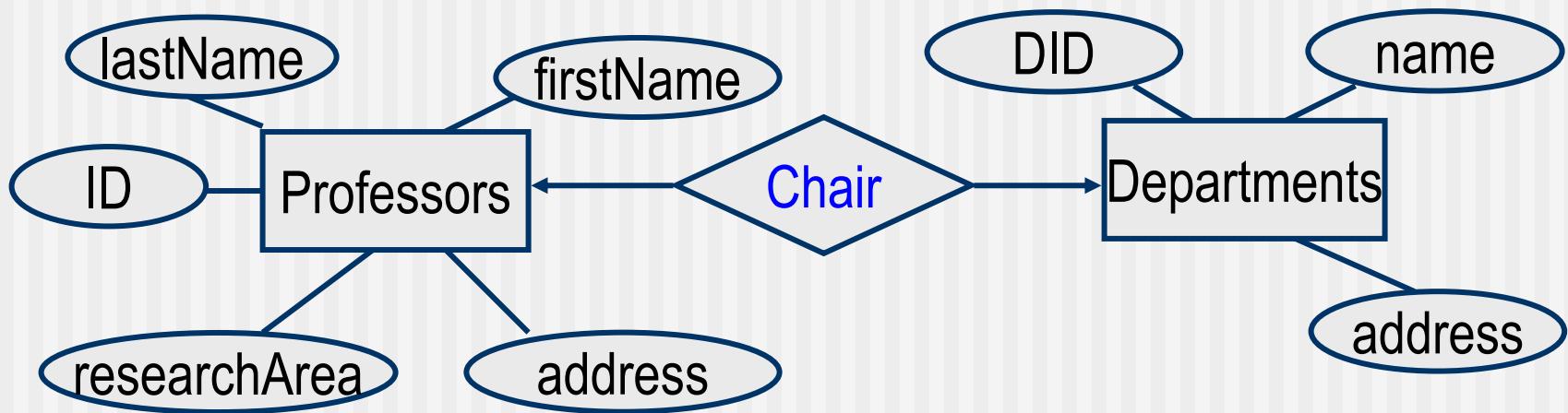
# Multiplicity of Binary Relationships



- The arrow indicates that each entity/tuple in relation **Movie** is “related to” **at most one** entity in **Studio**
- **Owns** relationship is **one-many** from **Studio** to **Movie**

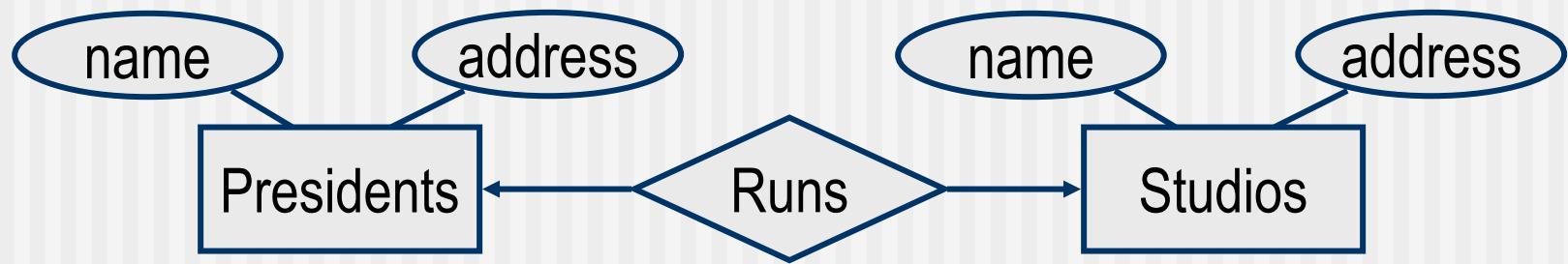
# Multiplicity of Binary Relationships

- The relationship **Chair** between the entity sets **Department** and **Professor** is **one-one**, indicating which professor is the chair of which department. Note that this represents also the situation where a department is assigned no chair.



# Multiplicity of Binary Relationships

---

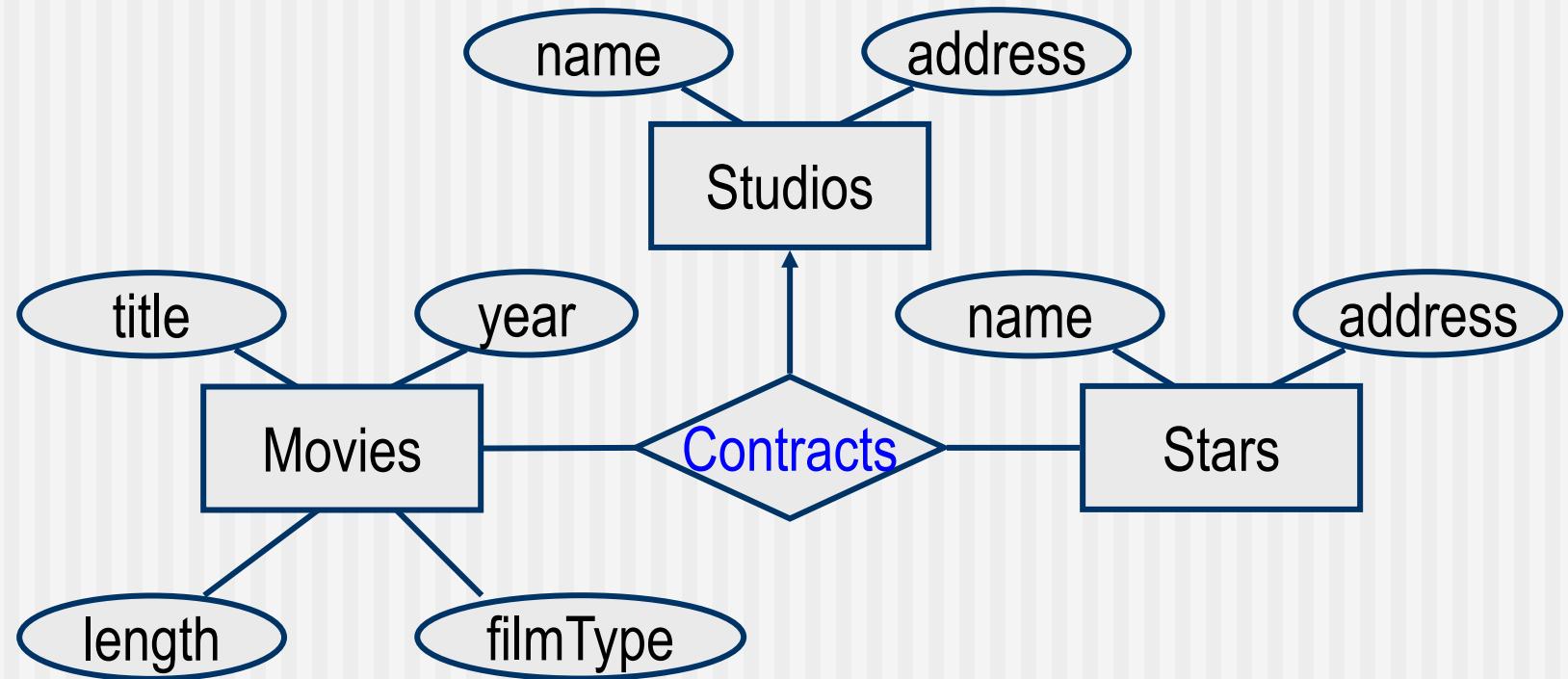


# Multiway Relationships

---

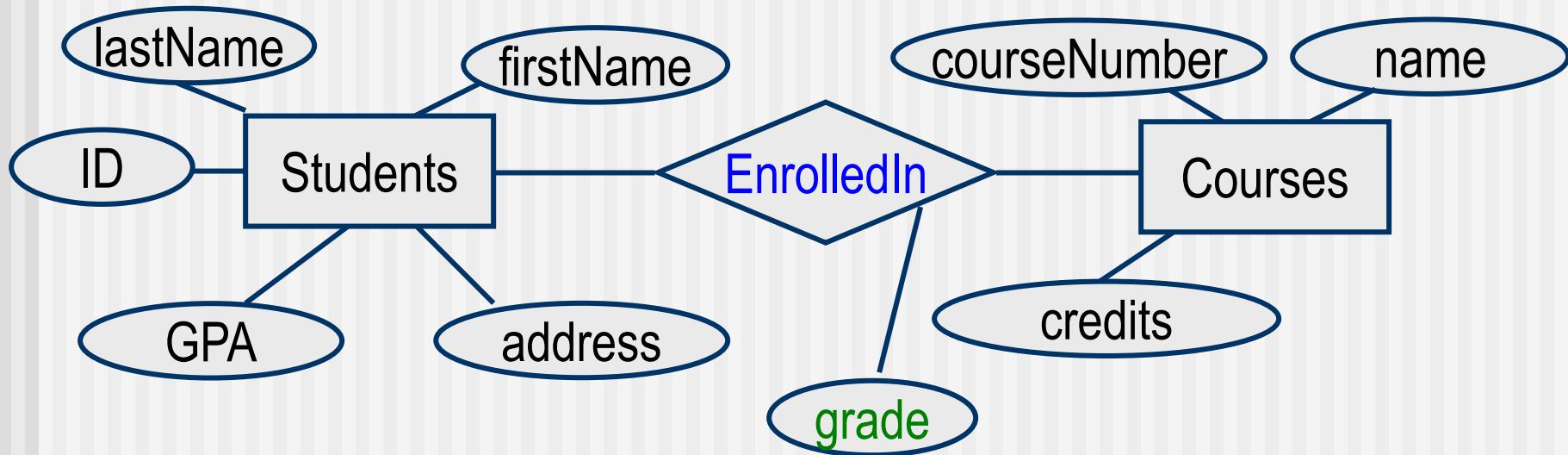
- **ODL** (an OO notation which we will introduce later) allows defining only **binary** relationships, i.e., relationships involving two classes.
- In general, we need to represent **n-ary** (multi-way) relationships, i.e., relationships involving more than two entity sets
- **E/R** model allows defining n-ary relationships *conveniently*.
- An n-ary relationship in **E/R** is represented by a diamond connecting all entity sets involved.

# Multiway Relationships



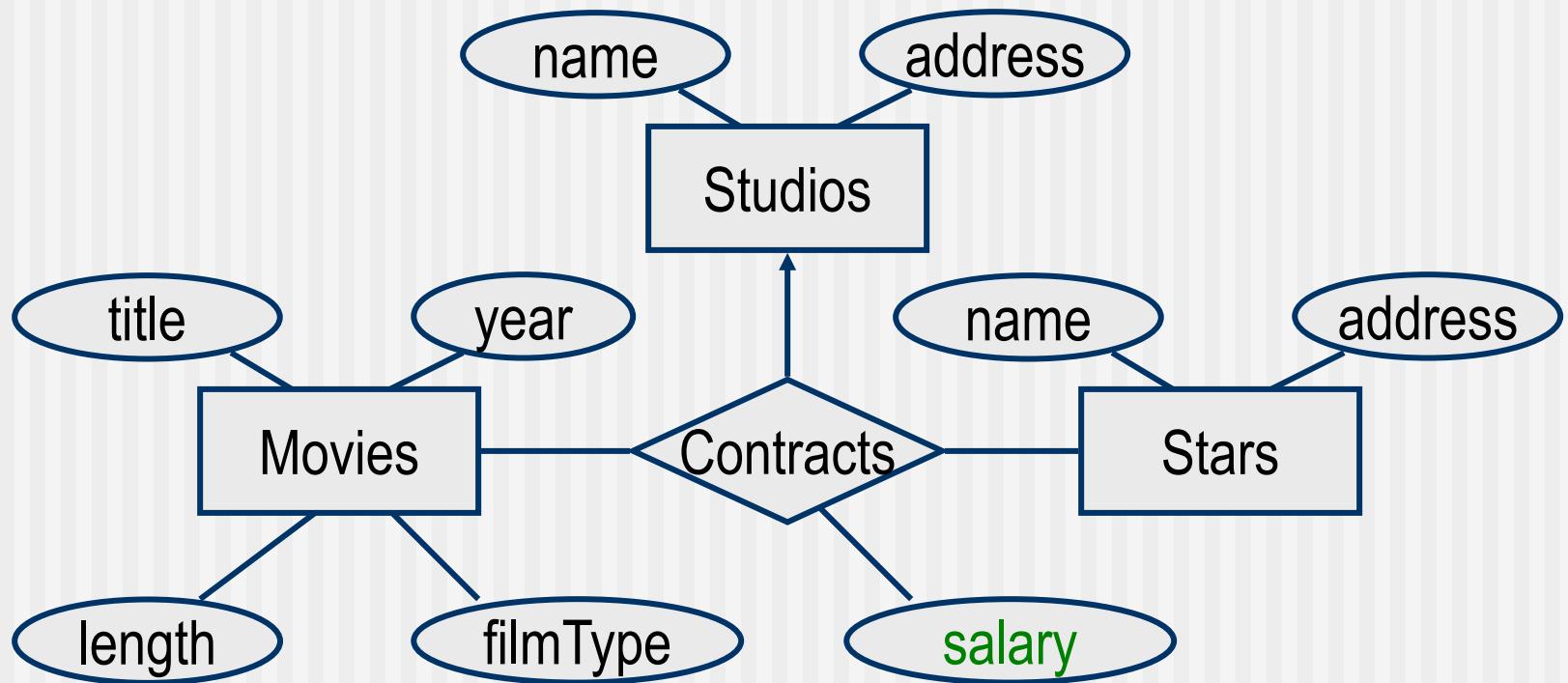
# Attributes on Relationships

- Consider **EnrolledIn** relationship between **Student** and **Course**



- We may wish to record student's **grade** for this course
- Where should it go?
- Sometimes, it is more appropriate to associate **attributes** with a **relationship** rather than with the entity sets involved

# Attributes on Relationships



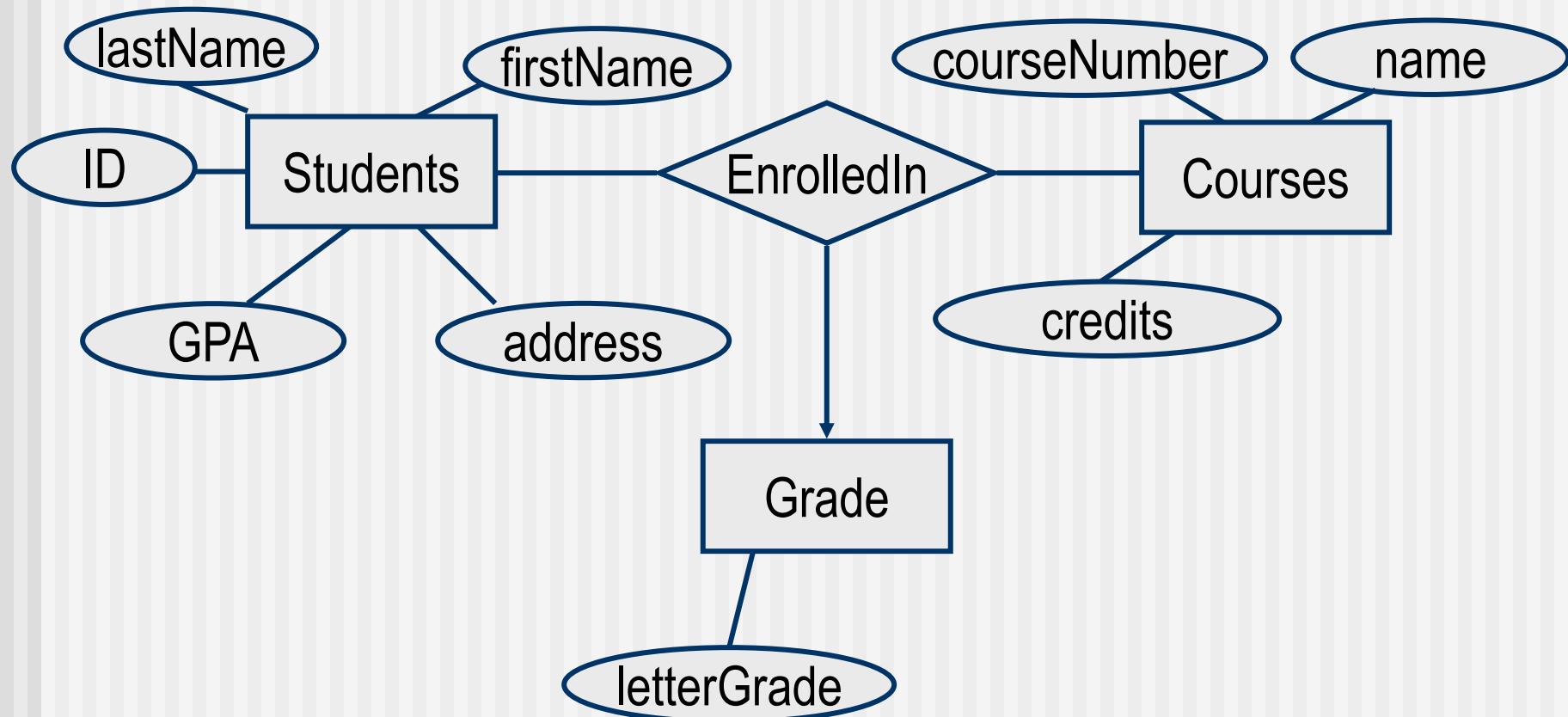
- Suppose we want to record the star's **salary** associated with this contract
- Where should it go?

# Moving an attribute to an entity set

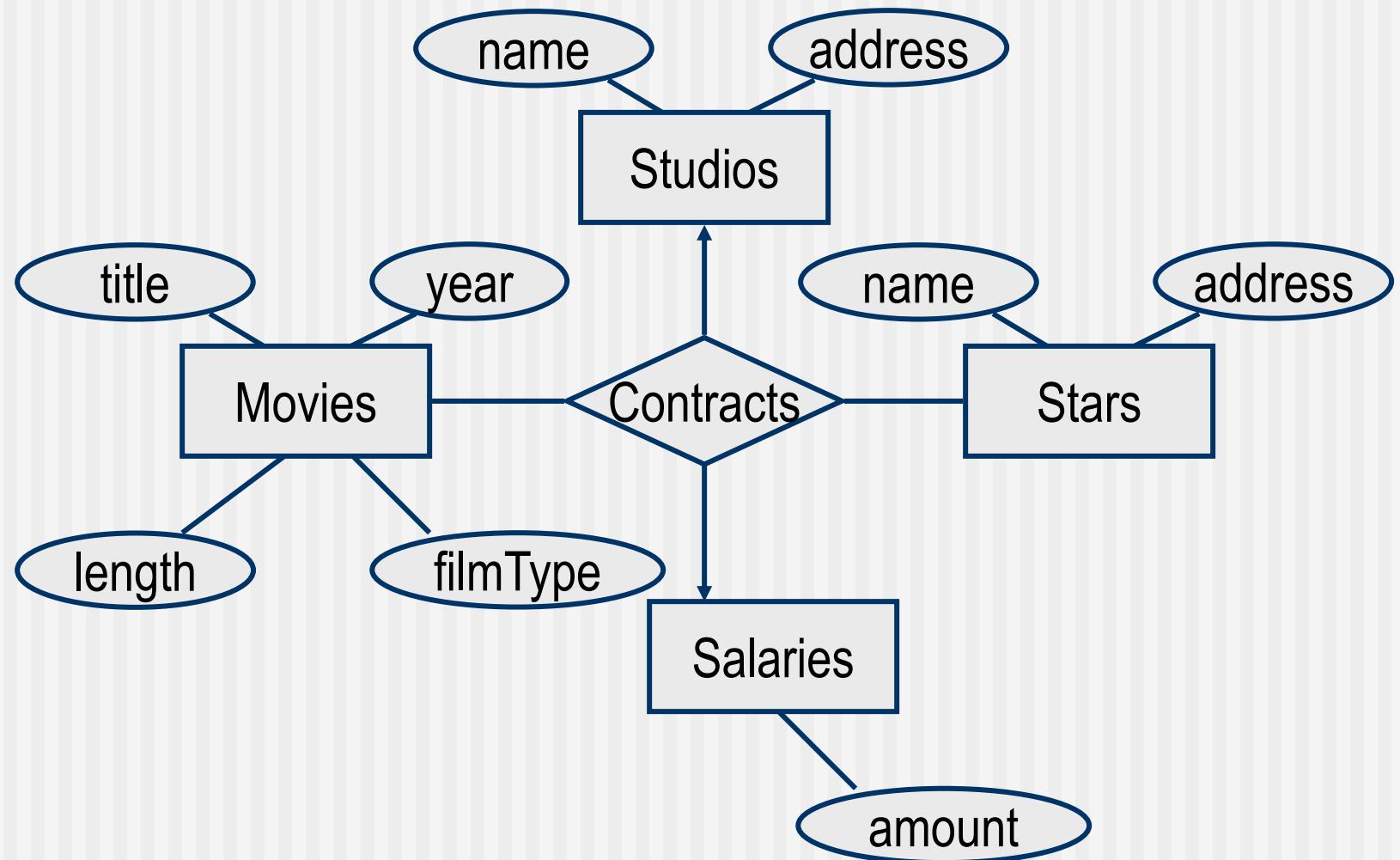
---

- We could add an attribute(s) to a **relationship**
- Alternatively, we could do the following:
  - invent a **new** entity set, whose entities have the attributes ascribed to the relationship
  - “connect/include” this entity set in the relationship
  - omit the attribute from the relationship itself
- Consider again the “**salary**” for the “**Contract**” example:

# Moving an attribute to an entity set



# Moving an attribute to an entity set

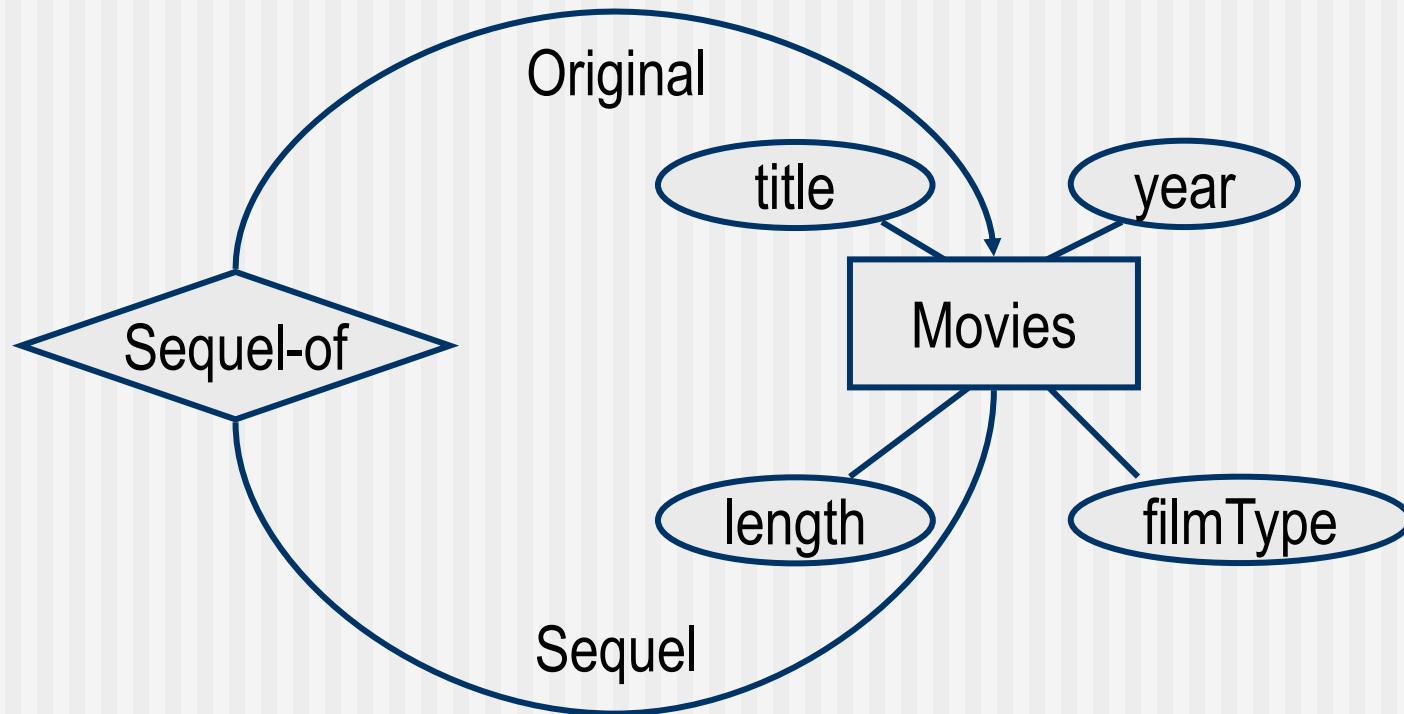


# Roles in Relationships

---

- It is possible that one entity set appears **two or more** times in a relationship
- Suppose, we want to capture relationship between two **movies**, one of which is the **sequel** of the other

# Roles in Relationships

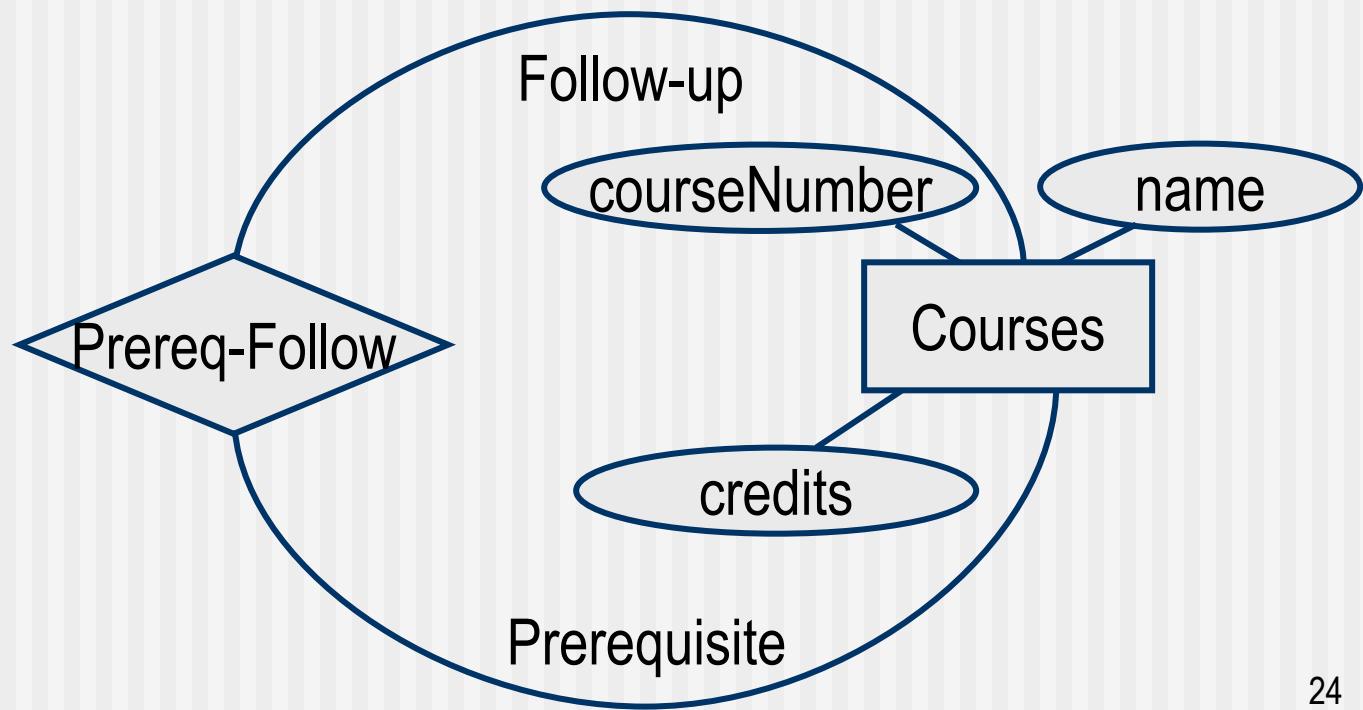


**Each line** to the entity set represents a different **role** that the entity set plays in the relationship

# Roles in Relationships

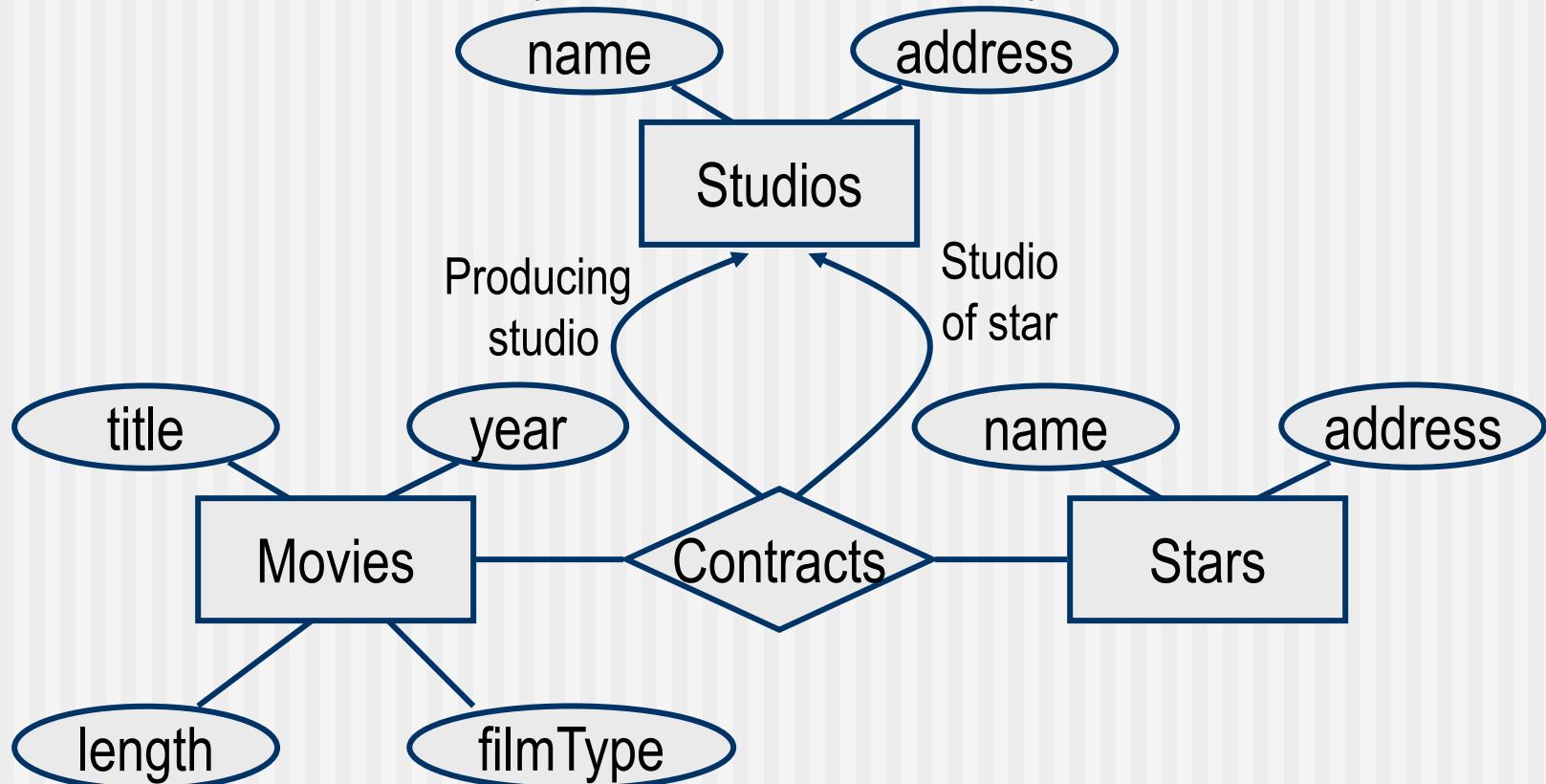
---

*Each line to the entity set represents a different **role** that the entity set plays in the relationship*



# A more complex example

- Suppose, each star is associated with exactly one studio
- Supp. studio  $s_1$  of star  $a_1$  may further contract with another studio  $s_2$  to allow  $a_1$  to play in movie  $m_2$  made by studio  $s_2$

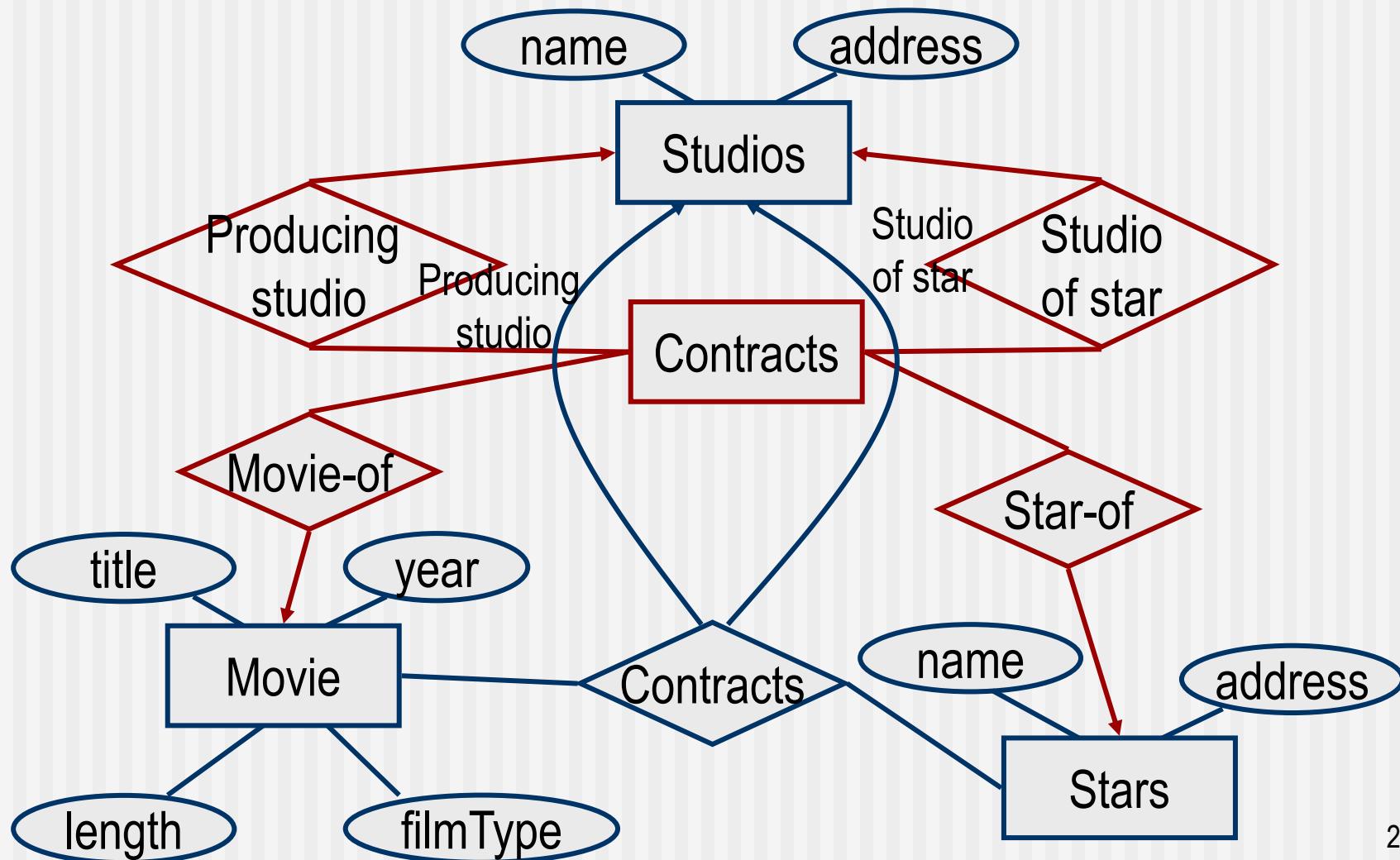


# Converting n-ary relationship to binary

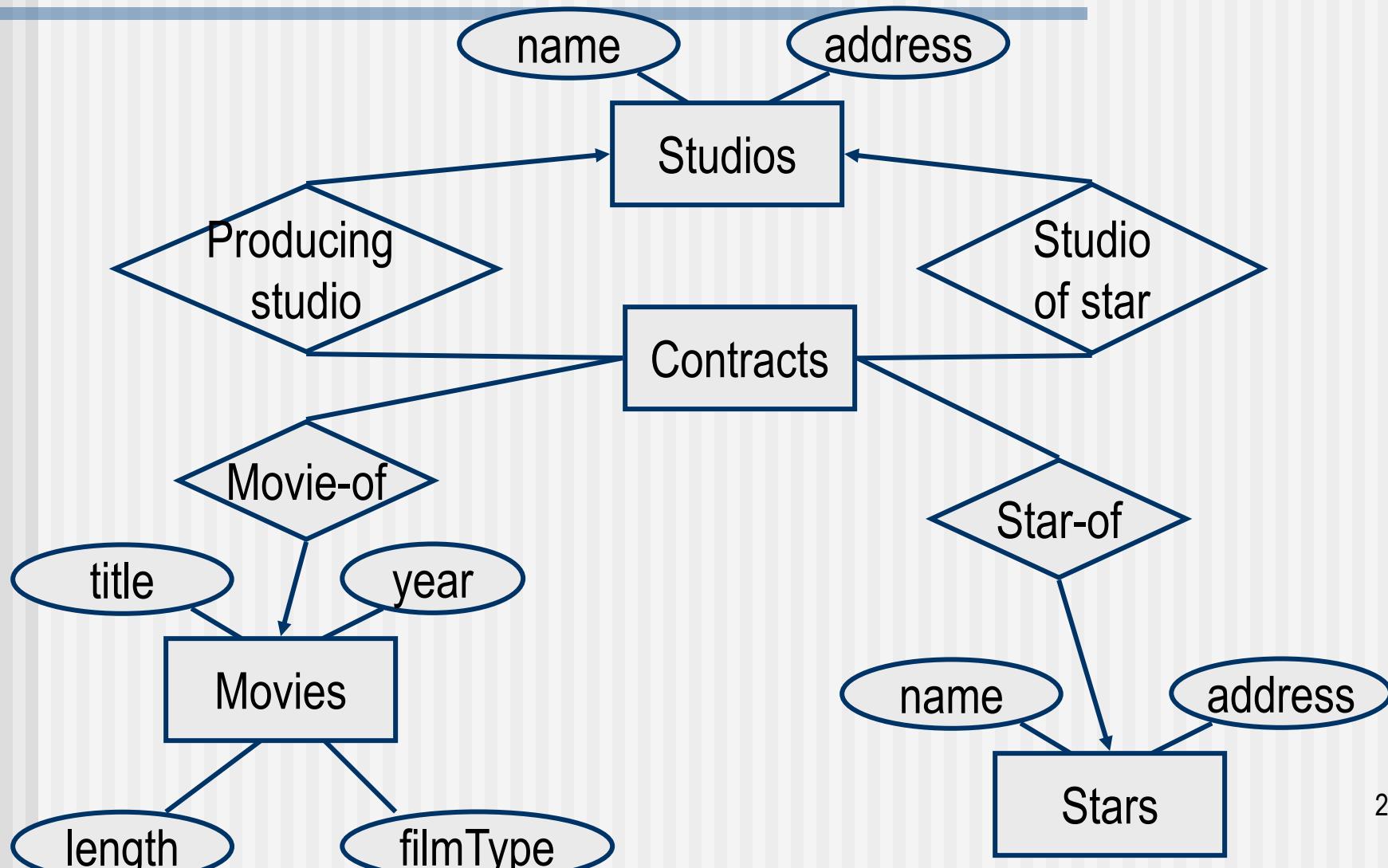
---

- Any n-ary relationship  $R$  can be converted into a **collection** of  $M-1$  **binary** relationships **without loosing** the information represented by  $R$
- To do this:
  - Introduce a **new** entity set  $E$ , called **connecting** entity set, whose entities might be thought of as tuples in  $R$  (the original n-ary relationship)
  - Introduce  $M-1$  relationships from  $E$  to each one of the entity sets involved in  $R$
  - If an entity set  $E$  plays **more than one** role, then  $E$  is the target of one relationship for **each role**

# Converting n-ary relationship to binary

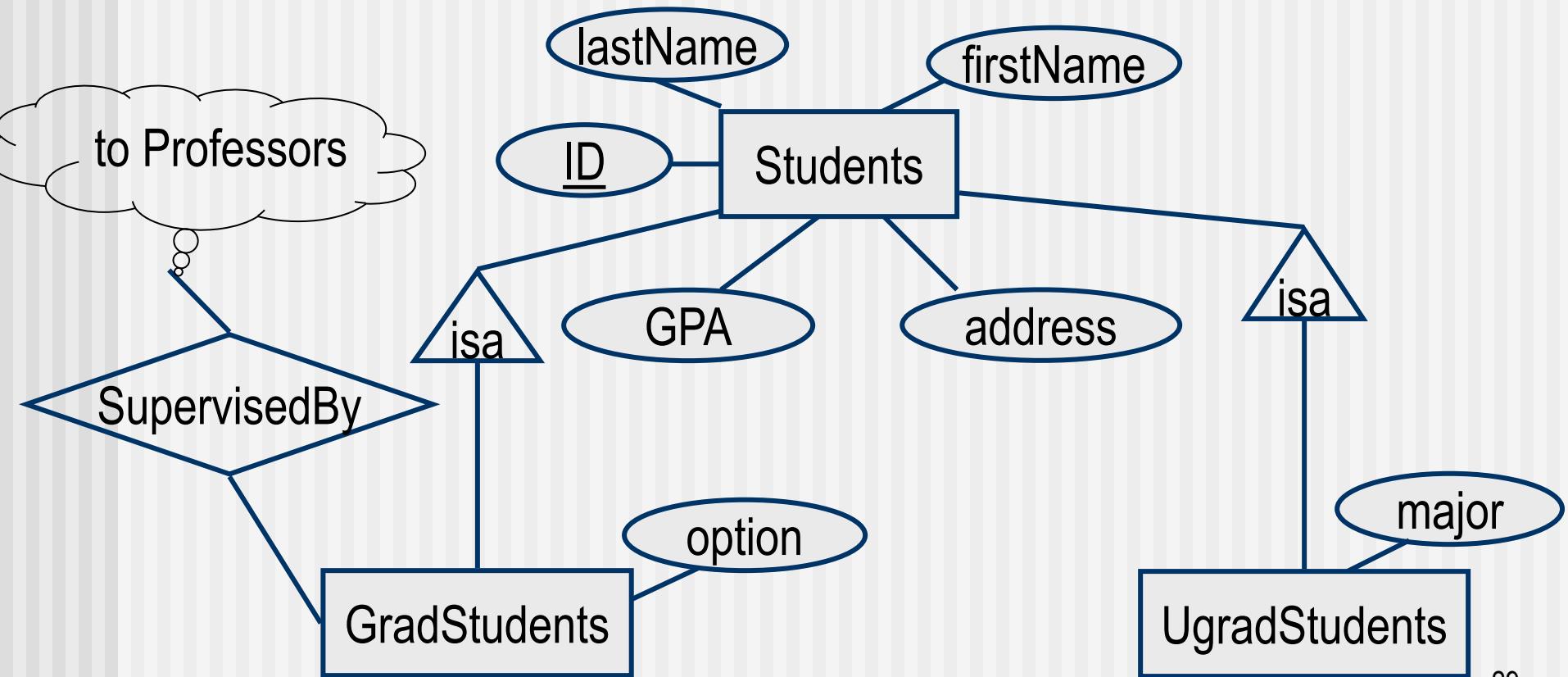


# Converting n-ary relationship to binary

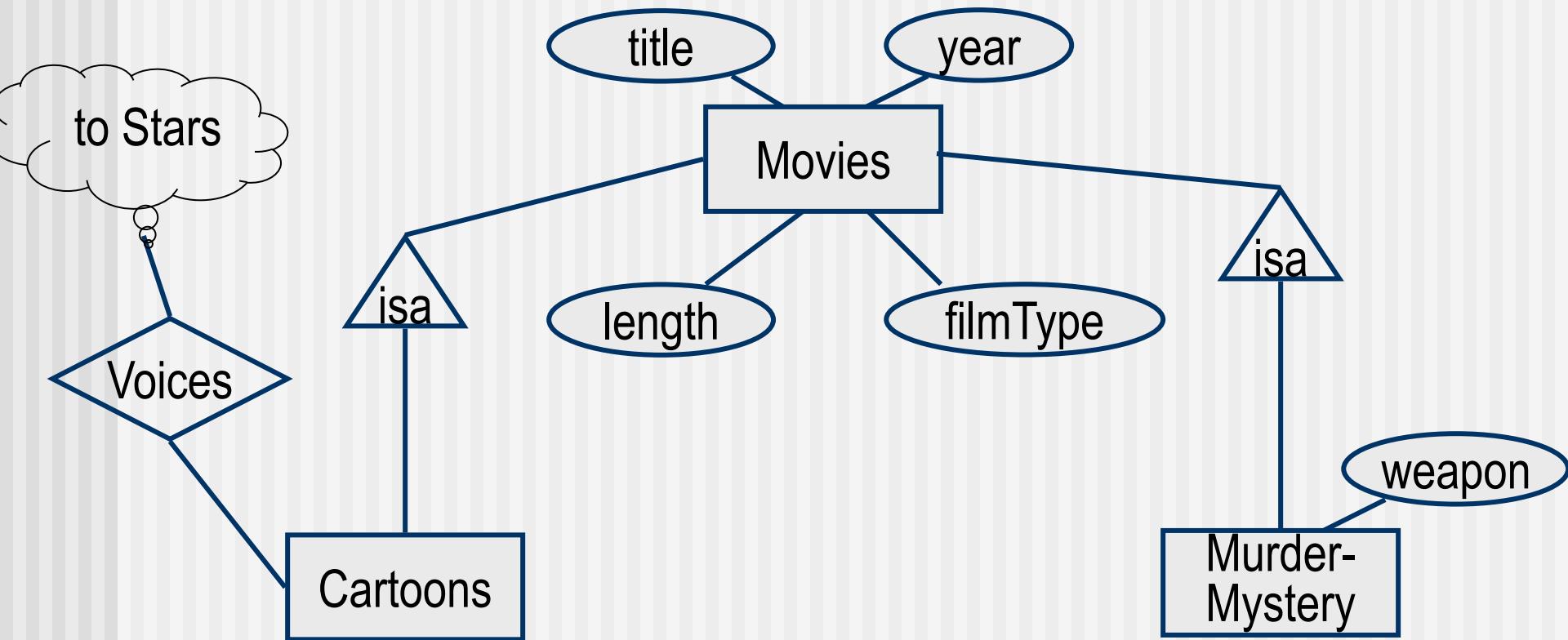


# Inheritance in E/R

- Inheritance in E/R is expressed by *isa* relationship



# Inheritance in E/R



# Inheritance in E/R

---

- There is a subtle difference between the concept of inheritance in an OO approach and in E/R
  - In OO, an object must be a member of exactly one class
  - In E/R
    - We consider **an entity** as having “components” belonging to several entity sets that are “part of” a single **isa**-hierarchy
    - The “components” are connected into a single entity by the **isa** relationships
    - The **entity** has whatever **attributes** any of its components has, and participates in whatever **relationships** its components participate in
- ➔ In an E/R diagram, we represent an entity set (e.g., **CartoonMurderMystery**) only if it has attributes and/or relationships of its own

# Constraints

---

- There are some important **aspects** of the **real world** that **cannot** be represented using the **ODL** or **E/R** model introduced so far
- The additional information about these aspects often takes the form of **constraints** on the data
- Sometimes modeling this additional information goes beyond the **structural** and **type** constraints imposed by **classes**, **entity sets**, **attributes**, and **relationships**

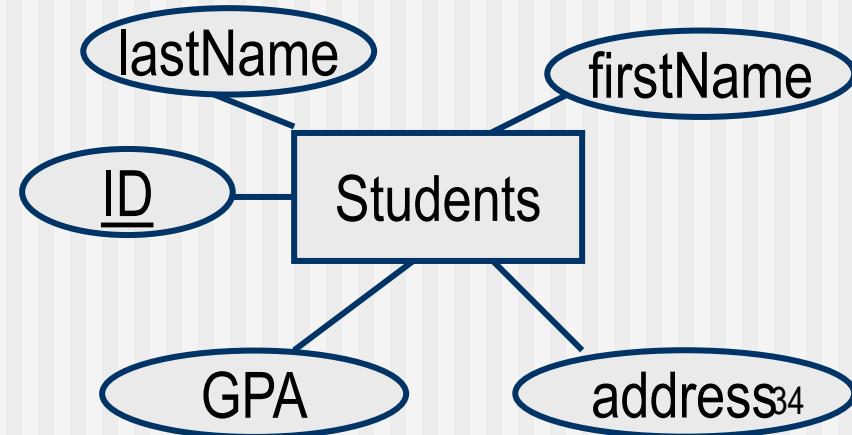
# A Classification of Constraints

---

- A **Key K** is a set of attribute(s) that **uniquely** identifies an object within its class or an entity within its entity set R; Defn:  $K \subseteq R$ .
  - That is, **no two entities may agree in all their key values**
- **Single-value constraints** are requirements that the value in a certain context/role be unique. In addition to *key constraints*, there are other sources of single-value constraints (e.g., M:1 or 1:1 relationships, like a department's chair)
- **Referential integrity constraints** are requirements that a value referred to by some object/entity must actually exist in the database; This means, **no dangling pointers**
- **Domain constraints** require that the value of an attribute must be drawn from a specific **set** of values (called attribute **domain**), or lies within a specific range
- **General constraints** – arbitrary assertions that must hold on the db

# Keys

- A *superkey* is a set of attributes whose values uniquely identify an entity (object) in the entity set (class); this set may not be minimal.
- A **minimal superkey** is called a (*candidate*) **key**.
- An entity set may have more than one candidate **key**. One of them is picked (*by ?*) as the **primary key**; others may be called *alternates*
- In E/R, we underline the attributes forming a key of an entity set
- No notation in E/R for alternate keys

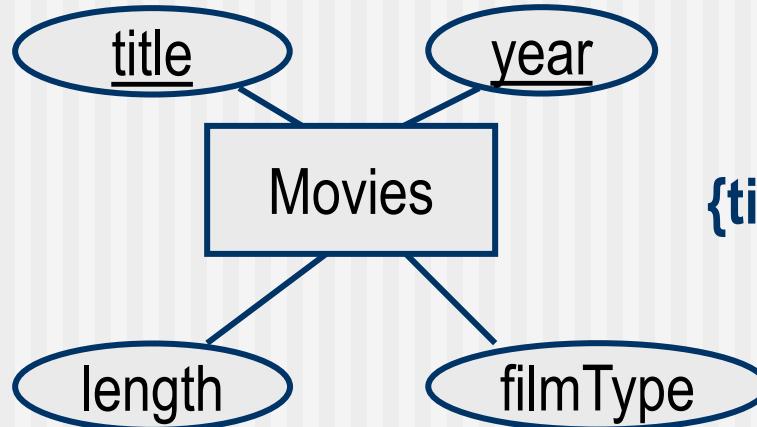


Here, ID is the key of **Student**

# Example

---

- What should we consider as a key for **Movie** ?
- **title**?
  - there could be different movies with the same name
- **{title, year}**?
  - there still could be two movies with the same title made in the same year, but that's very unlikely as we understand!



**{title, year}** is a key for **Movie**

# Example

---

- What should be the key for **Star = {name, address}**?
- **name**?
  - We may think that the name cannot serve to distinguish two people, but for stars, the names distinguish them, since traditionally they choose “stage names” as names



{**name**} is the key for **Star**

# Example

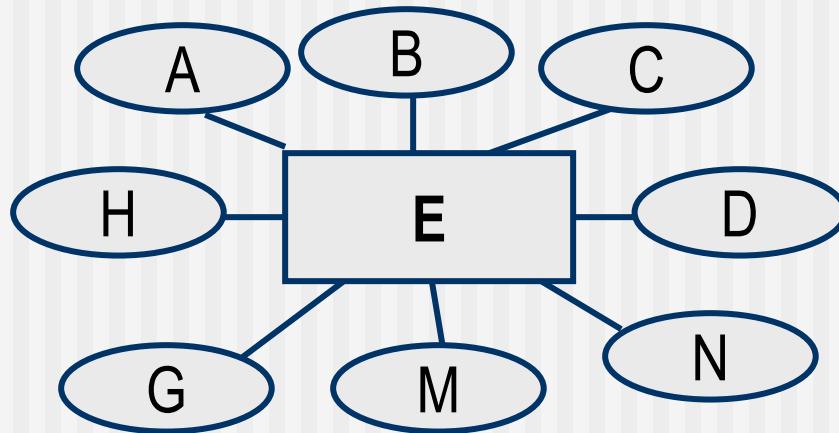
---

- What should be considered as the key for **Studio**?
- **name**?
  - It is “reasonable” to assume that there are **no** studios with the **same names**



{name} is the key for **Studio**

# Selecting Primary Key



Suppose candidate keys for **E** are :

1. {A, B}
2. {D, N}
3. {G, M, N}

- Which of the three should be pick as the **primary key**?

# Selecting Primary Key

---

- Some criteria to choose a **primary key** when there are alternate keys (i.e., more than one candidate):
  - Total size of the attributes forming a key
  - Number of attributes forming a key
  - Convenience/Natural choice
  - A combination of the above

# Single-Value Constraints

---

- In E/R:
  - attributes are **atomic** (first normal form, 1NF)
  - an arrow ( $\rightarrow$ ) can be used to express the multiplicity
  - What about multi-valued or structured in E/R?  
No for **attributes** but Yes for **relationships**

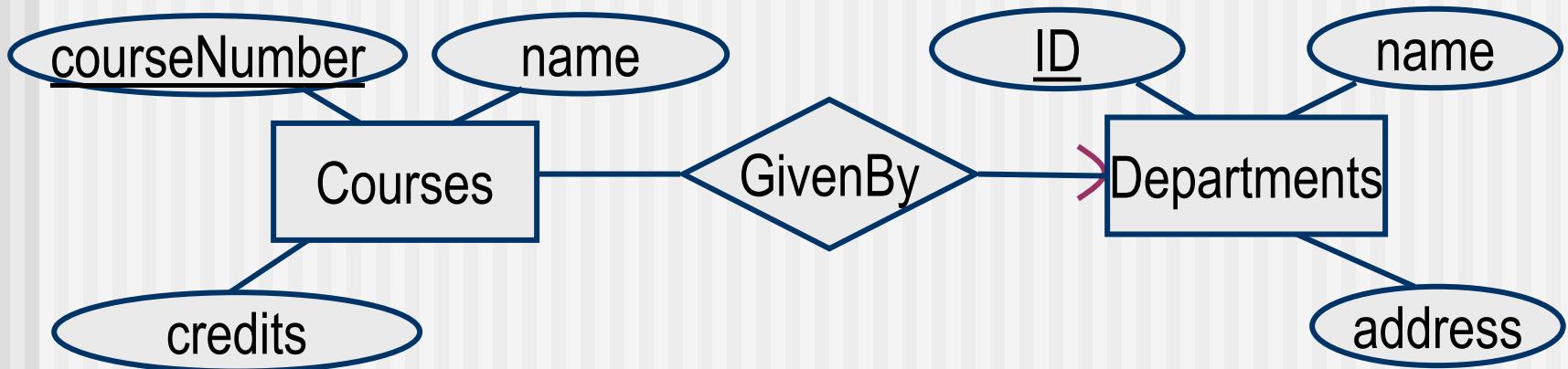
# Single-Value Constraints

---

- Suppose A is a single-valued attribute.
- Using the E/R model introduced so far, we can't:
  - Express that the value for A *must* be present  
(e.g., when A is the key or part of a key), or
  - Express that the value for A *may* be present  
(e.g., when A's value is optional, for which we use NULL)
- If the choice is not explicit, we use the following “defaults”:
  - The value for A must exist if A is part of the key
  - The value for A is optional, otherwise

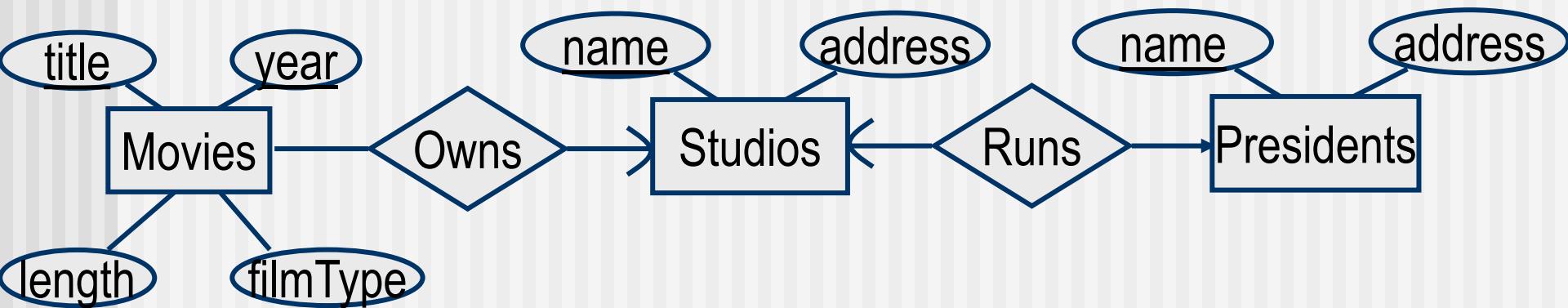
# Referential Integrity Constraints

- For relationships:
  - Single-value + Existence = Referential Integrity Constraint
- We extend the **arrow** notation to indicate a reference is mandatory (to support referential integrity)



- This means, there is no course listed in the database unless there is a department giving the course.

# Referential Integrity Constraints



- The studio for a movie must always be present in **Studio**
- If someone is a president, then the studio that he/she runs must exist in **Studio**
- However, the model allows studios without presidents (temporarily)

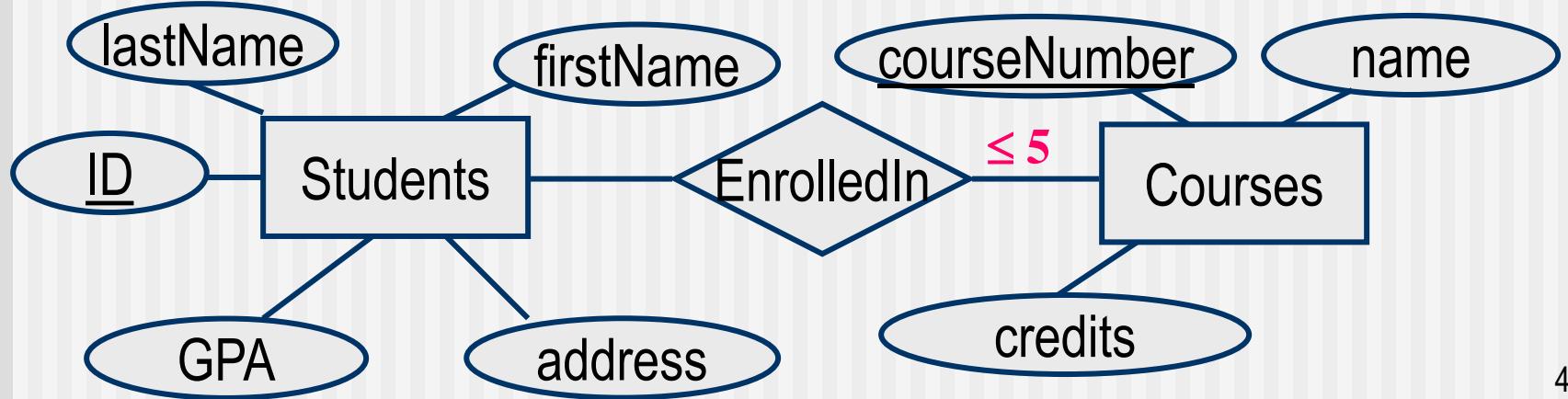
# Domain constraints

---

- Domain constraints restrict the values of an attribute to be drawn from a set
  - E/R does not support imposing domain constraints
  - ODL allows using types to limit/control possible values of the attributes
  - ODL does not support restricting the “range” of values allowed for an attribute

# Relationship degree constraints

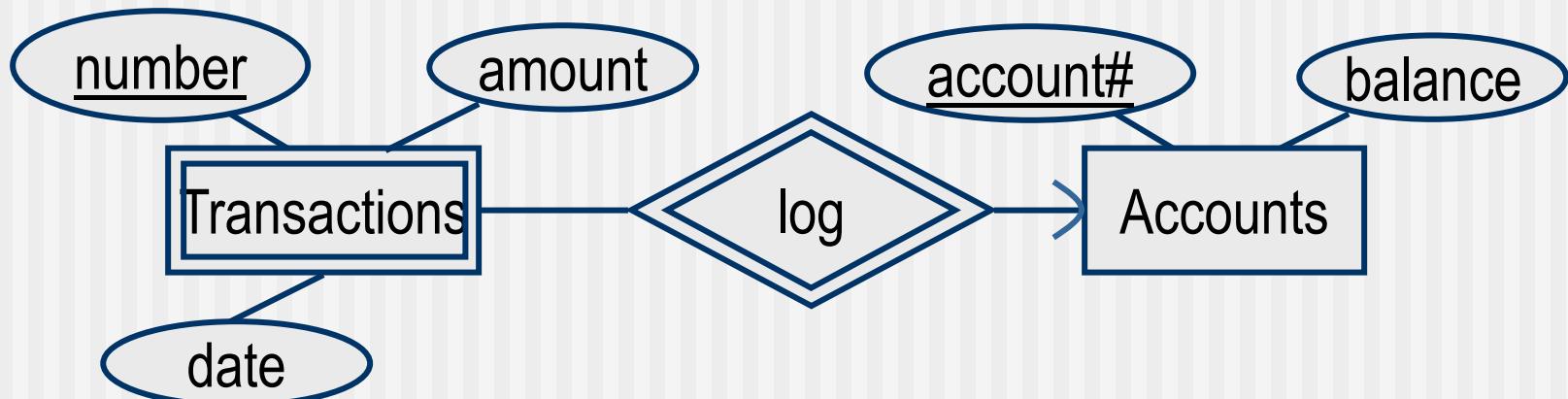
- **Relationship degree constraints** (multiplicity) restrict the number of entities in the entity sets involved in a relationship
- For example, we can impose a constraint that:  
*a student cannot be enrolled in more than 5 courses*
  - In E/R, we may attach a “bounding number” to the corresponding link
  - In ODL, a set of references or an **array** of size 5 (of reference type) will do



# Weak Entity / Relationship Sets: Example

---

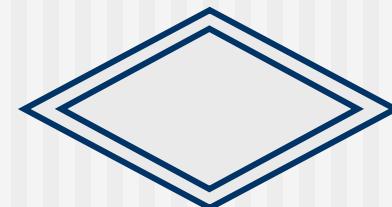
- Log records the transactions initiated through an ATM
- Each transaction has a number, a date, and an amount
- Different accounts might have transactions by the same number, on the same date, and for the same amount



# Weak Entity / Relationship Sets

---

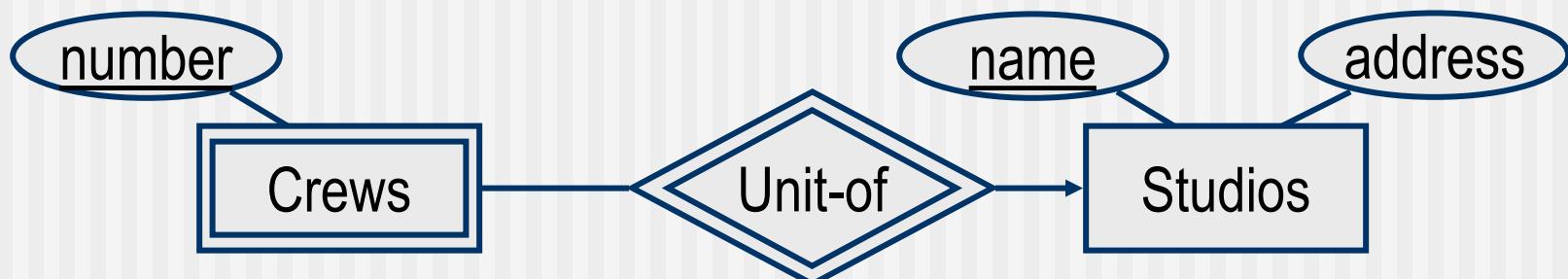
- A **strong** entity set has a key
- A **weak** entity set does not have sufficient attributes of its own to form a key. It participates in a M-1 relationship (with no descriptive attributes) with a strong entity set
- **Discriminator** of a weak entity set is a set of attributes that distinguishes among the entities corresponding to a strong entity
- **key** of a weak entity set = key of the strong entity together with the discriminator of the weak entity
- In E/R, these are represented by:



# Another example

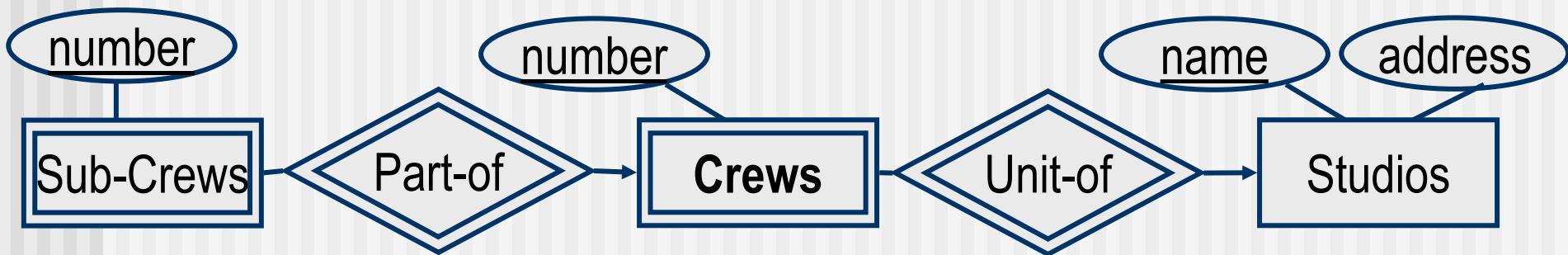
---

- A movie studio may have several film crews
- The crews in a studio may be designated as crew 1, crew 2, and so on
- All studios use the same designations for crews



# Sources of Weak Entity Sets

- Entity sets fall into a hierarchy
  - The entities in set **Crew** are **sub-units** of entities in set **Studio**
    - The “numbers” used for crew entities are not unique until we take into account the name of Studio with which she/he is associated (subordinate)
  - It is possible to have a chain of “weak” entity sets/relationships

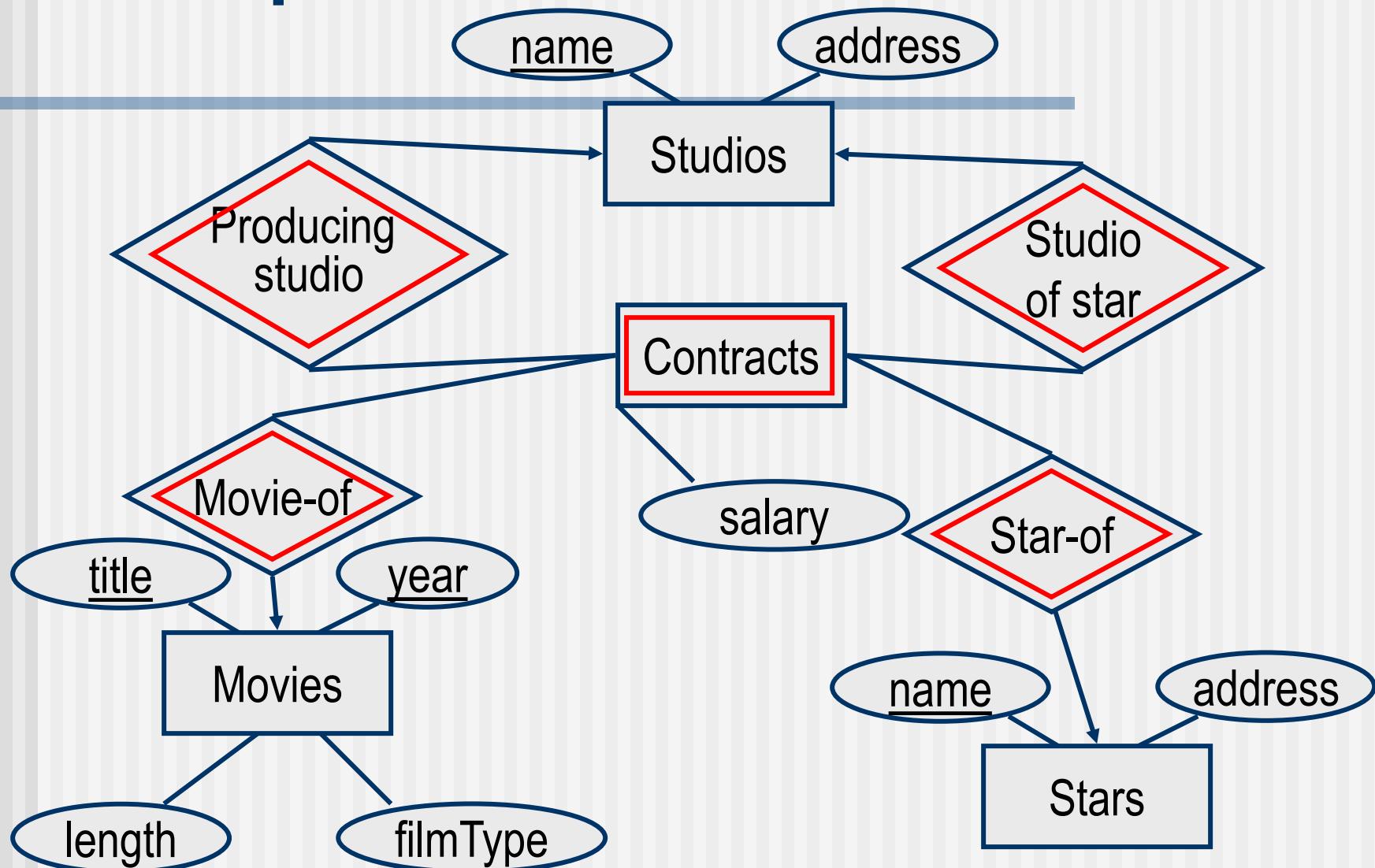


# Sources of Weak Entity Sets

---

- Connecting entity sets ([CES](#)):
  - CES's often have no attributes of their own
  - Their key is formed from the key attributes of the entity sets which they connect
- A connecting entity set is always weak

# Example



# Design Principles

---

- Design should
  - Reflect the “reality” we are trying to model – **faithful/realistic**
  - Avoid redundancy -- **minimal**
    - Redundant information takes space
    - Could cause inconsistency
  - Be as **simple** as possible
- Be careful when choosing between using attributes and using classes or entity sets. Remember that:
  - An attribute is **simpler** to implement than a class/entity set or a relationship
  - If something has more information associated with it than just its name, it should probably be modeled as an entity set or a class<sup>2</sup>

# A quick test!

---

- Which of the following statements is **NOT** correct?
  - A.** A data model is a collection of concepts for describing data and their relationships.
  - B.** A data model is a language for describing the data semantics and the constraints.
  - C.** A data model is a language for expressing queries and transactions over a database.
  - D.** A data model is an abstract representation of the structure of the data.

# **COMP353 Databases**

---

## **Relational Data Model**

# **Conceptual Database Design**

---

## **Relational Data Model: Introduction**

# The Relational Data Model

---

- **Relational database**
  - A set of **relations**
- **Relation**
  - A two-dimensional table in which data is arranged

# Example: Relation

| Attribute Names |      |        |          |
|-----------------|------|--------|----------|
| Title           | Year | Length | FilmType |
| Star Wars       | 1997 | 124    | color    |
| Mighty Ducks    | 1991 | 104    | color    |
| Wayne's World   | 1992 | 95     | color    |
| ...             | ...  | ...    | ...      |

Each row  
is a tuple

Components of the tuple

Attributes (type) are atomic (1NF)

# Relational Data Model

---

- Relation schema (or structure):  $R_i = \{A_1, \dots, A_m\}$ 
  - Relation name + a set of attribute names (+ attribute types)
- Relation instance:
  - The set of “current” tuples
- Database schema:
  - A set of relation schemas  $D = \{R_1, \dots, R_n\}$
- Database instance:
  - A collection of relation instances -- one for each relation in the database schema

# Relational Query Languages

---

- A major strength of the relational model is that it supports a powerful, high-level programming language – the Structured Query Language (SQL)

# **Logical Database Design**

---

**From E/R to Relational Model**

# Converting E/R to Relational Model

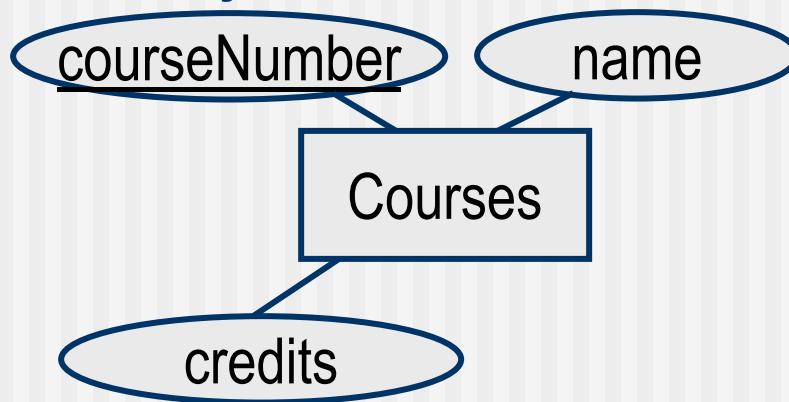
---

- Input:
  - An E/R diagram
- Output:
  - A relational database schema -- a collection of relations

# Converting Entity Sets to Relations

- For each entity set  $E$ , create a corresponding relation with the same attributes as in  $E$

**Entity Set:**



**Relation Instance:**

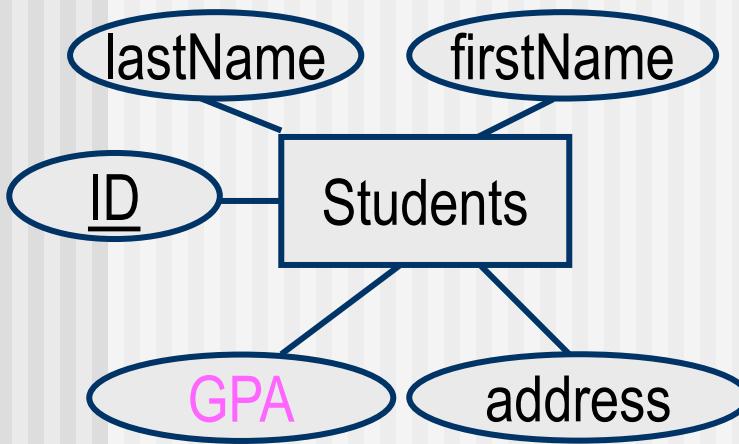
| courseNumber | name            | credits |
|--------------|-----------------|---------|
| Comp248      | C++ Prog.       | 3       |
| Comp352      | Data Structures | 4       |
| Comp353      | Databases       | 4       |

**Relation Schema:**

In theory, **Courses** = {courseNumber, name, credits}  
In practice, **Courses**(courseNumber, name, credits)

# Converting Entity Sets to Relations

Entity Set:



Relation Instance:

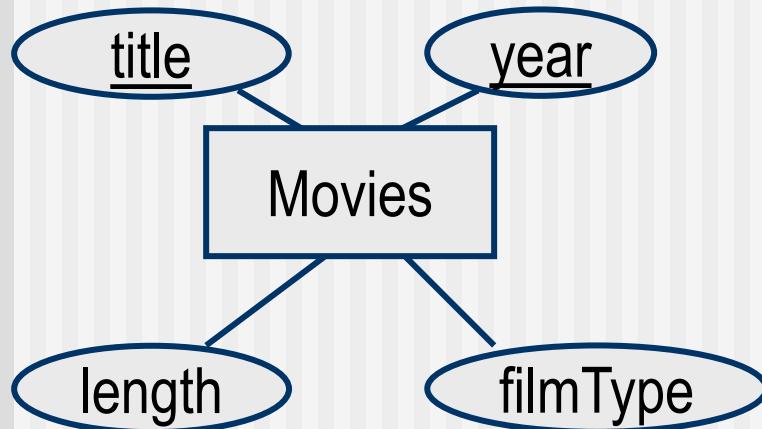
| ID  | firstName | lastName | GPA | address     |
|-----|-----------|----------|-----|-------------|
| 111 | Joe       | Smith    | 4.0 | 45 Pine av. |
| 222 | Sue       | Brown    | 3.1 | 71 Main St. |
| 333 | Ann       | John     | 3.7 | 39 Bay St.  |

Relation Schema:

**Students**(ID, firstName, lastName, **GPA**, address)

# Converting Entity Sets to Relations

Entity Set:



Relation Instance:

| title         | year | length | filmType |
|---------------|------|--------|----------|
| Star Wars     | 1997 | 124    | color    |
| Mighty Ducks  | 1991 | 104    | color    |
| Wayne's World | 1992 | 95     | b&w      |

Relation Schema:

**Movies**(title, year, length, filmType)

# Converting Entity Sets to Relations

---

**Entity Set:**



**Relation Instance:**

| name      | address   |
|-----------|-----------|
| Fox       | Hollywood |
| Disney    | Hollywood |
| Paramount | Hollywood |

**Relation Schema:**

**Studios(name, address)**

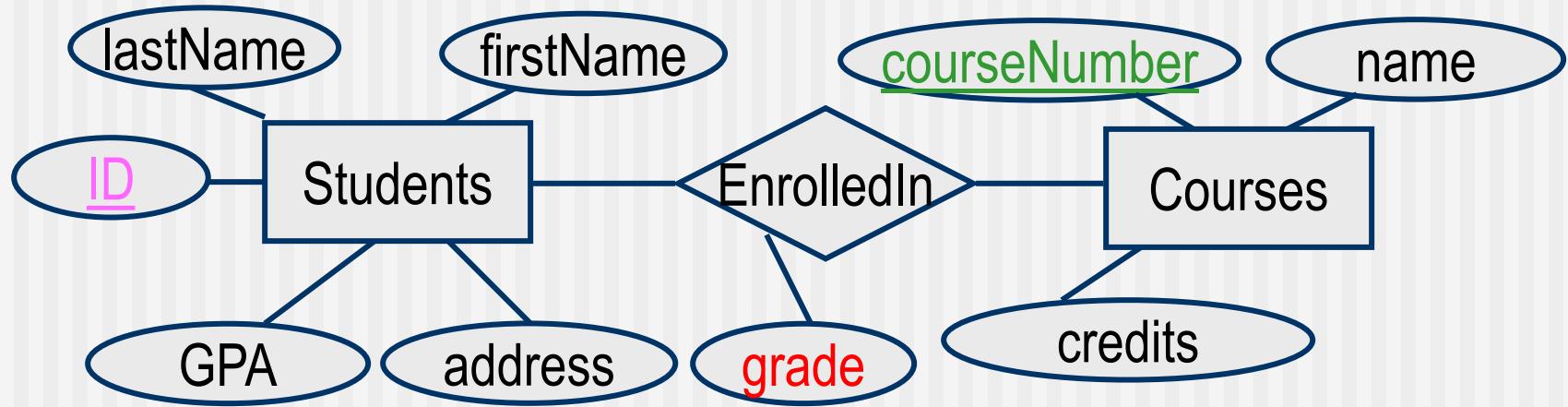
# Converting Relationships to Tables

---

- For each relationship set  $R$ , create the corresponding table (relation) and determine its attributes.
- The set of attributes of this table includes:
  - “Implicitly”: Key attribute(s) of the entity sets involved in the relationship  $R$
  - “Explicitly”: every attribute used “explicitly” in  $R$

# From Relationships to Tables

Relationship Set:



Relation Instance:

Relation Schema:

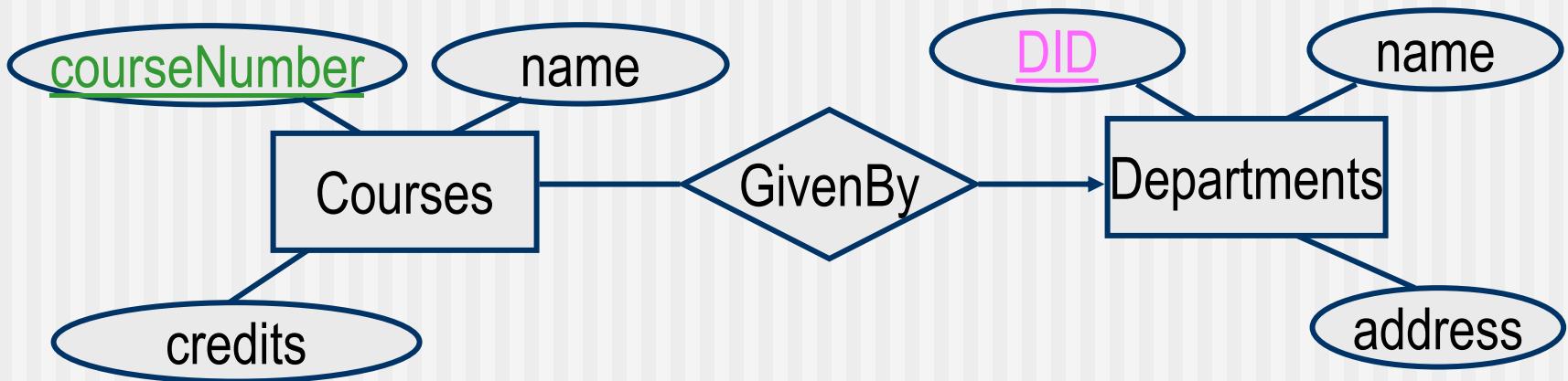
**EnrolledIn** (**ID**, **courseNumber**, **grade**)

What is the primary key of this relation?

| <b>ID</b> | <b>courseNumber</b> | <b>grade</b> |
|-----------|---------------------|--------------|
| 123       | Comp248             | A-           |
| 456       | Comp248             | B            |
| 123       | Comp353             | A+           |

# From Relationships to Tables

Relationship Set:



Relation Schema:

**GivenBy**(courseNumber, DID)

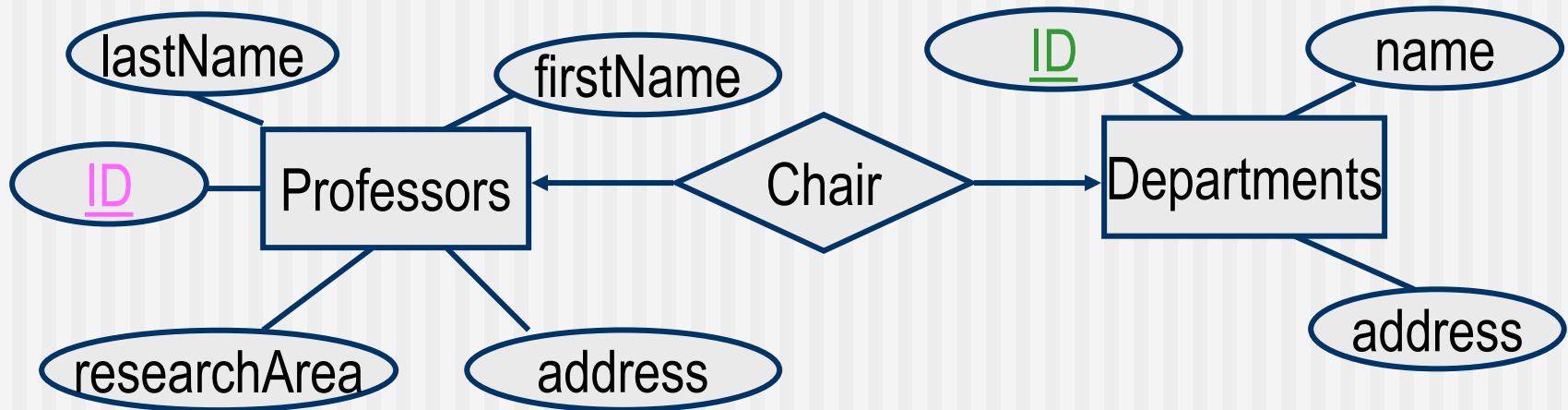
What is the primary key of this relation?

Relation Instance:

| courseNumber | DID |
|--------------|-----|
| Comp248      | 1   |
| Comp352      | 1   |
| Math207      | 9   |

# From Relationships to Tables

Relationship Set:



Relation Instance:

Relation Schema:

**Chair(PID, DID)**

What is the primary key of this relation?

| PID | DID |
|-----|-----|
| 234 | 1   |
| 451 | 2   |
| 778 | 9   |

# Identifying Key of Relationship R

---

- We are already familiar with the concept of key
- If **R** is a **binary** relationship between entity sets **E1** and **E2**, then the multiplicity of this relationship determines the key of **R**
  - If **R** is M-N, then the keys of **E1** and **E2** together are “part of” the key of **R**
  - If **R** is M-1 from **E1** to **E2**, then the key of **E1** is part of the key of **R**
  - If **R** is 1-1, then either **E1** or **E2** (but not both) is part of the key of **R**
- Do the above rules regarding the formation of keys apply to:
  - **Multi-way relationships?**
- How to determine keys for:
  - **Weak entity sets?**
  - **Entity sets and relationship sets in isa hierarchies?**

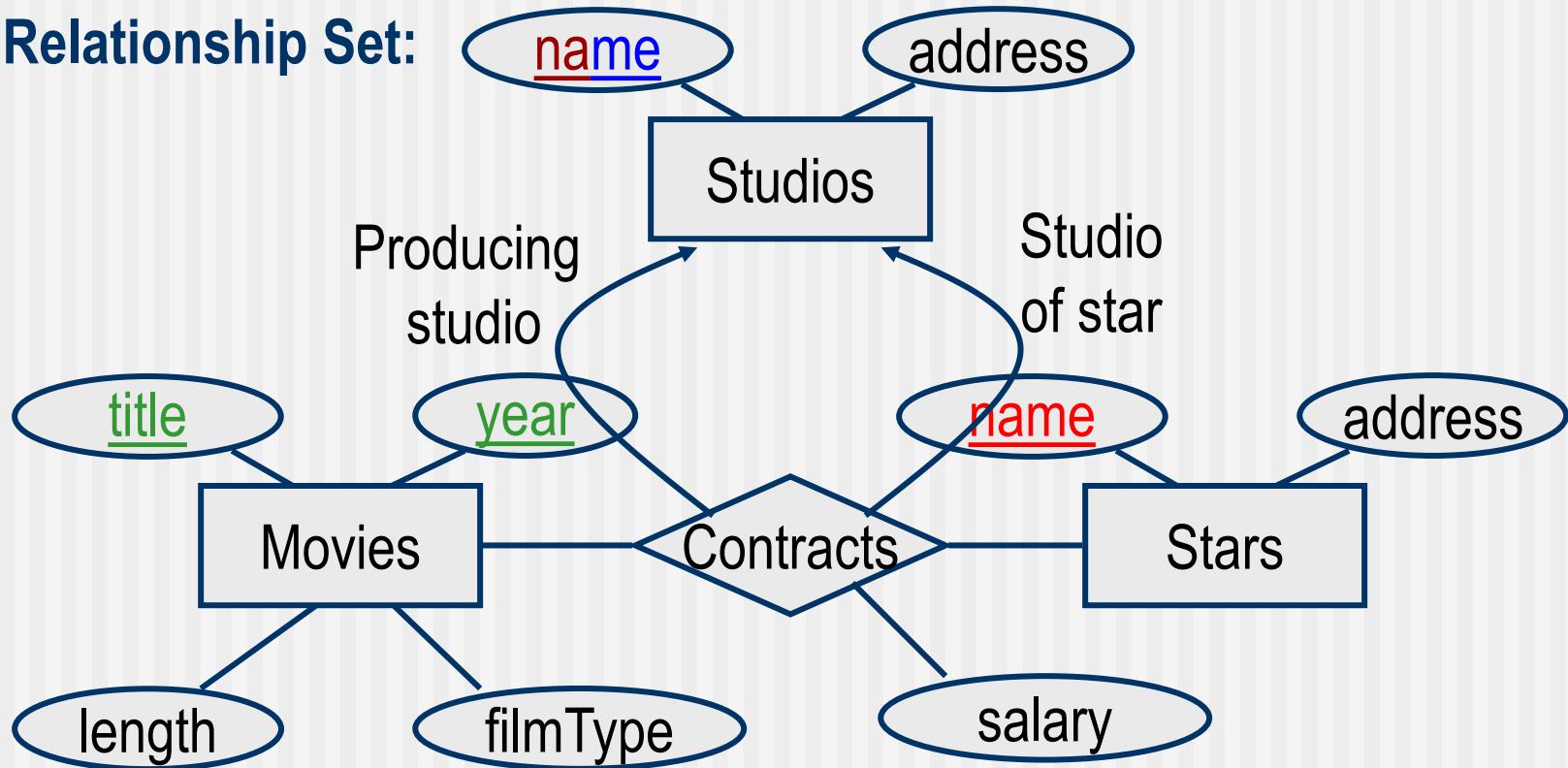
# Converting Relationships to Tables

---

- We should **rename** the attributes in the relations created when:
  - An entity set is involved in a relationship more than once
  - The **same attribute name** appears in the keys of different entity sets involved in the relationship (e.g., **ID** in previous example)
  - This is to avoid **ambiguity** in the schema and to be more clear in meanings

# Relationship Sets to Relations

Relationship Set:



Relation Schema:

What is the primary key for Contracts?

Contracts(starName, title, year, studioOfStar, producingStudio, salary )

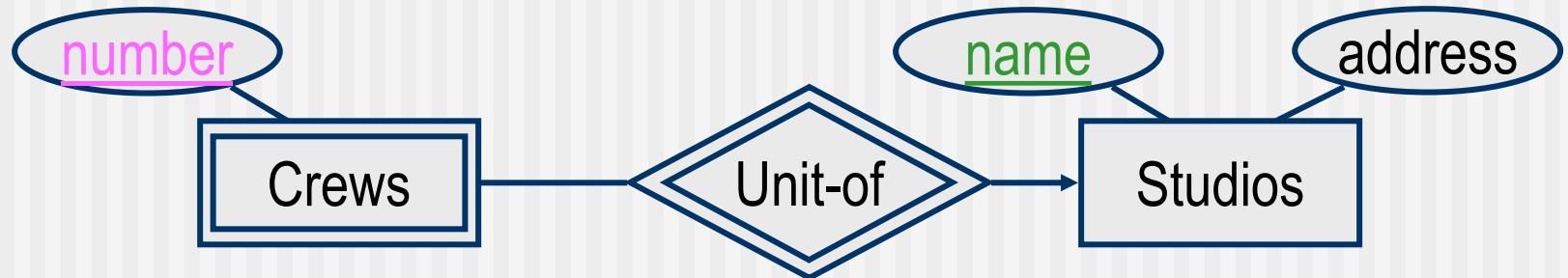
# Weak Entity Sets to Relations

---

- The relation/table  $W$  for the weak entity set  $W$ , must include all the attributes of  $W$  as well as the key attributes of the strong entity sets to which  $W$  is associated.
- Any relationship  $R$  to which the weak entity set  $W$  contributes, must include **all** the key attributes of  $W$ , i.e., the key attributes of every entity set that contributes to  $W$ 's key
- The weak relationships, from the weak entity set  $W$  to other entity sets that provide the key for  $W$ , need not be converted into a separate table, i.e., double diamonds connecting a weak entity set need not become a separate table.

# Weak Entity Sets to Relations

Weak Entity Set:



Relation Schema:

Crews (number, name)

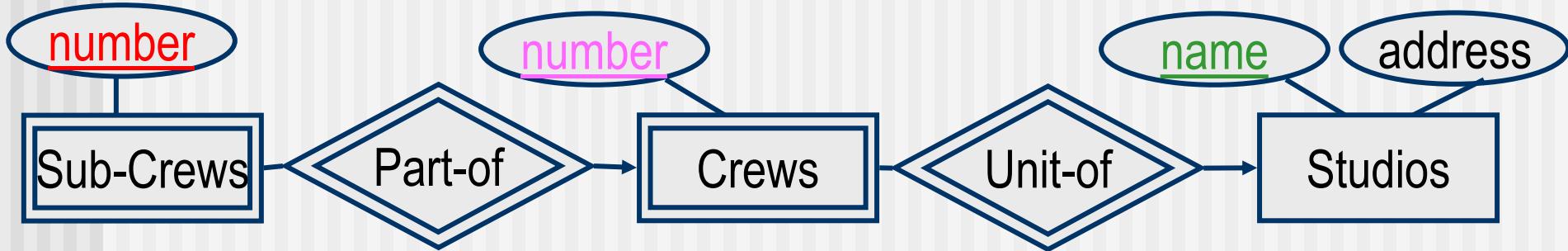
Unit-of (number, studioName, name)

Studios (name, address)

Do we need to keep the  
relation Unit-of?

# Weak Entity Sets to Relations

Weak Entity Set:

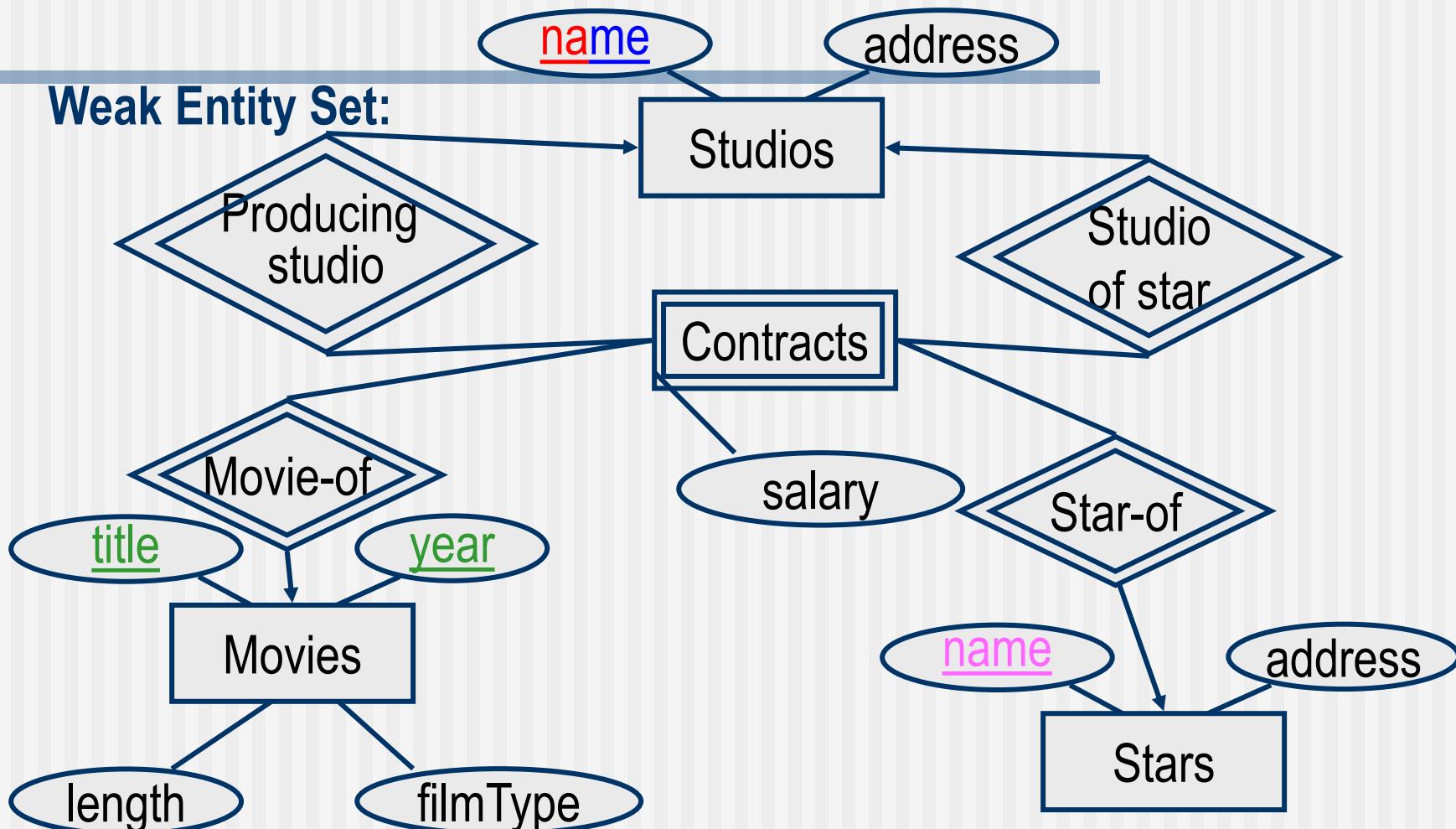


Relation Schemas:

**Sub-Crews** (number, crewNumber, name)

**Studios** (name, address)

# Weak Entity Sets to Relations



# Converting *isa*-Hierarchies to Relations

---

There are 3 approaches:

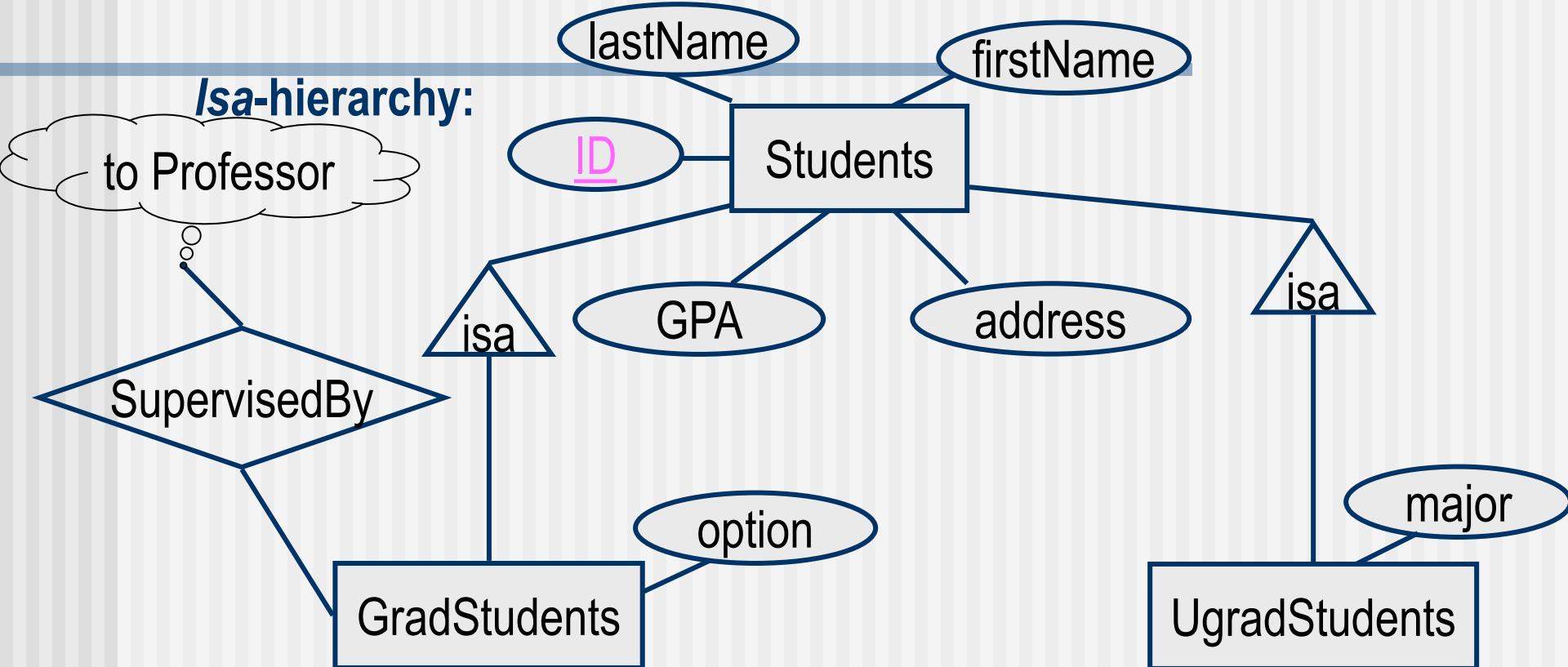
- Straight-E/R style method
  - In the E/R model, an entity (object) can be represented by entities that may belong to several entity sets, which are connected and related via *isa* hierarchies
  - The “connected” entities together represent the object and also determine the object’s properties (e.g., attributes and relationships)
- The object-oriented method
- The nulls method

# Converting *isa*-Hierarchy to Relations

---

- For each entity set **E**, create a relation (table) **e**, and give it attribute(s) **A**, whenever:
  - **A** belongs to **E**
  - **A** is the key attribute of the parent(s) relation
- No relation is created for the *isa*-relationship

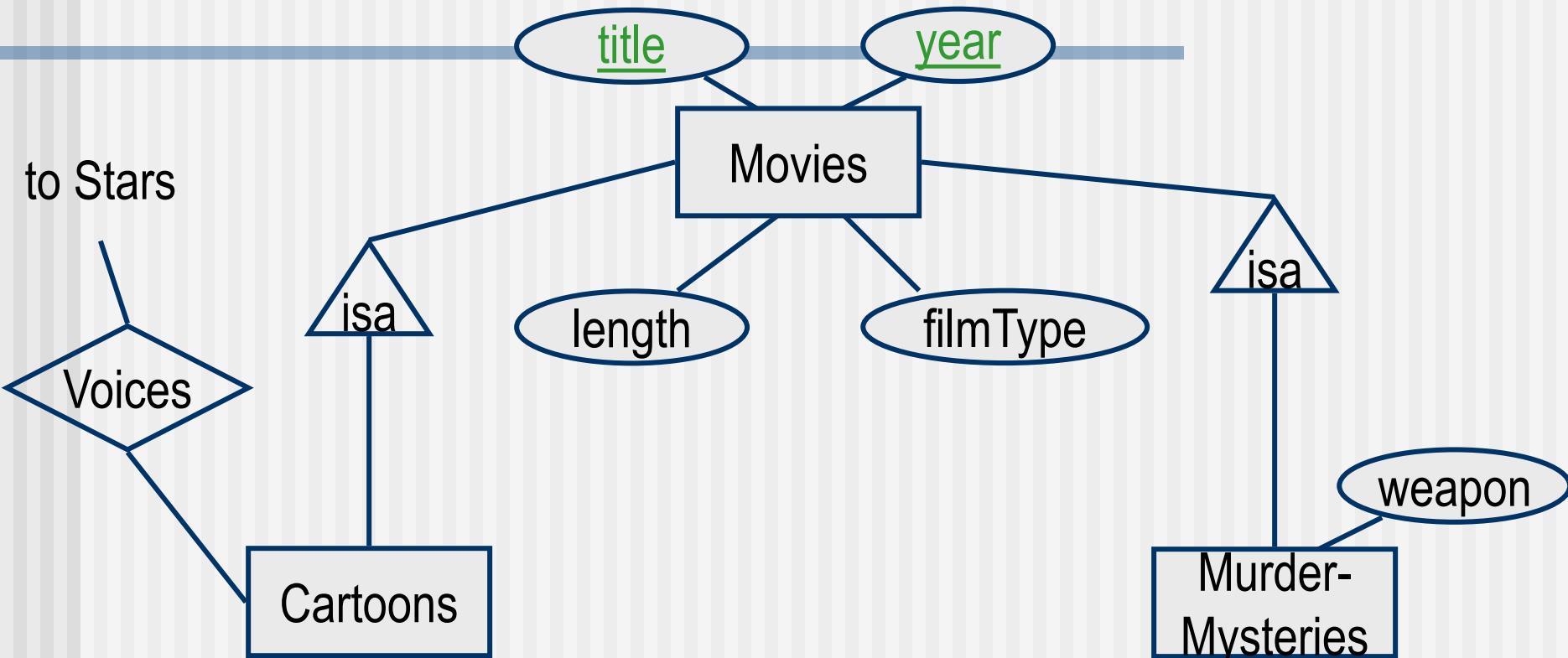
# Isa-Hierarchy to Relations



**Relation Schemas:**

- Students** (ID, lastName, firstName, GPA, address )
- GradStudents** (ID, option )
- UgradStudents** (ID, major )
- SupervisedBy** (StudentID, professorID)

# Isa-Hierarchy to Relations



**Relation Schemas:**

What about  
Cartoon-Murder-Mysteries?

**Movies** (title, year, length, film Type)

**Cartoons** (title, year) **← Do we really need this?**

**Murder-Mysteries** (title, year, weapon)

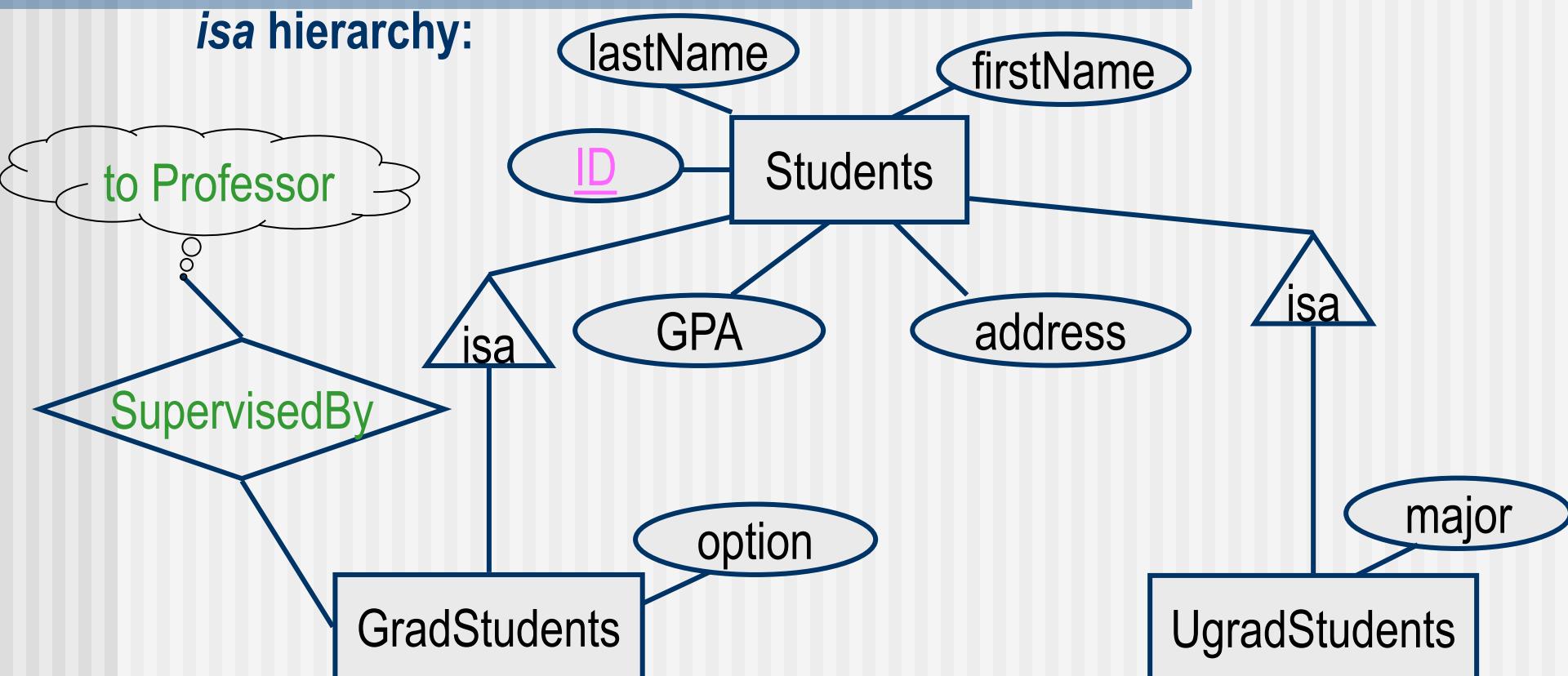
**Voices** (starName, title, year)

# The NULL Values Approach

---

- If we are **allowed** to use **NULL** as a value in tuples, we can handle a hierarchy of entity sets (classes) with a **single relation**
  - This relation has all the attributes belonging to any entity set (class) of the hierarchy.
  - An entity/object is represented by a single tuple that has NULL in each attribute that is not defined for that entity/object.

# Converting isa Hierarchy to Relations: The Null Approach



**Relation Schemas:**

`SupervisedBy` (`ID`, `professorID`)

`Student` (`ID`, `lastName`, `firstName`, `GPA`, `address`, `option`, `major`)

# NULL Approach

---

- The null approach: supports efficient query processing but is inefficient in space utilization. Why?
  - Answering queries: the nulls approach allows us to find, *in a single relation R*, every tuple/object from any set involved in the hierarchy
  - Allows us to find **all** the information about **an entity/object** in a single tuple in **R**
  - The down side is its *space utilization which is too costly for having* repeated and redundant information:
  - **Note:** Nulls are not allowed in the relational model theory, but practically, it is supported by commercial DBMS

# A quick test!

---

- Suppose R is a M-1 relationship from entity set E1={a1,a2} to E2={b1,b2}. Which of the following is NOT a ***valid instance*** of R?
  - $R = \{(a1, b1), (a1, b2)\}$ .
  - $R = \{(a1, b1)\}$ .
  - $R = \{(a2, b1)\}$ .
  - $R = \{(a1, b1), (a2, b1)\}$ .
  - $R = \{ \}$

# **COMP 353 Databases**

---

**Design Theory for Relational Databases**

**Functional Dependencies**

**Schema Refinement (Decomposition)**

**Normal Forms**

# Functional Dependencies (FDs)

---

- A **functional dependency** (FD) is a kind of **constraint**
- Suppose **R** is a relation schema and  $X, Y \subseteq R$ .
  - A FD on **R** is a statement of the form  $X \rightarrow Y$ , which asserts:  
“For every “**legal/valid**” instance  $r$  of **R**, and for all pairs of tuples  $t_1$  and  $t_2$  in  $r$ , if  $t_1$  and  $t_2$  **agree** on the values in  $X$ , then  $t_1$  and  $t_2$  **agree** also on the values in  $Y$ .”  
In symbols:  $\forall t_1, t_2 \in r: t_1[X] = t_2[X] \rightarrow t_1[Y] = t_2[Y]$ .
- We read  $X \rightarrow Y$  as:
  - $X$  (**functionally**) **determines**  $Y$  (or  $Y$  is determined by  $X$ )
- We say that the FD:  $X \rightarrow Y$  is **relevant** to **R** if  $XUY \subseteq R$ .

# Functional Dependencies

---

- Consider the relation schema:  
**Star** (name, SIN, street, city, postalCode, phone)
- Since we know the semantics of this relation from the design phase, we can answer the following question:
  - What are the functional dependencies on **Star**?
- Note that in general, FDs on a relation R may not be determined based on a given instance of R!

# Functional Dependencies

---

- Consider the relation:  
**Movie** (title, year, length, filmType)
- What are the FD's on the **Movie** relation?  
We use the semantics of this relation to answer.

# Keys

- The concept of FD generalizez the concept of key. How?
  - Let  $X \subseteq R$ . Then  $X$  is a key of  $R$  iff  $X \rightarrow R$
- $X$  is a (**candidate**) key of  $R$  (or a key, for short) if
  1.  $X \rightarrow R$ . That is, attributes in  $X$  functionally determine **all** the attributes of  $R$
  2. No proper subset of  $X$  is key, i.e., a candidate key must be **minimal**
- Is {title, year, filmType} a key for relation **Movie**?
- A set of attributes that contains a key is called a **superkey** (**that is, a superset of a key**)
  - Note that every key is a **superkey**, but not vice versa

# Functional Dependencies

---

- $X \rightarrow Y$  is called a **functional dependency** because, in principle, there is a function that takes a list of values, one for each attribute in  $X$ , and returns at most one value (i.e., a *unique* value or no value at all) for the attributes in  $Y$

# Functional Dependencies

---

- Consider the relation:  
**Movie** (title, year, length, filmType, studioName, starName)
- What are the functional dependencies?

$\{title, year\} \rightarrow length$

$\{title, year\} \rightarrow filmType$

$\{title, year\} \rightarrow studioName$

$\rightarrow \{title, year\} \rightarrow \{length, filmType, studioName\}$

Note:  $\{title, year\} \rightarrow starName$  does not hold

- What is the key of the **Movie** relation?

# Trivial FD's

---

- An FD  $X \rightarrow Y$  is said to be **trivial** if  $Y \subseteq X$ .
  - For example:  $\{\text{title}, \text{year}\} \rightarrow \text{title}$  is a trivial FD
- Otherwise, the FD is called **nontrivial**
  - For example:  $\{\text{title}, \text{year}\} \rightarrow \text{length}$  is a nontrivial FD

# Functional Dependencies

---

- Why are we interested in functional dependencies?

# Redundancy Problem

---

- Redundancy – a “piece” of information is unnecessarily repeated in different tuples in a relation
- Recall that ***redundancy*** is the main source of problems:
  - Storage waste
    - Some information stored repeatedly
  - Update anomalies
    - If a copy of such information is updated, an inconsistency may arise unless all its copies are updated
  - Insertion anomalies
    - Unless we allow nulls, it may not be possible to store some information unless we have all the information to store
  - Deletion anomalies
    - Deleting some information may result in losing some other information (which we don't want to lose)

# Is this a good design for relation R?

---

| Name | SSN        | Phone          |
|------|------------|----------------|
| Fred | 123-321-99 | (201) 555-1234 |
| Fred | 123-321-99 | (201) 572-4312 |
| Joe  | 909-438-44 | (908) 464-0028 |
| Mary | 938-401-54 | (201) 555-1234 |

The only FD on R is: **SSN → Name**

Therefore, the only key of R is: **{SSN, Phone}**

# What about this design, replacing R with R1 and R2?

---

| R1 | SSN        | Name |
|----|------------|------|
|    | 123-321-99 | Fred |
|    | 909-438-44 | Joe  |
|    | 938-401-54 | Mary |

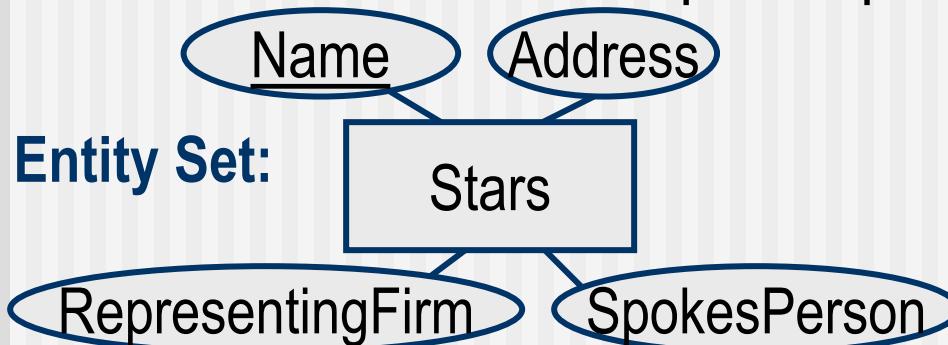
| R2 | SSN        | Phone           |
|----|------------|-----------------|
|    | 123-321-99 | (201) 555-1234  |
|    | 123-321-99 | (201) 572- 4312 |
|    | 909-438-44 | (908) 464-0028  |
|    | 938-401-54 | (201) 555-1234  |

$$D = \{R1(\underline{SSN}, \underline{\text{Name}}), \\ R2(\underline{SSN}, \underline{\text{Phone}})\}$$

FD's on R1 and R2?

# Another Example

- Suppose each star has a representing *firm* and each firm has one spokes person)



Relation Schema:  
**Star** (Name, Address,  
RepresentingFirm, SpokesPerson)

RepresentingFirm  
→  
SpokesPerson

Relation Instance:

| Name          | Address      | RepresentingFirm | SpokesPerson |
|---------------|--------------|------------------|--------------|
| Carrie Fisher | 123 Maple    | Star One         | Joe Smith    |
| Harrison Ford | 789 Palm rd. | Star One         | Joe Smith    |
| Mark Hamill   | 456 Oak rd.  | Movies & Co      | Mary Johns   |

# Redundancy Problem

---

*What is the role of FDs in detecting redundancy?*

- Consider the relation scheme  $R(A, B, C)$ 
  - Suppose no (nontrivial) FD holds on  $R$ 
    - There is no redundancy in any instance  $r$  of  $R$ .
  - Now suppose FD:  $A \rightarrow B$  holds on  $R$ 
    - If several tuples have the same  $A$  value  $\rightarrow$  they must all have the same  $B$  value; otherwise this FD is violated
- Presence of some FDs in a relation suggests possibility of redundancy

# Implications of FDs and Reasoning

---

- Consider relation  $R(A, B, C)$  with the set of FDs:  
 $F = \{A \rightarrow B, B \rightarrow C\}$
- We can deduce from  $F$  that  $A \rightarrow C$  also holds on  $R$ .  
**How?** Apply the definition...
- To detect possible data redundancy, is it **necessary** to consider “all” the FDs (implicit and explicit)?
  - As shown above, there might be some additional hidden (nontrivial) FDs “**implied**” by a given set of FD’s

# Implications of FDs

---

- Defn: If a relation instance  $r$  satisfies every FD in a given set  $F$  of FD's, then we say that  $r$  satisfies  $F$ .
  - In this case, we also say that  $r$  is a *legal/valid instance*.
- Given  $\langle R, F \rangle$ , we say that  $F$  **implies** a FD  $X \rightarrow Y$ , if *every* instance  $r$  of  $R$  that satisfies  $F$  also satisfies  $X \rightarrow Y$ .

**Formally**, we express this as:  $F \models X \rightarrow Y$ .

We may also say that  $X \rightarrow Y$  follows from  $F$ .

- To show  $F \not\models X \rightarrow Y$ , we may give a counter-example, i.e., an instance  $r$  of  $R$  that satisfies  $F$  but not  $X \rightarrow Y$ .

# FDs Implication (Cont'd)

---

- Consider  $R(A_1, A_2, A_3, A_4, A_5)$  with FDs:

$$F = \{ A_1 \rightarrow A_2, A_2 \rightarrow A_3, A_2A_3 \rightarrow A_4, A_2A_3A_4 \rightarrow A_5 \}$$

Prove that  $F \not\Rightarrow A_5 \rightarrow A_1$

Solution method: Provide a counter-example; give a relation instance  $r$  of  $R$  that satisfies every FD in  $F$  but not  $A_5 \rightarrow A_1$

A desired instance  $r$  of  $R$ :

|     | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
|-----|-------|-------|-------|-------|-------|
| t1: | 0     | 1     | 1     | 1     | 1     |
| t2: | 1     | 1     | 1     | 1     | 1     |

# Closure of a set F of FDs

---

- Defn: The **closure of F**, denoted by  $F^+$ , is the set of every FD:  $X \rightarrow Y$  that is implied by F.
- How can we determine  $F^+$ ?
  - Clearly,  $F^+$  includes F and possibly some more FDs
  - To answer the question we need to *reason* about FDs

# Equivalence of two sets of FD's

---

- Let  $R$  be a relation schema, and  $S, T$  be sets of FDs on  $R$ .
- Defn: we say  $S$  **covers**  $T$  ( $S \sqsupseteq T$ ) if for every instance  $r$  of  $R$ , whenever  $r$  satisfies (every FD in)  $S$ ,  $r$  also satisfies  $T$ .
- Defn:  $T$  and  $S$  are **equivalent** ( $S \equiv T$ ) iff  $S \sqsupseteq T$  and  $T \sqsupseteq S$ .
- Note:  $F$  and  $F^+$  are equivalent.

Example: Suppose  $R = \{A, B, C\}$ , and

$$S = \{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$$

$$T = \{A \rightarrow B, B \rightarrow C\}$$

We can show that  $S \equiv T$ .

# Armstrong's Axioms [1974]

---

- $R$  is a relation schema, and  $X, Y, Z$  are subsets of  $R$ .
- **Reflexivity**
  - If  $Y \subseteq X$ , then  $X \rightarrow Y$  (trivial FDs)
- **Augmentation**
  - If  $X \rightarrow Y$ , then  $XZ \rightarrow YZ$ , for every  $Z$
- **Transitivity**
  - If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$
- These are **sound** and **complete** inference rules for FDs

# Additional rules / axioms

---

Other useful rules that follow from Armstrong Axioms:

Suppose  $X$ ,  $Y$ ,  $Z$ , and  $W$  are sets of attributes.

## ■ Union (Combining) Rule

- If  $X \rightarrow Y$  and  $X \rightarrow Z$ , then  $X \rightarrow YZ$

## ■ Decomposition (Splitting) Rule

- If  $X \rightarrow YZ$ , then  $X \rightarrow Y$  and  $X \rightarrow Z$

## ■ Pseudotransitivity Rule

- If  $X \rightarrow Y$  and  $WY \rightarrow Z$ , then  $XW \rightarrow Z$

# Example – Discovering hidden FD's

---

- Consider  $R = \{A, B, C, G, H, I\}$  with the FDs:  
 $F = \{ A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H \}$
- Using Armstrong's rules, we can derive more FDs
  - Since  $A \rightarrow B$  and  $B \rightarrow H$ , then  $A \rightarrow H$ , by **transitivity**
  - Since  $CG \rightarrow H$  and  $CG \rightarrow I$ , then  $CG \rightarrow HI$ , by **union**
  - Since  $A \rightarrow C$  then  $AG \rightarrow CG$ , by **augmentation**  
Now, since  $AG \rightarrow CG$  and  $CG \rightarrow I$ , then  $AG \rightarrow I$ , by **transitivity** (and in a similar way, we get  $F \vdash AG \rightarrow H$ )
  - Many trivial dependencies can be derived(!) by **augmentation**

# Implication Problem

---

- Given a set  $F$  of FDs, does  $X \rightarrow Y$  follow from  $F$ ?
- In other words: is  $X \rightarrow Y$  in the closure of  $F$ ?

(In symbols, does  $F \models X \rightarrow Y$  hold, or is

$X \rightarrow Y \in F^+$  true?)

- How to answer this question?
  - Compute the closure of  $F$  & check if it includes  $X \rightarrow Y$
- What is the problem with this approach?
  - Computing  $F^+$  is *expensive*! Is there a better solution?

# Closure of a Set of Attributes

---

- Given  $\langle R, F \rangle$ ; Let  $X \subseteq R$ .

The closure of  $X$  under  $F$  is the set of all attributes  $Y$  in  $R$  that are determined by  $X$ . This yields  $X \rightarrow Y$ , i.e., every valid instance of  $R$  (that satisfies  $F$ ) also satisfies  $X \rightarrow Y$

- We denote the closure of a set of attributes  $X$  under  $F$  by  $X^+_F$ 
  - When  $F$  is known , we simply write  $X^+$  (and omit  $F$ )
  - Closure of  $\{A_1, A_2, \dots, A_n\}$  is denoted  $\{A_1, A_2, \dots, A_n\}^+$
- Note that  $X \subseteq X^+$ , for any set  $X$  of attributes (because  $X \rightarrow X$ )

# Computing the Closure of Attributes

---

- Given a set  $F$  of FD's and a set  $X$  of attributes, how to compute the closure of  $X$  w.r.t.  $F$ ?
  - Starting with set  $X^+ = X$ , we **repeatedly** expand  $X^+$  by adding the RHS  $Z$  for every FD:  $W \rightarrow Z$  in  $F$ , if the LHD  $W$  is already in  $X^+$ .
  - This process terminates when  $X^+$  could not be expanded further.
  - This process is expressed as an algorithm in the next slide.

# An Algorithm to Compute $X^+$ under $F$

---

$X^+ \leftarrow X$  (initialization step)

*repeat*

*for each FD  $W \rightarrow Z$  in  $F$  do:*

*if  $W \subseteq X^+$  then*

$X^+ \leftarrow X^+ \cup Z$  // add  $Z$  to the result

*until  $X^+$  does not change*

**Complexity?** In the worst case, how many times the “repeat” statement may be executed?

# Examples

---

- Consider a relation schema  $R = \{ A, B, C, D, E, H \}$  with the FD's  $F = \{ AB \rightarrow C, BC \rightarrow AD, D \rightarrow E, CH \rightarrow B \}$
- Suppose  $X=\{A,B\}$ . Compute  $X^+$
- Execution result at each iteration:
  - Initially,  $X^+ = \{A, B\}$
  - Using  $AB \rightarrow C$ , we get  $X^+ = \{A, B, C\}$
  - Using  $BC \rightarrow AD$ , we get  $X^+ = \{A, B, C, D\}$
  - Using  $D \rightarrow E$ , we get  $X^+ = \{A, B, C, D, E\}$
  - No more change to  $X^+$  is possible.  
    →  $X^+ = \{A, B\}^+ = \{A, B, C, D, E\}$
- Does the order in which FD's appear in  $F$  affects the computation?

# Implication Problem Revisited

---

- Given a set of FD's  $F$ , does an FD:  $X \rightarrow Y$  follow from  $F$ ?
  - That is, is FD  $X \rightarrow Y$  in  $F^+$ ?
- To answer this, we can compute  $X^+$  under  $F$ , and check if  $Y$  is in  $X^+$  or not
  - If yes, then the answer is positive! ( $F \models X \rightarrow Y$  ☺)
  - Otherwise, it is negative ( $F \not\models X \rightarrow Y$  ☹)

# Example

---

- Consider  $\langle R, F \rangle$  where  $R = \{ A, B, C, D, E, H \}$  and  
 $F = \{ AB \rightarrow C, BC \rightarrow AD, D \rightarrow E, CH \rightarrow B \}$
- Does  $AB \rightarrow D$  follow from  $F$ ?
- Two steps:
  1. Compute  $\{A, B\}^+ = \{A, B, C, D, E\}$
  2. Check if  $D \in \{A, B\}^+$
- So, here we conclude that  $AB \rightarrow D$  is implied by  $F$

# Example

---

- Consider a relation schema  $R = \{ A, B, C, D, E, H \}$  with FDs:  $F = \{ AB \rightarrow C, BC \rightarrow AD, D \rightarrow E, CH \rightarrow B \}$
- True/False: Does  $D \rightarrow A$  follow from  $F$ ?
- Two steps:
  1. Compute  $X^+ = \{D\}^+ = \{D, E\}$
  2. Check if  $A \in X^+$
- Since  $A \notin \{D, E\}$ , the answer is NO, i.e.,  $F \not\models D \rightarrow A$

# Closures and Keys

---

- Consider a case where  $X^+$  includes **all** the attributes of a relation  $R$ 
  - Clearly,  $X$  is a (super) key of  $R$
  - To check if  $X$  is a candidate key of  $R$ , we should check 2 things:
    1. If  $X^+$  is a superkey  $R$ , i.e., when  $X^+ = R$ , and
    2. If no proper subset of  $X$  is a key, i.e.,  $\forall A \in X: (X - \{A\})^+ \neq R$
- To find the keys of a relation, we can use the algorithm on slide 26
- This would be exponential in the number of attributes! Can do better?
- Knowledge about keys is essential to understand “Normal forms.”

# **COMP353 Databases**

---

**Schema Refinement:**  
**Minimal Bases**  
**(Canonical Covers)**

# Minimal Basis (Canonical Cover)

---

- Recall that the number of iterations to compute the closure of a set of attributes depends on the number of attributes
  - The complexity of some other algorithms which we will study (eg, decomposition algorithms) depend on the number of FD's
- To ease the situation, can we "minimize"  $F$ ?

# Covers/bases

---

- Note that FD's on a relation may be represented in different but equivalent ways.
- Recall that, given two sets of FD's  $F$  and  $G$  on  $R$ , we say that:  
“ $G$  follows from  $F$  ( $F \vDash G$ )”, provided for every instance  $r$  of  $R$ , if  $r$  satisfies  $F$ , then  $r$  satisfies  $G$ . In this case, we may also say:  
“ $F$  implies  $G$ ”, or “ $F$  covers  $G$ ”, or “ $G$  is implied by  $F$ ”.
- If  $F \vDash G$  and  $G \vDash F$  both hold, then we say that  $G$  and  $F$  are equivalent and denote this by  $F \equiv G$ . In this case, we may also say that  $F$  and  $G$  are covers for each other.
- Note that  $F \equiv G$  iff  $F^+ \equiv G^+$

# Canonical Cover (minimal basis)

---

- Let  $F$  be a set of FD's. A **canonical cover** of  $F$  is a set  $G$  of FD's that satisfies the following conditions:
  1.  $G$  is **equivalent** to  $F$ , that is,  $G \equiv F$
  2. Every FD in  $G$  has a single attribute on the right hand side.
  3.  $G$  is **minimal**, that is, if we obtain a set  $H$  of FD's from  $G$  by deleting some FD's in  $G$  or by reducing the left hand side of some FD's, then  $H$  won't be equivalent to  $F$  (that is,  $H \not\equiv F$ )

# Canonical Cover

---

- A canonical cover **G** is **minimal** in two respects:
1. Every FD in **G** is “**required**” in order for **G** to be equivalent to **F**
  2. Every FD in **G** is as “**small**” as possible, that is,
    - each attribute on the left hand side **X** is necessary.
    - Recall: the RHS of every FD in **G** is a single attribute

# Computing Canonical Cover

---

Given a set  $F$  of FD's, how to compute  $a$  canonical cover  $G$  of  $F$ ?

- **Step 1:** Put the FD's in the **simple** form (i.e., one attribute on the RHS)
  - Initialize  $G := F$
  - Replace each FD  $X \rightarrow A_1A_2\dots A_k$  in  $G$  with  $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_k$
- **Step 2: Minimize** the left hand side  $X$  of every FD
  - E.g., if  $AB \rightarrow C$  is in  $G$ , check if  $A$  or  $B$  on the LHS is redundant , i.e.,  
$$(G - \{AB \rightarrow C\} \cup \{A \rightarrow C\})^+ \equiv F^+ \text{ or}$$
$$(G - \{AB \rightarrow C\} \cup \{B \rightarrow C\})^+ \equiv F^+ ?$$
- **Step 3: Delete** redundant FD's, if any
  - For each FD  $X \rightarrow A$  in  $G$ , check if  $(G - \{X \rightarrow A\})^+ \equiv F^+ ?$

# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- Step 1 – put FD's in the simple form
  - All present FD's are simple

→  $G = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$

# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- Step 2 – Check every FD to see if it is left reduced
  - For every FD  $X \rightarrow A$  in  $G$ , check if the closure of a subset of  $X$  determines  $A$ . If so, remove the redundant attribute(s) from  $X$

# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- $G = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$ 
  - $A \rightarrow B$ 
    - obviously OK (no left redundancy)
  - $DE \rightarrow A$ 
    - $D^+ = D$
    - $E^+ = E$
    - OK (no left redundancy)

# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- $G = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$ 
  - $BC \rightarrow E$ 
    - $B^+ = B$
    - $C^+ = C$
  - OK (no left redundancy)

# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- $G = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, \textcolor{blue}{AC \rightarrow E}, BCD \rightarrow A, AED \rightarrow B \}$ 
  - $AC \rightarrow E$ 
    - $A^+ = AB$
    - $C^+ = C$
  - OK (no left redundancy)

# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- $G = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, \textcolor{blue}{BCD \rightarrow A}, AED \rightarrow B \}$ 
  - $BCD \rightarrow A$ 
    - $B^+ = B$
    - $C^+ = C$
    - $D^+ = D$
    - $\{BC\}^+ = BCE$
    - $\{CD\}^+ = CD$
    - $\{BD\}^+ = BD$
  - OK (no left redundancy)

# Computing Canonical Cover

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- $G = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, \textcolor{blue}{AED \rightarrow B} \}$ 
  - $AED \rightarrow B$ 
    - $A^+ = AB$
    - $E \& D$  are redundant  
→ we can remove them from  $\textcolor{blue}{AED \rightarrow B}$
- $G = \{ \cancel{A \rightarrow B}, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, A \rightarrow B \}$   
→  $G = \{ DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, A \rightarrow B \}$

# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- Step 3 – Find and remove redundant FDs
  - For every FD  $X \rightarrow A$  in  $G$ 
    - Remove  $X \rightarrow A$  from  $G$ ; call the result  $G'$
    - Compute  $X^+$  under  $G'$
    - If  $A \in X^+$ , then  $X \rightarrow A$  is redundant and hence we remove the FD  $X \rightarrow A$  from  $G$  (that is, we rename  $G'$  to  $G$ )

# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- $G = \{ DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, A \rightarrow B \}$ 
  - Remove  $DE \rightarrow A$  from  $G$ 
    - $G' = \{ BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, A \rightarrow B \}$
    - Compute  $DE^+$  under  $G'$ 
      - $\{DE\}^+ = DE$  (computed under  $G'$ )
    - Since  $A \notin DE$ , the FD  $DE \rightarrow A$  is not redundant
    - $G = \{ DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, A \rightarrow B \}$

# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- $G = \{ DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, A \rightarrow B \}$ 
  - Remove  $BC \rightarrow E$  from  $G$ 
    - $G' = \{ DE \rightarrow A, AC \rightarrow E, BCD \rightarrow A, A \rightarrow B \}$
    - Compute  $BC^+$  under  $G'$ 
      - $\{BC\}^+ = BC$ 
        - $\rightarrow BC \rightarrow E$  is not redundant
    - $G = \{ DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, A \rightarrow B \}$

# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- $G = \{ DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, A \rightarrow B \}$ 
  - Remove  $AC \rightarrow E$  from  $G$ 
    - $G' = \{ DE \rightarrow A, BC \rightarrow E, BCD \rightarrow A, A \rightarrow B \}$
    - Compute  $\{AC\}^+$  under  $G'$ 
      - $\{AC\}^+ = ACBE$
  - Since  $E \in ACBE$ ,  $AC \rightarrow E$  is redundant  $\rightarrow$  remove it from  $G$
  - $G = \{ DE \rightarrow A, BC \rightarrow E, BCD \rightarrow A, A \rightarrow B \}$

# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, H \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- $G = \{ DE \rightarrow A, BC \rightarrow E, \textcolor{blue}{BCD} \rightarrow A, A \rightarrow B \}$ 
  - Remove  $BCD \rightarrow A$  from  $G$ 
    - $G' = \{ DE \rightarrow A, BC \rightarrow E, A \rightarrow B \}$
  - Compute  $BCD^+$  under  $G'$ 
    - $\{BCD\}^+ = BCDEA$
  - This FD is redundant  $\rightarrow$  remove it from  $G$ 
    - $G = \{ DE \rightarrow A, BC \rightarrow E, A \rightarrow B \}$

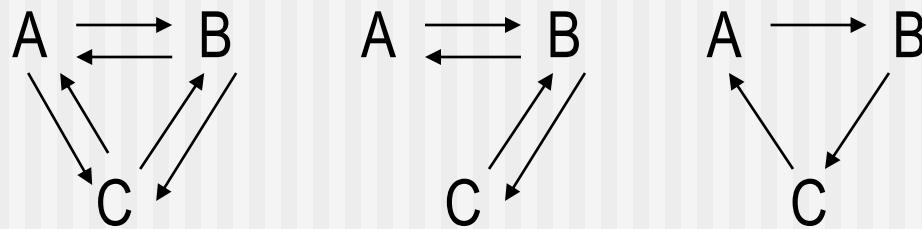
# Computing Canonical Cover

---

- $R = \{ A, B, C, D, E, F \}$
- $F = \{ A \rightarrow B, DE \rightarrow A, BC \rightarrow E, AC \rightarrow E, BCD \rightarrow A, AED \rightarrow B \}$
- $G = \{ DE \rightarrow A, BC \rightarrow E, A \rightarrow B \}$ 
  - Remove  $A \rightarrow B$  from  $G$ 
    - $G' = \{ DE \rightarrow A, BC \rightarrow E \}$
    - Compute  $A^+$  under  $G'$ 
      - $A^+ = A$
    - This FD is not redundant (Another reason why need  $A \rightarrow B$  ?)
    - $G = \{ DE \rightarrow A, BC \rightarrow E, A \rightarrow B \}$ 
      - $G$  is a minimal cover for  $F$

# Several Canonical Covers Possible?

- Relation  $R=\{A,B,C\}$  with  $F = \{A \rightarrow B, A \rightarrow C, B \rightarrow A, B \rightarrow C, C \rightarrow B, C \rightarrow A\}$
- Several canonical covers exist
  - $G = \{A \rightarrow B, B \rightarrow A, B \rightarrow C, C \rightarrow B\}$
  - $G = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$



Can you find more ?

# Computing a Canonical Cover

---

This example shows the order of steps 2 and 3 is important!

$R = \{A, B, C, D\}$  with  $F = \{ ABC \rightarrow D, AB \rightarrow C, D \rightarrow C \}$

1. (step 3; step 2): Doing step 3 first, no FD is redundant (Why?)

In step 2,  $ABC \rightarrow D$  is left reduced to  $AB \rightarrow D$ . No more changes.

We thus obtain  $G = \{ AB \rightarrow D, AB \rightarrow C, D \rightarrow C \}$  which is equivalent to  $F$  but is **not** minimal! (The red FD is **redundant**!).

2. (step 2; step 3): Following our algorithm, in step 2 we get  $G$  above. In step 3, we remove the redundant FD in  $G$ .

This yields  $\{ AB \rightarrow D, D \rightarrow C \}$  which is equivalent to  $F$  and minimal.

# How to Deal with Redundancy?

---

## Relation Schema:

**Star** (name, address, representingFirm, spokesPerson)

$F = \{ \text{name} \rightarrow \text{address}, \text{representingFirm}, \text{spokePerson},$   
 $\text{representingFirm} \rightarrow \text{spokesPerson} \}$

## Relation Instance:

| Name          | Address      | RepresentingFirm | SpokesPerson |
|---------------|--------------|------------------|--------------|
| Carrie Fisher | 123 Maple    | Star One         | Joe Smith    |
| Harrison Ford | 789 Palm dr. | Star One         | Joe Smith    |
| Mark Hamill   | 456 Oak rd.  | Movies & Co      | Mary Johns   |

- We can **decompose** this relation into two smaller relations

# How to Deal with Redundancy?

---

Given the relation schema below:

**Star** (name, address, **representingFirm**, **spokesperson**) with

$F = \{name \rightarrow address, representingFirm, spokePerson$   
 $representingFirm \rightarrow spokesPerson\}$

Decompose Star into the following 2 relations:

**Star** (name, address, **representingFirm**)

with  $F1=\{ name \rightarrow address, representingFirm \}$

and

**Firm** (representingFirm, **spokesPerson**)

with  $F2= \{ representingFirm \rightarrow spokesPerson \}$

# How to Deal with Redundancy?

Instance of Star before decomposition:

| Name          | Address      | RepresentingFirm | Spokesperson |
|---------------|--------------|------------------|--------------|
| Carrie Fisher | 123 Maple    | Star One         | Joe Smith    |
| Harrison Ford | 789 Palm dr. | Star One         | Joe Smith    |
| Mark Hamill   | 456 Oak rd.  | Movies & Co      | Mary Johns   |

The instance after the decomposition:

| Name          | Address      | RepresentingFirm |
|---------------|--------------|------------------|
| Carrie Fisher | 123 Maple    | Star One         |
| Harrison Ford | 789 Palm dr. | Star One         |
| Mark Hamill   | 456 Oak rd.  | Movies & Co      |

| RepresentingFirm | Spokesperson |
|------------------|--------------|
| Star One         | Joe Smith    |
| Movies & Co      | Mary Johns   |

# Decomposition

---

- A *decomposition* of a relation schema  $\mathbf{R}$  is obtained by splitting  $\mathbf{R}$  into two or more relations, denoted as  $\underline{\mathbf{R}} = \{\mathbf{R}_1, \dots, \mathbf{R}_m\}$ . Formally,  $\underline{\mathbf{R}}$  is a decomposition of  $\mathbf{R}$  if the following two conditions hold:
  1. No attribute of  $\mathbf{R}$  is lost or introduced (i.e.,  $\mathbf{R}_1 \cup \dots \cup \mathbf{R}_m = \mathbf{R}$ )
  2. No schema  $\mathbf{R}_i$  is a subset or equal to any relation  $\mathbf{R}_j$  (for  $i \neq j$ )
- When  $m = 2$ , the decomposition  $\underline{\mathbf{R}} = \{\mathbf{R}_1, \mathbf{R}_2\}$  is called *binary*
- Not every decomposition of  $\mathbf{R}$  is “desirable”. Why?
- Properties of a decomposition?
  - (1) Lossless-join – this is a **must**
  - (2) Dependency-preserving – this is **desirable**

Explanation follows ...

# Example

Relation Instance:

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 2 | 5 |

Decomposed into:

| A | B |
|---|---|
| 1 | 2 |
| 4 | 2 |

| B | C |
|---|---|
| 2 | 3 |
| 2 | 5 |

To “recover” information, we join the relations:

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 2 | 5 |
| 4 | 2 | 3 |
| 1 | 2 | 5 |

Why do we got new tuples?

# Lossless-Join Decomposition

---

- Suppose  $R$  is a relation and  $F$  is a set of FD's over  $R$ .  
A binary decomposition of  $R$  into relation schemas  $R_1$  and  $R_2$ , with attribute sets  $X$  and  $Y$  is said to be a **lossless-join decomposition with respect to  $F$** , if for every instance  $r$  of  $R$  that satisfies  $F$ , it holds that  $\pi_X(r) \bowtie \pi_Y(r) = r$
- **Thm:** Let  $R$  be a relation schema and  $F$  a set of FD's on  $R$ .  
A binary decomposition of  $R$  into  $R_1$  and  $R_2$  with attribute sets  $X$  and  $Y$  is lossless if  $X \cap Y \rightarrow X$  or  $X \cap Y \rightarrow Y$ , i.e., this binary decomposition is lossless if the common attributes of  $X$  and  $Y$  form a key of  $R_1$  or  $R_2$

# Example: Lossless-join

Relation Instance:

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 2 | 3 |

Decomposed into:

| A | B |
|---|---|
| 1 | 2 |
| 4 | 2 |

| B | C |
|---|---|
| 2 | 3 |

$$F = \{ B \rightarrow C \}$$

To recover the original relation  $r$ , we join the two relations:

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 2 | 3 |

No new tuples !

# Example: Dependency Preservation

Relation Instance:

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 5 | 7 |
| 4 | 3 | 6 | 8 |

$$F = \{ B \rightarrow C, B \rightarrow D, A \rightarrow D \}$$

Decomposed into:

| A | B |
|---|---|
| 1 | 2 |
| 4 | 3 |

| B | C | D |
|---|---|---|
| 2 | 5 | 7 |
| 3 | 6 | 8 |

Can we enforce  $A \rightarrow D$ ?  
How ?

# Dependency-Preserving Decomposition

---

- A dependency-preserving decomposition allows us to enforce every FD (on each insertion of a tuple or when modifying a tuple) by examining just one single relation instance
- Let  $R$  be a relation schema that is decomposed into two schemas with attribute sets  $X$  and  $Y$ , and let  $F$  be a set of FD's over  $R$ . The projection of  $F$  on  $X$  (denoted by  $F_X$ ) is the set of FD's in  $F^+$  that follow from  $F$  and involve only attributes in  $X$ 
  - Recall that a FD  $U \rightarrow V$  in  $F^+$  is in  $F_X$  if all the attributes in  $U$  and  $V$  are in  $X$ ; In this case, we say this FD is “relevant” to  $X$
- The decomposition of  $\langle R, F \rangle$  into two schemas with attribute sets  $X$  and  $Y$  is dependency-preserving if  $(F_X \cup F_Y)^+ \equiv F^+$

# Normal Forms

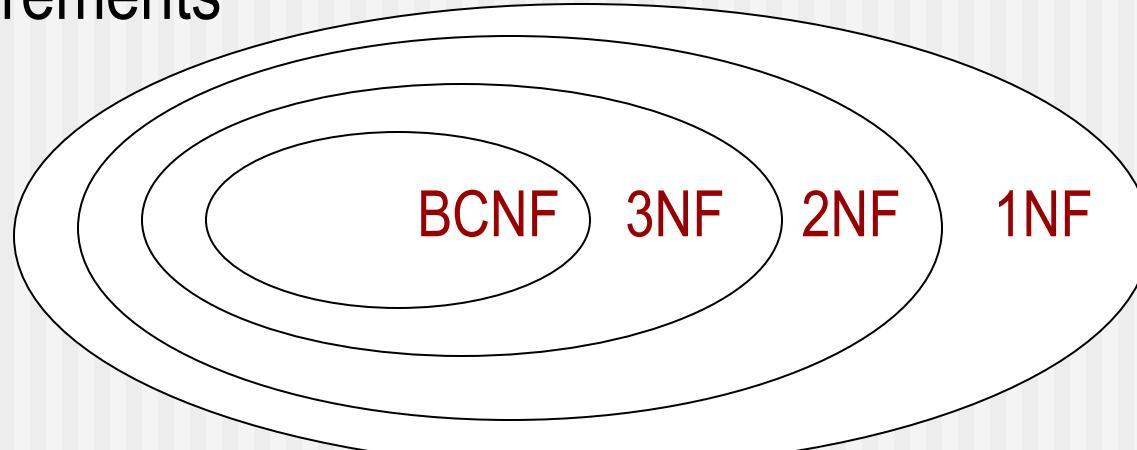
---

- Given a relation schema  $R$ , we must be able to determine whether it is “good” or we need to decompose it into smaller relations, and if so, how?
- To address these issues, we need to study **normal forms**
- If a relation schema is in one of these normal forms, we know that it is in some “good” shape in the sense that *certain kinds of problems (related to redundancy) cannot arise*

# Normal Forms

---

- The normal forms based on FD's are
  - First normal form (1NF)
  - Second normal form (2NF)
  - Third normal form (3NF)
  - Boyce-Codd normal form (BCNF)
- These normal forms have increasingly restrictive requirements



# Third Normal Form (3NF)

---

Let  $R$  be a relation schema,  $F$  a set of FD's on  $R$ ,  $X \subseteq R$ , and  $A \in R$ .

- We say  $R$  w.r.t.  $F$  is in 3NF (**third normal form**), if for every FD  $X \rightarrow A$  in  $F$ , at least one of the following conditions holds:
  - $A \in X$ , that is,  $X \rightarrow A$  is a trivial FD, **or**
  - $X$  is a superkey, **or**
  - If  $X$  is not a key, then  $A$  is part of some key of  $R$
- To determine if a relation  $\langle R, F \rangle$  is in 3NF:
  - We check whether the LHS of each nontrivial FD in  $F$  is a superkey
  - If not, we check whether its RHS is part of any key of  $R$

# Boyce-Codd Normal Form

---

Let  $R$  be a relation schema,  $F$  a set of FD's on  $R$ ,  $X \subseteq R$ , and  $A \in R$ .

- We say  $R$  w.r.t.  $F$  is in **Boyce-Codd normal form**, if for every FD  $X \rightarrow A$  in  $F$ , at least one of the following conditions holds:
  - $A \in X$ , that is,  $X \rightarrow A$  is a trivial FD, **or**
  - $X$  is a superkey
- To determine whether  $R$  with a given set of FD's  $F$  is in BCNF
  - Check whether the LHS  $X$  of each nontrivial FD in  $F$  is a superkey
    - How? Simply compute  $X^+$  (w.r.t.  $F$ ) and check if  $X^+ = R$

# Normal Forms for Relational Data

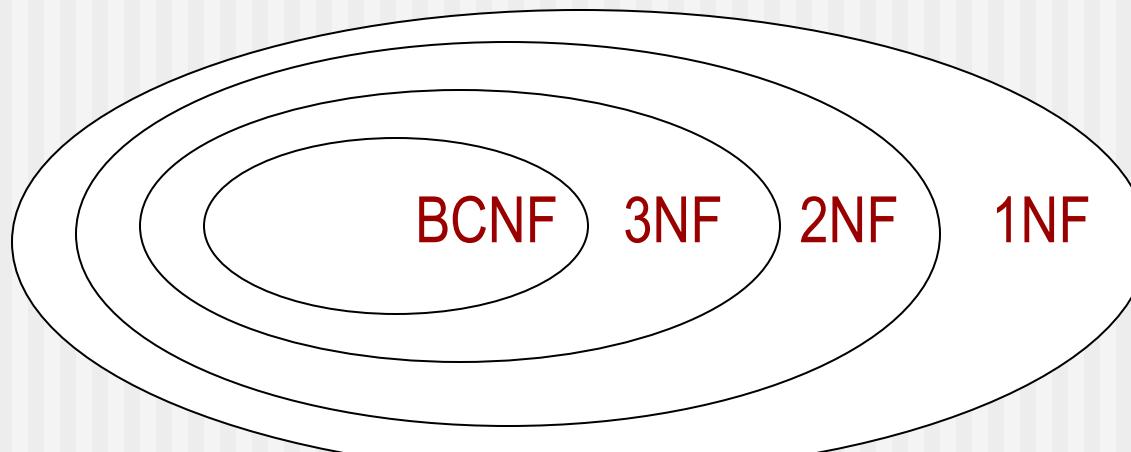
---

- If a relation schema is in some normal form, it means it is in some level of quality in the sense that *certain kinds of problems and anomalies (related to redundancy) will not arise*
- Given a relation schema  $R$  with FDs  $F$ , how to determine in what normal form it is? And if it is not in a particular normal form, how to convert it into a few smaller relations all of which are in that normal form.
- To address these issues, we study definitions and algorithms for different **normal forms**.

# Normal Forms

---

- The normal forms as defined and captured by FD's:
  - First normal form (1NF)
  - Second normal form (2NF)
  - ✓ Third normal form (3NF)
  - ✓ Boyce-Codd normal form (BCNF)
- The relationships among these normal forms:



# Third Normal Form (3NF)

---

Let  $R$  be a relation schema with a set of FD's  $F$ .

- We say  $R$  w.r.t.  $F$  is in 3NF (**third normal form**), if for every FD  $X \rightarrow A$  in  $F$ , at least one of the following conditions holds:
  - $X \rightarrow A$  is a trivial, i.e.,  $A \in X$ , or
  - $X$  is a superkey, or
  - $X$  is not a key but  $A$  is part of some key of  $R$
- ➔ Therefore, to determine if  $R$  is in 3NF w.r.t.  $F$ , we need to:
  - Check if the LHS of each nontrivial FD in  $F$  is a superkey
  - If not, check if its RHS is part of any key of  $R$

# Boyce-Codd Normal Form

---

Given: A relation schema  $R$  with a set of FD's  $F$  on  $R$ .

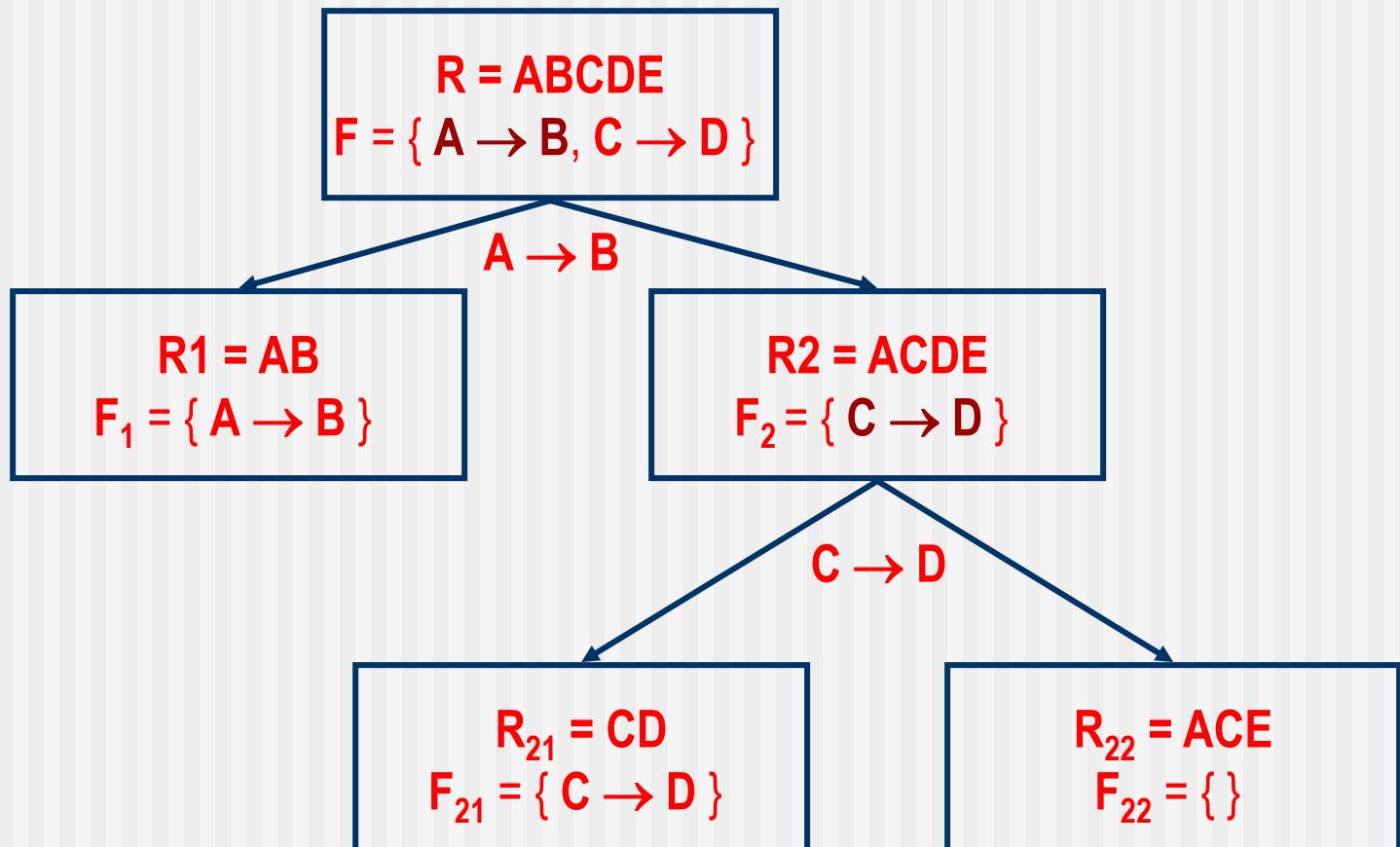
- We say  $R$  w.r.t.  $F$  is in **Boyce-Codd normal form**, if for every FD  $X \rightarrow Y$  in  $F$ , at least one of the following 2 conditions holds
  - $Y \subseteq X$ , that is,  $X \rightarrow Y$  is a trivial FD **or**
  - $X$  is a superkey
- To determine if  $R$  is in BCNF w.r.t.  $F$ ,  
For every FD  $X \rightarrow Y$ , check if its LHS  $X$  is a superkey.  
That is, compute  $X^+$ .
  - If  $X^+ = R$ , then no problem; the FD  $X \rightarrow Y$  passes.
  - If  $X^+ \neq R$ , then  $R$  is not in BCNF.

# Decomposition into BCNF relations

---

- Suppose relation  $R$  is in 1NF. Consider  $\langle R, F \rangle$
- If  $R$  is not in BCNF w.r.t.  $F$ , we can always obtain a *lossless-join decomposition* of  $R$  into a collection of BCNF relations
- However, this decomposition may not always be dependency preserving.
- The basic step of a BCNF decomposition alg. (done **recursively**):
  - Pick an FD  $X \rightarrow A \in F$  that violates the BCNF requirement:
  - 1. Decompose  $R$  into two relations:  $R_1 = X^+$  and  $R_2 = (R - X^+) \cup X$
  - 2. Project  $F$  onto  $R_1$  and  $R_2$ . Let's call them as  $F_1$  and  $F_2$ .
  - 3. If  $\langle R_1, F_1 \rangle$  or  $\langle R_2, F_2 \rangle$  is not in BCNF, decompose further.

# Example (Decomposition into BCNF relations)



# Decomposition into 3NF

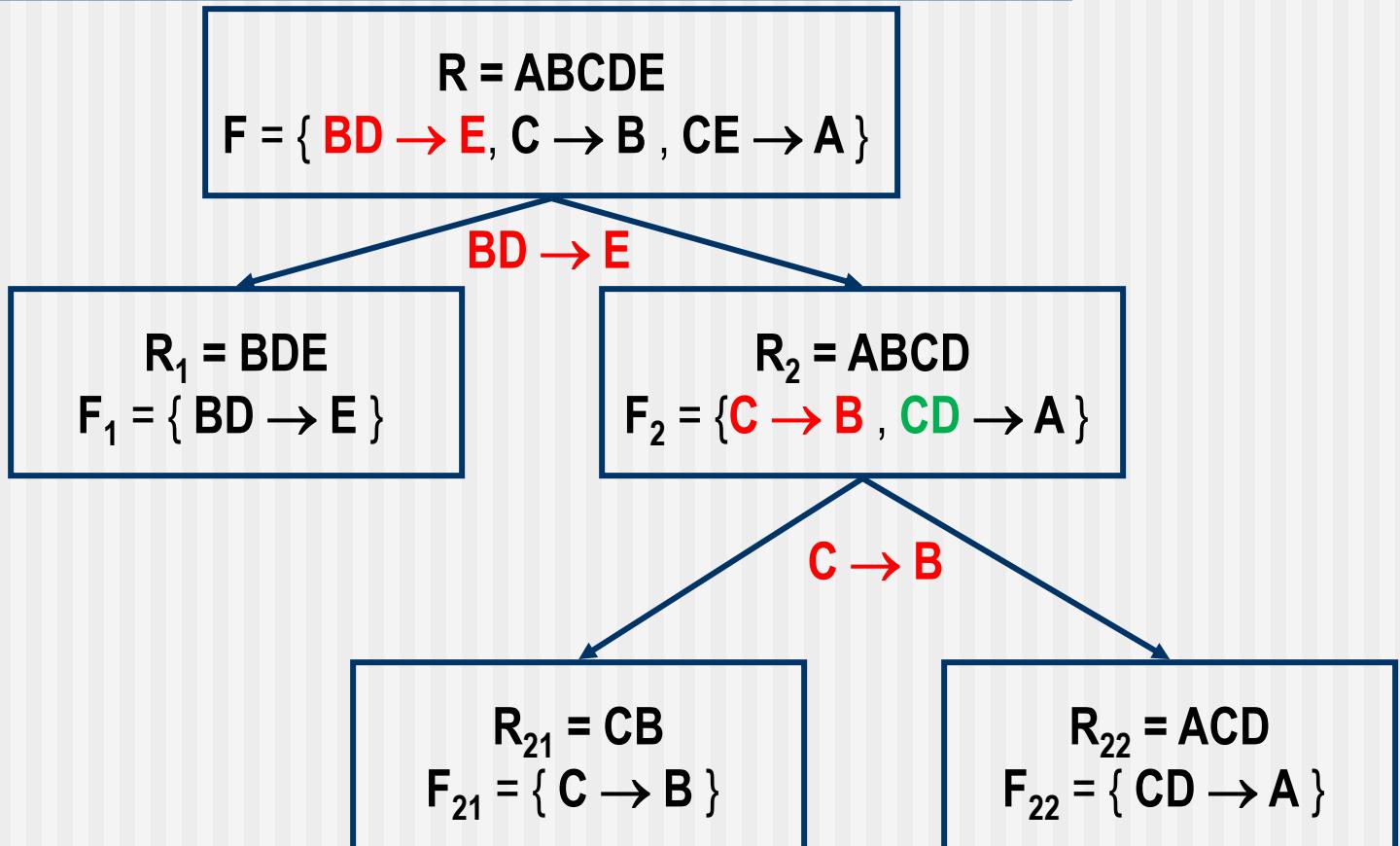
---

- We can always obtain a lossless-join, dependency-preserving decomposition of a relation into 3NF relations. How?
- We discuss 2 solution approaches for 3NF decomposition.
- **Approach 1:** using the *binary decomposition* method.

Let  $\underline{R} = \{ R_1, R_2, \dots, R_n \}$  be the result. Recall that this is always lossless-join, **but may not preserve all the FD's** → need to fix this!

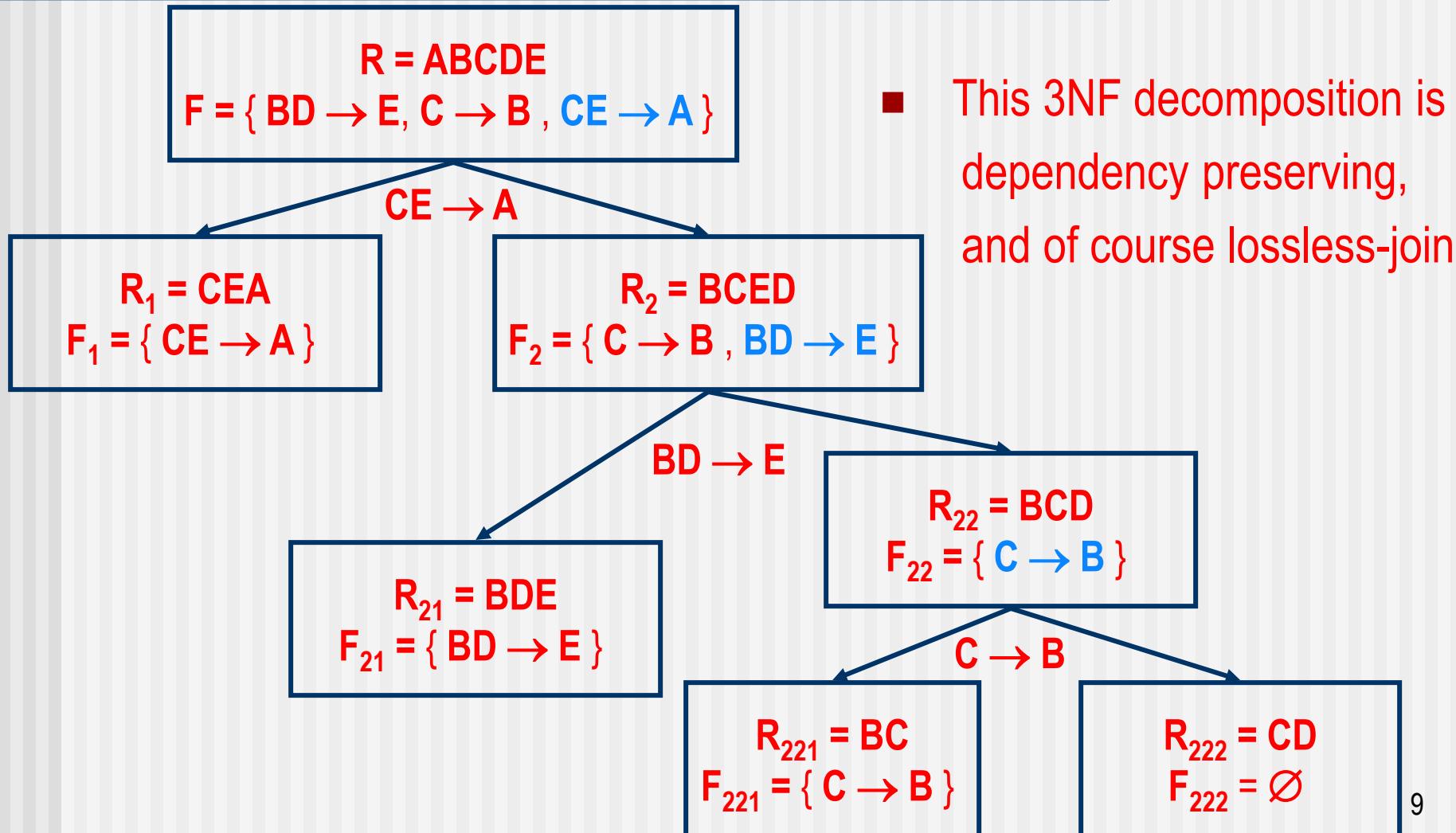
- Identify the set  $N$  of FD's in  $F$  which we lost in the decomposition proc.
- For each FD  $X \rightarrow A$  in  $N$ , create a relation schema  $XA$  and add it to  $\underline{R}$
- A refinement step to avoid creating MANY relations: if there are several FD's with the same LHS, e.g.,  $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_k$ , create **just one relation** with schema  $XA_1 \dots A_k$

# Example (3NF Decomposition)



- $CE \rightarrow A$  is **not** preserved, since  $A \notin \{CE\}^+$  w.r.t.  $F_1 \cup F_{21} \cup F_{22}$
- ➔ To fix this, We **add** to  $R$ , a new relation  $R_3 = CEA$  with  $F_3 = \{ CE \rightarrow A \}$

# Example (using a *different order*)



# Decomposition into 3NF

---

- **Method 1:** (binary decomposition):
    - Lossless-join ✓
    - May not be dependency preserving. If so, then add extra relations  $X\text{A}$ , for every FD  $X \rightarrow A$  we lost
  - **Method 2:** the *synthesis* approach
    - Dependency preservation ✓
    - However, may not be lossless-join. If so, we must add to  $\underline{R}$ , one extra relation that includes whose attributes form a key of  $R$
- What would be the FDs on this newly added relation?

# Decomposition into 3NF (Using the synthesis approach)

---

Consider  $\langle R, F \rangle$

- The synthesis approach:
  - Get a minimal cover  $F^c$  of  $F$
  - For each FD  $X \rightarrow A$  in  $F^c$ , add schema  $XA$  to  $\underline{R}$
  - If none of the relation schemas in  $\underline{R}$  from previous step is a superkey for  $R$ , add a relation whose schema is a key for  $R$

# Example

---

- $R = (A, B, C)$  with  $F = \{A \rightarrow B, C \rightarrow B\}$  is not in 3NF, so we decompose  $R$ . This yields:
- $\underline{R} = \{R_1, R_2\}$ , where  $R_1 = (A, B)$  with FDs  $F1 = \{A \rightarrow B\}$  and  $R_2 = (B, C)$  with  $F2 = \{C \rightarrow B\}$
- AC is the key for  $R$ , and since neither AB nor BC is a *superkey* for  $R$ , we add  $R_3 = (A, C)$  to  $\underline{R}$ .
- The decomposition  $\underline{R} = \{R_1, R_2, R_3\}$  is both lossless and dependency-preserving. (The FD on AC is  $\emptyset$ ).

# The Chase test to check lossless join

Suppose relation  $R\{A_1, \dots, A_k\}$  is decomposed into  $R_1, \dots, R_n$

To determine if this decomposition is lossless, we use table

$L[1 \dots n][1 \dots k]$ , and initialize it as follows.

**Initializing the table:**

```
for each relation R_i do
 for each attribute A_j do
 if A_j is an attribute in R_i
 then $L[i][j] \leftarrow a_j$
 else $L[i][j] \leftarrow b_{ij}$
```

# Chase test (to check lossless join)

---

**repeat**

**for each FD  $X \rightarrow Y$  in  $F$  do:**

for each pair of rows  $i$  and  $j$  in  $L$  that agree on  $X$  (that is,  $L[i] = L[j]$  for every attribute in  $X$ ), modify every column  $t$  in  $L$  corresponding to attribute  $A_t$  in  $Y$  as follows:

```
if $L[i][t] = a_t$
 then $L[j][t] \leftarrow a_t$
else if $L[j][t] = a_t$
 then $L[i][t] \leftarrow a_t$
else $L[j][t] \leftarrow L[i][t]$
```

**until no change**

The decomposition is lossless if, after this algorithm terminates, table  $L$  contains a row of all a's, that is, if there is a row  $i$  such that  $L[i][j] = a_j$  for every column  $j$  corresponding to each attribute  $A_j$  in  $R$

# Examples

---

- Given  $\langle R, F \rangle$ , where  $R = (A, B, C, D)$ , and  $F = \{A \rightarrow B, A \rightarrow C, C \rightarrow D\}$  is a set of FD's on  $R$
- Is the decomposition  $\underline{R} = \{R_1, R_2\}$  lossless, where  $R_1 = (A, B, C)$  and  $R_2 = (C, D)$ ?
  - To be discussed in class
- Now consider  $S = (A, B, C, D, E)$  with the FD's:  
$$G = \{AB \rightarrow CD, A \rightarrow E, C \rightarrow D\}$$
- Is decomposition of  $\underline{S} = \{S_1, S_2, S_3\}$  lossless, where  $S_1 = (A, B, C)$ ,  $S_2 = (B, C, D)$ , and  $S_3 = (C, D, E)$ ?
  - To be discussed in class

# Checking if a decomposition is Dependency-Preserving?

---

Inputs: Let  $\langle R, F \rangle$ , where  $F = \{X_1 \rightarrow Y_1, \dots, X_n \rightarrow Y_n\}$ .

Suppose  $\underline{R} = \{R_1, \dots, R_k\}$  is a decomposition of  $R$   
and  $F_i$  is the projection of  $F$  on schema  $R_i$

Method:

```
preserved ← TRUE
for each FD $X \rightarrow Y$ in F and while preserved = TRUE
do compute X^+ under $F_1 \cup \dots \cup F_k$;
 if $Y \not\subseteq X^+$ then {preserved ← FALSE; exit};
end
```

# Example

---

- Consider  $R = (A, B, C, D)$ ,  $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D\}$
- Is the decomposition  $\underline{R} = \{R_1, R_2\}$  dependency-preserving, where  $R_1 = (A, B)$ ,  $F_1 = \{A \rightarrow B\}$ ,  $R_2 = (A, C, D)$ , AND  $F_2 = \{C \rightarrow D, A \rightarrow D, A \rightarrow C\}$ ?
  - Check if  $A \rightarrow B$  is preserved
    - Compute  $A^+$  under  $\{A \rightarrow B\} \cup \{C \rightarrow D, A \rightarrow D, A \rightarrow C\}$ 
      - $A^+ = \{A, B, C, D\}$
      - Check if  $B \in A^+$
      - Yes
    - $A \rightarrow B$  is preserved
  - Check if  $B \rightarrow C$  is preserved
    - Compute  $B^+$  under  $\{A \rightarrow B\} \cup \{C \rightarrow D, A \rightarrow D, A \rightarrow C\}$ 
      - $B^+ = \{B\}$
      - Check if  $C \in B^+$
      - No
    - $B \rightarrow C$  is not preserved

→ The decomposition is not dependency-preserving

# **COMP353 Databases**

---

**Relational Algebra (RA)  
for Relational Data Model**

# Relational Algebra (RA)

---

- *Database Query languages* are specialized languages to ask for information (**queries**) in DB.
- **Relational Algebra (RA)** is a query language associated with the relational data model.
- Queries in RA are expressions using a collection of operators on relations in the DB.
- The input(s) and output of a RA query are relations
- A query is **evaluated** using the **current instance** of the input relations to produce the output

# Operations in “standard” RA

---

- The well-known set operations
  - ✓ Union (  $\cup$  )
  - ✓ Intersection (  $\cap$  )
  - ✓ Difference (  $-$  )
- Special DB operations that select “parts” of a relation instance
  - Selection (  $\sigma$  ) – selects some rows (tuples) & discards the rest
  - Projection (  $\pi$  ) – selects some columns (attributes) & discards the rest
- Operations that “combine” the tuples from the argument relations
  - ✓ Cartesian product (  $\times$  ) – pairs the tuples in all possible ways
  - Join (  $\bowtie$  ) – pairs particular tuples from the two input relations
- A unary operation to **rename** relations, called Rename (  $\rho$  )

**Note:** The output of a RA expression is an “unnamed” relation/set, i.e., RA expressions return **sets**, whereas SQL returns **multisets (bags)**

# Compatibility Requirement

---

- We can apply the **set operators** of **union**, **intersection**, and **difference** to instances of relations **R** and **S** if **R** and **S** are *compatible*, that is, they have “the same” schemas.
- **Definition:** Relations  $S(A_1, \dots, A_n)$  and  $R(B_1, \dots, B_m)$  are compatible if:
  - (1)  $n=m$  and
  - (2)  $\text{type}(A_i) = \text{type}(B_i)$  (or compatible types), for all  $1 \leq i \leq n$ .

# Set Operations on Relations

---

Let **R** and **S** be relation schemas, and **r** and **s** be any instances of them.

- The **union** of **r** and **s** is the set of all tuples that appear in either one or both. Each tuple **t** appears only once in the union, even if it appears in both;  $r \cup s = \{t \mid t \in r \vee t \in s\}$
- The **intersection** of **r** and **s**, is the set of all tuples that appear in both;  $r \cap s = \{t \mid t \in r \wedge t \in s\}$
- The **difference** of **r** and **s**, is the set of all tuples that appear in **r** but not in **s**;  $r - s = \{t \mid t \in r \wedge t \notin s\}$
- **Commutative** operations;  $r \text{ Op } s = s \text{ Op } r$   
Note: Set difference ( $-$ ) is not commutative, i.e.,  $(r - s \neq s - r)$

# Example

**Relation Schema:** **Star** (name, address, gender, birthdate )

**Instance r  
of Star:**

| Name          | Address     | Gender | Birthdate |
|---------------|-------------|--------|-----------|
| Carrie Fisher | 123 Maple   | F      | 9/9/99    |
| Mark Hamill   | 456 Oak rd. | M      | 8/8/88    |

**Instance s  
of Star:**

| Name          | Address      | Gender | Birthdate |
|---------------|--------------|--------|-----------|
| Carrie Fisher | 123 Maple    | F      | 9/9/99    |
| Harrison Ford | 789 Palm rd. | M      | 7/7/77    |

**r ∪ s:**

| Name          | Address      | Gender | Birthdate |
|---------------|--------------|--------|-----------|
| Carrie Fisher | 123 Maple    | F      | 9/9/99    |
| Mark Hamill   | 456 Oak rd.  | M      | 8/8/88    |
| Harrison Ford | 789 Palm rd. | M      | 7/7/77    |

# Example

**Relation Schema:** **Star** (name, address, gender, birthdate )

**Instance r  
of Star:**

| Name          | Address     | Gender | Birthdate |
|---------------|-------------|--------|-----------|
| Carrie Fisher | 123 Maple   | F      | 9/9/99    |
| Mark Hamill   | 456 Oak rd. | M      | 8/8/88    |

**Instance s  
of Star:**

| Name          | Address      | Gender | Birthdate |
|---------------|--------------|--------|-----------|
| Carrie Fisher | 123 Maple    | F      | 9/9/99    |
| Harrison Ford | 789 Palm rd. | M      | 7/7/77    |

**r ∩ s:**

| Name          | Address   | Gender | Birthdate |
|---------------|-----------|--------|-----------|
| Carrie Fisher | 123 Maple | F      | 9/9/99    |

# Example

**Relation Schema:** **Star** (name, address, gender, birthdate )

**Instance r  
of Star:**

| Name          | Address     | Gender | Birthdate |
|---------------|-------------|--------|-----------|
| Carrie Fisher | 123 Maple   | F      | 9/9/99    |
| Mark Hamill   | 456 Oak rd. | M      | 8/8/88    |

**Instance S  
of Star:**

| Name          | Address      | Gender | Birthdate |
|---------------|--------------|--------|-----------|
| Carrie Fisher | 123 Maple    | F      | 9/9/99    |
| Harrison Ford | 789 Palm rd. | M      | 7/7/77    |

**r – S:**

| Name        | Address     | Gender | Birthdate |
|-------------|-------------|--------|-----------|
| Mark Hamill | 456 Oak rd. | M      | 8/8/88    |

# Example

**Relation Schema:** **Star** (name, address, gender, birthdate)

**Instance r  
of Star:**

| Name          | Address     | Gender | Birthdate |
|---------------|-------------|--------|-----------|
| Carrie Fisher | 123 Maple   | F      | 9/9/99    |
| Mark Hamill   | 456 Oak rd. | M      | 8/8/88    |

**Instance S  
of Star:**

| Name          | Address      | Gender | Birthdate |
|---------------|--------------|--------|-----------|
| Carrie Fisher | 123 Maple    | F      | 9/9/99    |
| Harrison Ford | 789 Palm rd. | M      | 7/7/77    |

**S – r:**

| Name          | Address      | Gender | Birthdate |
|---------------|--------------|--------|-----------|
| Harrison Ford | 789 Palm rd. | M      | 7/7/77    |

# Projection ( $\pi$ )

---

- Let  $R$  be a relation schema.
- The **projection** operation ( $\pi$ ) is used to produce, from any instance  $r$  of  $R$ , a new relation that includes listed “columns” of  $R$
- The output of  $\pi_{A_1, A_2, \dots, A_j}(r)$  is a relation with columns  $A_1, A_2, \dots, A_j$ , in this order.
- Note: The subscript of  $\pi$  is a *list*, which defines the structure of the output as the ordered tuple  $(A_1, A_2, \dots, A_j)$ .

# Example

Relation Schema: Movie(title, year, length, filmType, studioName, producer)

Instance  
movie  
Of Movie:

| title         | year | length | filmType | studioName | producer |
|---------------|------|--------|----------|------------|----------|
| Star wars     | 1977 | 124    | color    | Fox        | 12345    |
| Mighty Ducks  | 1991 | 104    | color    | Disney     | 67890    |
| Wayne's World | 1992 | 95     | color    | Paramount  | 99999    |

Query:  $\pi$  title, year, length(movie)

| title         | year | length |
|---------------|------|--------|
| Star wars     | 1977 | 124    |
| Mighty Ducks  | 1991 | 104    |
| Wayne's World | 1992 | 95     |

# Example

**Relation Schema: Movie(title, year, length, filmType, studioName, producer)**

| Instance<br><b>movie</b><br>Of Movie: | title         | year | length | filmType | studioName | producer |
|---------------------------------------|---------------|------|--------|----------|------------|----------|
|                                       | Star wars     | 1977 | 124    | color    | Fox        | 12345    |
|                                       | Mighty Ducks  | 1991 | 104    | color    | Disney     | 67890    |
|                                       | Wayne's World | 1992 | 95     | color    | Paramount  | 99999    |

**Query:**  $\pi_{\text{filmType}}(\text{movie})$

**Result:**

| filmType |
|----------|
| color    |

# Selection ( $\sigma$ )

---

- The **selection** operator ( $\sigma$ ), applied to an instance  $r$  of relation  $R$ , returns a subset of  $r$
- We denote this operation/query by  $\sigma_c(r)$
- The output includes tuples satisfying condition  $C$
- The schema of the output is the same as  $R$

# Example

**Relation Schema: Movie(title, year, length, filmType, studioName, producer)**

Instance **movie**  
of Movie:

| title         | year | length | filmType | studioName | producer |
|---------------|------|--------|----------|------------|----------|
| Star wars     | 1977 | 124    | color    | Fox        | 12345    |
| Mighty Ducks  | 1991 | 104    | color    | Disney     | 67890    |
| Wayne's World | 1992 | 95     | color    | Paramount  | 99999    |

Query:  $\sigma_{\text{length} \geq 100}(\text{movie})$

Result:

| title        | year | length | filmType | studioName | producer |
|--------------|------|--------|----------|------------|----------|
| Star wars    | 1977 | 124    | color    | Fox        | 12345    |
| Mighty Ducks | 1991 | 104    | color    | Disney     | 67890    |

# Example

**Relation:** Movie(title, year, length, filmType, studioName, producer)

**Instance  
movie  
of Movie:**

| title         | year | length | filmType | studioName | producer |
|---------------|------|--------|----------|------------|----------|
| Star wars     | 1977 | 124    | color    | Fox        | 12345    |
| Mighty Ducks  | 1991 | 104    | color    | Disney     | 67890    |
| Wayne's World | 1992 | 95     | color    | Paramount  | 99999    |

**Query:**  $\sigma_{\text{length} \geq 100 \text{ AND } \text{studioName} = \text{'Fox'}}$  (Movie)

**Result:**

| title     | year | length | filmType | studioName | producer |
|-----------|------|--------|----------|------------|----------|
| Star wars | 1977 | 124    | color    | Fox        | 12345    |

# Cartesian Product ( $\times$ )

---

- Let **R** and **S** be relation schemas, and **r** and **s** be any instances of **R** and **S**, respectively.
- The **Cartesian Product** of **r** and **s** is the set of all tuples obtained by “**concatenating**” the tuples in **r** and **s**. Formally,  $r \times s = \{ t_1.t_2 \mid t_1 \in r \wedge t_2 \in s \}$
- The schema of result is the “union” of **R** and **S**
  - If **R** and **S** have some attributes in common, we need to invent new names for identical names, e.g., use **R.B** and **S.B**, if **B** appears in both **R** and **S**

# Example

Instance r of R:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |

Instance s of S:

| B | C  | D  |
|---|----|----|
| 2 | 5  | 6  |
| 4 | 7  | 8  |
| 9 | 10 | 11 |

$r \times s$ :

| A | R.B | S.B | C  | D  |
|---|-----|-----|----|----|
| 1 | 2   | 2   | 5  | 6  |
| 1 | 2   | 4   | 7  | 8  |
| 1 | 2   | 9   | 10 | 11 |
| 3 | 4   | 2   | 5  | 6  |
| 3 | 4   | 4   | 7  | 8  |
| 3 | 4   | 9   | 10 | 11 |

# Theta-join ( $\theta$ )

---

- Suppose  $R$  and  $S$  are relation schemas,  $r$  is an instance of  $R$ , and  $s$  is an instance of  $S$ . The **theta-join** of  $r$  and  $s$  is the set of all tuples obtained from concatenating all  $t_1 \in r$  and  $t_2 \in s$ , such that  $t_1$  and  $t_2$  satisfy some condition  $C$
- We denote  $\theta$ -join by  $r \triangleright\triangleleft_C s$
- The schema of the result is the same as the schema of  $R \times S$  (i.e., the union of  $R$  and  $S$ )
- $C$  is a Boolean expression, simple or complex, as in operation  $\sigma$

# Example

Instance r of R:

| A | B | C  |
|---|---|----|
| 1 | 2 | 3  |
| 6 | 5 | 8  |
| 9 | 7 | 11 |

Instance s of S:

| B | C | D  |
|---|---|----|
| 2 | 3 | 4  |
| 2 | 3 | 5  |
| 7 | 8 | 10 |

$r \triangleright\triangleleft_{A < D} s$ :

| A | R.B | R.C | S.B | S.C | D  |
|---|-----|-----|-----|-----|----|
| 1 | 2   | 3   | 2   | 3   | 4  |
| 1 | 2   | 3   | 2   | 3   | 5  |
| 1 | 2   | 3   | 7   | 8   | 10 |
| 6 | 5   | 8   | 7   | 8   | 10 |
| 9 | 7   | 11  | 7   | 8   | 10 |

# Equi-join

---

- The **equi-join** operator, is a special case of  $\theta$ -join, in which we may only use the equality relation ( $=$ ) in condition **C**
- It is denoted as  $r \bowtie_c s$  (i.e., the same as  $\theta$ -join)
- The schema of the output is the same as that of  $\theta$ -join

# Example

Instance r of R:

| A | B | C  |
|---|---|----|
| 1 | 2 | 3  |
| 6 | 5 | 8  |
| 9 | 7 | 11 |

Instance s of S:

| B | C | D  |
|---|---|----|
| 2 | 3 | 4  |
| 2 | 3 | 5  |
| 7 | 8 | 10 |

$r \triangleright\triangleleft_{R.C = S.C} s$ :

| A | R.B | R.C | S.B | S.C | D  |
|---|-----|-----|-----|-----|----|
| 1 | 2   | 3   | 2   | 3   | 4  |
| 1 | 2   | 3   | 2   | 3   | 5  |
| 6 | 5   | 8   | 7   | 8   | 10 |

# Natural Join ( $\bowtie$ )

- **Natural join**, is a special case of equi-join, where the equalities are not explicitly specified, rather they are assumed implicitly on the common attributes of **R** and **S**
- We denote this natural join operation by  $r \bowtie s$
- The schema of the output is similar to that of equi-join, except that each common attribute appears only once.

Note: If **R** and **S** do not have any common attribute, then the join operation becomes Cartesian product.

# Example

Instance r of R:

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 2 | 8 |
| 9 | 7 | 3 |

Instance s of S:

| B | C | D  |
|---|---|----|
| 2 | 3 | 4  |
| 2 | 3 | 5  |
| 7 | 8 | 10 |

$r \bowtie s$ :

| A | B | C | D |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 5 |

# Expressing Queries in RA

---

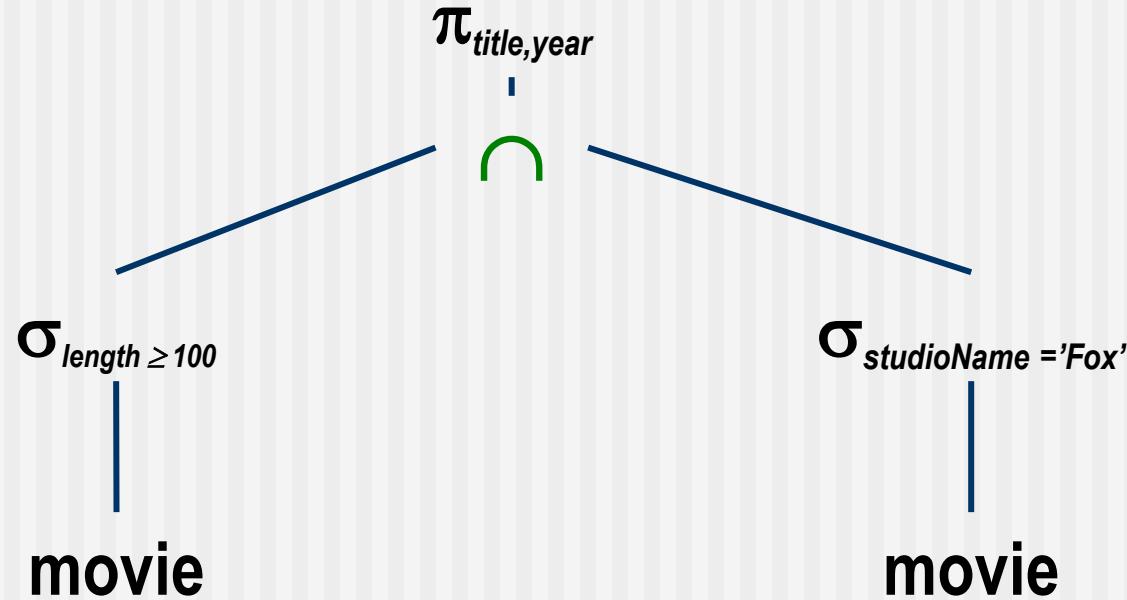
- Every standard RA operation has relation(s) as argument(s) and produces a relation (set) as the output  
*(Exception is the sort operator  $\tau$ )*
- This property of RA operations (that inputs and outputs are relations) makes it possible to formulate/express any query by *composing/nesting/grouping* subqueries.
- We can use parentheses for *grouping*, *in order* to improve *clarity* and *readability*

# Example: RA Query

---

- Relation schema:  
**Movie** (title, year, length, filmType, studioName)
- Query: List the title and year of every movie made by Fox studio whose length is at least 100 minutes?
- One way to express this query in RA is:  
$$\pi_{\text{title}, \text{year}} (\sigma_{\text{studioName} = \text{'Fox'} \text{ and } \text{length} \geq 100} (\text{movie}))$$
- Another way:
  - Select those **movie** tuples that have *length*  $\geq 100$
  - Select those **movie** tuples that have *studioName* = 'Fox'
  - Find the **intersection** of the above two results
  - Then project on the attributes **title** and **year**

# Example: RA Query

$$\pi_{title, year}(\sigma_{studioName = 'Fox' \text{ and } length \geq 100}(\text{movie}))$$

$$\pi_{title, year}(\sigma_{length \geq 100}(\text{movie}) \cap \sigma_{studioName = 'Fox'}(\text{movie}))$$

# Example: RA Query

---

- Relation schema:
  - Movie** (title, year, length, filmType, studioName)
  - StarsIn** (title, year, starName)
- Query: List the **names** of the stars of movies of  $length \geq 100$  minutes long.
- One expression in RA for this query:
  - Select **movie** tuples of  $length \geq 100$
  - Join the result with relation **StarsIn**
  - Project on the attribute **starName**
- **Exp1:**  $\pi_{starName}(\sigma_{length \geq 100}(\text{movie}) \bowtie \text{starsIn})$
- **Another solution:**  $\pi_{starName}(\sigma_{length \geq 100}(\text{movie} \bowtie \text{starsIn}))$

# Renaming Operator ( $\rho$ )

---

- To **control** manipulating the names of the attributes in formulating queries in relational algebra, we may need **renaming** of relations. May do this for *convenience* too
- **The Renaming Operator** is denoted by  $\rho_{s(A_1, A_2, \dots, A_n)}(r)$
- The result is a copy of the input relation instance  $r$ , but renamed to  $s$  and its attributes to  $A_1, \dots, A_n$ , in that order.
- Use  $\rho_s(r)$  to give relation  $r$  a new name  $s$  (**with the same attributes in  $r$** )

That is, in this case, schema of  $s$  is the same as that of  $r$ .

# Example

- **Query:**  $\pi \text{ starName} (\sigma_{\text{length} \geq 100} (\text{movie}) \bowtie \text{starsIn})$
- This query can be rewritten in 2 steps as follows:
  1.  $\rho_M(\text{title, year, length, filmType, studioName}) (\sigma_{\text{length} \geq 100} (\text{movie}))$   
or even simpler as:  $\rho_M (\sigma_{\text{length} \geq 100} (\text{movie}))$  if used in the same formula
  2. Or use  $M := \sigma_{\text{length} \geq 100} (\text{movie})$  as a separate formula and then formulate the query as:  $\pi \text{ starName} (M \bowtie \text{starsIn})$
- Consider **takes**(sid, cid, grade)
- **Query:** Find ID of every student who has taken at least 2 courses.
- $\pi_{\text{takes.sid}} (\sigma_{(\text{takes.sid} = T.\text{sid}) \text{ and } (\text{takes.cid} \neq T.\text{cid})} (\text{takes} \times \rho_T (\text{takes})))$

# Dependent and Independent Operations

---

- Some RA operations can be expressed based on other operations. Examples include:
  - $r \cap s = r - (r - s)$
  - $r \triangleright\triangleleft_C s = \sigma_C(r \times s)$
  - $r \triangleright\triangleleft s = \pi_L(\sigma_{r.A1 = s.A1 \text{ AND } \dots \text{ AND } r.An = s.An}(r \times s)),$   
where **L** is the list of attributes in **R** followed by those attributes in **S** that are not in **R**, and **A1, ..., An** are the common attributes of **R** and **S**

# Relational Algebra with Bag Semantics

---

- Relations stored in DB are called **base** relations/tables.
- Base relations are normally sets; no duplicates.
- In some situations, e.g., during query processing, it is allowed for relations to have duplicate tuples.
- If duplicates are allowed in a collection, it is called ***bag/multiset***.

Instance  
r of R:

| A | B | C  |
|---|---|----|
| 1 | 2 | 3  |
| 6 | 5 | 8  |
| 6 | 5 | 8  |
| 1 | 2 | 3  |
| 9 | 7 | 11 |

Here, r is a bag

# Why Bags?

---

- **1. Faster projection operations**
  - Bag projection is faster, since otherwise returning distinct values is expensive (as we need sorting for duplicate elimination).  
Another example: *Computing the bag union ( $r \cup^B s$ ) is much cheaper than computing the standard set union  $r \cup s$ .*  
Formally, if  $r$  and  $s$  have  $n$  and  $m$  tuples, then the bag and set union operations will cost  $O(n+m)$  and  $O(n*m)$ , respectively.
- **2. Correct computation with some aggregation**
  - For example, to compute the average of values for attribute A in the previous relation, we must consider the bag of those values

# Set Operations on Bags

---

- $r \cup^B s$ , the **bag union** of  $r$  and  $s$ , is the bag of tuples that are in  $r$ , in  $s$ , or in both. If a tuple  $t$  appears  $n$  times in  $r$ , and  $m$  times in  $s$ , then  $t$  appears  $n+m$  times in bag  $r \cup^B s$

$$r \cup^B s = \{ t:k \mid t:n \in r \wedge t:m \in s \wedge k = n+m \}$$

- $r \cap^B s$ , the **bag intersection** of  $r$  and  $s$ , is the bag of tuples that appear in both  $r$  and  $s$ . If a tuple  $t$  appears  $n$  times in  $r$ , and  $m$  times in  $s$ , then the number of occurrences of  $t$  in bag  $r \cap^B s$  is  $\min(n,m)$

$$r \cap^B s = \{ t:k \mid t:n \in r \wedge t:m \in s \wedge k = \min(n,m) \}$$

- $r -^B s$ , the **bag difference** of  $r$  and  $s$  is defined as follows:

$$r -^B s = \{ t:k \mid t:n \in r \wedge t:m \in s \wedge k = \max(0, n-m) \}$$

$$s -^B r = \{ t:k \mid t:n \in r \wedge t:m \in s \wedge k = \max(0, m-n) \}$$

# Example

Bag r:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |

Bag s:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |

$r \cup^B s$ :

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |

# Example

---

Bag r:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |

Bag s:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |

$r \cap^B s:$

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |

# Example

---

Bag r:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |

Bag s:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |

$r -^B s$ :

| A | B |
|---|---|
| 1 | 2 |
| 1 | 2 |

# Example

Bag r:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |

Bag s:

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 3 | 4 |
| 5 | 6 |

$s -^B r$ :

| A | B |
|---|---|
| 3 | 4 |
| 5 | 6 |

# Bag Projection $\pi^B$

---

- Let  $R$  be a relation scheme, and  $r$  be a collection of tuples over  $R$ , which could have duplicates.  
The **bag projection** operator is used to produce, from  $r$ , a bag of tuples over some of  $R$ .
- Even when  $r$  does not have duplicates, we may get duplicates when projecting on some attributes of  $R$ .  
That is,  $\pi^B$  does not eliminate the duplicates and hence corresponds exactly to the SELECT clause in SQL.

# Example

---

Bag r:

| A | B | C |
|---|---|---|
| 1 | 2 | 5 |
| 3 | 4 | 6 |
| 1 | 2 | 7 |
| 1 | 2 | 8 |

$\pi^B_{A, B}(r)$ :

| A | B |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 1 | 2 |
| 1 | 2 |

# Example

Relation Schema: movie(title, year, length, filmType, studioName, producer)

Instance:

| title         | year | length | filmType | studioName | producer |
|---------------|------|--------|----------|------------|----------|
| Star wars     | 1977 | 124    | color    | Fox        | 12345    |
| Mighty Ducks  | 1991 | 104    | color    | Disney     | 67890    |
| Wayne's World | 1992 | 95     | color    | Paramount  | 99999    |

$\pi^B_{\text{filmType}}(\text{movie})$ :

| filmType |
|----------|
| color    |
| color    |
| color    |

# Selection on Bags

---

- The selection operator  $\sigma_c$  applied to an instance  $r$  of relation  $R$  will return a subset of  $r$ 
  - The tuples returned are those that satisfy the specified condition  $C$  (which involves attributes of  $R$ )
  - Duplicates are **not** eliminated from the result of a bag-selection

**Note:** The selection operation  $\sigma$  in RA is **different** from the **SELECT** clause in SQL

# Example

---

Bag r:

| A | B | C |
|---|---|---|
| 1 | 2 | 5 |
| 3 | 4 | 6 |
| 1 | 2 | 7 |
| 1 | 2 | 7 |

$\sigma_{C \geq 6}(r)$ :

| A | B | C |
|---|---|---|
| 3 | 4 | 6 |
| 1 | 2 | 7 |
| 1 | 2 | 7 |

# Cartesian Product of Bags

---

- The **Cartesian Product** of bags  $r$  and  $s$  is the bag of tuples that can be formed by concatenating pairs of tuples, the first of which comes from  $r$  and the second from  $s$ . In symbols,  $r \times s = \{ t_1.t_2 \mid t_1 \in r \wedge t_2 \in s \}$ 
  - Each tuple of one relation is paired with each tuple of the other, regardless of whether it is a duplicate or not
  - If a tuple  $t_1$  appears  $m$  times in a relation  $r$ , and a tuple  $t_2$  appears  $n$  times in relation  $s$ , then tuple  $t_1.t_2$  appears  $m*n$  times in their bag-product,  $r \times s$

# Example

---

Bag r:

| A | B |
|---|---|
| 1 | 2 |
| 1 | 2 |

Bag s:

| B | C |
|---|---|
| 2 | 3 |
| 4 | 5 |
| 4 | 5 |

$r \times s$ :

| A | R.B | S.B | C |
|---|-----|-----|---|
| 1 | 2   | 2   | 3 |
| 1 | 2   | 2   | 3 |
| 1 | 2   | 4   | 5 |
| 1 | 2   | 4   | 5 |
| 1 | 2   | 4   | 5 |
| 1 | 2   | 4   | 5 |

# Join of Bags

---

- The bag join is computed in the same way as the standard join operation
- Duplicates are not eliminated in a bag join operation

Bag r:

| A | B |
|---|---|
| 1 | 2 |
| 1 | 2 |

Bag s:

| B | C |
|---|---|
| 2 | 3 |
| 4 | 5 |

$r \bowtie s$ :

| A | B | C |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 2 | 3 |

# Constraints on Relations

---

- RA offers a convenient way to express a wide variety of constraints, e.g., referential integrity and FD's.
- There are **two ways** to express constraints in RA
  1. If  $r$  is an expression in RA, then the constraint  $r = \emptyset$  says:  
*“ $r$  has no tuples, i.e., or  $r$  is empty”*
  2. If  $r$  and  $s$  are RA expressions, then the constraint  $r \subseteq s$  says:  
*“every tuple in (the result of)  $r$  is in (the result of)  $s$ ”*

These constraints hold also when  $r$  and  $s$  are bags.

# Constraints on Relations

---

- Note that these two types of constraints are not independent. Why?
  - The constraint  $r \subseteq s$  could also be written as
$$r - s = \emptyset$$

This follows from the definition of “−”, because  $r \subseteq s$  iff  $r - s = \emptyset$ , meaning that there is no tuple in  $r$  which is not in  $s$

# Referential Integrity Constraints

---

- Referential integrity in relational data model means:
    - if there is a value  $v$  in a tuple  $t$  in a relation  $r$ , then it is expected that  $v$  appears in a particular component (attribute) of some tuple  $s$  in relation  $s$
- E.g., if tuple  $(s, c, g)$  is in table ***takes***(*sid*, *cid*, *grade*), then there must be a **student** with  $\textcolor{green}{sid} = s$  and a **course** with  $\textcolor{green}{cid} = c$  such that  $s$  has taken  $c$
- IOW, the mentions of values  $s$  and  $c$  in ***takes*** “refers” to some values outside this relation, and these values **must** exist*

# Example

---

- Relation schemas:
  - Movie** (title, year, length, filmType)
  - StarsIn** (title, year, starName)
- Constraint:
  - the **title** and **year** of every movie that appears in relation **starsIn** *must* appear also in **movie**; otherwise there is a violation in referencing in **starsIn**
- Query in RA:
  - $\pi_{\text{title}, \text{year}}(\text{starsIn}) \subseteq \pi_{\text{title}, \text{year}}(\text{movie})$
  - or equivalently
  - $\pi_{\text{title}, \text{year}}(\text{starsIn}) - \pi_{\text{title}, \text{year}}(\text{movie}) = \emptyset$

# Functional Dependencies

---

- Any functional dependency  $X \rightarrow Y$  can be expressed as an expression in RA
- Example:  
Consider the relation schema:  
**Star** (name, address, gender, birthdate)
- How to express the FD: **name  $\rightarrow$  address** in RA?

# Functional Dependencies

---

- Relation schema:  
**Star (name, address, birthdate)**
- With the FD: **name → address**
- The **idea** is that if we construct all pairs of **star** tuples, we **must not** find a pair that agree on **name** but disagree on **address**
- To “construct” the pairs in RA, we use **Cartesian product**, and to find pairs that violate this FD, we use **selection**
- We are then ready to express this **FD** by equating the result to  $\emptyset$ , as follows...

# Example

Star:

| Name          | Address      | Birthdate |
|---------------|--------------|-----------|
| Carrie Fisher | 123 Maple    | 9/9/99    |
| Mark Hamill   | 456 Oak rd.  | 8/8/88    |
| Harrison Ford | 789 Palm rd. | 7/7/77    |

$\rho_{S1(name, address, birthdate)}(\text{star})$

| Name          | Address      | Birthdate |
|---------------|--------------|-----------|
| Carrie Fisher | 123 Maple    | 9/9/99    |
| Mark Hamill   | 456 Oak rd.  | 8/8/88    |
| Harrison Ford | 789 Palm rd. | 7/7/77    |

$\rho_{S2(name, address, birthdate)}(\text{star})$

| Name          | Address      | Birthdate |
|---------------|--------------|-----------|
| Carrie Fisher | 123 Maple    | 9/9/99    |
| Mark Hamill   | 456 Oak rd.  | 8/8/88    |
| Harrison Ford | 789 Palm rd. | 7/7/77    |

# Example

---

$s1 \times s2:$

| S1.Name       | S1.Address   | S1.Birthdate | S2.Name       | S2.Address   | S2.Birthdate |
|---------------|--------------|--------------|---------------|--------------|--------------|
| Carrie Fisher | 123 Maple    | 9/9/99       | Carrie Fisher | 123 Maple    | 9/9/99       |
| Carrie Fisher | 123 Maple    | 9/9/99       | Mark Hamill   | 456 Oak rd.  | 8/8/88       |
| Carrie Fisher | 123 Maple    | 9/9/99       | Harrison Ford | 789 Palm rd. | 7/7/77       |
| Mark Hamill   | 456 Oak rd.  | 8/8/88       | Carrie Fisher | 123 Maple    | 9/9/99       |
| Mark Hamill   | 456 Oak rd.  | 8/8/88       | Mark Hamill   | 456 Oak rd.  | 8/8/88       |
| Mark Hamill   | 456 Oak rd.  | 8/8/88       | Harrison Ford | 789 Palm rd. | 7/7/77       |
| Harrison Ford | 789 Palm rd. | 7/7/77       | Carrie Fisher | 123 Maple    | 9/9/99       |
| Harrison Ford | 789 Palm rd. | 7/7/77       | Mark Hamill   | 456 Oak rd.  | 8/8/88       |
| Harrison Ford | 789 Palm rd. | 7/7/77       | Harrison Ford | 789 Palm rd. | 7/7/77       |

$$\sigma_{S1.name=S2.name \text{ AND } S1.address \neq S2.address} (s1 \times s2) = \emptyset$$

# Functional Dependencies

---

- Relation schema:  
**Star (name, address, birthdate)**
- With the FD: **name → address**
- In RA:

$$\sigma_{S1.name=S2.name \text{ AND } S1.address \neq S2.address}(\rho_{S1}(star) \times \rho_{S2}(star)) = \emptyset$$

# Domain Constraints

---

- Relation schema:  
**Star (name, address, gender, birthdate)**
- How to express the following constraint?  
**Valid values for gender are 'F' and 'M'**
- In RA:
  - $\sigma_{\text{gender} \neq 'F' \text{ AND } \text{gender} \neq 'M'}(\text{star}) = \emptyset$
  - This is an example of ***domain constraints***

# Domain Constraints

---

- Relation schema:  
**Employee** (eid, name, address, salary)
- How to express the constraint:  
Maximum employee salaries is \$150,000
- In RA:
  - $\sigma_{\text{salary} > 150000}(\text{employee}) = \emptyset$

# "For All" Queries (1)

---

- Given the database schema:

Student(Sid, Sname, Addr)

Course(Cid, Cname, Credits)

Enrolled (Sid, Cid)

- Consider the query:

*“Find students enrolled in all the courses.”*

- A first attempt (below) fails!

$\pi_{\text{Sid}}(\text{Enrolled})$

- This RA query returns those students enrolled in some courses.
- So, how to correctly express “For All” types of queries?

# "For All" Queries (2)

---

- A *solution strategy* would be to:
  - First find the list of “all” students (**all guys**), from which we then subtract those who have not taken at least a course (**bad guys**)
- Then “**good guys**”, would be “all guys” from which we remove the “bad guys”, i.e.,

**Answer (Good guys) = All guys – Bad guys**

# "For All" Queries (3)

---

- Set of all students that we should consider:
  - All Courses :=  $\pi_{Cid}(\text{Course})$
  - All Students :=  $\pi_{Sid}(\text{Student})$
- Steps to find **students not enrolled in all courses**
  1. Create all possible “student-course” pairs:  
**All: Student-Course Pairs :=  $\pi_{Sid}(\text{Student}) \times \pi_{Cid}(\text{Course})$**
  2. Extract the “actual” student-course pairs from **Enrolled**
  3. Using 1 & 2, we then find students not enrolled in all courses:  
**Bad :  $\pi_{Sid}(\pi_{Sid}(\text{Student}) \times \pi_{Cid}(\text{Course})) - \text{Enrolled}$**
- **Answer:** All – Bad

# The Division Operation ( $\div$ )

---

- The previous query can be expressed in RA using the ***division*** operator  $\div$ 
  - Divide Enrolled by  $\pi_{Cid}(\text{Course})$   
that is,  $\text{Enrolled} \div \pi_{Cid}(\text{Course})$
  - Schema of the result is  $\{\text{Sid}, \text{Cid}\} - \{\text{Cid}\}$
- $R \div S$  requires that the attributes of  $S$  to be a subset of the attributes of  $R$ .
  - The schema of the output would be  $R - S$

# Example: Enrolled (student, sport)

---

Find students enrolled in all sports {Hockey, Football}.

Enrolled (Student, sport)

|     |          |
|-----|----------|
| Jim | Hockey   |
| Joe | Football |
| Jim | Football |
| Sue | Hockey   |

# Example: Enrolled(student, sport)

$$\pi_{student}(\text{Enrolled}) \times \pi_{sport}(\text{Enrolled}) - \text{Enrolled} =$$

|     |          |
|-----|----------|
| Jim | Hockey   |
| Jim | Football |
| Joe | Hockey   |
| Joe | Football |
| Sue | Hockey   |
| Sue | Football |

—

|     |          |
|-----|----------|
| Jim | Hockey   |
| Joe | Football |
| Jim | Football |
| Sue | Hockey   |

=

|     |          |
|-----|----------|
| Joe | Hockey   |
| Sue | Football |

$$\pi_{student}(\text{Enrolled}) - \pi_{student}(\pi_{student}(\text{Enrolled}) \times \pi_{sport}(\text{Enrolled})) - \text{Enrolled}$$

All

Bad

—

|     |
|-----|
| Jim |
| Joe |
| Sue |

= ? →

Jim is the only student enrolled in all sports

# Another Example

---

- $r \div s = \pi_{R-S}(r) - \pi_{R-S}(\pi_{R-S}(r) \times s - r)$
- Consider the following DB schema in a banking application:
  - Customer(cid, name)
  - Branch(bid, district)
  - Account(cid, bid)
- Query: "*Find the names of those customers who have an account in every branch in the Westmount area*"
- Solution?
  - $\pi_{name}(\text{Customer} \triangleright\!\!\!< \text{Account} \triangleright\!\!\!< (\sigma_{district = "Westmount"}(\text{Branch})))$ ?
- **No**, this query returns those customers who have an account at some branch in Westmount, but not necessarily at all such branches.

# Database:

Customer(cid, name), Branch(bid, district), Account(cid, bid)

---

- We can use the division operator  $\div$ 
  - Find all customer-branch pairs (cid, bid) for which customer (cid) has an account at branch (bid):  
 $\pi_{cid,bid}$  (customer  $\triangleright\triangleleft$  account)
  - Divide the above by bid's of all branches in Westmount  
 $\pi_{bid} (\sigma_{district = "Westmount"} (branch))$
  - That is:  $\pi_{name} ((customer \triangleright\triangleleft account) \div \pi_{bid} (\sigma_{district = "Westmount"} (branch)))$
  - The division “ $r \div s$ ” of relation r by s is defined as:  
 $r \div s = \pi_{R-S}(r) - \pi_{R-S}(\pi_{R-S}(r) \times s - r))$

# COMP353 Databases

---

*More on SQL Queries*

# SQL Queries: Review

---

- SQL query has a form

**SELECT** . . .

**FROM** . . .

**WHERE** . . . ;

- The **SELECT** clause says **which Attribute(s)** we are interested in
- The **FROM** clause says **which Relation(s)** we refer to
- The **WHERE** clause says **which Tuple(s)** we refer to

# Case Insensitivity

---

- SQL is case *insensitive*
- So, keyword FROM maybe written as:
  - FROM or
  - From or
  - FrOm
- Only in **strings**, SQL distinguishes between uppercase and the lowercase letters
  - So, the following are different strings:
    - 'FROM'
    - 'From'
    - 'FrOm'

# Select Clause

---

- In place of \* in the **SELECT** clause, we can put any attribute “we wish” to project on
- In the **SELECT** clause, we can also do **renaming**

```
SELECT title AS name, length AS duration
FROM Movie;
```

→ the structure of the query output “appears as”:   
(name, duration)

# Select Clause

---

- We can also use a *formula* in place of an attribute

```
SELECT title AS name, length/60 AS lengthInHours
FROM Movie;
```

➔ the structure of the output: (name, lengthInHours)

# Select Clause

---

- SQL even allows using a constant as an item in the **SELECT** clause, as shown below.

```
SELECT title AS name, length/60 AS length, 'hrs.' AS inHours
FROM Movie;
```

→ The structure of the output: (name, length, inHours)

|                    |      |      |
|--------------------|------|------|
| Gone with the wind | 1.98 | hrs. |
| King Kong          | 0.75 | hrs. |

- Why? To put some “useful” words into the output that SQL displays

# Comparison of Strings

---

- Two strings are equal if they have the same sequence of characters/symbols
- Strings are compared alphabetically
  - 'fodder' < 'foo'
  - 'bar' < 'bargain'
- WHERE R.A = T.B AND **s LIKE p**
  - s is an attribute of type *string* and p is a pattern; e.g.  
WHERE title LIKE 'Gone%'
  - “*Ordinary*” characters in p matches ordinary characters in s
  - What about “*Special*” characters in p: % , \_
  - “%” in p matches **any** sequence of zero or more characters in s
  - “\_” in p matches any **one** character in s

# Comparison of Strings

---

- Suppose we remember a movie “Star *something*”, and we do remember that “the something” has four letters

```
SELECT title
FROM Movie
WHERE title LIKE 'Star ____';
```

```
SELECT title
FROM Movie
WHERE title LIKE 'Star%';
```

# Comparison of Strings

---

What if the pattern p includes ', %, or \_?

- Find all movies with a possessive ( 's ) in their title

**LIKE '%s%'**

**SELECT title**

**FROM Movie**

**WHERE title LIKE '%''s%';**

- The convention is that two apostrophes " in a string represent one single apostrophe ('), and not the end of string

# Comparison of Strings

---

- What if **p** involves the *special characters* **%** or **\_**?
  - We should “escape” their special meaning using “some” **escape character**
  - SQL allows using **any** character as escape character
- **s LIKE 'x% %x%' ESCAPE 'x';**
  - Here, **X** is the escape character → x% means the character %, and not its usual meaning (the special character)
  - The pattern '**x% %x%**' matches strings: **%whatever%**

# Ordering the Output

---

- We may wish the output of a query to be displayed in some order. This could be done using the SQL clause:
  - **ORDER BY** <list of attributes>
- E.g., List Disney movies in 1990 by their **length**, shortest first, and then by the alphabetical order of the **titles**:

**SELECT \***

**FROM Movie**

**WHERE** studioName = 'Disney' **AND** year = 1990

**ORDER BY** length, title;

- Default ordering is **ASCending**, unless we use the **DESC** keyword
- Ties are broken by the “next” attribute in the **ORDER BY** list.

# Products and Joins

---

- SQL has a simple way to couple relations in a query
  - How? Simply list each relation in the **FROM** clause
- All the relations in the **FROM** clause are coupled through **Cartesian product**
- Then we can put conditions in the **WHERE** clause in order to get a desired kind of **join**

# Join (Example, Recall)

---

- Relation schemas:

- Movie** (title, year, length, filmType)

- Owns** (title, year, studioName)

- Query:

- Find** titles **and** lengths **of all movies produced by Disney**

- Query in SQL:

- SELECT** Movie.title, Movie.length

- FROM** Movie, Owns

- WHERE** Movie.title = Owns.title **AND** Movie.year = Owns.year

- AND** Owns.studioName = 'Disney';

# Union, Intersection, and Difference

---

- We can apply the common set operations of **union**, **intersection**, and **difference** to relations **R** and **S**, if they are *compatible*.
- When the output of two or more SQL queries are compatible, we may “combine” the queries using:
  - **UNION**
  - **INTERSECT**
  - **EXCEPT** (or **MINUS** in Oracle)

# Union, Intersection, and Difference

---

## ■ Relation schemas:

**Movie** (title, year, length, filmType)

**StarsIn** (title, year, starName)

## ■ Query:

Find titles and years of movies that appeared in either **Movie** or  
**StarsIn** relations

## ■ Query in SQL:

**SELECT** title, year

**FROM** Movie

**UNION**

**SELECT** title, year

**FROM** StarsIn;

# Union, Intersection, and Difference

---

- Relation schemas:

**Star**(name, address, gender, birthdate)

**Exec**(name, address, cert#, netWorth)

- Query:

Find names and addresses of all female movie stars who are also movie executives with a net worth of over \$10,000,000

- Query in SQL:

**SELECT** name, address

**FROM** Star

**WHERE** gender = 'F'

**INTERSECT**

**SELECT** name, address

**FROM** Exec

**WHERE** netWorth > 10000000;

# Union, Intersection, and Difference

---

- Relation schemas:

Star (name, address, gender, birthdate)

Exec (name, address, cert#, netWorth)

- Query:

Find names and addresses of movie stars who are **not** movie executives

- Query in SQL:

SELECT name, address

FROM Star

EXCEPT

//or MINUS in Oracle//

SELECT name, address

FROM Exec;

# Duplicate Elimination

---

*Note that in SQL:*

- The **union**, **intersection**, and **difference** operations normally eliminate duplicates (the set semantics)
- To retain duplicates, hence preventing duplicate elimination, we must use the keyword **ALL** after the operator **UNION**, **INTERSECT**, and **EXCEPT**
  - $R \text{ UNION ALL } S$  (*the only bag operation supported in Oracle*)
  - $R \text{ INTERSECT ALL } S$
  - $R \text{ EXCEPT ALL } S$

# Retaining Duplicates

---

## ■ $R \text{ UNION ALL } S$

- The bag of elements that are in  $R$ ,  $S$ , or in both. If  $R$  is a bag in which tuple  $t$  appears  $n$  times, and  $S$  is a bag in which  $t$  appears  $m$  times, then the number of occurrences of tuple  $t$  in bag  $R \cup S$  is  $n + m$

## ■ $R \text{ INTERSECT ALL } S$

- The bag of elements that are in both  $R$  and  $S$ . If  $R$  is a bag in which tuple  $t$  appears  $n$  times, and  $S$  is a bag in which  $t$  appears  $m$  times, then the number of occurrences of  $t$  in bag  $R \cap S$  is  $\min(n, m)$

## ■ $R \text{ EXCEPT ALL } S$

- The bag of elements that are in  $R$  but not in  $S$ . If  $R$  is a bag in which tuple  $t$  appears  $n$  times, and  $S$  is a bag in which  $t$  appears  $m$  times, then the number of occurrences of  $t$  in bag  $R - S$  is  $\max(0, n - m)$

# Retaining Duplicates in Union

---

- Relation schemas:

**Movie** (title, year, length, filmType)

**StarsIn** (title, year, starName)

- Query:

List the title and year of every movie that appears in **Movie** or  
**StarsIn**

- Query in SQL:

**SELECT** title, year

**FROM** Movie

**UNION ALL**

**SELECT** title, year

**FROM** StarsIn;

# **COMP353 Databases**

---

**More on SQL:**

**Nested Querie  
Views**

# Scalar Values

---

- An SQL query is an expression that evaluates to a **collection** of tuples, i.e., it produces a **relation/bag**
- This “collection” may have only one attribute
- It is also possible that there will be only one single value produced for that attribute
- If all these hold, then we say that the query produces a **scalar** value
  - **Scalar values** – example include simple values such as integers, reals, strings, dates, etc.

# Queries that Produce Scalar Values

---

- Relation schema:

**Movie**( title, year, length, filmType, studioName, producerC# )

- Query:

Find certificate number of the producer of “Star Wars”

- Query in SQL:

**SELECT** producerC#

**FROM** Movie

**WHERE** title = 'Star Wars';

*Assuming that we have only one such movie.*

# Subqueries

---

- Conditions in the **WHERE** clause may have comparisons that involve scalar values
- A SQL query can produce a scalar value
- If so, we can use such SELECT-FROM-WHERE expression, surrounded by parentheses, **as if it were a constant**
- **Subquery** – a query within a query  
The result of a SQL subquery is a collection (relation/bag)

# Example

---

- Relation schemas:

**Movie** (title, year, length, filmType, studioName, producerC#)  
**Exec** (name, address, cert#, netWorth)

- Query:

Find the name of the producer of “Star Wars”

- Query in SQL:

**SELECT** Exec.name

**FROM** Movie, Exec

**WHERE** Movie.title = 'Star Wars' **AND**

Movie.producerC# = Exec.cert#;

# Example

---

- Relation schemas:

**Movie** (title, year, length, filmType, studioName, **producerC#**)

**Exec** (**name**, address, cert#, netWorth)

- Query:

Find the **name** of the producer of “Star Wars”

- Query with Subquery:

**SELECT** **name**

**FROM** Exec

**WHERE** cert# = ( **SELECT** producerC#

**FROM** Movie

**WHERE** title = 'Star Wars' );

# Conditions Involving Relations

---

- There are a number of SQL checks/conditions that can be done on a relation **R** and produce a **boolean** value
- These conditions can be **negated** by putting a **NOT** before them
- Typically, **R** above is the result of an SQL subquery, shown as: **(R)**
- If such a condition involves a scalar value **s** or a tuple, we should make sure its type matches **R**.

# Conditions Involving Relations

---

- “**EXISTS ( $R$ )**” is a condition that is true iff  $R$  is not empty
- “ **$s$  IN ( $R$ )**” is true iff  $s$  is equal to **one** of the values in  $R$ 
  - “ **$s$  NOT IN ( $R$ )**” is true iff  $s$  is not equal to any value in  $R$
- “ **$s > \text{ALL } (R)$** ” is true iff  $s$  is greater than **every** value in  $R$ 
  - “ $>$ ” could be replaced by other operators with the analogous meaning
  - **Note:** “ **$s <> \text{ALL } (R)$** ” is the same as “ **$s$  NOT IN  $R$** ”
- “ **$s > \text{ANY } (R)$** ” is true iff  $s$  is  $>$  **at least one** value in  $R$ 
  - “ $>$ ” could be replaced by any of the other 5 comparison operators with the analogous meaning
  - **Note:** “ **$s = \text{ANY } (R)$** ” is the same as “ **$s$  IN  $R$** ”

# Conditions Involving Tuples

---

- A tuple in SQL is represented by a parenthesized list of scalar values; the concept “tuple” can be viewed as an *extension* of the concept of scalar;
  - $(123, \text{'foo'})$
- Mixing of **constants** and **attributes** is also permitted in tuples
  - $(123, \text{Movie.title})$
- If a tuple  $t$  has the same number of components as a relation  $R$ , then it makes sense to compare  $t$  and  $R$  like:
  - $t \text{ IN } (R)$  -- this is true iff  $t$  is in  $R$
  - $t <> \text{ANY } (R)$  -- this is true  $R$  includes a tuple other than  $t$

# Example

---

- Relation schemas:

**Movie** (title, year, length, filmType, studioName, producerC#)

**Exec** (name, address, cert#, netWorth)

**StarsIn** (title, year, starName)

- Query: Find the names of the producers of Harrison Ford's movies

- Query in SQL:

```
SELECT name
```

```
FROM Exec
```

```
WHERE cert# IN (SELECT producerC#
```

```
 FROM Movie
```

```
 WHERE (title, year) IN (SELECT title, year
```

```
 FROM StarsIn
```

```
 WHERE starName = 'Harrison Ford'));
```

# Example

---

- Relation schemas:

**Movie**(title, year, length, filmType, studioName, producerC#)

**Exec**(name, address, cert#, netWorth)

**StarsIn**(title, year, starName)

- Query: Find names of the producers of Harrison Ford's movies

- Query in SQL:

**SELECT** Exec.name

**FROM** Exec, Movie, StarsIn

**WHERE** Exec.cert# = Movie.producerC# **AND**

    Movie.title = StarsIn.title **AND**

    Movie.year = StarsIn.year **AND**

    starName = 'Harrison Ford';

# Correlated Subqueries

---

- Simple subqueries can be evaluated once and the result be used in a higher level (calling) query
- A more complex use of nested subquery requires the subquery to be evaluated many times, once for each assignment of a value (to some term in the subquery) that comes from a tuple variable in the calling query
- A subquery of this type is called ***correlated subquery***

# Correlated Subqueries

---

- Relation schema:

**Movie**(title, year, length, filmType, studioName, producerC#)

- Query:

Find movie titles that appear more than once

- Query in SQL:

**SELECT** title

**FROM** Movie Old

**WHERE** year < ANY (**SELECT** year

**FROM** Movie

**WHERE** title = Old.title);

Note the **scopes** of the variables in this query.

# Correlated Subqueries

---

- Query in SQL

**SELECT title**

**FROM Movie Old**

**WHERE year < ANY (SELECT year**

**FROM Movie**

**WHERE title = Old.title);**

- The condition in the outer **WHERE** is true only if there is a movie with same title as *Old.title* that has a **later** year
  - ➔ The query will produce a title **one fewer times** than there are movies with that title
- What would be the result if we used “**<>**”, instead of “**<**” ?
  - ➔ For a movie title appearing 3 times, we would get 3 copies of the title in the output

# Views

---

- **View** is a table/relation defined in a database but has no tuples explicitly stored for it in the database but rather computed, when needed, from the **view definition**
- The view mechanism provides support for:
  - Logical data independence:
    - Views can be used to define relations in the external schema that mask, from the applications/users, changes in the *conceptual database schema*
    - If the schema of a relation is changed, we can define a view with the old schema so that applications that use the old schema can continue using it
  - Security:
    - Views can be used to restricts the users access only the information they are allowed to “see and operate on”

# Views

---

- Relation schema:  
**Movie(title, year, length, filmType, **studioName**, producerC#)**
- View:  
**Create the Paramount's movies (title and year)**
- View in SQL:  
**CREATE VIEW ParamountMovie AS  
SELECT title, year  
FROM Movie  
WHERE studioName = 'Paramount';**

# Views

---

- A view can be used in defining new queries/views in exactly the same way as an explicitly stored table may be used
- Example to query the (virtual) relation ParamountMovie

```
SELECT title
FROM ParamountMovie
WHERE year = 1979;
```

- This query is translated, by the query processor, into:

```
SELECT title
FROM Movie
WHERE studioName = 'Paramount' AND year = 1979;
```

# Views

---

- Relation schema:

**ParamountMovie** (title, year )

**StarsIn**(title, year, starName)

- Query:

List the stars of the movies made by Paramount.

- Query in SQL

**SELECT DISTINCT StarsIn.starName**

**FROM ParamountMovie, StarsIn**

**WHERE ParamountMovie.title = StarsIn.title AND**

**ParamountMovie.year = StarsIn.year;**

# Views

---

- Relation schema:

**Movie** (title, year, length, filmType, studioName, **producerC#**)  
**Exec** (name, address, cert#, netWorth)

- View:

Define a view of Movie (titles and executives/producers)

- View in SQL:

```
CREATE VIEW MovieProd AS
SELECT Movie.title, Exec.name
FROM Movie, Exec
WHERE Movie.producerC# = Exec.cert#;
```

# Views

---

- Relation schema:

**MovieProd** (title, name)

- Query:

Find the name of the producer of 'Gone With the Wind'?

- Query in SQL:

**SELECT** name

**FROM** MovieProd

**WHERE** title = 'Gone With the Wind';

# Views

---

- Renaming attributes used in view definitions
  - We can give new names to view attributes rather than using the names that come out of query defining the view

- Example:

```
CREATE VIEW MovieProd (MovieTitle, ProducerName) AS
SELECT Movie.title, Exec.name
FROM Movie, Exec
WHERE Movie.producerC# = Exec.cert#;
```

# Views

---

- Relation schema:

**MovieProd** (MovieTitle, ProducerName)

- Query:

Find the name of the producer of 'Gone With the Wind'?

- Query in SQL:

**SELECT** ProducerName

**FROM** MovieProd

**WHERE** MovieTitle = 'Gone With the Wind';

# Updating Views?

---

- We saw that a view can appear in a query in exactly the same way as a “base” table may appear.
- What about modifications/updates?
- What does it mean to update a view?
  - Translate modification of the view to the corresponding modification on the base tables used in the view definition
- Should we allow updates on views?
  - Yes, in principle, but some problems may arise
- Some “simple” views can be updated
  - Such views are called **updatable views**
- Many views **cannot** be updated
  - This is due to the so called **view-update anomaly**

# Insertion into Views?

---

- Relation schema:

**Movie**(title, year, length, filmType, **studioName**, producerC#)

- View: Recall the definition of **ParamountMovie**

**CREATE VIEW** ParamountMovie **AS**

**SELECT** title, year

**FROM** Movie

**WHERE** studioName = 'Paramount';

- Update statement:

**INSERT INTO** ParamountMovie (**title, year**) **VALUES**('KK', 2002);

- Result: the following tuple being added to Movie

('KK', 2002, NULL, NULL, **NULL**, **NULL**) **What's the problem?**

# Insertion into Views?

---

- Relation schema:

**Movie**(title, year, length, filmType, studioName, producerC#)

- An updatable view:

**CREATE VIEW** ParamountMovie **AS**

**SELECT** title, year, **studioName**

**FROM** Movie

**WHERE** studioName = 'Paramount';

- Update statement:

**INSERT INTO** ParamountMovie **VALUES**('KK',2002,'Paramount');

- Result: the following tuple is being added to Movie

('KK', 2002, NULL, NULL, 'Paramount', NULL) Problem solved!

# Insertion into Views?

- Relation schemas:

**Movie**(title, year, length, filmType, studioName, **producerC#** )  
**Exec**(name, address, cert#, netWorth)

- View in SQL:

```
CREATE VIEW MovieProd AS
SELECT Movie.title, Exec.name
FROM Movie, Exec
WHERE Movie.producerC# = Exec.cert#;
```

- Update statement

**INSERT INTO** MovieProd (**title**,**name**) **VALUES**('The Movie', 'J. Smith');

- Result: these tuples are added to the corresponding relations:

**Movie**('The Movie', **NULL**, **NULL**, **NULL**, **NULL**, **NULL**)

**Exec**('J. Smith', **NULL**, **NULL**, **NULL**)      Problems? The insertion command will fail !

# Deletion from Views?

---

- Relation schema:

**Movie**(title, year, length, filmType, studioName, producerC#)

- View: Recall the definition :

**CREATE VIEW** ParamountMovie **AS**

**SELECT** title, year, studioName

**FROM** Movie

**WHERE** studioName = 'Paramount';

- Delete statement:

**DELETE FROM** ParamountMovie **WHERE** title **LIKE** '%K%';

- Translated query:

**DELETE FROM** Movie

**WHERE** studioName = 'Paramount' **AND** title **LIKE** '%K%';

# Updating Views?

---

- Relation schema:

**Movie**(title, year, length, filmType, studioName, producerC#)

- View:

**CREATE VIEW** ParamountMovie **AS**

**SELECT** title, year, studioName

**FROM** Movie

**WHERE** studioName = 'Paramount';

- The view update statement:

**UPDATE** ParamountMovie **SET** year = 1797 **WHERE** title = 'KK';

- We may drop a view: **DROP VIEW** ParamountMovie;

# Updating Views?

---

- Recall: updating views includes insertion, deletion, and changing data
- SQL provides a formal definition of when modifications to a view are permitted
- Roughly, this is permitted when the view is defined by selecting some attributes from **one** relation **R**, which could be an “updatable” view itself
  - The list in the **SELECT** clause includes “enough” attributes that for every tuple inserted into the view, the tuple inserted into the base relation will “yield” the inserted tuple of the view
  - The **NOT NULL** constraints on the base table will not be violated
  - The view definition uses **SELECT** (but not **SELECT DISTINCT**)
  - The **WHERE** clause does not involve **R** in a subquery

# **COMP353 Databases**

---

**More on SQL:**

**Null value**

**Triggers**

# Domains

---

- SQL allows user defined data types - **domains**
- We can define a domain as follows:
  - The keyword ***CREATE DOMAIN***
  - The name of the domain
  - The keyword **AS**
  - Type description
  - Optional default value, constraints
- Example

**CREATE DOMAIN <name> AS <type description>**

# Domains

---

- To create a domain:
  - **CREATE DOMAIN MovieDomain AS VARCHAR(50);**

- Example:

```
CREATE TABLE Movie (
 title MovieDomain,
 year DATE,

);
```

# Domains

---

- To create a domain with default value:
  - **CREATE DOMAIN MovieDomain AS VARCHAR(50)  
DEFAULT 'unknown';**
- To change the default for a domain:
  - **ALTER DOMAIN MovieDomain SET DEFAULT 'no such title';**
- To delete a domain definition:
  - **DROP DOMAIN MovieDomain;**

# NULLs

---

- We use NULL in place of a value in a tuple's component when:
  - The value is *unknown* -- don't know
  - The value is *inapplicable* -- NA
  - The exact value *does not matter* -- don't care
- There could be many reasons why a value is not present in a relation, e.g., when *inserting* a tuple into a relation, we don't have/wish to specify all the values for the attributes.

# Arithmetic operations on NULLs

---

- Result of an arithmetic operator, when at least one of the operands has a value of NULL, is NULL
- Example:
  - Suppose the value of attribute x is NULL
    - ➔ The value  $x+3$  is also NULL
  - Note however that NULL is not a constant!
    - ➔  $\text{NULL} + 3$  is *illegal*

# Arithmetic operations on NULLs

---

- Some key laws in math fail to hold with NULLs
- Suppose  $x$  is an attribute with numeric value
  - Example 1:
    - We know that  $x * 0 = 0$ , but
    - If  $x$  is NULL →  $x * 0$  is NULL
  - Example 2:
    - We also know that  $x - x = 0$ , but
    - If  $x$  is NULL →  $x - x$  is NULL

# Comparison operations on NULLs

---

- The result of a comparison is “usually” **TRUE** or **FALSE**
- That is, 2 possible values or **2-valued logic**
- Comparisons involving NULLs give rise to a **3rd** truth value, **UNKNOWN**, and hence we are dealing with a 3-valued logic
- In this case, the 3 possible values are **true**, **false**, **unknown**

# 3-Valued Logic

---

- We may assume that:

- **TRUE** = 1
- **FALSE** = 0
- **UNKNOWN** = 1/2

- Then:

- $x \text{ AND } y = \min(x, y)$
- $x \text{ OR } y = \max(x, y)$
- $\text{NOT } x = 1 - x$

# Truth table for 3-Valued Logic

| X       | Y       | X AND Y | X OR Y  | NOT X   |
|---------|---------|---------|---------|---------|
| TRUE    | TRUE    | TRUE    | TRUE    | FALSE   |
| TRUE    | UNKNOWN | UNKNOWN | TRUE    | FALSE   |
| TRUE    | FALSE   | FALSE   | TRUE    | FALSE   |
| UNKNOWN | TRUE    | UNKNOWN | TRUE    | UNKNOWN |
| UNKNOWN | UNKNOWN | UNKNOWN | UNKNOWN | UNKNOWN |
| UNKNOWN | FALSE   | FALSE   | UNKNOWN | UNKNOWN |
| FALSE   | TRUE    | FALSE   | TRUE    | TRUE    |
| FALSE   | UNKNOWN | FALSE   | UNKNOWN | TRUE    |
| FALSE   | FALSE   | FALSE   | FALSE   | TRUE    |

# 3-Valued Logic

---

- Some key laws in logic fail to hold with NULLs
- Example:
  - Law of the excluded middle
    - $x \text{ OR NOT } x = \text{TRUE}$
  - For 3-valued logic
    - if  $x = \text{UNKNOWN} \rightarrow x \text{ OR (NOT } x) = \max(1/2, (1 - 1/2)) = 1/2 = \text{UNKNOWN} \neq (\text{TRUE})$
    - Note: Do not treat NULL as a “value”  
(e.g., see the next example)

# Example

---

- Relation schema:

**Movie** ( title, year, length, filmType, studioName, producerC#)

- Consider query:

```
SELECT *
FROM Movie
WHERE length <= 120 OR length > 120;
```

- Don't we expect to get a copy of the **Movie** relation?
- Yes, if there is no Movie tuple whose length is NULL
- This query returns a subset of **Movie** tuples in general.  
It returns only each **Movie** tuple whose length is not NULL.

# Example

---

- Read the box on page 254 in the textbook for some rules on NULLs
- The value NULL is ignored in any aggregation.
  - Query:  
**SELECT COUNT(A) FROM R;**  
returns the number of non-null values under attribute A in R.
- Query:  
**SELECT COUNT(\*) FROM R;**  
returns the number of tuples in R.
- NULL is treated as an ordinary value in a “group by” attribute.
  - Query:  
**SELECT A, AVG(B) FROM R GROUP BY A;**  
produces a tuple for each distinct value A, including the null, if exists.

# Joins in SQL

---

- How to express the Cartesian product in SQL?
  - List the relation names in the **FROM** clause
- How do we express various joins in SQL?
  - Follow the Cartesian Product with conditions in the **WHERE** clause for the desired join

# Joins in SQL2

---

- In SQL2, there are other forms for expressing  $\times$  and  $\bowtie$
  - Cartesian Product of Movie and StarsIn
    - Movie **CROSS JOIN** StarsIn; (Movie  $\times$  StarsIn, in RA)
  - Theta- (or equi-) join of Movie and StarsIn
    - Movie **JOIN** StarsIn **ON**  
**StarsIn.title = Movie.title AND StarsIn.year = Movie.year;**
- Note:** The result may have some redundant columns;  
We can use the above expression as a subquery in a **FROM** clause and use **SELECT** to remove these undesired attributes
- Natural join of Movie and StarsIn
    - Movie **NATURAL JOIN** StarsIn;

# Example

---

- Relation schemas:

**Exec**(name, address, cert#, netWorth)

**Star** (name, address, gender, birthdate)

- Query:

Find all info on all stars who are also movie executives

- Query in SQL:

**SELECT \***

**FROM Star NATURAL JOIN Exec;**

- The join expression appears in a **FROM** clause
- Parenthesized **SELECT-FROM-WHERE** is also allowed in a **FROM** clause

# Example

---

- Relation schemas:

- Exec(name, address, cert#, netWorth)**

- Star (name, address, gender, birthdate)**

- Query: **Find “all” information on all stars**

- Query in SQL:

- SELECT \***

- FROM Star;**

- This query does not return all information on stars who are also movie executives

- Query in SQL

- SELECT \***

- FROM Star NATURAL JOIN Exec ;**

- This query does NOT return stars who are not movie executives

# Outer joins

---

- $R$  OUTER JOIN  $S$  -- computes the join of  $R$  and  $S$  with **dangling** tuples padded with **NULLs**
- A tuple in  $R$  is **dangling** if it doesn't join with any tuple in  $S$
- Outer join could be
  - **FULL OUTER JOIN**
    - It pads, with nulls, the dangling tuples of  $R$  and  $S$
  - **LEFT OUTER JOIN**
    - It pads dangling tuples of  $R$  only
  - **RIGHT OUTER JOIN**
    - It pads dangling tuples of  $S$  only

# Example

Instance R:

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |

Instance S:

| B | C |
|---|---|
| 2 | 5 |
| 2 | 6 |
| 7 | 8 |

**SELECT \* FROM R NATURAL FULL OUTER JOIN S:**

| A    | B | C    |
|------|---|------|
| 1    | 2 | 5    |
| 1    | 2 | 6    |
| 2    | 3 | NULL |
| NULL | 7 | 8    |

# Example

Instance R:

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |

Instance S:

| B | C |
|---|---|
| 2 | 5 |
| 2 | 6 |
| 7 | 8 |

R NATURAL LEFT OUTER JOIN S:

| A | B | C    |
|---|---|------|
| 1 | 2 | 5    |
| 1 | 2 | 6    |
| 2 | 3 | NULL |

# Example

Instance R:

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |

Instance S:

| B | C |
|---|---|
| 2 | 5 |
| 2 | 6 |
| 7 | 8 |

R NATURAL RIGHT OUTER JOIN S:

| A    | B | C |
|------|---|---|
| 1    | 2 | 5 |
| 1    | 2 | 6 |
| NULL | 7 | 8 |

# Outer joins in SQL2

---

$R$  [**NATURAL**] [**LEFT** | **RIGHT** | **FULL**] **OUTER JOIN**  $S$  [**ON** ...]

- Can do either a natural join  $\bowtie$  or a theta-join  $\bowtie_{\theta}$
- Use **NATURAL** for  $\bowtie$ , and use **ON** for theta-outer join, but not both (NATURAL and ON cannot be used together)
- Can use any one of **LEFT**, **RIGHT**, or **FULL**
- Examples:
  - $R$  **NATURAL FULL OUTER JOIN**  $S$ ;
  - *Star NATURAL LEFT OUTER JOIN Exec*; (for slide #17)
  - Movie **RIGHT OUTER JOIN** StarsIn **ON**  
StarsIn.title = Movie.title **AND** StarsIn.year = Movie.year;

# Constraints and Triggers (Chapter 7)

---

- SQL provides a variety of ways for expressing *integrity constraints* as part of the database schema
- Constraints' checking, in essence, provide users with more control over the database content
- An *active* element is a statement that we write once, store it in the database, and “expect” it to be executed at “appropriate” times
- The *time of action* might be when certain events occur (e.g., insertion of a tuple into a relation or any change(s) made to the database so that certain condition(s) becomes true

# Constraints

---

- Declaration of primary keys
- Foreign key constraints (also see referential integrity constraints)
  - E.g., if in relation **StarsIn**, it says that a star has a role in a movie **m**, then there should be a movie tuple **m** in **Movie**
- Constraints on attributes, tuples, and relations
- SQL2 Assertions = global/general constraints (inter-relations)
  - Not supported in Oracle
- Triggers
  - are substitutes for “general” assertions
- SQL3 triggers and assertions

# Primary Key

---

- A key constraint is declared by the DDL command **CREATE TABLE**
- Can use the keywords **PRIMARY KEY** or **UNIQUE**
  - Oracle treats them as synonyms
  - A table can only have one primary key but any number of "unique" declarations
- Two ways to declare a primary key in the **CREATE TABLE** statement:
  - After an attribute type, if the attribute is a key by itself
  - As a separate line, for any number of attributes forming the key:

**PRIMARY KEY ( list of attribute(s) )**

- Should use the second way if the key is not a singleton

# Primary Key

---

## ■ Example:

```
CREATE TABLE Star(
 name CHAR(30) PRIMARY KEY,
 address VARCHAR(255),
 gender CHAR(1),
 birthdate DATE);
```

Three consequences of declaring a primary key:

1. Repeated values of the key attributes will not be allowed  
Any violation will be rejected/failed by the DBMS
2. NULLs are not allowed for the key attribute(s)
3. (Another possible consequence) Creating an index on the primary key, or alternatively keeping the table sorted on the key attribute(s).

# Primary Key

---

- Example:

```
CREATE TABLE Star (
 name CHAR(30) UNIQUE,
 address VARCHAR(255) UNIQUE,
 gender CHAR(1),
 birthdate DATE);
```

This is to say: *no two movie stars have the same address*

# Primary Key

---

## ■ Example:

```
CREATE TABLE Star(
 name CHAR(30),
 address VARCHAR(255),
 gender CHAR(1),
 birthdate DATE,
 UNIQUE (name)
);
```

Note the distinction between **UNIQUE** and **PRIMARY KEY**:  
When using **UNIQUE**, NULLs are allowed for (all or some of) the key attribute(s).

# Primary Key

---

## ■ Example:

```
CREATE TABLE Movie (
 title CHAR(20),
 year INT,
 length INT,
 PRIMARY KEY(title, year)
);
```

Here, the key declaration must appear on a separate line, since the key consists of more than 1 attribute (title, year)

# Foreign Keys

---

- Referential integrity constraints:

Intuitively, values for certain attributes must “make sense”

- That is, every non-Null value in attribute **A** of relation **R** must appear in attribute **B** of relation **S** -- **Inclusion dependency**
- In SQL, we declare an attribute (or a set K of attributes) in a relation (**R**) to be a *foreign key*, if they reference/point to some attribute(s) G of some related relation **S**.

That is,  $t[K] = s[G]$  if tuple  $t$  in **R** refers to tuple  $s$  in **S**

- The set of attributes **G** must be declared as the *primary key* of **S**
- The notion of referential integrity constraint is the one that connects or relates tuples in different relations.

# Foreign Keys

---

- Two ways to declare foreign keys:
  - If the foreign key is a single attribute A, we may write the following after the attribute *name* and its *type*:
    - (1) **REFERENCES** <referenced-table> (A)
  - When the foreign key includes more than one attribute, write it as a separate line in the **CREATE TABLE** declaration:
    - (2) **FOREIGN KEY** (<attributes>) **REFERENCES** <table> (<attributes>)
      - Use form (2) if the foreign key includes 2 or more attributes

# Foreign Keys

---

## ■ Relation Schemas:

**Exec**(eName, address, cert#, netWorth)

**Studio** (sName, address,  presidentC#)

Here, Studio(presidentC#) refers to Exec(cert#).

```
CREATE TABLE Studio(
 sName CHAR(30) PRIMARY KEY,
 address VARCHAR(255),
 presidentC# INT REFERENCES Exec(cert#)
)
```

# Foreign Keys

---

## ■ Relation Schemas:

**Exec**(eName, address, cert#, netWorth)

**Studio** (sName, address, presidentC#)

**CREATE TABLE** Studio(

sName **CHAR**(30) **PRIMARY KEY**,

address **VARCHAR**(255),

presidentC# **INT**,

**FOREIGN KEY** presidentC# **REFERENCES** Exec(cert#);

In either declaration form, when a value  $v$  appears for presidentC# in a Studio tuple,  $v$  MUST already exist for an Exec tuple.

*Note: An exception to this requirement is when  $v$  is NULL.*

# Maintaining Referential Integrity

---

**Exec**(eName, address, cert#, netWorth)

**Studio** (sName, address, presidentC#)

- How to maintain referential integrity when the database is modified?
- Possible situations **violating** foreign key constraints:
  - Insert:
    - 1. Insert a new Studio tuple with presidentC# that is not cert# of any Exec tuple
  - Update:
    - 2. Update a Studio tuple to change its presidentC# to a non-Null value that is not the cert# of any tuple in table Exec
    - 3. Update an Exec tuple **e** that changes **e.cert#**, when the old cert# was presidentC# of some Studio tuple **s** (i.e., **e.cert#** is referenced by **s**)
  - Delete:
    - 4. Delete an Exec tuple when its cert# is presidentC# of 1 or more Studio tuples  
Recall that cert# is the key of the Exec table

# Maintaining Referential Integrity

---

- There are *three* policies, when there is a transaction that violates a referential integrity:
  - The **reject** policy (*default*)  
The system will reject any transaction violating referential integrity constraints -- a run-time error will be generated and the database state will *not* change.
  - If we update (3) or delete (4) a referenced item:
    - The **cascade** policy: changes to the referenced attributes are “mimicked” at the foreign key (e.g. presidentC#)
    - The “**set null**” policy: set the referencing attribute to NULL (e.g., presidentC# in Studio)

# Maintaining Referential Integrity

---

**Exec**(eName, address, cert#, netWorth)

**Studio** (sName, address, presidentC#)

**Example:** For the (default) reject policy, the system will **reject** the following modifications/transactions:

- Insert a new Studio tuple with presidentC# that is not in any Exec tuple
- Update a Studio tuple to change the presidentC# to a non-null value that is not in any Exec tuple
- Update an Exec tuple to change cert#, when the old cert# value is presidentC# in some Studio tuple(s)
- Delete an Exec tuple when its cert# is presidentC# in some Studio tuple(s)

# Maintaining Referential Integrity

---

**Exec**(eName, address, cert#, netWorth)

**Studio** (sName, address, presidentC#)

- The **cascade policy**:
  - In case of **deleting (updating)** a tuple in a referenced relation (Exec), the DBMS will delete (update) the referencing tuple(s) in Studio
- Example:
  - The system will cascade the following modifications:
    - Update Exec tuple to change cert#, when its cert# is presidentC# in some Studio tuple
      - The system will update presidentC# in corresponding Studio tuple(s)
    - Delete an Exec tuple when its cert# is presidentC# in some Studio tuple
      - The system will delete corresponding tuple(s) in Studio

# Maintaining Referential Integrity

---

- The set-null policy:
  - In case of **deleting** or **updating** a tuple in a referenced relation (Exec), the DBMS will **set to NULL** the corresponding values in the referencing tuples (in Studio)
- Example:
  - The system will do the following modifications:
    - Update Exec tuple to change cert#, when its cert# is presidentC# in some Studio tuple
      - The DBMS will set to NULL presidentC# in corresponding Studio tuple(s)
    - Delete an Exec tuple when its cert# is presidentC# in some Studio tuple(s)
      - The DBMS will set to NULL presidentC# in the corresponding Studio tuple(s)

# Selecting a Policy

---

- Reaction options/policies may be chosen (by **?**) for deletes and updates, in an *independent* way
- **ON [DELETE | UPDATE] [CASCADE | SET NULL]**

# Selecting a Policy

---

## ■ Example:

```
CREATE TABLE Studio(
 name CHAR(30) PRIMARY KEY,
 address VARCHAR(255),
 presidentC# INT,
 FOREIGN KEY presidentC# REFERENCES Exec(cert#)
 ON DELETE SET NULL (or SET DEFAULT)
 ON UPDATE CASCADE
);
```

# Selecting a Policy

---

- "Correct" or "right" policy is a design decision
- Example:
  - If a studio president retires (and its tuple gets deleted), the studio should exist with a NULL value for the president and not be deleted along with the president
  - If the certificate number cert# of a studio president was changed (by an update), it should be updated in all corresponding places (presidentC# in Studio, in our case); *we should not lose the information on who is the current president of a given studio*

# Not-Null Constraints

---

- We can assert that the value of an attribute may not be NULL
- Example:

**presidentC# INT REFERENCES Exec(cert#) NOT NULL**

## Two consequences:

1. We can't insert a tuple into Studio by just specifying name and address; *the value presidentC# must also be present.*
2. We can't use the “set-null” policy to fix foreign key violations by changing presidentC# to be NULL

# Attribute-Based Checks

---

- Aside from “referencing”, more complex constraints can be attached to an attribute declaration, followed by  
**CHECK (condition-on-attribute)**
- That is, we want the condition to hold for that attribute in every tuple in the relation
- Condition may involve the checked attribute
- Other attributes and relations may be involved, but only in subqueries (see slide #46)
  - **Oracle** : No subqueries allowed in the **condition**

# Attribute-Based Checks

---

- Example:

```
CREATE TABLE Studio(
 name CHAR(30) PRIMARY KEY,
 address VARCHAR(255),
 presidentC# INT
 CHECK (presidentC# >= 100000)
)
```

This requires certificate numbers to be at least 6 digits.

# Attribute-Based Checks

---

When an attribute-based check/condition is (not) done?

- The condition is checked
  - When the associated attribute changes, i.e., an insert or update occurs
- The condition is **NOT** checked
  - When the relations involved in the subquery of the condition are changed; see the next slide

# Attribute-Based Checks

---

- Example:

```
CREATE TABLE Studio(
```

```
 name CHAR(30) PRIMARY KEY,
```

```
 address VARCHAR(255),
```

```
 presidentC# INT
```

```
 CHECK (presidentC# IN (SELECT cert# FROM Exec))
```

```
);
```

- Is this check the same as a foreign-key constraint?

- Not really! The above check is done only when we insert a tuple in **Studio** or change the presidentC# in an existing tuple in **Studio**, not when deleting or update a tuple in **Exec**

# Attribute-Based Checks

---

- Example:

```
CREATE TABLE Star(
 name CHAR(30) PRIMARY KEY,
 address VARCHAR(255),
 birthdate DATE,
 gender CHAR(1) CHECK (gender IN ('F', 'M'))
)
```

The above condition uses an explicit set/relation with two tuples, providing possible values for attribute *gender*

# Tuple-Based Checks

---

- We restrict some components of the tuples of a relation by a **tuple-based check**
- Tuple-based check must appear on a separate element/part in a table declaration command
- Format:  
**CHECK** (condition)
  - Condition may involve any attribute(s) of the table
  - Other attributes and tables may be involved, but only in subqueries
    - **Oracle**: Does not support subqueries in condition

# Tuple-Based Checks

---

## ■ Example:

```
CREATE TABLE Star(
 name CHAR(30) PRIMARY KEY,
 address VARCHAR(255),
 gender CHAR(1),
 birthdate DATE,
 CHECK (gender = 'F' OR name NOT LIKE 'Ms.%')
);
```

This constraint says that if a “star is male” (M), then “his name must not begin with Ms.” ( $\sim L$ ), i.e.,  $M \rightarrow \sim L$ , or  $\sim M \vee \sim L$

# Modifications of Constraints

---

- We can add, modify, or delete constraints at any time
- Give names to constraints so we can “refer” to them later

Examples of *defining* constraints:

name **CHAR(30)** **CONSTRAINT** Name-Is-Key **PRIMARY KEY**,  
OR

**CONSTRAINT** RightTitle **CHECK** (gender = 'F' **OR** name **NOT LIKE** 'Ms.%')

Examples of *modifying* (deleting/adding) constraints:

**ALTER TABLE** Star **DROP CONSTRAINT** Name-Is-Key;

OR

**ALTER TABLE** Star **ADD CONSTRAINT** NamelsKey **PRIMARY KEY** (*name*);

# Assertions

---

- Assertions, also called “*general constraints*”, are boolean-valued SQL expressions that must *always* be satisfied
- Sometimes we need a constraint that involves a relation as a whole or part of the database schema  
(This is not supported in Oracle)
- Assertions are checked when a specified relation changes
- Syntax:

**CREATE ASSERTION < assertion-name > CHECK (< condition >);**

**Note:** Unlike attribute/tuple based checks, assertions are defined outside table declarations.

# Assertions

---

- Relation schemas:

**Exec (name, address, cert#, netWorth)**

**Studio (name, address, presidentC#)**

- Assertion in SQL:

**CREATE ASSERTION RichPresident CHECK**

**(NOT EXIST (SELECT \***

**FROM Studio, Exec**

**WHERE Studio.presidentC# = Exec.cert# AND**

**Exec.netWorth < 10000000**

**)**

**);**

- This constraint is checked when **Studio** and/or **Exec** tables change

# Triggers

---

- Also called **event-condition-action (ECA)** rules
- *Event*
  - DB transactions (Insertion, Deletion, Update)
- *Condition*
  - A condition to check if a trigger applies
- *Action*
  - One or more SQL statements to be executed if a triggering event occurred and the condition(s) holds.

# Triggers

---

- Triggers are not supported in SQL2
- Oracle's version and SQL3 version differ from Checks or SQL2 assertions in that:
  - For triggers, event is “programmable”, rather than **implied** by the kind of check
  - For checks, we don't have conditions specified
  - (Re)action could be any sequence of database operations

→ Active Database Management Systems (ADBMS)

# Triggers

Relation Schema: **Exec(name, address, cert#, networth)**

Example of an SQL trigger:

**CREATE TRIGGER** NetWorthTrigger

**AFTER UPDATE OF** netWorth **ON** Exec  
**REFERENCING**

**OLD ROW AS** OldTuple,

**NEW ROW AS** NewTuple

**FOR EACH ROW**

**WHEN** (OldTuple.netWorth > NewTuple.netWorth) ← [Condition]

**UPDATE** Exec

← Event

**SET** netWorth = OldTuple.netWorth

**WHERE** cert# = NewTuple.cert#;

← Action

# Triggers

---

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
 OLD ROW AS OldTuple,
 NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
 UPDATE Exec
 SET netWorth = OldTuple.netWorth
WHERE cert# = NewTuple.cert#;
```

---

- **AFTER** could be changed to either:
  - **BEFORE**
  - **INSTEAD OF**

# Triggers

---

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
 OLD ROW AS OldTuple,
 NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
 UPDATE Exec
 SET netWorth = OldTuple.netWorth
WHERE cert# = NewTuple.cert#;
```

---

- **OF** in '**UPDATE OF**' is optional
  - If present, it defines the event to be only an update of the attribute(s) listed after the keyword **OF**

# Triggers

---

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
 OLD ROW AS OldTuple,
 NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
 UPDATE Exec
 SET netWorth = OldTuple.netWorth
WHERE cert# = NewTuple.cert#;
```

---

- **UPDATE (OF whatever)** could be changed to:
  - **INSERT**
  - **DELETE**

# Triggers

---

**CREATE TRIGGER** NetWorthTrigger

**AFTER INSERT** ON Exec

**REFERENCING**

**OLD ROW AS** OldTuple,

**NEW ROW AS** NewTuple

**FOR EACH ROW**

**WHEN** (NewTuple.netWorth < 10000000)

**DELETE** Exec

**WHERE** cert# = NewTuple.cert#;

← **OLD makes no sense!**

- 
- If the event is **INSERT**:

- **REFERENCING**

- NEW ROW AS** NewTuple

# Triggers

---

```
CREATE TRIGGER NetWorthTrigger
AFTER DELETE ON Exec
REFERENCING
 OLD ROW AS OT,
 NEW ROW AS NT
FOR EACH ROW
WHEN (10000000 > (SELECT AVG(netWorth) FROM Exec))
 INSERT INTO Exec
 VALUES (OT.name,OT.address,OT.cerf#,OT.netWorth);
```

← NEW makes no sense!

- 
- If event is **DELETE**:
    - **REFERENCING**  
**OLD ROW AS** OldTuple

# Triggers

---

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
 OLD ROW AS OldTuple,
 NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
 UPDATE Exec
 SET netWorth = OldTuple.netWorth
WHERE cert# = NewTuple.cert# ;
```

---

- The action could be any sequence of SQL statements, separated by semicolons and embedded in a pair of **BEGIN** and **END**, e.g.,  
**WHEN condition BEGIN S1; ... Sk; END;**

# Triggers

---

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
```

```
OLD ROW AS OldTuple,
```

```
NEW ROW AS NewTuple
```

```
FOR EACH ROW
```

```
WHEN (OldTuple.netWorth > NewTuple.netWorth)
```

```
UPDATE Exec
```

```
SET netWorth = OldTuple.netWorth
```

```
WHERE cert# = NewTuple.cert#;
```

---

- This trigger is **row-level trigger**
- If we omit “**FOR EACH ROW**”, the trigger becomes a **statement-level trigger**, which is the default

# Triggers

---

- If we update an “entire” table with an SQL statement
  - A **row-level trigger** will be executed once for each tuple
  - A **statement-level trigger** will be executed only once for the entire update
- In a statement-level trigger:
  - We can not refer to old and new tuples
  - Instead, we can/should refer to
    - The **collection** of old tuples as **OLD TABLE**
    - The **collection** of new tuples as **NEW TABLE**

# Triggers

---

- Ex: Constraint: “The *average* net worth of executives may NOT drop below \$500,000.” This could be violated e.g., by updating netWorth, or deleting or inserting tuples from/into Exec.
- Below is a trigger for update; similar triggers must be written for delete and insert.

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
 OLD TABLE AS OldStuff,
 NEW TABLE AS NewStuff
FOR EACH STATEMENT
WHEN (500000 > (SELECT AVG(networth) FROM Exec))
BEGIN
 DELETE FROM Exec
 WHERE (name, address, cert#, netWorth) IN NewStuff ;
 INSERT INTO Exec(SELECT * FROM OldStuff) ;
END;
```

# Triggers

---

**Relation scheme:** Employee(name, empld, salary, dept, supervisorId)

**Constraint:** No employee gets a salary more than his/her supervisor.

**CREATE TRIGGER** Inform\_supervisor

**BEFORE INSERT OR UPDATE OF** salary, supervisorId **ON** Employee

**NEW ROW AS** new

**FOR EACH ROW**

**WHEN** (new.salary > (**SELECT** salary

**FROM** Employee

**WHERE** empld=new.supervisorId))

**Begin**

**ROLLBACK;**

    Inform\_Supervisor(new.supervisorId, new.empld);

**End;**

# Instead-Of Triggers

---

**Relation Scheme:** Movie(title, year, length, filmType, studioName, producerC#)

```
CREATE VIEW ParamountMovie AS
SELECT title, year
FROM Movie
WHERE studioName = 'Paramount';
```

The following *trigger* replaces an insertion on the view (ParamountMovie) with an insertion on its underlying base table (Movie)

```
CREATE TRIGGER ParamountInsert
INSTEAD OF INSERT ON ParamountMovie
REFERENCING NEW ROW AS NewRow
FOR EACH ROW
INSERT INTO Movie(title, year, studioName)
VALUES (NewRow.title, NewRow.year, 'Paramount');
```

# A test on Triggers!

Suppose relation **rel(a, b)** has no tuples. Consider the trigger:

CREATE TRIGGER T

AFTER INSERT ON rel

REFERENCING NEW ROW AS newRow

FOR EACH ROW

WHEN (newRow.a \* newRow.b > 10)

INSERT INTO rel VALUES(newRow.a - 1, newRow.b + 1);

Inserting which of the following tuples into relation **rel** results in having exactly 2 tuples in **rel**?

1. (3, 5)
2. (4, 3)
3. (3, 4)
4. (5, 3)

# **COMP353 Databases**

---

**Logical Query Languages:**  
**Datalog**

# Logical Query Languages (Section 5.3)

---

## ■ Motivation

- Logical *if-then* rules extend rather “naturally” and easily to **recursive queries**; Relational algebra doesn’t!
  - Recursion is considered in SQL3
- Logical rules (Datalog) form a basis for development of many concepts and techniques in database and knowledge base systems, with many applications such as *data integration*

# Datalog

---

**AlongMovie**(Title, Year)  $\leftarrow$  movie>Title, Year, Length, Type), Length  $\geq$  100.

- The **head** – the left hand side of the arrow/implication
- The **body** – the right hand side is a conjunction (AND) of predicates (called subgoals)
- NOTE: The book uses **AND** in the rule bodies instead of commas.

**AlongMovie**(Title, Year)  $\leftarrow$  movie>Title, Year, Length, Type) **AND** Length  $\geq$  100.

- The **head** is a positive predicate (atom) and the subgoals in the rule **body** are *Atoms*
- Atom - a formula of the form  $p(T_1, \dots, T_n)$ , where **p** is a predicate and  $T_i$ 's are *terms*
  - Predicate – **normal** (ordinary) relation name (e.g., movie, p) or **built-in** predicates (e.g.,  $\geq$  in the above example)
  - Terms (arguments) – In Datalog,  $T_i$  is either a *variable* or a *constant*
  - Subgoals in the rule body may be “negated” using NOT

# Datalog

---

`longMovie(Title, Year) ← movie(Title, Year, Length, Type), Length >= 100.`

- A variable in a rule body is called *local* if it appears only in the rule body, e.g., `Length` and `Type`
- The head is “true” if there are values for local variables that make every subgoal (in the rule body) true
- If the body includes no negation, then the rule can be viewed as a join of relations in the rule body followed by a *projection* on the head variable(s)

# Datalog

---

`longMovie(Title, Year) ← movie(Title, Year, Length, Type), Length >= 100.`

- This rule may be expressed in RA as:

$$\rho_{\text{longMovie}}(\pi_{\text{Title}, \text{Year}}(\sigma_{\text{Length} \geq 100}(\text{movie})))$$

# Variable-Based Interpretations of Rules

---

- In principle, given the rule
- $r : H \leftarrow B_1, \dots, B_k.$   
we consider all possible assignments  $\mathcal{I}$  of values (constants in the domain) to the variables in the rule.
- Such assignments  $\mathcal{I}$  are called **interpretations**.
- For every interpretation  $\mathcal{I}$  of *the rule*, if the body is true under  $\mathcal{I}$ , we add to the head relation, the tuple defined by  $H$  under  $\mathcal{I}$ .  
(we only consider *ground* interpretations/substitutions).
  - That is, if  $\mathcal{I}(B_i)$  is true,  $\forall i \in \{1, \dots, k\}$ , then  $\mathcal{I}(H)$  is true.
  - In this case, we say that " $\mathcal{I}$  satisfies  $r$ " or " $\mathcal{I}$  is a model for  $r$ ",  
(this is denoted as  $\mathcal{I} \models r$ )

# Example

---

$s(X, Y) \leftarrow r(X, Z), r(Z, Y), \text{NOT } r(X, Y).$

**Instance r:**

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |

- The only assignments that make the first subgoal true are:
  1.  $I_1: X \rightarrow 1, Z \rightarrow 2$
  2.  $I_2: X \rightarrow 2, Z \rightarrow 3.$
- In case (1),

**Instance s:**

| A | B |
|---|---|
| 1 | 3 |

- $Y \rightarrow 3$  makes the second subgoal  $r(Z, Y)$  true
- Since  $(1, 3) \notin r$ , then “**NOT**  $r(X, Y)$ ” is also true
- Thus, we infer tuple  $(1, 3)$  for the head relation,  $s$
- In case (2),
- No value of “ $Y$ ” makes the second subgoal true

# Tuple-Based Interpretations of Rules

---

- Consider tuple variables for each positive normal subgoals that range over their relations
- For each assignment of tuples to each of these subgoals, we determine the implied assignment  $\theta$  of values to variables
- If the assignment  $I$  is:
  - consistent and also
  - satisfies all the subgoals (normal and built-ins) in the bodythen we add to the head relation, the tuple defined by the head  $H$  under  $I$

# Example

---

$s(X, Y) \leftarrow r(X, Z), r(Z, Y), \text{NOT } r(X, Y).$

Instance r:

| A | B |
|---|---|
| 1 | 2 |
| 2 | 3 |

- Have 4 assignments of tuples to subgoals:

$r(X, Z) \ r(Z, Y)$

1.  $(1, 2) \ (1, 2)$
2.  $(1, 2) \ (2, 3)$
3.  $(2, 3) \ (1, 2)$
4.  $(2, 3) \ (2, 3)$

Instance s:

| A | B |
|---|---|
| 1 | 3 |

- Only the second assignment
  - is consistent for the value assigned to Z and satisfies the negative subgoal “**NOT**  $r(X, Y)$ ”  
→  $(1, 3)$  is the only tuple we get for s

# Datalog Programs

---

- A datalog program is a finite collection of rules
- Note: while standard datalog does not allow negation, in our presentation here, the programs and rules are actually in datalog extended with negation and built-in predicates.
- Predicates/relations can be divided into two classes
  - **EDB** Predicates (input relations), also called *FACTS*
    - Extensional database = relations stored explicitly in DB
  - **IDB** Predicates (derived/output relations), defined by rule(s)
    - Intensional database
    - They are similar to views in relational databases
- Note: **EDB** predicates appear only in the rule body and **IDBs** appear in the head and possibly in the body

# Operations in Datalog

---

- The usual set operations
- Consider relation schemas  $r(X,Y)$  and  $s(X,Y)$ 
  - Intersection
    - RA:  $Q = r \cap s$
    - Datalog:  $q(X,Y) \leftarrow r(X,Y), s(X,Y).$
  - Union
    - RA:  $Q = r \cup s$
    - Datalog: the following two rules:
      1.  $q(X,Y) \leftarrow r(X,Y) .$
      2.  $q(X,Y) \leftarrow s(X,Y) .$
  - Difference
    - RA:  $Q = r - s$
    - Datalog:  $q(X,Y) \leftarrow r(X,Y), \text{NOT } s(X,Y).$

# Operations in Datalog

---

## ■ Projection operation

- RA:  $p = \pi_x(r)$
- Datalog:  $p(X) \leftarrow r(X, Y).$

# Operations in Datalog

---

## ■ Selection operation

- RA:  $s = \sigma_{X > 10 \text{ AND } Y = 5}(r)$
- Datalog:  $s(X, Y) \leftarrow r(X, Y), X > 10, Y = 5.$

# Operations in Datalog

---

## ■ Selection operation. Recall the schema of $r(X, Y)$ .

- RA:  $s = \sigma_{x > 10 \text{ OR } y = 5}(r)$
- Datalog: the following two rules:
  1.  $s(X, Y) \leftarrow r(X, Y), X > 10.$
  2.  $s(X, Y) \leftarrow r(X, Y), Y = 5.$

Note: the following Datalog program is equivalent to the above.

1.  $s(A, B) \leftarrow r(A, B), A > 10.$
2.  $s(C, D) \leftarrow r(C, D), D = 5.$

# Operations in Datalog

---

- Cartesian Product operation
- Consider relation schemas  $r(A, B)$  and  $s(C, D)$ 
  - RA:  $Q = r \times s$
  - Datalog:  $q(X, Y, Z, W) \leftarrow r(X, Y), s(Z, W).$

# Operations in Datalog

---

## ■ Join operation

- Theta-join with an **AND** condition, e.g., “ $c_1 \text{ AND } c_2$ ”
  - RA:  $tj1 = r \bowtie_{x>z \text{ AND } y<w} s$
  - Datalog:  $tj1(X, Y, Z, W) \leftarrow r(X, Y), s(Z, W), X > Z, Y < W.$
- Theta-join with an **OR** condition, e.g., “ $c_1 \text{ OR } c_2$ ”
  - RA:  $tj2 = r \bowtie_{x>z \text{ OR } y<w} s$
  - Datalog: the following two rules:
$$tj2(X, Y, Z, W) \leftarrow r(X, Y), s(Z, W), X > Z.$$
$$tj2(X, Y, Z, W) \leftarrow r(X, Y), s(Z, W), Y < W.$$

# Operations in Datalog

---

## ■ Join operation

### ■ Equi-join

- RA:  $\text{ej3} = r \bowtie_{Y=z} s$
- Datalog:  $\text{ej3}(X,Y,Z,W) \leftarrow r(X,Y), s(Z,W), Y = Z.$   
OR even better (simpler):

$$\text{ej3}(X,Y,Y,W) \leftarrow r(X,Y), s(Y,W).$$

# Operations in Datalog

---

- Join operation
  - Natural join
    - RA:  $nj4 = r \bowtie s$
    - Datalog:  $nj4(X, Y, W) \leftarrow r(X, Y), s(Y, W).$

# Example:Datalog Queries/Programs

---

- Database schema:

**movie( Title, Year, Length, FilmType, StudioName)**  
**starsIn( Title, Year, StarName)**

- Query: Find the names of stars of movies that are at least 100 minutes long
- Relational Algebra Expression:

$$Q = \pi_{\text{starName}} (\sigma_{\text{length} \geq 100}(\text{movie}) \bowtie \text{starsIn})$$

- Datalog program:

```
r1>Title, Year, Length, Type, Studio ←
 movie>Title, Year, Length, Type, Studio), Length >= 100.
r2>Title, Year, Length, Type, Studio, Name) ←
 r1>Title, Year, Length, Type, Studio), starsIn>Title, Year, Name).
q>Name) ← r2>Title, Year, Length, Type, Studio, Name).
```

As in RA case, we could express this query using just one rule, as follows:

```
q>Name) ← movie>Title, Year, Length, Type, Studio), Length >= 100,
 starsIn>Title, Year, Name).
```

# Expressive Power of Datalog

---

- Relational algebra = Nonrecursive Datalog+ negation
- Datalog can express SQL **SELECT-FROM-WHERE** statements that do **not** use aggregation and/or grouping
- The SQL-99 standard supports recursion but it is not part of the “core” SQL-99 standard that every DBMS should support
- Some DBMS implementations, e.g. DB2, support *linear recursion*

# Example

trainSchedule(From,To)

■ Datalog:

- $\text{fromMontreal}(C) \leftarrow \text{trainSchedule}(\text{'Montreal'}, C).$
- $\text{fromMontreal}(TC) \leftarrow \text{fromMontreal}(C), \text{trainSchedule}(C, TC).$

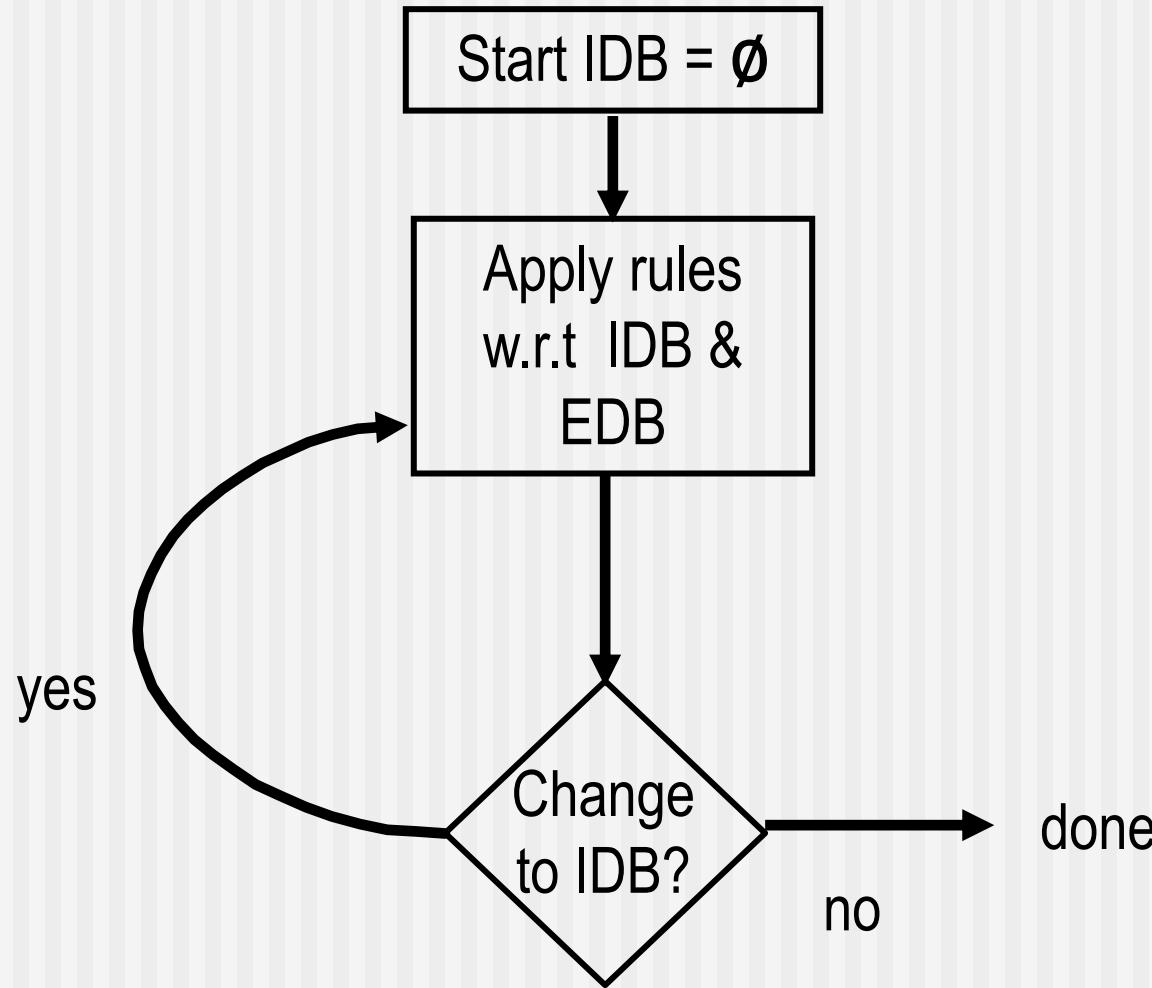
trainSchedule:

| From     | To          |
|----------|-------------|
| Toronto  | Calgary     |
| NYC      | Boston      |
| NYC      | Albany      |
| Chicago  | Detroit     |
| Montreal | Toronto     |
| Montreal | NYC         |
| Boston   | Quebec City |

fromMontreal:

|             |
|-------------|
| Toronto     |
| NYC         |
| Calgary     |
| Boston      |
| Albany      |
| Quebec City |

# Evaluation of Recursive Rules



# Example

**Relation Schema: `sequelOf(Movie, Sequel)`**

**Instance:**

| Movie         | Sequel        |
|---------------|---------------|
| Star wars     | Star wars II  |
| Naked Gun     | Naked Gun 2   |
| Naked Gun 2   | Naked Gun 2 ½ |
| Star wars II  | Star wars III |
| Naked Gun 2 ½ | Naked Gun 3   |

**For a given movie, find all the follow-up movies, i.e., a sequel, a sequel of a sequel, and so on**

# Example

Relation Schema: **sequelOf(Movie, Sequel)**

Instance:

| Movie         | Sequel        |
|---------------|---------------|
| Star wars     | Star wars II  |
| Naked Gun     | Naked Gun 2   |
| Naked Gun 2   | Naked Gun 2 ½ |
| Star wars II  | Star wars III |
| Naked Gun 2 ½ | Naked Gun 3   |

**followUp(X, Y)  $\leftarrow$  sequelOf(X,Y).**

**followUp(X, Y)  $\leftarrow$  sequelOf(X,Z), followUp(Z,Y).**

# Recursion

---

- Let  $P$  be any datalog program
- We say an IDB predicate  $r$  in  $P$  *depends on* predicate  $s$  if there is a rule in  $P$  with  $r$  as the head and  $s$  as a subgoal in the rule body
- Construct the (dependency) graph of  $P$ :
  - **Nodes** -- IDB predicates in  $P$
  - **Arcs** -- an arc from node  $r$  to  $s$  if  $r$  depends on  $s$
  - Label the arc with ‘ $\neg$ ’ for negated subgoals
- $P$  is recursive iff its dependency graph has a cycle

# Example

---

**followUp(X, Y)  $\leftarrow$  sequelOf(X,Y).**

**followUp(X, Y)  $\leftarrow$  sequelOf(X,Z), followUp(Z, Y).**



# Safety

---

- It is possible to write a rule that makes “no sense”.
- Example of such rules:
  - $s(X) \leftarrow r(Y).$
  - $s(X) \leftarrow \text{NOT } r(X).$
  - $s(X) \leftarrow r(Y), X < Y.$
- In each of these rules, the IDB relation **s** (output relation) could be infinite, even if (the input) relation **r** is finite
- Such rules are said to be not **SAFE**

# Safety

---

- For a rule to be safe, the following conditions must hold:
  - If a variable **X** appears in the rule head, then **X** must appear in an “ordinary” predicate in the body or be equal to such a variable (directly or indirectly), e.g., **X=Y**, and **Y** appears in an ordinary predicate in the rule body.

Recall: the predicates could be ordinary or built-in.

# **COMP353 Databases**

---

**Database Design:  
Object Definition  
Language (ODL)**

# ODL

---

- **ODL** (Object Definition Language) is a standard text-based language for describing the structure of databases
- **ODL** is an extension of **IDL** (Interface Description Language), a component of **CORBA** (Common Object Request Broker Architecture)

# Object Oriented World

---

- In an object oriented design, the “**world**” we want to model is thought of as being **composed of objects**
- Everything is an **object**
  - *people*
  - *bank accounts*
  - *airline flights*
- Every object has a unique object id (OID)
- Every **object** is an **instance** of a **class**
- A **class** simply represents a grouping of **similar objects**
- All objects that are instances of the same class have the same **properties** and **behaviors**

# Class Declarations

---

- A declaration of a **class** in ODL consists of:

- The keyword **class**
- The **name** of the class
- A bracketed { ... } list of **properties** of the class

```
class <name> {
 <list of properties>
};
```

```
class Movie {
```

```
...
```

```
};
```

# Properties of ODL classes

---

- ODL classes can have three kinds of properties:
  - **Attributes**
    - properties whose types are built from **primitive/basic types** such as integers, strings,...
  - **Relationships**
    - properties whose type is either a **reference** to an object or a **collection** of such references
  - **Methods**
    - **functions** that may be applied to objects of the class

# Attributes in ODL

---

- Attributes are the **simplest kinds** of properties
- An attribute **describes some aspect of an object** by associating, with the object, a value of some **simple type**
- For example, attributes of a **Student** object
  - Student ID
  - Name
  - Address
  - E-mail

# Keys in ODL

---

- In **ODL**, we declare keys using the keyword **key**
  - If a key has more than one attribute, we surround them by (...)

- Example: (two attributes forming a key)

```
class Movie

 (extent Movies key (title, year)) {

 attribute string title;

 ...

 };
```

- If a class has > one key, we may list them all, separated by commas

- Example: (A class with two keys)

```
class Employee

 (extent Employees key emplID, SIN) {...};
```

# Single-Value Constraints in ODL

---

- Often, we should *enforce* properties in the database saying that there is **at most one** value playing a particular role
  - For example:
    - that a movie object has a **unique** title, year, length, etc
    - that a movie is owned by a **unique** studio

# Single-Value Constraints

---

- In ODL:
  - An attribute is **not** of a collection type  
(Set, Bag, Array, List, Dictionary are **collection types**.)
  - A relationship is either a class type or (a single use of) a collection type constructor applied to a class type.
- Recall that in the **E/R** notation:
  - attributes are **atomic**
  - an arrow ( $\rightarrow$ ) can be used to express the multiplicity of relationships (1:1), (1:M), and (N:M)

# Type system

---

A **type system** consists of

- **Basic types**
- **Type constructors**
  - recursive rules whereby **complex types** are built from simpler ones

# Basis of types in ODL

---

- Primitive types (atomic)

- Integer
- Float
- Char
- Character String
- Boolean
- Date
- Enumeration (a **list of names** declared to be **synonyms for integers**)

- Class types

- Movie

# Type constructors in ODL

---

- **Set**
  - Set <integer>
  - Set <Movie>
- **Bag**
  - Bag <integer>
  - Bag <Movie>
- **Array**
  - Array <integer, 10>
  - Array <Movie, 3>
- **Structure**
  - Struct Address {string street, string city}
- **List**
  - List <integer>
  - List <Student>
- **Dictionary <keyType, valueType>**
  - Dictionary<Student, string>

## ■ Note:

- Set, Bag, Array, List and Dictionary are called **collection types**
- Collection type cannot be applied repeatedly (nested)
  - E.g., it is **illegal** to write Set<Array<integer,10>>

# Example

---

```
class Movie {
 attribute string title;
 attribute integer year;
 attribute integer length;
 attribute enum Film {color, blackAndWhite} filmType;
};
```

(“Gone with the Wind”, 1939, 231, color) is a Movie object.

# Example (non-atomic type)

---

```
class Star {
 attribute string name;
 attribute Struct Address {
 string street,
 Array<char, 10> city
 } homeAddress;
 attribute Address officeAddress;
};
```

# Example

---

```
class Student {
 attribute string ID;
 attribute string lastName;
 attribute string firstName;
 attribute date dob; /* date is a basic type in ODL */
 attribute string program;
 attribute Struct Address {
 string street,
 string city
 } homeAddress;
};
```

# Example

---

```
class Course {
 attribute string courseNumber;
 attribute string courseName;
 attribute integer noOfCredits;
 attribute string department;
};
```

# Relationships in ODL

---

- If we are designing a database about **Movies** and **Stars**, what are we missing? The relationships....
- How are **Movies** and **Stars** related?
- Every movie has a star (or stars)

# Example

---

- Can we write “ **attribute Star starOf;** ” ?

```
class Movie {
 attribute string title;
 attribute integer year;
 attribute integer length;
 attribute enum Film {color, blackAndWhite} filmType;
attribute Star starOf;
};
```

- **No**, the attribute types **must not** be classes

# Example

---

- **starOf** is a relationship between **Movie** and **Star**

```
class Movie {
 attribute string title;
 attribute integer year;
 attribute integer length;
 attribute enum Film {color, blackAndWhite} filmType;
 relationship Star starOf;
};
```

# Inverse Relationships

---

- How are **Movies** and **Stars** related?
- Not only every movie has a star but also every star has a role in some movie(s)
- To fix this in the **Star** class, we add the line:  
**relationship Movie starredIn;**

# Example

---

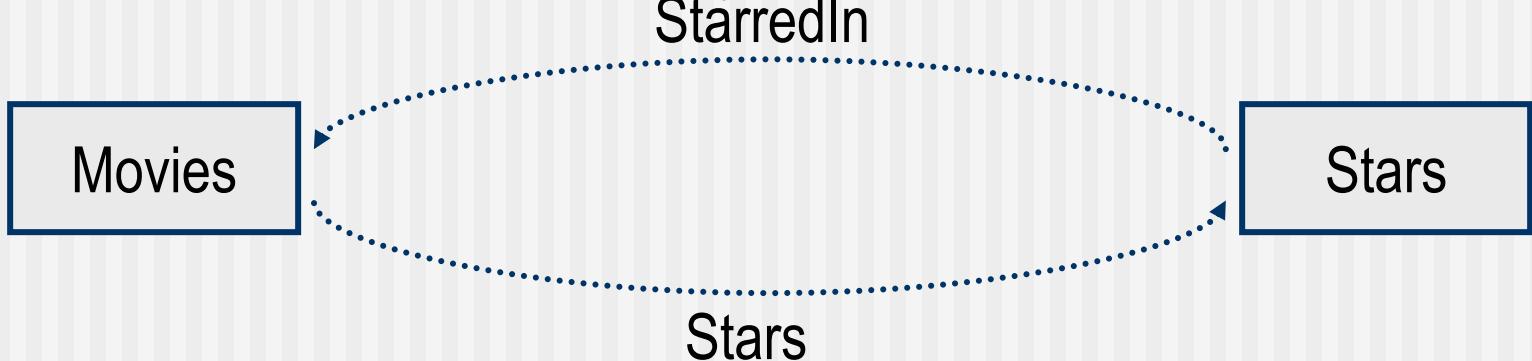
```
class Star {
 attribute string name;
 attribute Struct Address {
 string street,
 string city
 } address;
 relationship Movie starredIn;
};
```

- What is the problem here?

# Inverse Relationships

---

- We are omitting a very important aspect of the relationship between movies and stars
- We need a way to ensure that if a star **S** is connected to a movie **M** via **stars**, then conversely, **M** is connected to **S** via **starredIn**
- In ODL that is done by **inverse** of a relationship



# Example

---

```
class Movie {
 attribute string title;
 attribute integer year;
 attribute integer length;
 attribute enum Film {color, blackAndWhite} filmType;
 relationship Star stars
 inverse Star::starredIn;
};
```

# Example

---

```
class Star {
 attribute string name;
 attribute Struct Address {
 string street,
 string city
 } address;
 relationship Movie starredIn
 inverse Movie::stars;
};
```

# Relationships in ODL

---

- Our design is missing another important point!
- A movie typically has several stars
- A star usually plays in more than one movie
- To fix this, we write:

**relationship Set<Star> stars;**

# Example

---

```
class Movie {
 attribute string title;
 attribute integer year;
 attribute integer length;
 attribute enum Film {color, blackAndWhite} filmType;
 relationship Set<Star> stars
 inverse Star::starredIn;
};
```

# Example

---

```
class Star {
 attribute string name;
 attribute Struct Address {
 string street,
 string city
 } address;
 relationship Set<Movie> starredIn
 inverse Movie::stars;
};
```

# Example

---

- Suppose we introduce another class, **Studio**, representing the studios, i.e., companies that produce movies

```
class Studio {
 attribute string name;
 attribute string address;
};
```

# Example

---

- How are **Movies** and **Studios** related?
- Every **Studio** owns several **Movies**

```
class Studio {
 attribute string name;
 attribute string address;
 relationship Set<Movie> owns
 inverse Movie::ownedBy;
};
```

# Example

---

- What about inverse?
- Every **Movie** is owned by some **Studio**

```
class Movie {
 attribute string title;
 attribute integer year;
 attribute integer length;
 attribute enum Film {color, blackAndWhite} filmType;
 relationship Set<Star> stars inverse Star::starredIn;
 relationship Studio ownedBy inverse Studio::owns;
};
```

# Multiplicity of relationships

---

- In general, when we have a pair of inverse relationships, there are four cases:
  - The relationship is unique in both directions (1)
  - The relationship is unique in just one direction (2)
  - The relationship is not unique in any direction (1)
  - The *multiplicity* is thus referred to the kinds of these 4 relationships, also denoted as 1-1 (read as one-one), 1-M (one-many), M-1 (many-one), and M-N (many-many).

# Multiplicity of relationships

---

- A **many-many** relationship from a class **C** to a class **D** is one in which, for each **C** there is a set of **Ds** associated with **C**, and in the inverse relationship, associated with each **D** is a set of **Cs**
  - For example, each student can take many courses and each course can be taken by more than one student

```
class Student {
 ...
 relationship Set<Course> takes inverse Course::takenBy;
};

class Course {
 ...
 relationship Set<Student> takenBy inverse Student:: takes;
};
```

# Multiplicity of relationships

---

- A **many-one** relationship from class **C** to a class **D**, is one where for each **C** there is at most one **D**, but no such a constraint in the reverse direction (similarly for one-many)
  - For example, many employees may work in the same department, but each employee works only in one department

```
class Employee {
 ...
 relationship Department worksIn inverse Department::workers;
};

class Department {
 ...
 relationship Set< Employee > workers inverse
 Employee::worksIn;
};
```

# Multiplicity of relationships

---

- A **one-one** relationship from class **C** to class **D** is one that for each **C** there is at most one **D**, and conversely, for each **D** there is at most one **C**
  - For example, each department has at most one professor as its chairperson and each professor can be the chair of at most one department

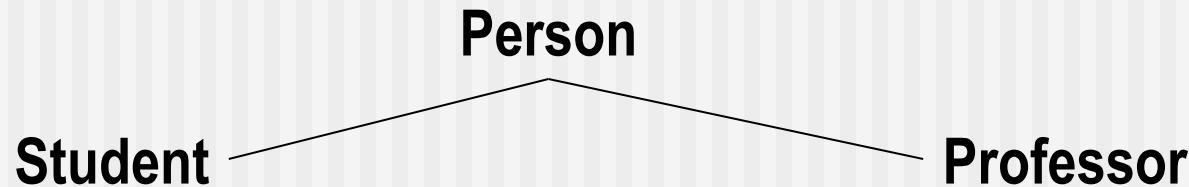
```
class Professor {
 ...
 relationship Department chairOf inverse Department::chair;
};

class Department {
 ...
 relationship Professor chair inverse Professor:: chairOf;
};
```

# Inheritance in Object Oriented World

---

- Objects can be organized into a hierarchical inheritance/is structure
- A child class (or subclass) will inherit properties from a parent class (or all the superclasses) higher in the hierarchy.



# Subclasses in ODL

---

- Often, a class contains some objects that have **special properties** not associated with all members of the class
- If so, we find it useful to organize the class into **subclasses**, each subclass having its **own special** attributes and/or relationships

# Subclasses in ODL

---

- We define a class **C** to be a subclass of another class **D** by following the name **C** in its declaration with a keyword **extends** and the name **D**

```
class Cartoon extends Movie {
 relationship Set<Star> voices;
};
```

A subclass *inherits* all the properties of its superclasses  
So, each cartoon object has *title*, *year*, *length*, *filmType*, and  
inherits relationships *stars* and *ownedBy* from *Movie*, in addition  
to its own relationship *voices*.

# Example

---

```
class Person {
 attribute string lastName;
 attribute string firstName;
 attribute integer age;
 attribute Struct Address {
 string street,
 string city
 } homeAddress;
};
```

```
class Student extends Person {
 attribute string ID;
 attribute string program;
};
```

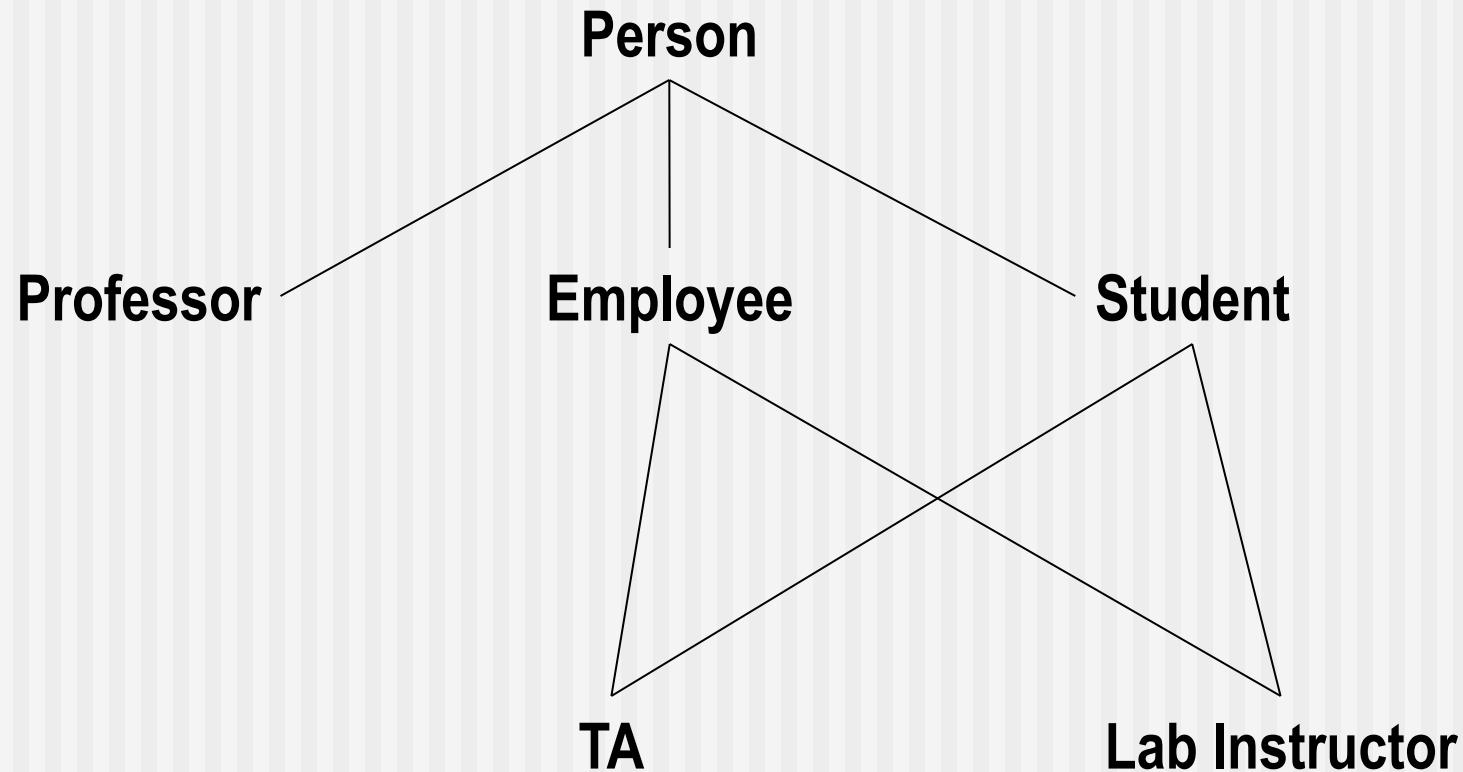
# Inheritance in ODL

---

- A class may have **more than one** subclass.
- A class may have more than one class from which it inherits properties; those classes are its superclasses
- Subclasses may themselves have subclasses, yielding a **hierarchy** of classes where each class inherits the properties of its ancestors.

# Multiple Inheritance in ODL

---



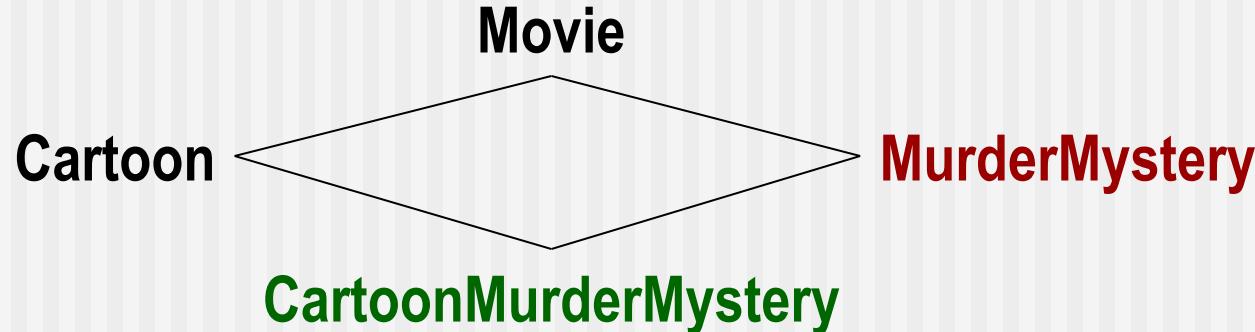
# Example

---

```
class MurderMystery extends Movie {
```

```
 attribute string weapon;
};
```

```
class CartoonMurderMystery extends Cartoon : MurderMystery;
```



- Thus, a **CartoonMurderMystery** object is defined to have all the properties of both of its superclasses: **Cartoon** and **MurderMystery**.

# "For All" Queries (1)

- Given the database schema:

$\text{Student}(\text{Sid}, \text{Sname}, \text{Addr})$

$\text{Course}(\text{Cid}, \text{Cname}, \text{Credits})$

$\text{Enrolled } (\underline{\text{Sid}}, \underline{\text{Cid}})$

- Consider the following query:

*"Find the students who are enrolled in all courses."*

- Solution in RA?

$\pi_{\text{Sid}}(\text{Enrolled})$

- No, this returns the students enrolled in some courses.
- Any ideas about what is wrong?

# "For All" Queries (2)

- A *solution strategy* would be to:
  - start with all students mentioned in the Enrolled relation (all guys), from which we then subtract those not enrolled in all courses (bad guys)
- That is, to find all the “good guys”, we need to determine “all guys” and “bad guys”, i.e.,  
Answer = Good guys = All guys – Bad guys

# "For All" Queries (3)

- Set of all students we need to consider:  
 $A \leftarrow \pi_{Sid}(\text{Enrolled})$
- Set of all students not enrolled in all courses
  1. Create all possible “student-course” pairs:  
 $\pi_{Sid}(\text{Enrolled}) \times \pi_{Cid}(\text{Course})$
  2. Create all “actual” pairs made of one student and one course: which is **Enrolled**
  3. Students who are not enrolled in all courses:  
 $B \leftarrow \pi_{Sid}(\pi_{Sid}(\text{Enrolled}) \times \pi_{Cid}(\text{Course}) - \text{Enrolled})$
- **The query:**  $A - B$

# The Division Operation ( $\div$ )

- The previous query can be conveniently expressed in RA using the division operator
  - Divide  $\text{Enrolled}$  by  $\pi_{Cid}(\text{Course})$
  - Here,  $\text{Enrolled} \div \pi_{Cid}(\text{Course})$
  - Schema of the result is  $\{\text{Sid}, \text{Cid}\} - \{\text{Cid}\}$
- $r \div s$  requires that all attributes of  $S$  to be a subset of attributes of  $R$ .
  - The schema of the result would be  $R - S$

# Example: Enrolled(student,sport)

$$\pi_{student}(\text{Enrolled}) \times \pi_{sport}(\text{Enrolled}) - \text{Enrolled} =$$

|     |        |
|-----|--------|
| Jim | Hockey |
| Jim | Soccer |
| Joe | Hockey |
| Joe | Soccer |
| Sue | Hockey |
| Sue | Soccer |

-

|     |        |
|-----|--------|
| Jim | Hockey |
| Joe | Soccer |
| Jim | Soccer |
| Sue | Hockey |

=

|     |        |
|-----|--------|
| Joe | Hockey |
| Sue | Soccer |

$$\pi_{student}(\text{Enrolled}) - \pi_{student}(\pi_{student}(\text{Enrolled}) \times \pi_{sport}(\text{Enrolled}) - \text{Enrolled})$$

All

Bad

|     |
|-----|
| Jim |
| Joe |
| Sue |

|     |
|-----|
| Joe |
| Sue |



Jim is the only student enrolled in all sports

# Another Example

- Given:
  - $\text{Customer}(\underline{\text{id}}, \text{name})$
  - $\text{Branch}(\underline{\text{bid}}, \text{district})$
  - $\text{Account}(\underline{\text{cid}}, \underline{\text{bid}})$
- Query: "*Find the names of all customers who have an account in every branch located in Westmount area*"
- Solution?
  - $\pi_{\text{name}} (\text{Customer} \triangleright\triangleleft \text{Account} \triangleright\triangleleft (\sigma_{\text{district} = \text{"Westmount"}} \text{Branch}))$  ?
- **No**, this returns the names of all customers who have an account at some branch in Westmount, not all such brances.

# Solution

Customer(cid, name), Branch(bid, district), Account(cid, bid)

- We can apply the division operator
  - Find all pairs (cid, bid) for which the customer has an account at a branch  
 $\pi_{cid,bid} (\text{customer} \triangleright\triangleleft \text{account})$
  - Divide it by all bid's of branches in Westmount  
 $\pi_{bid} (\sigma_{district = "Westmount"} \text{branch})$
- $\pi_{name} ( (\text{customer} \triangleright\triangleleft \text{account}) \div \pi_{bid} (\sigma_{district = "Westmount"} \text{branch}) )$
- So, we define  $r \div s$  as follows:  
 $r \div s = \pi_{R-s}(r) - \pi_{R-s}((\pi_{R-s}(r) \times s) - \pi_{R-s,s}(r))$