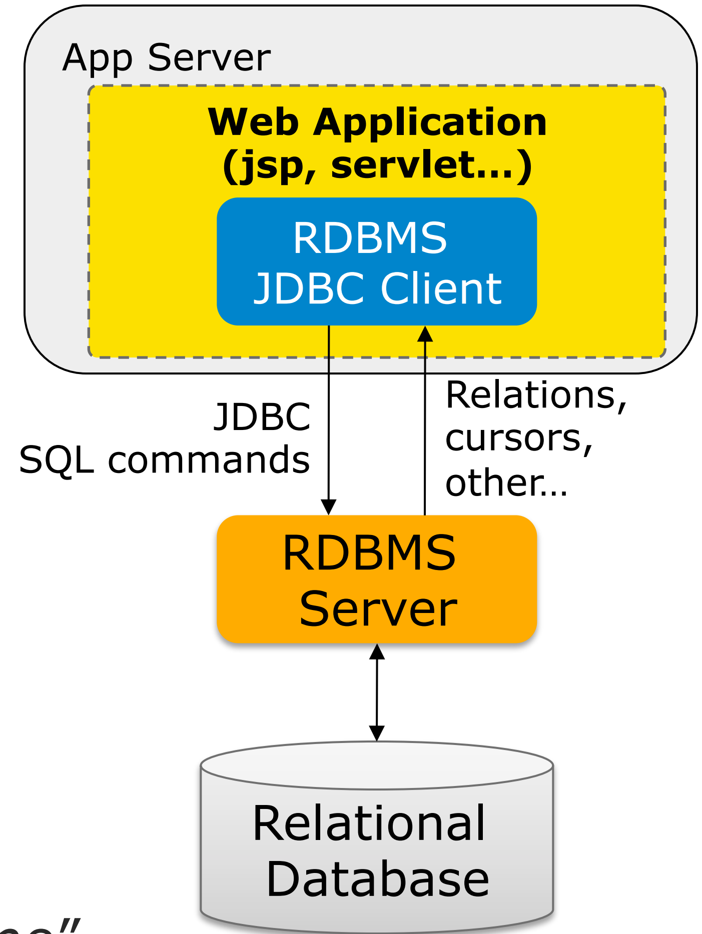
# MAS 201

## Database design & SQL programming

# Applications' View of a Relational Database Management System (RDBMS): Why use it?

- Persistent data structure
  - Large volume of data
- **High-level language/API for reading (querying)** & writing (inserting, deleting, updating)
  - Automatically optimized
- Transaction management (ACID)
  - Atomicity: all or none happens, despite failures &errors
  - Consistency
  - Isolation: appearance of "one at a time"
  - Durability: recovery from failures and other errors



# OLTP Vs OLAP use cases

OLTP

- Support quick ACID transactions
- Eg, Bank application that manages transactions

OLAP

- Perform analytics on the database
- Eg, Bank application analyzing customer profiles towards marketing

- All well-known databases can do both
- But may not be very efficient in analytics
- Many new databases focused on analytics
  - Organizations may have two databases – OLTP vs OLAP
    - Or 3+
- The jury is out on whether two kinds of databases will be needed

# Data Structure: Relational Model

- **Relational Databases:** Schema + Data
- **Schema:**
  - collection of *tables* (also called *relations*)
  - each table has a set of *attributes* (aka *columns*)
  - no repeating table names, no repeating attributes in one table
- **Data** (also called *instance*):
  - set of *tuples* (aka *rows*)
  - tuples have one atomic *value* for each attribute

**Movie**

| ID | Title | Actor |
|---|---|---|
| 1 | Wild | Winger |
| 2 | Sky | Winger |
| 3 | Reds | Beatty |
| 4 | Tango | Brando |
| 5 | Tango | Winger |
| 7 | Tango | Snyder |

**Schedule**

| ID | Theater | Movie |
|---|---|---|
| 1 | Odeon | 1 |
| 2 | Forum | 3 |
| 3 | Forum | 2 |

# Data Structure: Primary Keys; Foreign Keys are value-based pointers

**Schedule**

| ID | Theater | Movie |
|---|---|---|
| 1 | Odeon | 1 |
| 2 | Forum | 3 |
| 3 | Forum | 2 |

**Movie**

| ID | Title | Director | Actor |
|---|---|---|---|
| 1 | Wild | Lynch | Winger |
| 2 | Sky | Berto | Winger |
| 3 | Reds | Beatty | Beatty |
| 4 | Tango | Berto | Brando |
| 5 | Tango | Berto | Winger |
| 7 | Tango | Berto | Snyder |

- "**`ID`** is *primary key* of **`Schedule`**" => its value is unique in **`Schedule.ID`**
- "**`Schedule.Movie`** is foreign key (referring) to **`Movie.ID`**" means every **`Movie`** value of **`Schedule`** also appears as **`Movie.ID`**
- Intuitively, Schedule.Movie operates as pointer to Movie(s)

# Schema design has its own intricacies

**Schedule**

| ID | Theater | Movie |
| --- | --- | --- |
| 1 | Odeon | 1 |
| 2 | Forum | 3 |
| 3 | Forum | 2 |

**Movie**

| ID | Title | Director | Actor |
| --- | --- | --- | --- |
| 1 | Wild | Lynch | Winger |
| 2 | Sky | Berto | Winger |
| 3 | Reds | Beatty | Beatty |
| 4 | Tango | Berto | Brando |
| 5 | Tango | Berto | Winger |
| 7 | Tango | Berto | Snyder |

- This example is a bad schema design!
- Problems
  - Change the name of a theater
  - Change the name of a movie’s director
  - What about theaters that play no movie?

# How to Design a Database and Avoid Bad Decisions

- With experience…
- Normalization rules of database design instruct how to turn a “bad” design into a “good” one
  - a well-developed mathematical theory
  - no guidance on how to start
  - does not solve all problems
- MAS 201: Think **entities and relationships** – then translate them to tables
- MAS 201: The special case of star & snowflake schemas

# Designing Schemas Using Entity-Relationship modeling

The Basics

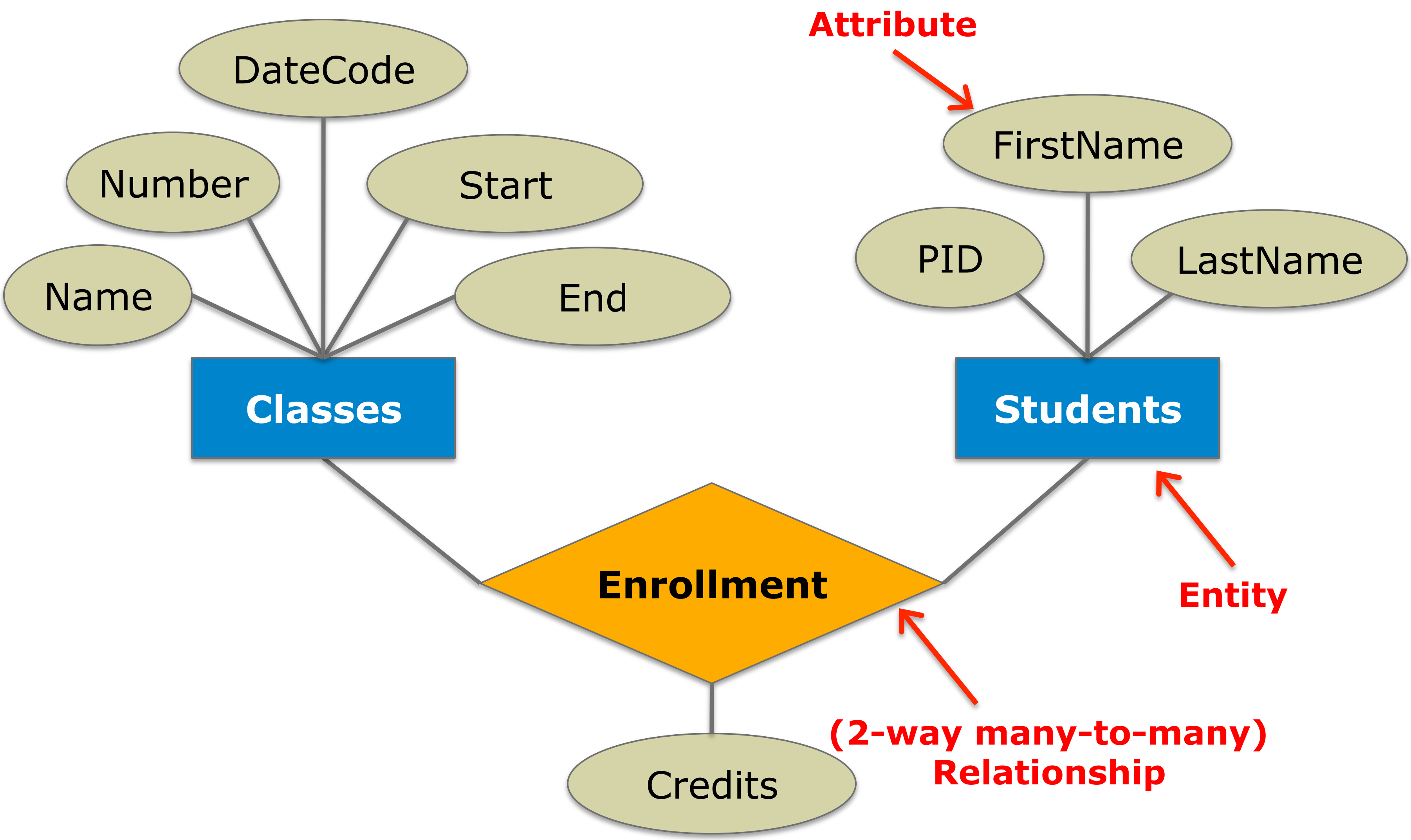
# Data Structure: Relational Model

**Example Problem:**

- Represent the students classes of the CSE department in Winter, including the enrollment of students in classes.
- Students have pid, first name and last name.
- Classes have a name, a number, date code (TR, MW, MWF) and start/end time.
  - Dismiss the possibility of two Winter classes (or class sections) for the same course
- A student enrolls for a number of credits in a class.

**Solution:**…

# Example 1a: E/R-Based Design



# E/R→ Relational Schema: Basic Translation

- For every entity
  - create corresponding table
  - For each attribute of the entity, add a corresponding attribute in the table
  - Include an ID attribute in the table even if not in E/R
- For every many-to-many relationship
  - create corresponding table
  - For each attribute of the relationship, add a corresponding attribute in the table
  - For each referenced entity $E_i$ include in the table a *required foreign key* attribute referencing ID of $E_i$

# Sample relational database, per previous page's algorithm

**Classes**

| id | name | number | date_code | start_time | end_time |
|---|---|---|---|---|---|
| 1 | Web stuff | MAS201 | TuTh | 2:00 | 3:20 |
| 2 | Databases | CSE132A | TuTh | 3:30 | 4:50 |
| 4 | VLSI | CSE121 | F | *null* | *null* |

**Enrollment**

| id | class | student | credits |
|---|---|---|---|
| 1 | 1 | 1 | 4 |
| 2 | 1 | 2 | 3 |
| 3 | 4 | 3 | 4 |
| 4 | 1 | 3 | 3 |

**Students**

| id | pid | first_name | last_name |
|---|---|---|---|
| 1 | 8888888 | John | Smith |
| 2 | 1111111 | Mary | Doe |
| 3 | 2222222 | *null* | Chen |

# Declaration of schemas in SQL's Data Definition Language

```
CREATE TABLE classes (
    ID            SERIAL PRIMARY KEY,
    name          TEXT,
    number        TEXT,
    date_code     TEXT,
    start_time    TIME,
    end_time      TIME
)
CREATE TABLE students (
    ID            SERIAL PRIMARY KEY,
    pid           INTEGER,
    first_name    TEXT,
    last_name     TEXT
)
CREATE TABLE enrollment (
    ID                SERIAL,
    class             INTEGER REFERENCES classes (ID) NOT NULL,
    student           INTEGER REFERENCES students (ID) NOT NULL,
    credits           INTEGER
)
```

If we had "ID **INTEGER** PRIMARY KEY" we would be responsible for coming up with ID values. **SERIAL** leads to a counter that automatically provides ID values upon insertion of new tuples

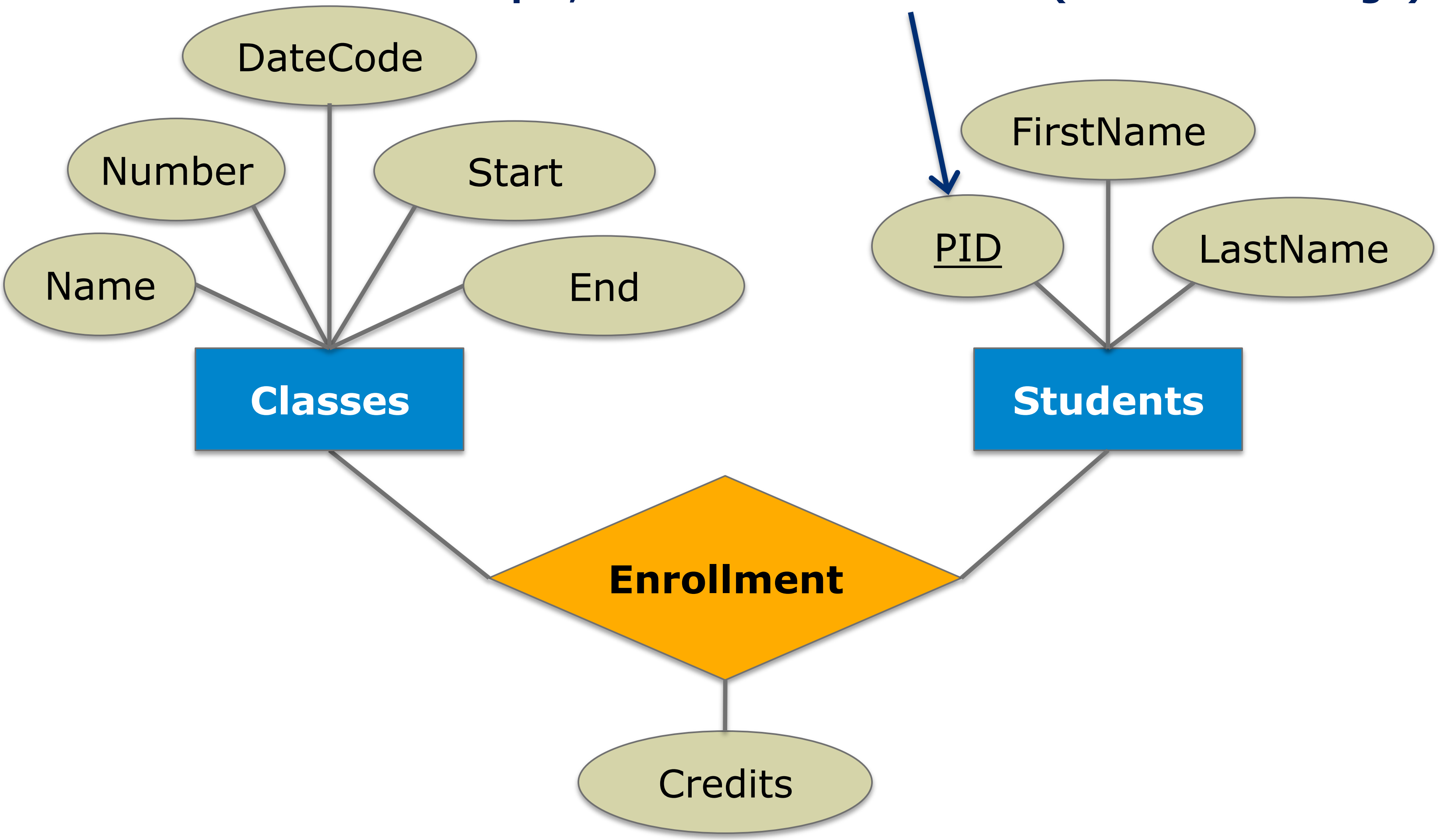
Changed name from "end" to "end_time" since "end" is reserved keyword

Foreign key declaration: Every value of **enrollment.class** must also appear as **classes.ID**

Declaration of "required" constraint: **enrollment.student** cannot be null (notice, it would make no sense to have an enrollment tuple without a student involved)

# Example 1b: Using a semantic, immutable key

**Assume that each PID (the id number on UCSD cards) is unique, not null and immutable (will never change)**



# Example 1b: Sample, using the pid instead of the id to identify students

**Classes**

| id | name | number | date_code | start_time | end_time |
|---|---|---|---|---|---|
| 1 | Web stuff | MAS201 | TuTh | 2:00 | 3:20 |
| 2 | Databases | CSE132A | TuTh | 3:30 | 4:50 |
| 4 | VLSI | CSE121 | F | *null* | *null* |

**Enrollment**

| id | class | student | credits |
|---|---|---|---|
| 1 | 1 | 8888888 | 4 |
| 2 | 1 | 1111111 | 3 |
| 3 | 4 | 2222222 | 4 |
| 4 | 1 | 2222222 | 3 |

**Students**

| ~~id~~ | pid | first_name | last_name |
|---|---|---|---|
| ~~1~~ | 8888888 | John | Smith |
| ~~2~~ | 1111111 | Mary | Doe |
| ~~3~~ | 2222222 | *null* | Chen |

# Example 1b: Schema revisited, for using pid for students’ primary key

```
CREATE TABLE classes (
    ID          SERIAL PRIMARY KEY,
    name        TEXT,
    number      TEXT,
    date_code   TEXT,
    start_time  TIME,
    end_time    TIME
)
CREATE TABLE students (
    ID          SERIAL PRIMARY KEY,
    pid         INTEGER PRIMARY KEY,
    first_name  TEXT,
    last_name   TEXT
)
CREATE TABLE enrollment (
    ID              SERIAL,
    class           INTEGER REFERENCES classes (ID) NOT NULL,
    student         INTEGER REFERENCES students (pid) NOT NULL,
    credits         INTEGER
)
```

# … some easy hands-on experience

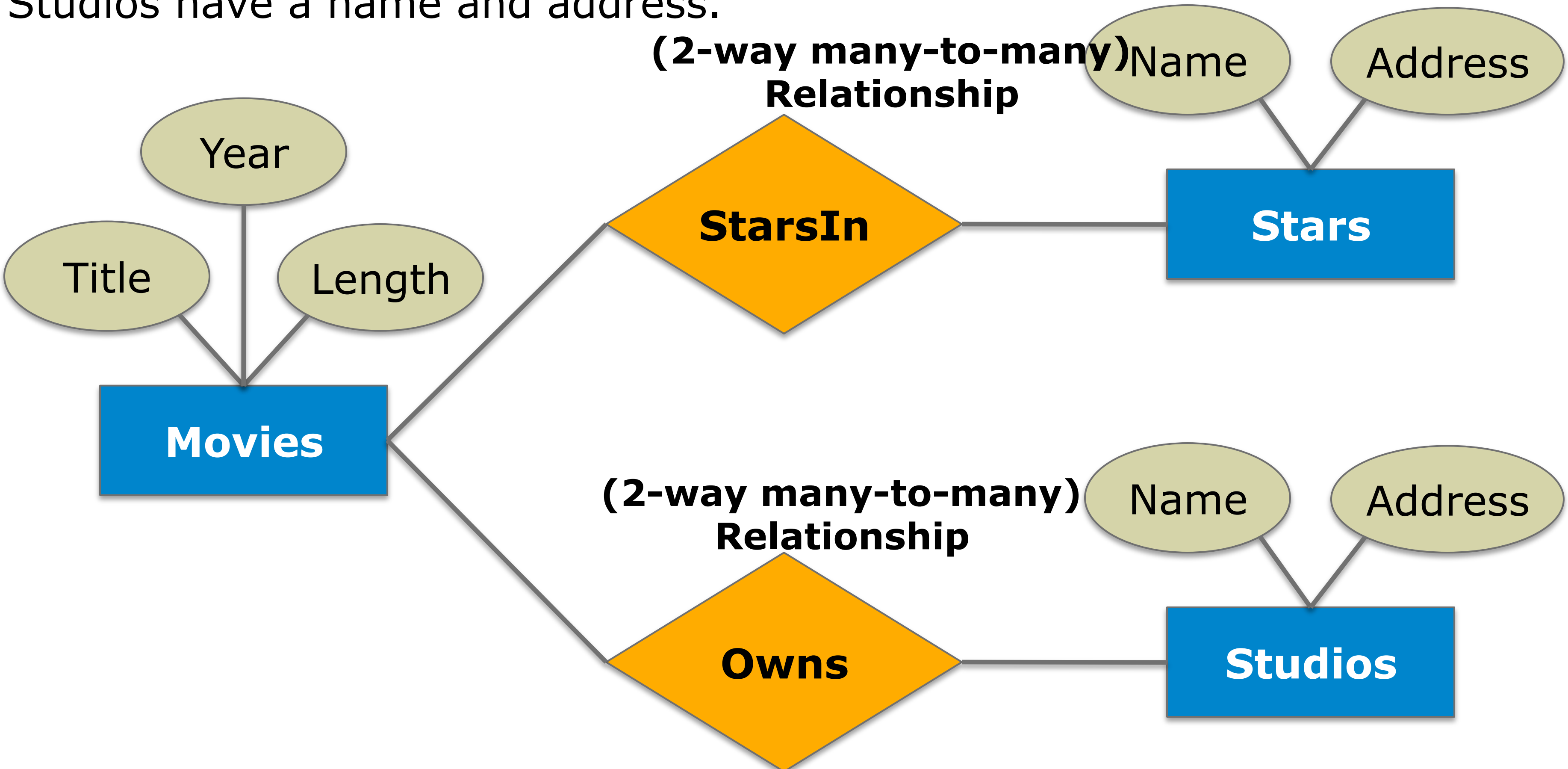
- Install the Postgresql open source database
- For educational and management purposes use the pgAdmin client to define schemas, insert data,
- For managing and accessing the Postgresql server, use the pgAdmin graphical client
  - Right click on `Postgresql`, and select `Connect`
  - Right click on `Databases`, and select `New Database`
  - Enter a new name for the database, and click `Okay`
  - Highlight the database, and select `Tools -> Query Tool`
  - Write SQL code (or open the examples), and select `Query -> Execute`

# Creating a schema and inserting some data

- Open file enrollment.sql
- Copy and paste its CREATE TABLE and INSERT commands in the Query Tool
- Run it – you now have the sample database!
- Run the first 3 SELECT commands to see the data you have in the database
  - You can run a command by highlighting it with the cursor and click run

# Example 2a

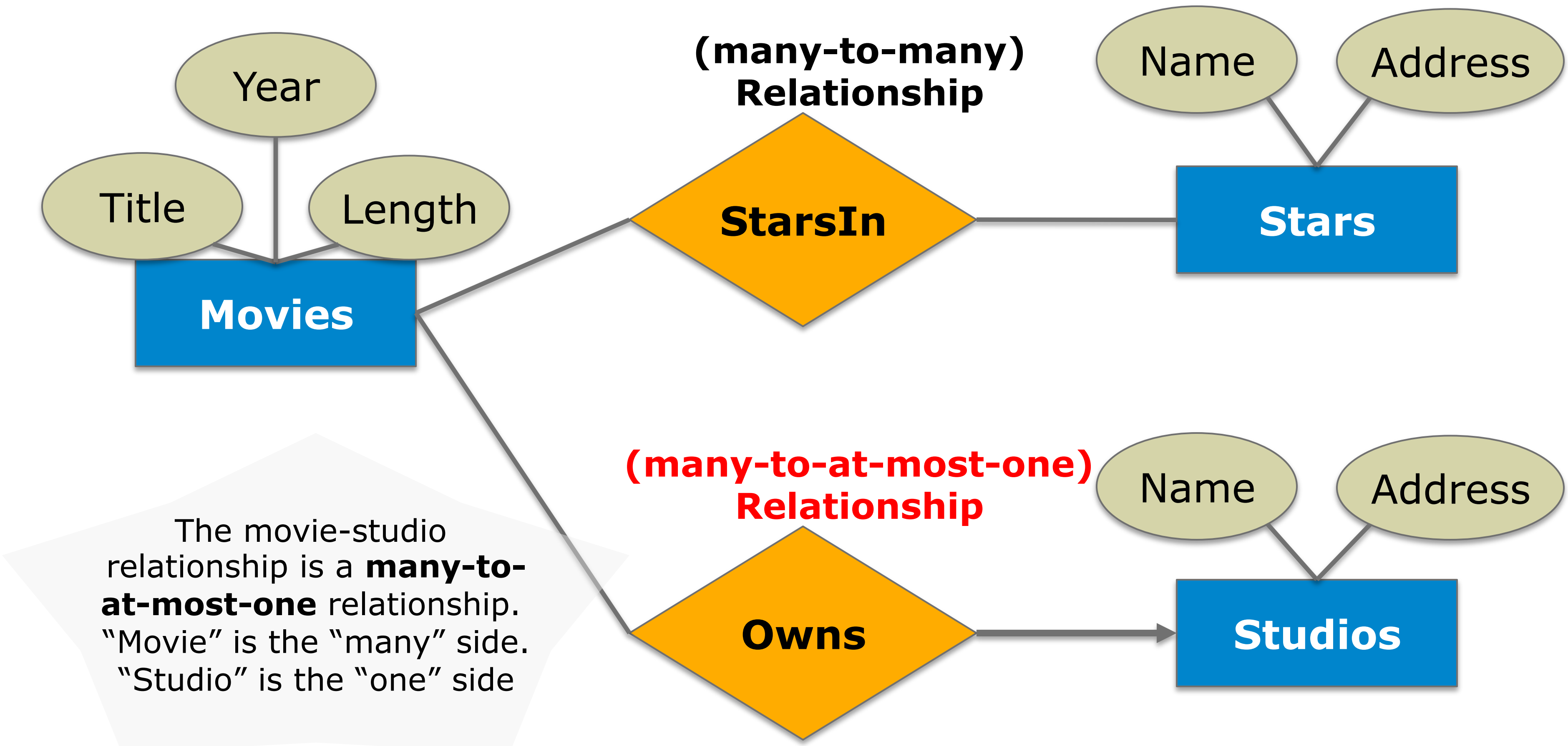
Movies have a title, a year of release and length (in minutes).
Actors have names and address.
Actors appear in movies.
A movie is (co-)owned by studios.
Studios have a name and address.



```
CREATE TABLE movies (
    ID          SERIAL PRIMARY KEY,
    title       TEXT,
    year        INTEGER,
    length      INTEGER,
)
CREATE TABLE stars (
    ID          SERIAL PRIMARY KEY,
    name        TEXT,
    address     TEXT
)
CREATE TABLE studios (
    ID          SERIAL PRIMARY KEY,
    name        TEXT,
    address     TEXT
)
CREATE TABLE starsin (
    ID              SERIAL,
    movie           INTEGER REFERENCES movies (ID) NOT NULL,
    star            INTEGER REFERENCES stars (ID) NOT NULL
)
CREATE TABLE ownership (
    ID              SERIAL,
    movie           INTEGER REFERENCES movies (ID) NOT NULL,
    owner           INTEGER REFERENCES studios (ID) NOT NULL
)
```

# Example 2b: many-to-at-most-one relationship

Modification to Example 2a:
A movie is owned by **at most one** studio.



# E/R→ Relational: Translation revisited for many-to-at-most-one relationship

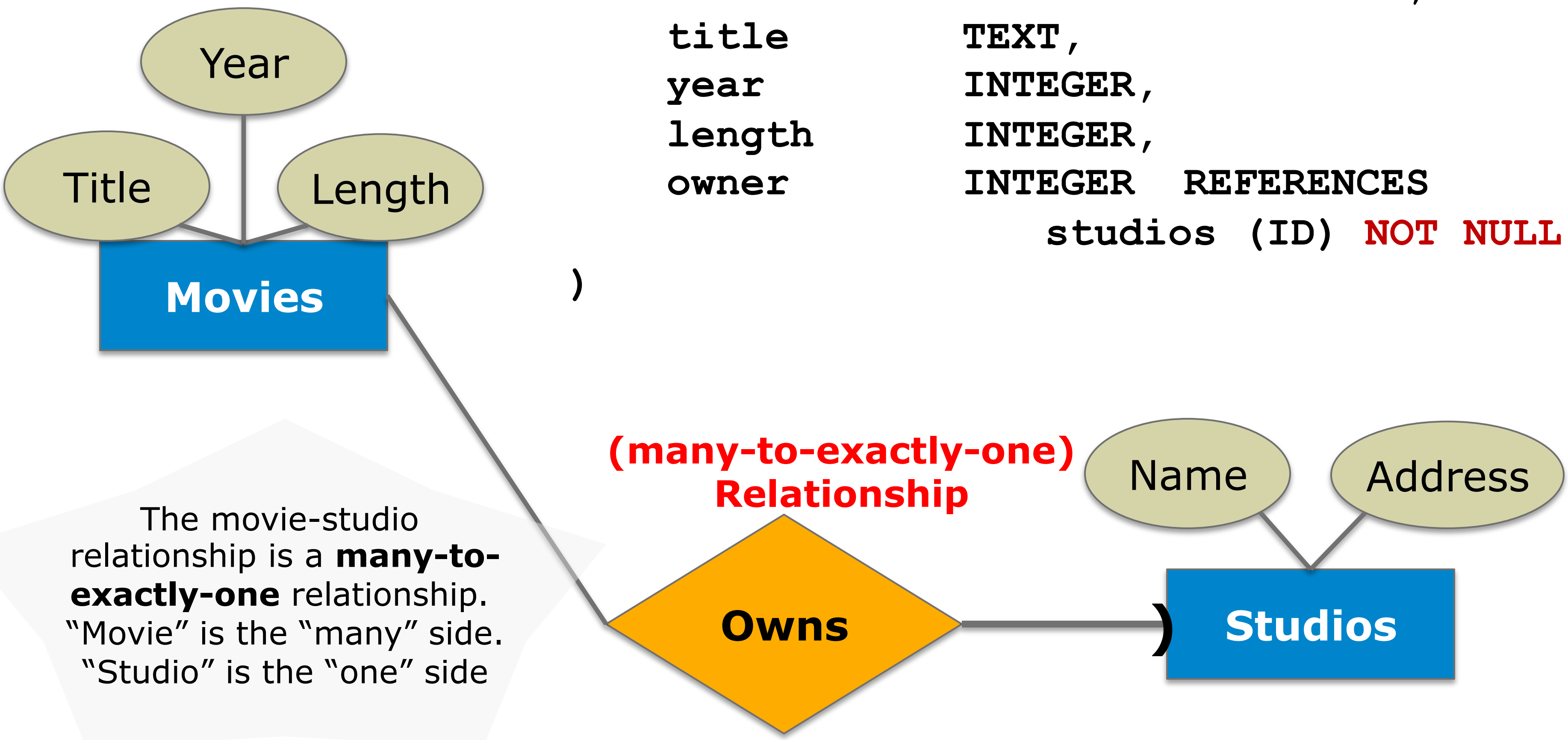
- For every entity, do the usual…
- For every **many-to-many** relationship, do the usual…
- For every **2-way many-to-at-most-one** relationship, where
  - $E_m$ is the “many” side
  - $E_o$ is the “one” side (pointed by the arrow)
  - **do not** create table, instead:
  - In the table corresponding to $E_m$ add a (non-required, i.e., potentially NULL) foreign key attribute referencing the ID of the table corresponding to $E_o$

```
CREATE TABLE movies (
    ID            SERIAL PRIMARY KEY,
    title         TEXT,
    year          INTEGER,
    length        INTEGER,
    owner         INTEGER  REFERENCES studios (ID)
)
CREATE TABLE stars (
    ID            SERIAL PRIMARY KEY,
    name          TEXT,
    address       TEXT
)
CREATE TABLE studios (
    ID            SERIAL PRIMARY KEY,
    name          TEXT,
    address       TEXT
)
CREATE TABLE starsin (
    ID                SERIAL,
    movie             INTEGER REFERENCES movies (ID) NOT NULL,
    star              INTEGER REFERENCES stars (ID) NOT NULL
)
```

# Example 2c: many-to-exactly-one relationship

Modification to Example 2a:
A movie **must** be owned by **one** studio.

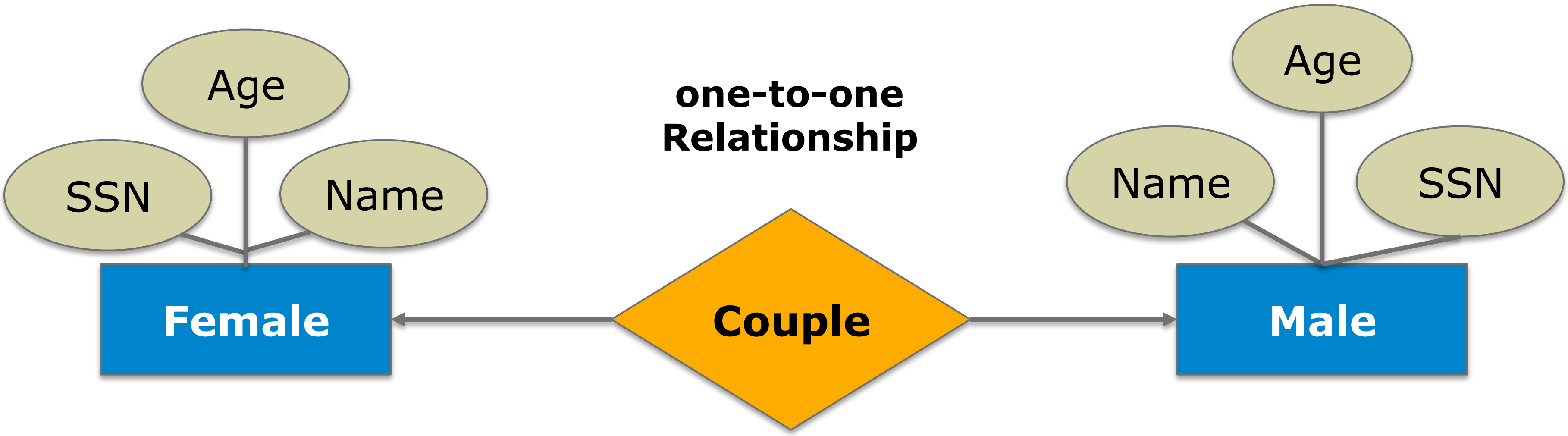
```
CREATE TABLE movies (
     ID              SERIAL PRIMARY KEY,
     title           TEXT,
     year            INTEGER,
     length          INTEGER,
     owner           INTEGER  REFERENCES
                         studios (ID) NOT NULL
)
```




The movie-studio relationship is a **many-to-exactly-one** relationship.
"Movie" is the "many" side.
"Studio" is the "one" side

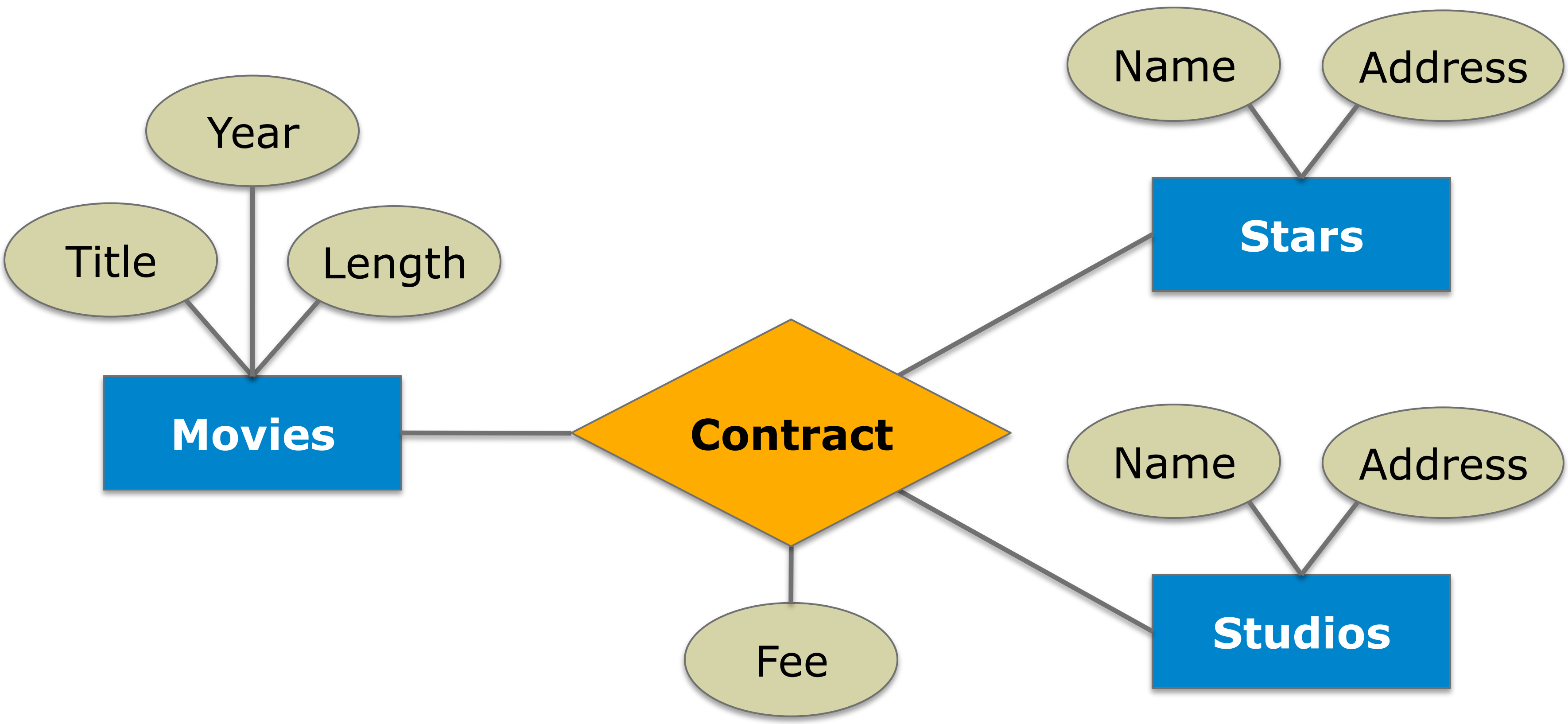
# Example 2d: one-to-one relationship

Consider a database of heterosexual couples
(we neglect homosexual couples, amazons and Warren Jeffs followers)




```
CREATE TABLE couple (
      husband INTEGER REFERENCES
               females (ID) NOT NULL UNIQUE,
      wife INTEGER REFERENCES
                  males (ID) NOT NULL UNIQUE
)
```

# Example 3: 3-Way Many-to-Many Relationship

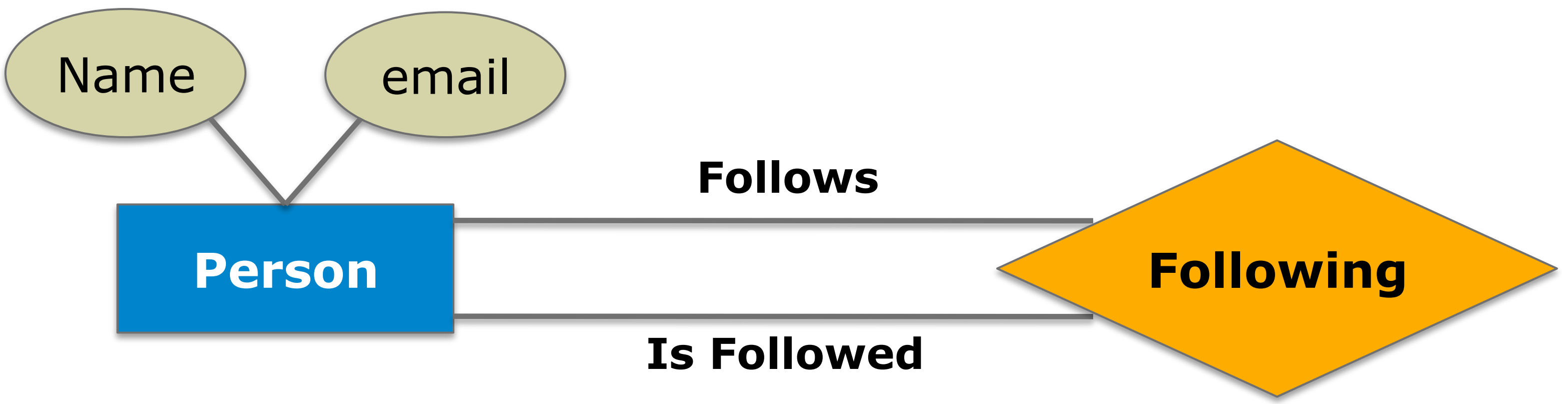



- A studio has contracted with a particular star to act in a particular movie
  - No ownership of movies by studios

```
CREATE TABLE contract (
    ID                SERIAL,
    movie             INTEGER REFERENCES movies (ID) NOT NULL,
    star              INTEGER REFERENCES stars (ID) NOT NULL,
    owner             INTEGER REFERENCES studios (ID) NOT NULL,
    fee               INTEGER
)
```

# Example 4a : Self-Relationships with Roles

Twitter Use Case



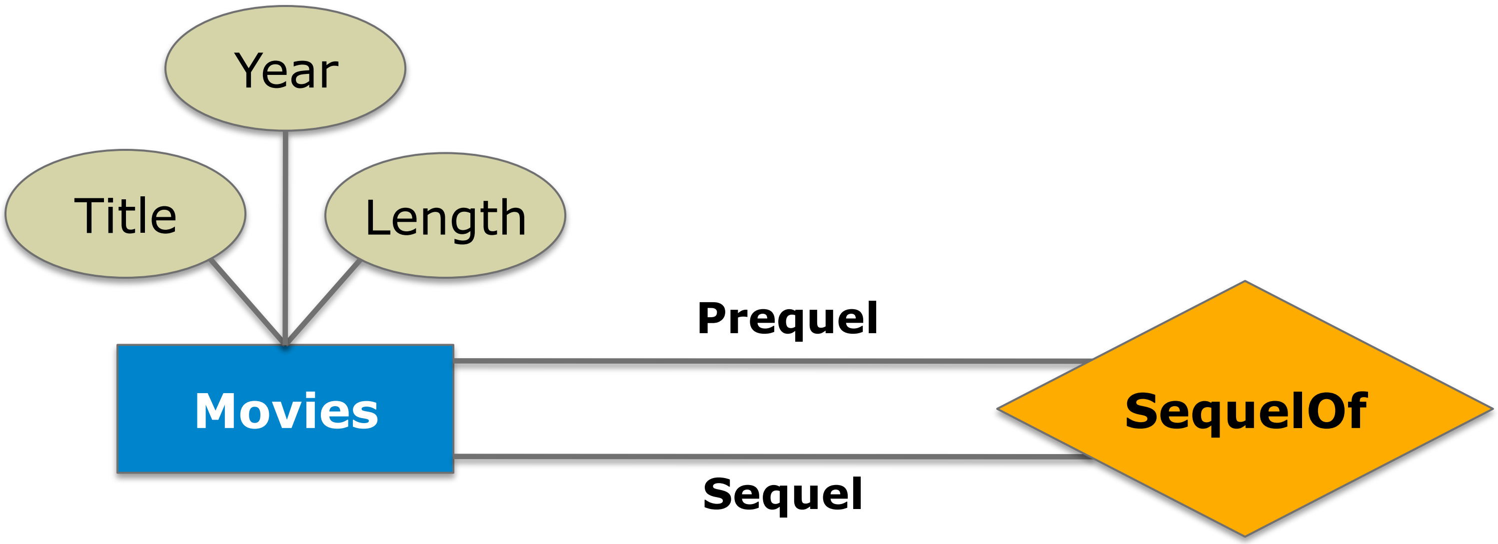
```
CREATE TABLE persons (
    ID          SERIAL PRIMARY KEY,
    …
)

CREATE TABLE following (
    ID              SERIAL,
    follows         INTEGER REFERENCES persons (ID) NOT NULL,
    isFollowed      INTEGER REFERENCES persons (ID) NOT NULL
)
```

Notice the use of roles as attributes names for the foreign keys

# Example 4b : Self-Relationships with Roles

Prequels and Sequels

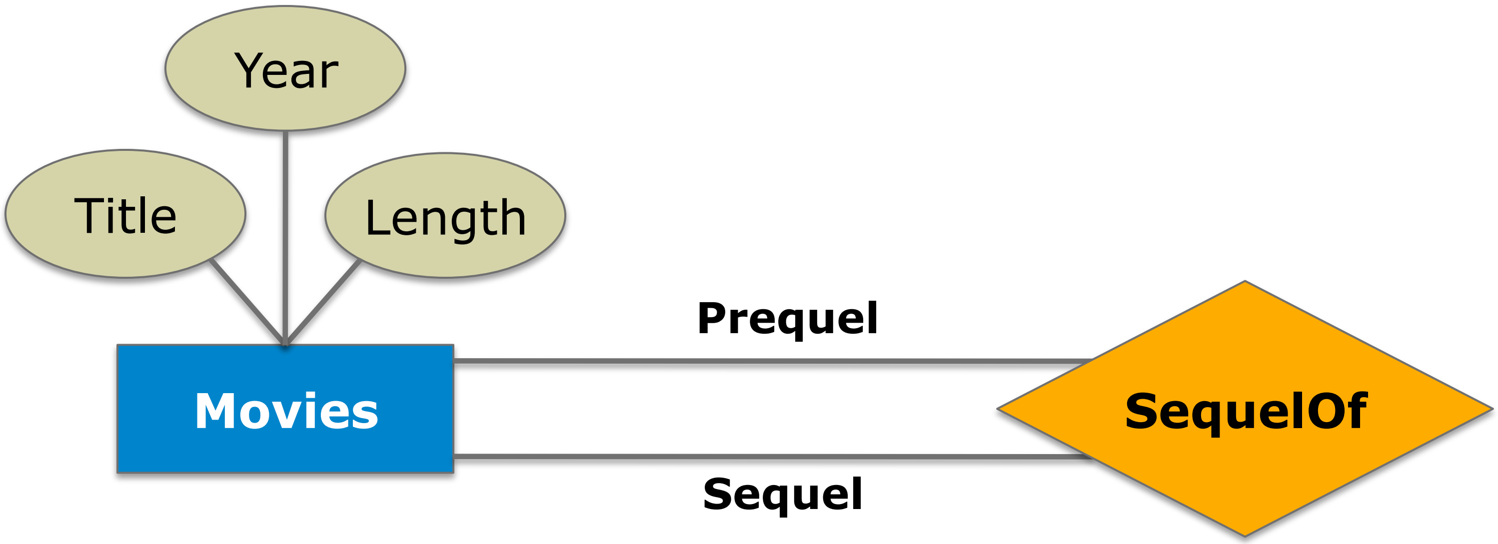


```
CREATE TABLE movies (
    ID          SERIAL PRIMARY KEY,
    …
)

CREATE TABLE sequelof (
    ID              SERIAL,
    prequel         INTEGER REFERENCES movies (ID) NOT NULL,
    sequel          INTEGER REFERENCES movies (ID) NOT NULL
)
```

# Example 4b : Self-Relationships with Roles – Questions on Meaning

What exactly are the prequel-sequel pairs?




"Terminator II: Judgment Day" is a sequel of "Terminator"

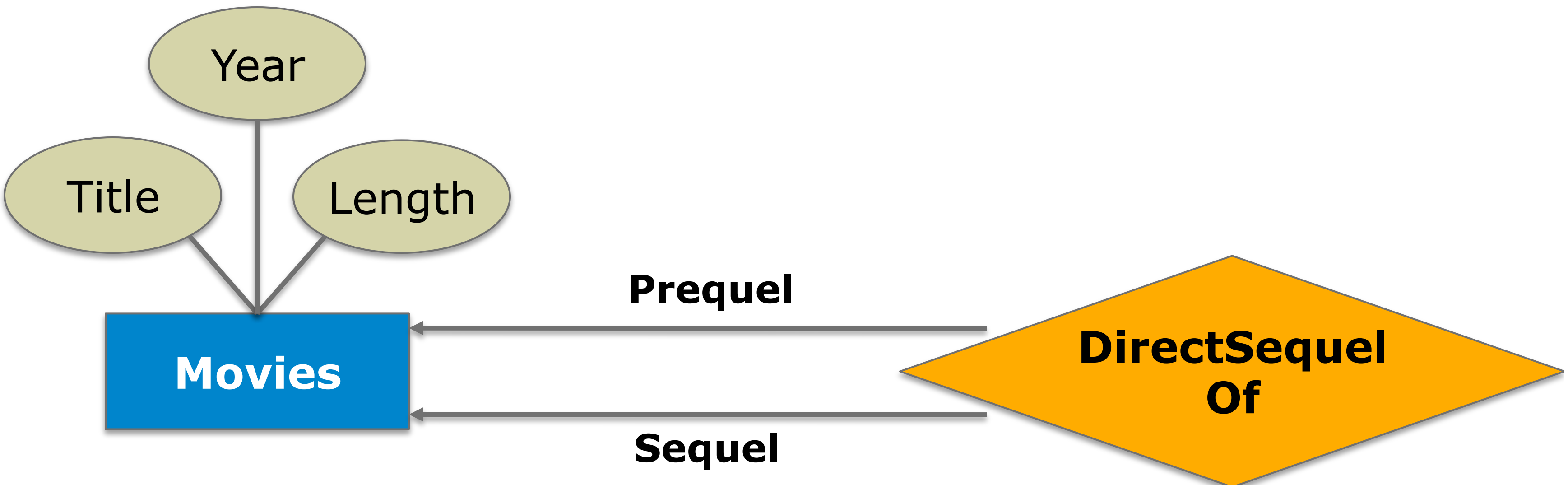
"Terminator III: Raise of the Machines" is a sequel of "Terminator II: Judgment Day"

**Is "Terminator III: Raise of the Machines" a sequel of "Terminator" ?**

# Example 4c : Interpreting sequels non-transitively

**Is "Terminator III: Raise of the Machines" a direct sequel of "Terminator" ? NO**

A movie has at most one direct "prequel" and at most one direct "sequel"




Modeling movie sequels by "DirectSequelOf" is preferable in OLTP to using transitive "SequelOf"

A lesson about good (OLTP?) database design:
- Good designs avoid redundancy.
- No stored piece of data should be inferable from other stored pieces of data

# To be Redundant or Not to be?

NOT

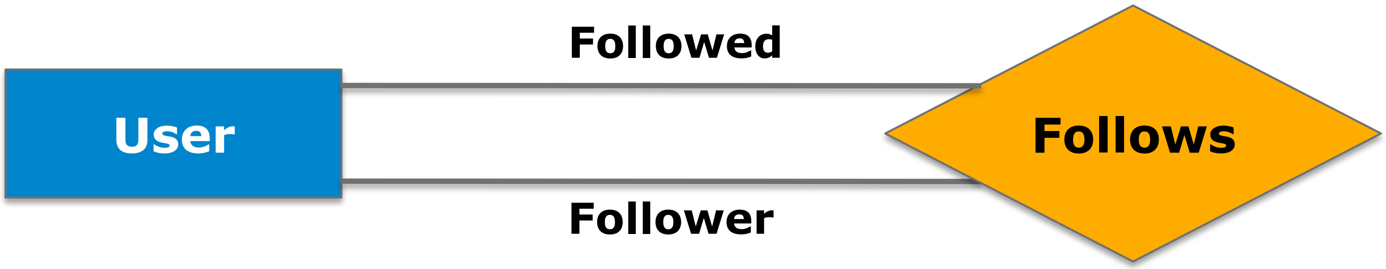
- Too many Friends-of-Friends
  - Even more Friends-of-Friends-of-Friends
    - If “Six Degrees of Separation” is true, the 6-step friends is not even saying anything
- A database with derivative data is harder to maintain

YES

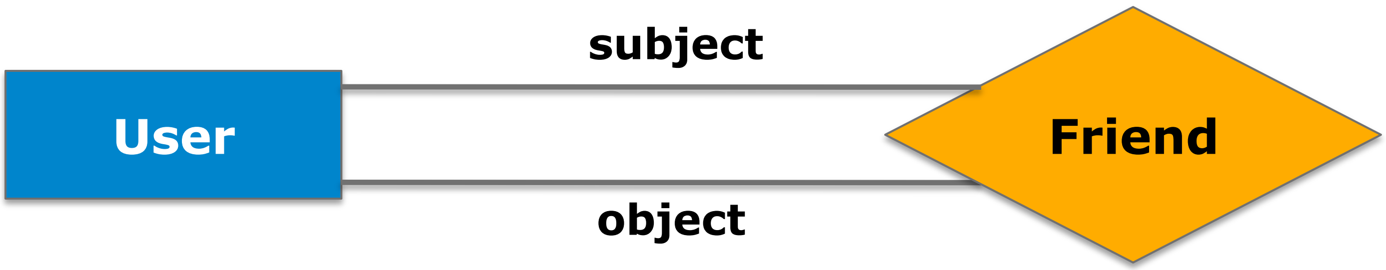
- Some derivations, interesting to OLAP, are too expensive to compute live
- If OLAP, maintenance is not primary concern

# Self-relationships without roles

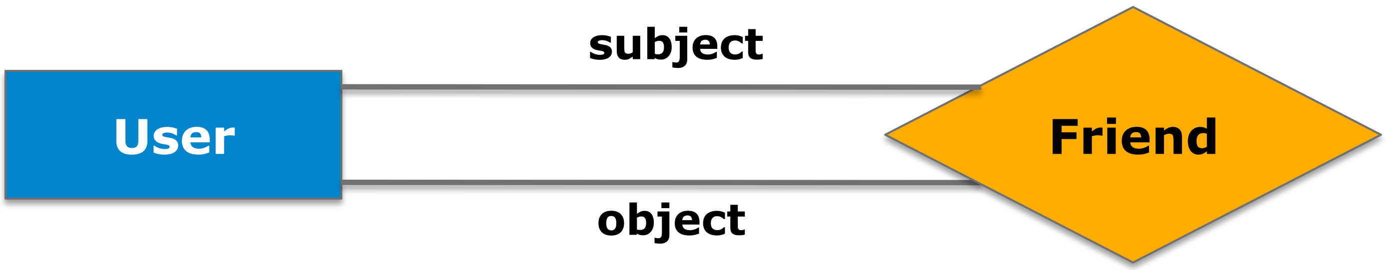
Twitter “followship” is a self-relationship with roles




Facebook “friendship” is a self-relationship without real roles



# A case where redundancy may be welcome




```
CREATE TABLE friend (
      subject INTEGER REFERENCES user (ID) NOT NULL,
      object INTEGER REFERENCES user (ID) NOT NULL
)
```

If Subject is Facebook friend of Object,
then Object is Facebook friend of Subject.
Is it redundant to explicitly represent both facts in “friend”?
Yes, but makes some queries much easier and faster.

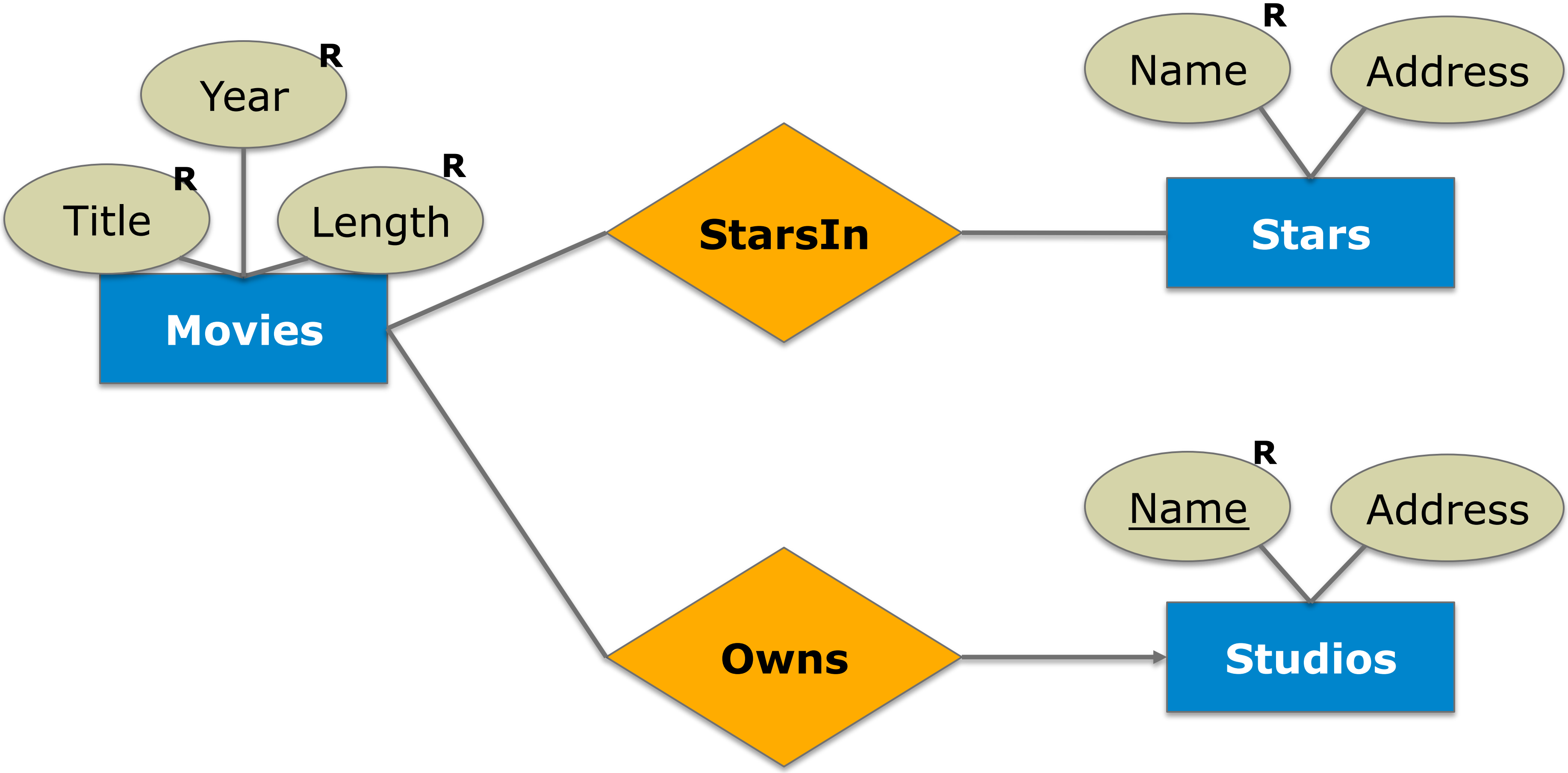
# Designing Schemas Using Entity-Relationship modeling

Additional Topics

# Example 5a: Constraints: uniqueness; required attributes

In addition to Example 2b's assumptions, let us also assume that:

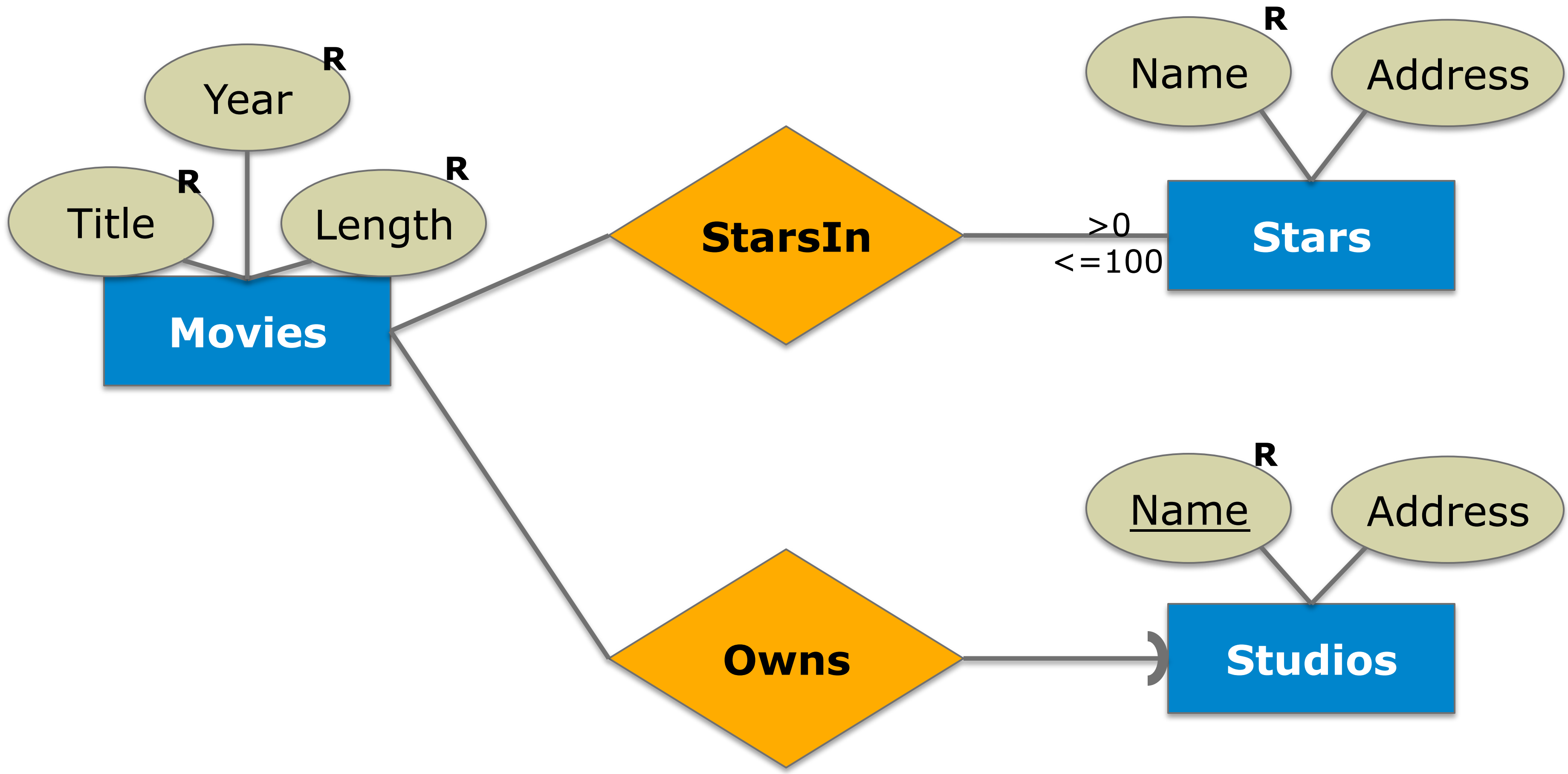
- title, year, length, star name and studio name are required attributes of the respective entities
    - default is that an attribute value may be **null**
- studios have unique names, i.e., no two studios may have the same name



# Example 5b: Constraints: Required relationship; cardinality ranges

In addition to Example 2c's assumptions, let us also assume that:

- a movie is owned by **exactly one** studio
  - so far we had not assumed that the owning studio has to be known (not null)
- a movie must have at least one actor and no more than 100



# SQL Schema for Examples 5a, 5b

```
CREATE TABLE movies (
    ID          SERIAL PRIMARY KEY,
    title       TEXT NOT NULL,
    year        INTEGER NOT NULL,
    length      INTEGER NOT NULL,
    owner       INTEGER  REFERENCES studios (ID) NOT NULL
)
CREATE TABLE stars (
    ID          SERIAL PRIMARY KEY,
    name        TEXT NOT NULL,
    address     TEXT
)
CREATE TABLE studios (
    ID          SERIAL PRIMARY KEY,
    name        TEXT NOT NULL UNIQUE,
    address     TEXT
)
CREATE TABLE starsin (
    ID              SERIAL,
    movie           INTEGER REFERENCES movies (ID) NOT NULL,
    star            INTEGER REFERENCES stars (ID) NOT NULL
)
```

# A sample database

**stars**

| ID | name | address |
|---|---|---|
| 1 | Al Pacino | New York, NY |
| 2 | Harrison Ford | Beverly Hills, CA |
| 3 | Tom Hanks | Santa Monica, CA |

**studios**

| ID | name | address |
|---|---|---|
| 1 | 20th Century Fox | Century City, CA |
| 2 | Paramount Productions | Hollywood, CA |
| 3 | Universal Pictures | Universal City, CA |

**starsin**

| ID | movie | star |
|---|---|---|
| 1 | 1 | 3 |
| 2 | 2 | 1 |
| 3 | 3 | 2 |
| 4 | 4 | 1 |

**movies**

| ID | title | year | length | owner |
|---|---|---|---|---|
| 1 | Forrest Gump | 1994 | 142 | 2 |
| 2 | The Godfather | 1972 | 175 | 2 |
| 3 | Star Wars | 1977 | 121 | 1 |
| 4 | Scent of a Woman | 1992 | 157 | 3 |

# Why do we want constraints? What happens when they are violated?

- Protect the database from erroneous data entry
- Prevent database states that are inconsistent with the rules of the business process you want to capture
- Whenever you attempt to change (insert, delete, update) the database in a way that violates a constraint the database will prevent the change
  - Try it out on the sample databases of the class page

# Some constraints are not implemented by some SQL database systems

- Consider the cardinality constraint that a movie has between 1 and 100 actors.
- The SQL standard provides a way, named CHECK constraints, to declare such
  - its specifics will make more sense once we have seen SQL queries
- However, no open source database implements the CHECK constraints.

- Project Phase I: Introduce such constraints on your E/R, despite the fact that you will not be able to translate them to the SQL schema

# Vice versa: SQL allows some constraints that are not in plain E/R

Notable cases:

- Attribute value ranges
  - Example: Declare that the year of movies is after 1900
- Multi-attribute UNIQUE
  - Example: Declare that the (title, year) attribute value combination is unique

# Added constraints of previous slide

```
CREATE TABLE movies (
    ID          SERIAL PRIMARY KEY,
    title       TEXT NOT NULL,
    year        INTEGER NOT NULL CHECK (year > 1900),
    length      INTEGER NOT NULL,
    owner       INTEGER  REFERENCES studios (ID) NOT NULL,
    UNIQUE (title, year)
)
CREATE TABLE stars (
    ID          SERIAL PRIMARY KEY,
    name        TEXT NOT NULL,
    address     TEXT
)
CREATE TABLE studios (
    ID          SERIAL PRIMARY KEY,
    name        TEXT NOT NULL UNIQUE,
    address     TEXT
)
CREATE TABLE starsin (
    ID              SERIAL,
    movie           INTEGER REFERENCES movies (ID) NOT NULL,
    star            INTEGER REFERENCES stars (ID) NOT NULL
)
```

# Example 6: tricky flavors of one-to-one relationships

Assume that a president manages exactly one studio and
a studio may have at most one president.
Notice: a studio may not have a president but
in order to be a president one has to manage a studio.

# 1st candidate

```
CREATE TABLE presidents (
    ID          SERIAL PRIMARY KEY,
    name        TEXT,
    age         INTEGER,
)

CREATE TABLE studios (
    ID          SERIAL PRIMARY KEY,
    name        TEXT,
    address     TEXT
)

CREATE TABLE management (
     manager INTEGER REFERENCES presidents (ID) NOT NULL UNIQUE
     manages INTEGER REFERENCES studios (ID) NOT NULL UNIQUE
)
```

One may be a president WITHOUT managing any studio
=> This design fails to capture a given constraint

# 2nd candidate

```
CREATE TABLE presidents (
    ID            SERIAL PRIMARY KEY,
    name          TEXT,
    age           INTEGER,
    manages       INTEGER  REFERENCES studios (ID) NOT NULL UNIQUE
)

CREATE TABLE studios (
    ID            SERIAL PRIMARY KEY,
    name          TEXT,
    address       TEXT
)
```

Guarantees that in order to be president, one has to manage a studio

Guarantees that no two presidents may manage the same studio

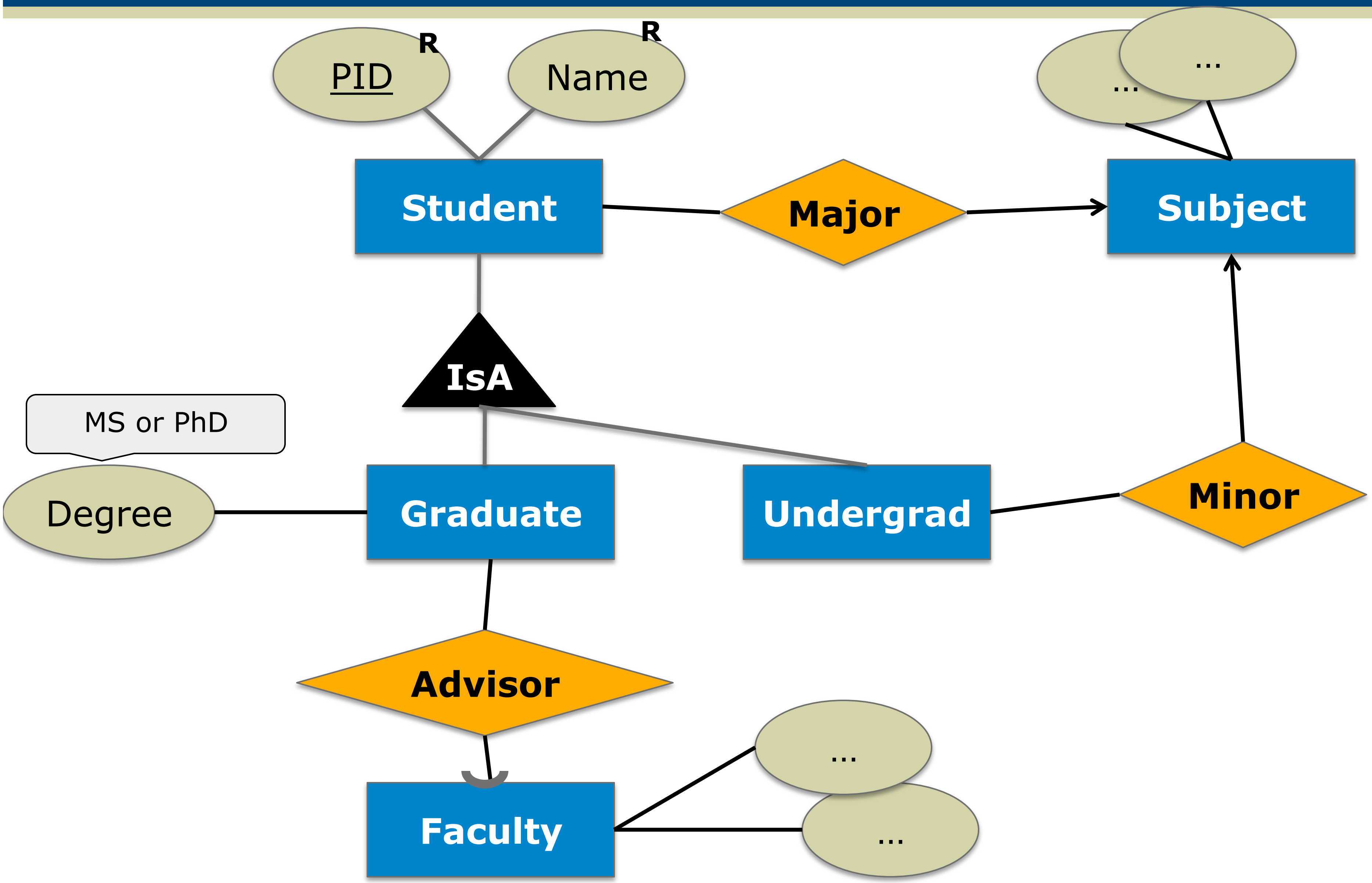
# 3rd candidate

3rd candidate is also not preferred. Why? What constraint it misses?

```
CREATE TABLE presidents (
    ID          SERIAL PRIMARY KEY,
    name        TEXT,
    age         INTEGER
)

CREATE TABLE studios (
    ID          SERIAL PRIMARY KEY,
    name        TEXT,
    address     TEXT,
    managedBy   INTEGER  REFERENCES presidents (ID) UNIQUE
)
```

# Example 6: Subclassing



# Schemas for subclassing: Candidate 1

```
CREATE TABLE student(
        ID      SERIAL PRIMARY KEY,
        pid      TEXT NOT NULL UNIQUE,
        name     TEXT NOT NULL,
        major   INTEGER REFERENCES subject(ID)
)
CREATE TABLE undergrad(
        studentid       INTEGER REFERENCES student(ID) NOT NULL,
        minor           INTEGER REFERENCES subject(ID)
)
CREATE TABLE graduate(
        studentid      INTEGER REFERENCES student(ID) NOT NULL,
        degree  TEXT NOT NULL CHECK (degree="PhD" OR degree="MS"),
        advisor         INTEGER REFERENCES faculty(ID) NOT NULL
)
CREATE TABLE subject(
        ID      SERIAL PRIMARY KEY,
        …
)
CREATE TABLE faculty(
        ID      SERIAL PRIMARY KEY,
        …
)
```

+ captures constraints
- Information about graduates is spread on two tables
- Creating a report about students is a tricky query

To appreciate the above wait till we discuss SQL

# Schemas for subclassing: Candidate 2

```
CREATE TABLE student(
       ID      SERIAL PRIMARY KEY,
       pid      TEXT NOT NULL UNIQUE,
       name     TEXT NOT NULL,
       kind     CHAR(1) CHECK (kind='U' OR kind='S'),
       major   INTEGER REFERENCES subject(ID),
       minor    INTEGER REFERENCES subject(ID),
       degree          TEXT CHECK (degree="PhD" OR degree="MS"),
       advisor         INTEGER REFERENCES faculty(ID)
)
CREATE TABLE subject(
       ID      SERIAL PRIMARY KEY,
       …
)
CREATE TABLE faculty(
       ID      SERIAL PRIMARY KEY,
       …
)
```

-misses constraints
E.g., notice that it does not capture that a graduate student must have an advisor since we had to make the advisor attribute non-required in order to accommodate having undergraduates in the same table

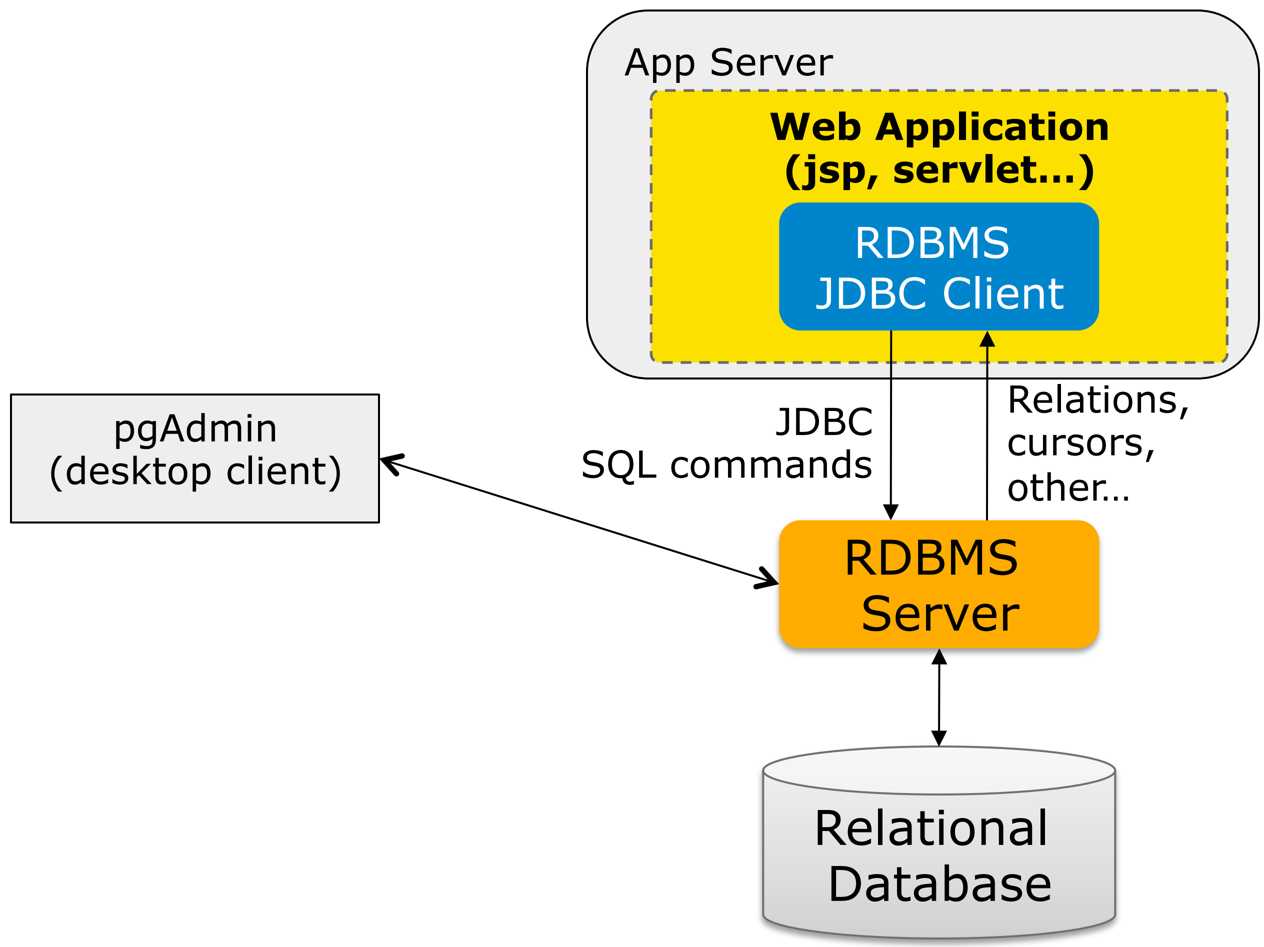
# Not covered E/R features

- Weak entities
  - double-lined entities and relationships
- Many-to-Many-to-One 3-way (or more) relationships
- Necessary participation of entity in relationship
- … more

# Programming on Databases with SQL

# Writing programs on databases: JDBC

- How client opens connection with a server
- How access & modification commands are issued
- …



# … some easy hands-on experience

- Install the Postgresql open source database
- For educational and management purposes use the pgAdmin client to define schemas, insert data,
  - See online instructions
- For managing and accessing the Postgresql server, use the pgAdmin graphical client
  - Right click on `Postgresql`, and select `Connect`
  - Right click on `Databases`, and select `New Database`
  - Enter a new name for the database, and click `Okay`
  - Highlight the database, and select `Tools -> Query Tool`
  - Write SQL code (or open the examples), and select `Query -> Execute`

# Creating a schema and inserting some data

- Open file enrollment.sql
- Copy and paste its CREATE TABLE and INSERT commands in the Query Tool
- Run it – you now have the sample database!
- Run the first 3 SELECT commands to see the data you have in the database

# Access (Query) & Modification Language: SQL

- SQL
  - used by the database user
  - **declarative**:we only describe **what** we want to retrieve
  - based on tuple relational calculus
- The result of a query is a table
- Internal Equivalent of SQL: Relational Algebra
  - used internally by the database system
  - **procedural** (operational): describes **how** query is executed
- The solutions to the following examples are on the class page download

# SQL: Basic, single-table queries

- Basic form

```
SELECT r.A1,…,r.AN
FROM R r
WHERE <condition>
```

- **WHERE** clause is optional
- Have tuple variable r range over the tuples of R, qualify the ones that satisfy the (boolean) condition and return the attributes $A_1, \ldots, A_N$

Find first names and last names of all students

```
SELECT  s.first_name, s.last_name
FROM    students s;
```

Display all columns of all students whose first name is John; project all attributes

```
SELECT s.id, s.pid, s.first_name,
          s.last_name
FROM students s
WHERE s.first_name = 'John'
```

… and its shorthand form

```
SELECT  *
FROM    students s
WHERE   s.first_name = 'John';
```

# SQL Queries: Joining together multiple tables

- Basic form

```
SELECT …,ri.Aj,…
FROM R1 r1,…,RM rM
WHERE <condition>
```

- When more than one relations in the **FROM** clause have an attribute named **A**, we refer to a specific **A** attribute as **<RelationName>.A**
- Hardest to get used to, yet most important feature of SQL

Produce a table that shows the pid, first name and last name of every student enrolled in the class with ID 1, along with the number of credit units in the "class 1" enrollment

```
SELECT  s.pid, s.first_name,
        s.last_name, e.credits
FROM    students s, enrollment e
WHERE   s.id = e.student
          AND e.class = 1 ;
```

# (repeat)

**Classes**

| id | name | number | date_code | start_time | end_time |
|---|---|---|---|---|---|
| 1 | Web stuff | MAS201 | TuTh | 2:00 | 3:20 |
| 2 | Databases | CSE132A | TuTh | 3:30 | 4:50 |
| 4 | VLSI | CSE121 | F | *null* | *null* |

**Enrollment**

| id | class | student | credits |
|---|---|---|---|
| 1 | 1 | 1 | 4 |
| 2 | 1 | 2 | 3 |
| 3 | 4 | 3 | 4 |
| 4 | 1 | 3 | 3 |

**Students**

| id | pid | first_name | last_name |
|---|---|---|---|
| 1 | 8888888 | John | Smith |
| 2 | 1111111 | Mary | Doe |
| 3 | 2222222 | *null* | Chen |

# Take One: Understanding FROM as producing all combinations of tuples from the tables of the FROM clause

SELECT s.pid, s.first_name, s.last_name, e.credits
**FROM students s, enrollment e**
WHERE s.id = e.student AND e.class = 1

"FROM" produces all 12 tuples made from one "students" tuple and one "enrollment" tuple

| Student s part of the tuple | | | | Enrollment e part of the tuple | | | |
|---|---|---|---|---|---|---|---|
| s.id | s.pid | s.first_name | s.last_name | e.id | e.class | e.student | e.credits |
| 1 | 88.. | John | Smith | 1 | 1 | 1 | 4 |
| 1 | 88.. | John | Smith | 2 | 1 | 2 | 3 |
| 1 | 88.. | John | Smith | 3 | 4 | 3 | 4 |
| 1 | 88.. | John | Smith | 4 | 1 | 3 | 3 |
| 2 | 11.. | Mary | Doe | 1 | 1 | 1 | 4 |
| 2 | 11.. | Mary | Doe | 2 | 1 | 2 | 3 |
| 2 | 11.. | Mary | Doe | 3 | 4 | 3 | 4 |
| 2 | 11.. | Mary | Doe | 4 | 1 | 3 | 3 |
| 3 | 22.. | *null* | Chen | 1 | 1 | 1 | 4 |
| 3 | 22.. | *null* | Chen | 2 | 1 | 2 | 3 |
| 3 | 22.. | *null* | Chen | 3 | 4 | 3 | 4 |
| 3 | 22.. | *null* | Chen | 4 | 1 | 3 | 3 |

# Take One: or understanding FROM as nested loops (producing all combinations)

SELECT s.pid, s.first_name, s.last_name, e.credits
FROM **students s, enrollment e**
WHERE s.id = e.student AND e.class = 1 ;

for **s** ranging over **students** tuples
  for **e** ranging over **enrollment** tuples
    output a tuple with all **s** attributes and **e** attributes

Student part of the tuple ⟷ Enrollment part of the tuple

| s.id | s.pid | s.first_name | s.last_name | e.id | e.class | e.student | e.credits |
|---|---|---|---|---|---|---|---|
| 1 | 88.. | John | Smith | 1 | 1 | 1 | 4 |
| 1 | 88.. | John | Smith | 2 | 1 | 2 | 3 |
| 1 | 88.. | John | Smith | 3 | 4 | 3 | 4 |
| 1 | 88.. | John | Smith | 4 | 1 | 3 | 3 |
| 2 | 11.. | Mary | Doe | 1 | 1 | 1 | 4 |
| 2 | 11.. | Mary | Doe | 2 | 1 | 2 | 3 |
| 2 | 11.. | Mary | Doe | 3 | 4 | 3 | 4 |
| 2 | 11.. | Mary | Doe | 4 | 1 | 3 | 3 |
| 3 | 22.. | *null* | Chen | 1 | 1 | 1 | 4 |
| 3 | 22.. | *null* | Chen | 2 | 1 | 2 | 3 |
| 3 | 22.. | *null* | Chen | 3 | 4 | 3 | 4 |
| 3 | 22.. | *null* | Chen | 4 | 1 | 3 | 3 |

# The order in FROM clause is unimportant

- FROM **students s, enrollment e**

- FROM **enrollment e, students s**

produce the same combinations (pairs) of student + enrollment

# … with shorter column names

SELECT s.pid, s.first_name, s.last_name, e.credits
FROM **students s, enrollment e**
WHERE **s.id = e.student AND e.class = 1** ;

"FROM" produces all 12 tuples made from one "students" tuple and one "enrollment" tuple

Student part of the tuple — Enrollment part of the tuple

| s.id | pid | first_name | last_name | e.id | class | student | credits |
|---|---|---|---|---|---|---|---|
| 1 | 88.. | John | Smith | 1 | 1 | 1 | 4 |
| 1 | 88.. | John | Smith | 2 | 1 | 2 | 3 |
| 1 | 88.. | John | Smith | 3 | 4 | 3 | 4 |
| 1 | 88.. | John | Smith | 4 | 1 | 3 | 3 |
| 2 | 11.. | Mary | Doe | 1 | 1 | 1 | 4 |
| 2 | 11.. | Mary | Doe | 2 | 1 | 2 | 3 |
| 2 | 11.. | Mary | Doe | 3 | 4 | 3 | 4 |
| 2 | 11.. | Mary | Doe | 4 | 1 | 3 | 3 |
| 3 | 22.. | *null* | Chen | 1 | 1 | 1 | 4 |
| 3 | 22.. | *null* | Chen | 2 | 1 | 2 | 3 |
| 3 | 22.. | *null* | Chen | 3 | 4 | 3 | 4 |
| 3 | 22.. | *null* | Chen | 4 | 1 | 3 | 3 |

# Understanding WHERE as qualifying the tuples that satisfy the condition

SELECT s.pid, s.first_name, s.last_name, e.credits
FROM students s, enrollment e
**WHERE s.id = e.student AND e.class = 1 ;**

| s.id | s.pid | s.first_name | s.last_name | e.id | e.class | e.student | e.credits |
|---|---|---|---|---|---|---|---|
| 1 | 88.. | John | Smith | 1 | 1 | 1 | 4 |
| 1 | 88.. | John | Smith | 2 | 1 | 2 | 3 |
| ~~1~~ | ~~88..~~ | ~~John~~ | ~~Smith~~ | ~~3~~ | ~~4~~ | ~~3~~ | ~~4~~ |
| ~~1~~ | ~~88..~~ | ~~John~~ | ~~Smith~~ | ~~4~~ | ~~1~~ | ~~3~~ | ~~3~~ |
| ~~2~~ | ~~11..~~ | ~~Mary~~ | ~~Doe~~ | ~~1~~ | ~~1~~ | ~~1~~ | ~~4~~ |
| ~~2~~ | ~~11..~~ | ~~Mary~~ | ~~Doe~~ | ~~2~~ | ~~1~~ | ~~2~~ | ~~3~~ |
| 2 | 11.. | Mary | Doe | 3 | 4 | 3 | 4 |
| ~~2~~ | ~~11..~~ | ~~Mary~~ | ~~Doe~~ | ~~4~~ | ~~1~~ | ~~3~~ | ~~3~~ |
| ~~3~~ | ~~22..~~ | ~~*null*~~ | ~~Chen~~ | ~~1~~ | ~~1~~ | ~~1~~ | ~~4~~ |
| ~~3~~ | ~~22..~~ | ~~*null*~~ | ~~Chen~~ | ~~2~~ | ~~1~~ | ~~2~~ | ~~3~~ |
| ~~3~~ | ~~22..~~ | ~~*null*~~ | ~~Chen~~ | ~~3~~ | ~~4~~ | ~~3~~ | ~~4~~ |
| ~~3~~ | ~~22..~~ | ~~*null*~~ | ~~Chen~~ | ~~4~~ | ~~1~~ | ~~3~~ | ~~3~~ |

# Understanding SELECT as keeping the listed columns (highlighted below)

| Students.id | pid | first_name | last_name | Enrollment.id | class | student | credits |
|---|---|---|---|---|---|---|---|
| 1 | 88.. | John | Smith | 1 | 1 | 1 | 4 |
| ~~1~~ | ~~88..~~ | ~~John~~ | ~~Smith~~ | ~~2~~ | ~~1~~ | ~~2~~ | ~~3~~ |
| 1 | 88.. | John | Smith | 3 | 4 | 3 | 4 |
| ~~1~~ | ~~88..~~ | ~~John~~ | ~~Smith~~ | ~~4~~ | ~~1~~ | ~~3~~ | ~~3~~ |
| 2 | 11.. | Mary | Doe | 1 | 1 | 1 | 4 |
| ~~2~~ | ~~11..~~ | ~~Mary~~ | ~~Doe~~ | ~~2~~ | ~~1~~ | ~~2~~ | ~~3~~ |
| 2 | 11.. | Mary | Doe | 3 | 4 | 3 | 4 |
| ~~2~~ | ~~11..~~ | ~~Mary~~ | ~~Doe~~ | ~~4~~ | ~~1~~ | ~~3~~ | ~~3~~ |
| ~~3~~ | ~~22..~~ | ~~*null*~~ | ~~Chen~~ | ~~1~~ | ~~1~~ | ~~1~~ | ~~4~~ |
| ~~3~~ | ~~22..~~ | ~~*null*~~ | ~~Chen~~ | ~~2~~ | ~~1~~ | ~~2~~ | ~~3~~ |
| ~~3~~ | ~~22..~~ | ~~*null*~~ | ~~Chen~~ | ~~3~~ | ~~4~~ | ~~3~~ | ~~4~~ |
| ~~3~~ | ~~22..~~ | ~~*null*~~ | ~~Chen~~ | ~~4~~ | ~~1~~ | ~~3~~ | ~~3~~ |

**SELECT s.pid, s.first_name, s.last_name, e.credits**

| Students .pid | Students.first_name | Students.last_name | Enrollment.credits |
|---|---|---|---|
| 88.. | John | Smith | 4 |
| 11.. | Mary | Doe | 3 |
| 22.. | *null* | Chen | 3 |

# Take Two on the previous exercises: The algebraic way

Produce a table that shows the pid, first name and last name of every student enrolled in the class with ID 1, along with the number of credit units in the "class 1" enrollment

SELECT s.pid, s.first_name, s.last_name, e.credits
FROM **students s JOIN enrollment e**
**ON s.id = e.student**
WHERE e.class = 1 ;

# Take two cont'd

FROM clause result

Student part of the tuple

Enrollment part of the tuple

| s.id | pid | first_name | last_name | e.id | class | student | credits |
|---|---|---|---|---|---|---|---|
| 1 | 88.. | John | Smith | 1 | 1 | 1 | 4 |
| 2 | 11.. | Mary | Doe | 2 | 1 | 2 | 3 |
| 3 | 22.. | *null* | Chen | 3 | 4 | 3 | 4 |
| 3 | 22.. | *null* | Chen | 4 | 1 | 3 | 3 |

WHERE clause result

| s.id | pid | first_name | last_name | e.id | class | student | credits |
|---|---|---|---|---|---|---|---|
| 1 | 88.. | John | Smith | 1 | 1 | 1 | 4 |
| 2 | 11.. | Mary | Doe | 2 | 1 | 2 | 3 |
| 3 | 22.. | *null* | Chen | 3 | 4 | 3 | 4 |
| ~~3~~ | ~~22..~~ | ~~*null*~~ | ~~Chen~~ | ~~4~~ | ~~1~~ | ~~3~~ | ~~3~~ |

Net result of the query is

| s.pid | first_name | last_name | credits |
|---|---|---|---|
| 88.. | John | Smith | 4 |
| 11.. | Mary | Doe | 3 |
| 22.. | *null* | Chen | 3 |

# Heuristics on writing queries

- Do you understand how queries work but have difficulty writing these queries yourself?

- The following heuristics will help you translate a requirement expressed in English into a query
  - The key point is to translate informal English into a precise English statement about which tuples your query should find in the database

# Hints for writing FROM/WHERE: Rephrase the statement, see it as a navigation across primary/foreign keys

Produce a table that shows the pid, first name and last name of every student enrolled in class 1, along with the number of credit units in his/her class 1 enrollment

- Find every enrollment tuple e
- that is an enrollment in class 1
  - i.e., e.class = 1
- and find the student tuple s that is connected to e
  - i.e., the student's id s.id appears in the enrollment tuple e as the foreign key e.student
- display the pid, first_name, last_name of s and the credits of e

e

**Enrollment**

| id | class | student | credits |
|---|---|---|---|
| 1 | 1 | 1 | 4 |
| … | | | |

s.id = e.student

s

**Students**

| id | pid | first_name | last_name |
|---|---|---|---|
| 1 | 8888888 | John | Smith |
| … | | | |

- **Find every enrollment tuple e**
- that is an enrollment in class 1
  - i.e., e.class = 1
- and find the student tuple s that is connected to e
  - i.e., the student's id s.id appears in the enrollment tuple e as the foreign key e.student
- display the pid, first_name, last_name of s and the credits of e

FROM **enrollment e**

- Find every enrollment tuple e
- **that is an enrollment in class 1**
  - **i.e., e.class = 1**
- and find the student tuple s that is connected to e
  - i.e., the student's id s.id appears in the enrollment tuple e as the foreign key e.student
- display the pid, first_name, last_name of s and the credits of e

FROM enrollment e
**WHERE e.class = 1**

- Find every enrollment tuple e
- that is an enrollment in class 1
    - i.e., e.class = 1
- **and find the student tuple s that is connected to e**
    - **i.e., the student's id s.id appears in the enrollment tuple e as the foreign key e.student**
- display the pid, first_name, last_name of s and the

We could have also said "and find **every** student tuple s that is connected" but we used our knowledge that there is exactly one connected student and instead said "**the** student"

```
FROM    enrollment e, students s
WHERE   e.class = 1
          AND e.student = s.id
```

```
FROM    enrollment e
          JOIN  students s
          ON e.student = s.id
WHERE   e.class = 1
```

- Find every enrollment tuple e
- that is an enrollment in class 1
    - i.e., e.class = 1
- and find the student tuple s that is connected to e
    - i.e., the student's id s.id appears in the enrollment tuple e as the foreign key e.student
- **display the pid, first_name, last_name of s and the credits of e**

```
SELECT s.pid, s.first_name, s.last_nam
          e.credits
FROM    enrollment e, students s
WHERE   e.class = 1
          AND e.student = s.id
```

```
SELECT s.pid, s.first_name, s.last_na
          e.credits
FROM    enrollment e
          JOIN  students s
          ON e.student = s.id
WHERE   e.class = 1
```

# SQL Queries: Nesting

- The **`WHERE`** clause can contain predicates of the form
  - **`attr/value IN <query>`**
  - **`attr/value NOT IN <query>`**
  - **`attr/value = <query>`**
- The predicate is satisfied if the **`attr`** or **`value`** appears in the result of the nested **`<query>`**
- Also
  - **`EXISTS <query>`**
  - **`NOT EXISTS <query>`**

# Nesting: Break the task into smaller

Produce a table that shows the pid, first name and last name of every student enrolled in the class named `MAS201', along with the number of credit units in his/her `MAS201' enrollment
Note: We assume that there are no two classes with the same name

```
SELECT  s.pid, s.first_name, s.last_name, e.credits
FROM    students s, enrollment e
WHERE   e.class = (SELECT c.id
                   FROM classes c
                   WHERE c.number = 'MAS201'
                  )
 AND s.id = e.student
```

{[id:1]} -> 1

Nested queries modularize the task:
Nested query finds the id of the MAS201 class.
Then the outer query behaves as if there were a "1" in lieu of the subquery

# IN

Produce a table that shows the pid, first name and last name of every student enrolled in the class named `MAS201', along with the number of credit units in his/her `MAS201' enrollment
Note: We assume that there are no two classes with the same name

```
SELECT  s.pid, s.first_name, s.last_name, e.credits
FROM    students s, enrollment e
WHERE   e.class IN (SELECT c.id
                     FROM classes c
                     WHERE c.number = 'MAS201'
                    )
 AND s.id = e.student
```

{[id:1]}

# Students + enrollments in class 1 Vs Students who enrolled in class 1

Imagine a weird university where a student is allowed to enroll multiple times in the same class

Produce a table that shows the pid, first name and last name of every student enrolled in the class with ID 1, along with the number of credit units in the "class 1" enrollment
=> The same student may appear many times, once for each enrollment

```
SELECT  s.pid, s.first_name,
        s.last_name, e.credits
FROM    students s, enrollment e
WHERE   s.id = e.student
          AND e.class = 1
```

Produce a table that shows the pid, first name and la st name of every student who has enrolled at least once in the "class 1".
=> Each student will appear at most once

```
SELECT  s.pid, s.first_name,
        s.last_name
FROM    students s
WHERE   s.id IN ( SELECT e.student
                  FROM enrollment e
                  WHERE e.class = 1
                )
```

# Uncorrelated subquery

```
SELECT  s.pid, s.first_name,
        s.last_name
FROM    students s
WHERE   s.id IN ( SELECT e.student
                  FROM enrollment e
                  WHERE e.class = 1
                )
```

"Uncorrelated" in the sense that the nested query could be a standalone query

Some nested queries are correlated (example later)

# EXISTS

Display the students enrolled in class 1,
only if the enrollment of class 2 is not empty

```
SELECT  s.pid, s.first_name, s.last_name
FROM    students s
WHERE   s.id IN ( SELECT e.student
                  FROM enrollment e
                  WHERE e.class = 1
        )
   AND EXISTS ( SELECT *
                FROM enrollment e
                WHERE e.class = 2
              )
```

Uncorrelated, also

# Correlated with EXISTS

Display the students enrolled in class 1

```
SELECT  s.pid, s.first_name, s.last_name
FROM    students s
WHERE   EXISTS ( SELECT e.student
                 FROM enrollment e
                 WHERE e.class = 1
                       AND e.student = s.id )
```

Correlation: the variable **s** comes from the outer query

# Exercise, on thinking cardinalities of tuples in the results

```
SELECT  s.pid, s.first_name, s.last_name, e.credits
FROM    students s, enrollment e
WHERE   e.class IN (SELECT c.id
                    FROM classes c
                    WHERE c.number = 'MAS201'
                   )
        AND s.id = e.student
```

EXERCISE: Under what condition the above query always produces the same result with the query below?

```
SELECT  s.pid, s.first_name, s.last_name, e.credits
FROM    students s, enrollment e, classes c
WHERE   c.number = 'MAS201'
        AND s.id = e.student
        AND e.class = c.id
```

# Exercise: Multiple JOINs

Produce a table that shows the pid, first name and last name of every student enrolled in the MAS201 class along with the number of credit units in his/her 135 enrollment

Take One:

```
SELECT  s.pid, s.first_name, s.last_name, e.credits
FROM    students s, enrollment e, classes c
WHERE   c.number = 'MAS201' AND s.id = e.student AND e.class = c.id
```

Take Two:

```
SELECT  s.pid, s.first_name, s.last_name, e.credits
FROM    (students s JOIN enrollment e ON s.id = e.student)
         JOIN classes c ON e.class = c.id
WHERE   c.number = 'MAS201'
```

# You can omit table names in SELECT, WHERE when attribute is unambiguous

```
SELECT  pid, first_name, last_name, credits
FROM    students, enrollment, classes
WHERE   number = 'MAS201'
        AND students.id = student
        AND class = classes.id ;
```

# SQL Queries, Aliases

- Use the same relation more than once in the same query or even the same `FROM` clause
- **Problem:** Find the Friday classes taken by students who take MAS201
  - also showing the students, i.e., produce a table where each row has the data of a MAS201 student and a Friday class he/she takes

Find the MAS201 students who take a Friday 11:00 am class

```
SELECT  s.id, s.first_name, s.last_name, cF.number
FROM    students s, enrollment eF, classes cF
WHERE date_code = 'F'
        AND eF.class = cF.id
        AND s.id = eF.student
        AND s.id IN
        (
            SELECT  student
            FROM    enrollment e201, classes c201
            WHERE   c201.id = e201.class
            AND c201.number = 'MAS201'
          )
```

Nested query computes the id's of students enrolled in MAS201

# Multiple aliases may appear in the same FROM clause

Find the MAS201 students who take a Friday 11:00 am class

```
SELECT  s.first_name, s.last_name, cF.number
FROM    students s, enrollment eF, classes cF,
        enrollment e201, classes c201
WHERE cF.date_code = 'F'
      AND eF.class = cF.id
      AND s.id = eF.student
      AND s.id = e201.student
      AND c201.id = e201.class
      AND c201.number = 'MAS201`
```

Under what conditions it computes the same result with previous page?

# DISTINCT

Find the other classes taken by MAS201 students
(I don't care which students)

```
SELECT  DISTINCT cOther.number
FROM    enrollment eOther, classes cOther,
        enrollment e201, classes c201
WHERE eOther.class = cOther.id
      AND eOther.student = e201.student
      AND c201.id = e201.class
      AND c201.number = 'MAS201`
```

# UNION ALL

Find all student ids for students who have taken class 1 or are named 'John'

```
( SELECT e.student
   FROM enrollment e
   WHERE e.class=1 )
UNION ALL
( SELECT s.id AS student
   FROM student s
   WHERE s.first_name='John'
)
```

If a student named John takes class 1 he will appear twice in the result

# UNION with non –duplicate results

```
( SELECT e.student
   FROM enrollment e
   WHERE e.class=1 )
UNION
( SELECT s.id AS student
   FROM student s
   WHERE s.first_name=‘John’
)
```

# SQL Queries: Aggregation & Grouping

- Aggregate functions: **SUM**, **AVG**, **COUNT**, **MIN**, **MAX**, and recently user defined functions as well
- **GROUP BY**

**Employee**

| Name | Dept | Salary |
|---|---|---|
| Joe | Toys | 45 |
| Nick | PCs | 50 |
| Jim | Toys | 35 |
| Jack | PCs | 40 |

**Example**: Find the average salary of all employees:

```
SELECT AVG(Salary) AS AvgSal
FROM Employee
```

| AvgSal |
|---|
| 42.5 |

**Example**: Find the average salary for each department:

```
SELECT Dept, AVG(Salary) AS AvgSal
FROM Employee
GROUP BY Dept
```

| Dept | AvgSal |
|---|---|
| Toys | 40 |
| PCs | 45 |

# SQL Grouping: Conditions that Apply on Groups

- **`HAVING <condition>`** may follow a **`GROUP BY`**clause
- If so, the condition applies to each group, and groups not satisfying the condition are eliminated

- **Example**: Find the average salary in each department that has more than 1 employee:

```
SELECT Dept,AVG(Salary) AS AvgSal
FROM Employee
GROUP BY Dept
HAVING COUNT(Name) >1
```

# Let’s mix features we’ve seen: Aggregation after joining tables

- **Problem:** List all enrolled students and the number of total credits for which they have registered

```
SELECT   students.id, first_name, last_name, SUM(credits)
FROM     students, enrollment
WHERE    students.id = enrollment.student
GROUP BY students.id, first_name, last_name
```

# ORDER BY and LIMIT

Order the student id’s of class 2 students according to their class 2 credits, descending

```
SELECT e.student
FROM enrollment e
WHERE e.class = 2
ORDER BY e.credits DESC
```

Order the student id’s of class 2 students according to their class 2 credits, descending **and display the Top 10**

```
SELECT e.student
FROM enrollment e
WHERE e.class = 2
ORDER BY e.credits DESC
LIMIT 10
```

# Combining features

Find the Top-5 classes taken by students of class 2,
i.e., the 5 classes (excluding class 2 itself)
with the highest enrollment of class 2 students,
display their numbers and how many class 2 students they have

```
SELECT  cF.number, COUNT(*)
FROM    enrollment eF, classes cF
WHERE eF.class = cF.id
      AND NOT(eF.class = 2)
      AND eF.student IN
      (
        SELECT  student
        FROM    enrollment e2
        WHERE   e201.class = 2
      )
GROUP BY cF.id, cF.number
ORDER BY cF.number
LIMIT 5
```

Grouping by both id and number ensures correctness even if multiple classes have the same number

# The outerjoin operator

- New construct in FROM clause
- R LEFT OUTER JOIN S ON R.<attr of R>=S.<attr of J>
- R FULL OUTER JOIN S ON R.<attr of R>=S.<attr of J>

**R**

| RJ | RV |
|---|---|
| 1 | RV1 |
| 2 | RV2 |

**S**

| SJ | SV |
|---|---|
| 1 | SV1 |
| 3 | SV3 |

```
SELECT *
FROM R LEFT OUTERJOIN S ON R.RJ=S.SJ
```

| RJ | RV | SJ | SV |
|---|---|---|---|
| 1 | RV1 | 1 | SV1 |
| 2 | RV2 | Null | Null |

```
SELECT *
FROM R FULL OUTERJOIN S ON R.RJ=S.SJ
```

| RJ | RV | SJ | SV |
|---|---|---|---|
| 1 | RV1 | 1 | SV1 |
| 2 | RV2 | Null | Null |
| Null | Null | 3 | SV3 |

# An application of outerjoin

- **Problem:** List all students and the number of total credits for which they have registered
  - Notice that you must also list non-enrolled students

```
SELECT   students.id, first_name, last_name, SUM(credits)
FROM     students LEFT OUTER JOIN enrollment ON
    students.id = enrollment.student
GROUP BY students.id, first_name, last_name
```

# SQL: More Bells and Whistles …

- Pattern matching conditions
  - `<attr> LIKE <pattern>`

Retrieve all students whose name contains “Sm”

```
SELECT *
FROM Students
WHERE name LIKE '%Sm%'
```

# …and a Few “Dirty” Points

- **Null values**
  - All comparisons involving NULL are **false** by definition
  - All aggregation operations, except **`COUNT(*)`**, ignore NULL values

# Null Values and Aggregates

- Example:

R

| a | b |
|---|---|
| x | 1 |
| x | 2 |
| x | null |
| null | null |
| null | null |

```
SELECT COUNT(a),COUNT(b),AVG(b), COUNT(*)
FROM R
GROUP BY a
```

| count(a) | count(b) | avg(b) | count(*) |
|---|---|---|---|
| 3 | 2 | 1.5 | 3 |
| 0 | 0 | null | 2 |

# Universal Quantification by Negation (difficult)

Problem:

- Find the students that take **every** class 'John Smith' takes

Rephrase:

- Find the students such that there is no class that 'John Smith' takes and they do not take

# Hints for writing FROM/WHERE: Rephrase the statement, see it as a navigation across primary/foreign keys

Produce a table that shows the pid, first name and last name of every student enrolled in the MAS201 class along with the number of credit units in his/her 135 enrollment

- Find every combination of a students tuple s, an enrollment tuple e, c where
- the students tuple s,
- is connected to the enrollment tuple e
  - i.e., the s.id appears in the enrollment tuple e as e.student,
- and e is connected to the classes tuple c
  - i.e., the e.class of e appears as c.id of the tuple c,
- whose c.number is MAS201

c

**Classes**

| id | name | number | date_code | start_time | end_time |
|---|---|---|---|---|---|
| 1 | Web stuff | MAS201 | TuTh | 2:00 | 3:20 |
| … | | | | | |

c.id = e.class

e

**Enrollment**

| id | class | student | credits |
|---|---|---|---|
| 1 | 1 | 1 | 4 |
| … | | | |

s.id = e.student

s

**Students**

| id | pid | first_name | last_name |
|---|---|---|---|
| 1 | 8888888 | John | Smith |
| … | | | |

- Find any students tuple s,
- that is connected to an enrollment tuple e
    - i.e., whose s.id appears in an enrollment tuple e as e.student,
- and e is connected to a classes tuple c
    - i.e., the e.class of e appears as c.id of the tuple c,
- whose c.number is MAS201

```
FROM students AS s
```

---

- Find any students tuple s,
- that is connected to an enrollment tuple e
    - i.e., whose s.id appears in an enrollment tuple e as e.student,
- and e is connected to a classes tuple c
    - i.e., the e.class of e appears as c.id of the tuple c,
- whose c.number is MAS201

*Take One: Declarative*

```
FROM students AS s,
     enrollment AS e
WHERE s.id = e.student
```

*Take Two: Algebraic*

```
FROM students AS s
     JOIN enrollment AS e
     ON s.id = e.student
```

- Find any students tuple s,
- that is connected to an enrollment tuple e
    - i.e., whose s.id appears in an enrollment tuple e as e.student,
- and e is connected to a classes tuple c
    - i.e., the e.class of e appears as c.id of the tuple c,
- whose c.number is MAS201

*Take One: Declarative*

```
FROM students AS s,
        enrollment AS e,
        classes AS c
WHERE s.id = e.student
        AND c.id = e.class
```

*Take Two: Algebraic*

```
FROM ( students AS s
        JOIN enrollment AS e
        ON s.id = e.student )
        JOIN classes AS c
        ON c.id = e.class
```

- Find any students tuple s,
- that is connected to an enrollment tuple e
    - i.e., whose s.id appears in an enrollment tuple e as e.student,
- and e is connected to a classes tuple c
    - i.e., the e.class of e appears as c.id of the tuple c,
- whose c.number is MAS201

*Take One: Declarative*

```
FROM students AS s,
     enrollment AS e,
     classes AS c
WHERE s.id = e.student
     AND c.id = e.class
     AND c.number = 'MAS201'
```

*Take Two: Algebraic*

```
FROM ( students AS s
     JOIN enrollment AS e
     ON s.id = e.student )
     JOIN classes AS c
     ON c.id = e.class
WHERE c.number = 'MAS201'
```

- Find any students tuple s,
- that is connected to an enrollment tuple e
    - i.e., whose s.id appears in an enrollment tuple e as e.student,
- and e is connected to a classes tuple c
    - i.e., the e.class of e appears as c.id of the tuple c,
- whose c.number is MAS201

```
FROM students AS s
```

---

- Find any students tuple s,
- that is connected to an enrollment tuple e
    - i.e., whose s.id appears in an enrollment tuple e as e.student,
- and e is connected to a classes tuple c
    - i.e., the e.class of e appears as c.id of the tuple c,
- whose c.number is MAS201

*Take One: Declarative*

```
FROM students AS s,
     enrollment AS e
WHERE s.id = e.student
```

*Take Two: Algebraic*

```
FROM students AS s,
     JOIN enrollment AS e
     ON s.id = e.student
```

# Breaking a query into pieces with WITH

Find the 5 classes whose students have the busiest courseload, i.e., the 5 classes whose students have the highest average of quarter credits

Defines a table "courseload" that lives for the duration of this query only

```
WITH courseload AS
( SELECT e.student, SUM(credits) AS total_credits
  FROM enrollment e
  GROUP BY e.student )
SELECT e.class, AVG(c.total_credits)
FROM enrollment e, courseload c
WHERE e.student = c.student
GROUP BY e.class
ORDER BY AVG(c.total_credits) DESC
LIMIT 5
```

# Avoid repeating aggregates

```
WITH courseload AS
( SELECT e.student, SUM(credits) AS total_credits
  FROM enrollment e
  GROUP BY e.student )
SELECT e.class, AVG(c.total_credits)
FROM enrollment e, courseload c
WHERE e.student = c.student
GROUP BY e.class
ORDER BY AVG(c.total_credits) DESC
LIMIT 5
```

Equivalent

```
WITH courseload AS
( SELECT e.student, SUM(credits) AS total_credits
  FROM enrollment e
  GROUP BY e.student )
SELECT e.class, AVG(c.total_credits) AS credits_avg
FROM enrollment e, courseload c
WHERE e.student = c.student
GROUP BY e.class
ORDER BY credits_avg DESC
LIMIT 5
```

# Breaking a query into pieces with nesting in the FROM clause

Find the 5 classes whose students have the busiest courseload, i.e., the 5 classes whose students have the highest average of quarter credits

Also defines a table, identical to the "courseload" except it has no name

```
SELECT e.class, AVG(c.total_credits) AS credits_avg
FROM enrollment e,
 ( SELECT e.student, SUM(credits) AS total_credits
   FROM enrollment e
   GROUP BY e.student ) c
WHERE e.student = c.student
GROUP BY e.class
ORDER BY credits_avg DESC
LIMIT 5
```

# and nesting in the SELECT clause

Find the 5 classes whose students have the busiest courseload, i.e., the 5 classes whose students have the highest average of quarter credits

```
SELECT e.class, AVG(
                    ( SELECT SUM(es.credits)
                      FROM enrollment es
                      WHERE es.student = e.student )
                    ) AS credits_avg
FROM enrollment e
GROUP BY e.class
ORDER BY credits_avg DESC
LIMIT 5
```

# Discussed in class and discussion section

How to solve in easy steps the following complex query:

Create a table that shows all time slots (date, start time, end time) when students of MAS201 attend a lecture of another class; Show also how many students attend a class at each time slot.

# SQL as a Data Manipulation Language: Insertions

- Inserting tuples

```
INSERT INTO R(A1,…,Ak)
VALUES (v1,…,vk);
```

- Some values may be left NULL
- Use results of queries for insertion

```
INSERT INTO R
SELECT …
FROM …
WHERE …
```

- Insert in Students 'John Doe' with A# 99999999

INSERT INTO students

(pid, first_name, last_name)

VALUES

('9999999', 'John', 'Doe')

- Enroll all MAS201 students into CSE132A

INSERT INTO enrollment (class, student)

SELECT c132a.id, student

FROM classes AS c135, enrollment, classes AS c132a

WHERE c135.number='MAS201' AND enrollment.class=c135.id AND c132a.number='CSE132A'

# SQL as a Data Manipulation Language: Updates and Deletions

- Deletion basic form: delete every tuple that satisfies **<cond>**:

```
DELETE FROM R
WHERE <cond>
```

- Update basic form: update every tuple that satisfies **<cond>** in the way specified by the **SET** clause:

```
UPDATE R
SET A1=<exp1>,…,Ak=<expk>
WHERE<cond>
```

$$A_1=\langle exp_1\rangle, \ldots, A_k=\langle exp_k\rangle$$

- Delete “John” “Smith”
- DELETE FROM students WHERE first_name='John' AND last_name='Smith'

---

- Update the registered credits of all MAS201 students to 5

UPDATE enrollment

SET credits=5

WHERE class=1

---

UPDATE enrollment

SET credits=5

WHERE class IN

(SELECT id FROM classes
WHERE number='MAS201')