

*W4111 – Introduction to Databases  
Section 002, Fall 2021*

*Lecture 5: ER, Relational, SQL (IV)*



# *Contents*

# Contents

- Questions, answers, discussion.
- Codd's 12 Rules revisited.
- SQL/RDB:
  - Sub-query
  - Metadata, DDL, Catalog
  - Views
  - Integrity Constraints
- Advanced ER Concepts
- A worked example involving:
  - ER modeling.
  - SQL DDL and data model creation.
  - Top-down and bottom-up modeling.

# *Questions, Answers, Discussion*

# *Codd's 12 Rules Revisited*

# Codd's 12 Rules

## Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

## Rule 2: Guaranteed Access Rule

Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.

## Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following – data is missing, data is not known, or data is not applicable.

## Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as data dictionary, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

## Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation.

## Rule 6: View Updating Rule

**All the views of a database, which can theoretically be updated, must also be updatable by the system.**

# Codd's 12 Rules

## Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

## Rule 8: Physical Data Independence

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

## Rule 9: Logical Data Independence

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rule to apply.

## Rule 10: Integrity Independence

**A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.**

## Rule 11: Distribution Independence

The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

## Rule 12: Non-Subversion Rule

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

# *Sub-query*

# Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.
- The nesting can be done in the following SQL query

```
select A1, A2, ..., An
  from r1, r2, ..., rm
 where P
```

as follows:

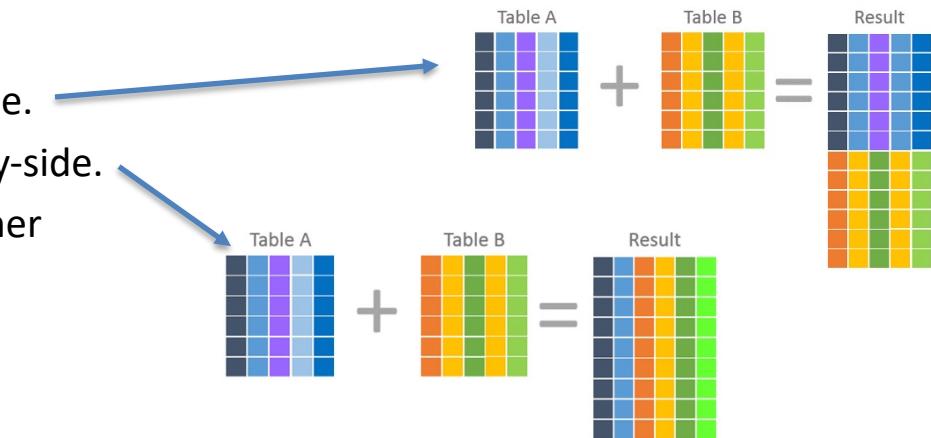
Note:

- This is a little cryptic.
- I think I know what they mean.
- There are some operations we will see later in the material, e.g IN, EXISTS, ... ...

- **From clause:**  $r_i$  can be replaced by any valid subquery
- **Where clause:**  $P$  can be replaced with an expression of the form:  
 $B <\text{operation}> (\text{subquery})$   
*B* is an attribute and *<operation>* to be defined later.
- **Select clause:**  $A_i$  can be replaced be a subquery that generates a single value.

# Nested Subquery

- The slides that come with the book have surprisingly little material on nested subqueries.
- The concept is:
  - Extremely important.
  - Students often find subqueries more confusing than joins.
  - The relationship/difference of subqueries to joins is often, initial unclear.
- We have seen:
  - Union sort of puts a table on top of a table.
  - Join puts tables sort of puts tables side-by-side.
  - Subquery enables one query to call another during execution like a subfunction.



# Consider Some Tables

Takes

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	None

Student

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

# Consider a Subquery Tables

select \*, (select name from student where student.id=takes.id) as name from takes;

Takes

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	None

Switch to Notebook

Student

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

# Set Membership

- Find courses offered in Fall 2017 and in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
course_id in (select course_id
from section
where semester = 'Spring' and year= 2018);
```

- Find courses offered in Fall 2017 but not in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
course_id not in (select course_id
from section
where semester = 'Spring' and year= 2018);
```

# Set Membership (Cont.)

- Name all instructors whose name is neither “Mozart” nor Einstein”

```
select distinct name  
from instructor  
where name not in ('Mozart', 'Einstein')
```

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

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

# Set Comparison – “some” Clause

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

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

- Same query using **> some** clause

```
select name  
from instructor  
where salary > some (select salary  
                      from instructor  
                     where dept name = 'Biology');
```

# Definition of “some” Clause

- $F <\text{comp}> \text{some } r \Leftrightarrow \exists t \in r \text{ such that } (F <\text{comp}> t)$

Where  $\text{comp}$  can be:  $<$ ,  $\leq$ ,  $>$ ,  $=$ ,  $\neq$

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

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

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

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

$(= \text{some}) \equiv \text{in}$   
However,  $(\neq \text{some}) \not\equiv \text{not in}$

# Set Comparison – “all” Clause

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

```
select name  
from instructor  
where salary > all (select salary  
                     from instructor  
                     where dept name = 'Biology');
```

# Definition of “all” Clause

- $F \text{ <comp> } \mathbf{all} \ r \Leftrightarrow \forall t \in r \ (F \text{ <comp> } t)$

( $5 < \mathbf{all}$   ) = false

( $5 < \mathbf{all}$   ) = true

( $5 = \mathbf{all}$   ) = false

( $5 \neq \mathbf{all}$   ) = true (since  $5 \neq 4$  and  $5 \neq 6$ )

$(\neq \mathbf{all}) \equiv \mathbf{not \ in}$   
However,  $(= \mathbf{all}) \not\equiv \mathbf{in}$

# Test for Empty Relations

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

# Use of “exists” Clause

- Yet another way of specifying the query “Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester”

```
select course_id  
from section as S  
where semester = 'Fall' and year = 2017 and  
exists (select *  
        from section as T  
        where semester = 'Spring' and year= 2018  
          and S.course_id = T.course_id);
```

- Correlation name** – variable S in the outer query
- Correlated subquery** – the inner query

# Use of “not exists” Clause

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

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

- First nested query lists all courses offered in Biology
- Second nested query lists all courses a particular student took

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

# Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to “true” if a given subquery contains no duplicates .
- Find all courses that were offered at most once in 2017

```
select T.course_id  
  from course as T  
 where unique ( select R.course_id  
                  from section as R  
                 where T.course_id= R.course_id  
                   and R.year = 2017);
```

# Subqueries in the Form Clause

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

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

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

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

# With Clause

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

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

# Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total(dept_name, value) as
  (select dept_name, sum(salary)
   from instructor
   group by dept_name),
dept_total_avg(value) as
  (select avg(value)
   from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

# Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

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

- Runtime error if subquery returns more than one result tuple

# Practice

- Switch to Jupyter notebook

# *Views*



# Views

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

```
select ID, name, dept_name  
from instructor
```

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



# View Definition

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

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

where <query expression> is any legal SQL expression. The view name is represented by *v*.

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



# View Definition and Use

- A view of instructors without their salary

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

- Find all instructors in the Biology department

```
select name  
    from faculty  
    where dept_name = 'Biology'
```

- Create a view of department salary totals

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



# Views Defined Using Other Views

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



# Views Defined Using Other Views

- **create view *physics\_fall\_2017* as**  
**select course.course\_id, sec\_id, building, room\_number**  
**from course, section**  
**where course.course\_id = section.course\_id**  
**and course.dept\_name = 'Physics'**  
**and section.semester = 'Fall'**  
**and section.year = '2017';**
- **create view *physics\_fall\_2017\_watson* as**  
**select course\_id, room\_number**  
**from *physics\_fall\_2017***  
**where building= 'Watson';**



# View Expansion

- Expand the view :

```
create view physics_fall_2017_watson as
    select course_id, room_number
        from physics_fall_2017
    where building= 'Watson'
```

- To:

```
create view physics_fall_2017_watson as
    select course_id, room_number
        from (select course.course_id, building, room_number
              from course, section
             where course.course_id = section.course_id
               and course.dept_name = 'Physics'
               and section.semester = 'Fall'
               and section.year = '2017')
    where building= 'Watson';
```



## View Expansion (Cont.)

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

**repeat**  
    Find any view relation  $v_i$  in  $e_1$   
    Replace the view relation  $v_i$  by the expression defining  $v_i$   
**until** no more view relations are present in  $e_1$
- As long as the view definitions are not recursive, this loop will terminate



# Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called **Materialized view**:
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **Maintain** the view, by updating the view whenever the underlying relations are updated.



# Update of a View

- Add a new tuple to *faculty* view which we defined earlier

```
insert into faculty
```

```
values ('30765', 'Green', 'Music');
```

- This insertion must be represented by the insertion into the *instructor* relation

- Must have a value for salary.

- Two approaches

- Reject the insert
  - Insert the tuple

```
('30765', 'Green', 'Music', null)
```

into the *instructor* relation



# Some Updates Cannot be Translated Uniquely

- **create view** *instructor\_info* **as**  
**select** *ID, name, building*  
**from** *instructor, department*  
**where** *instructor.dept\_name= department.dept\_name;*
- **insert into** *instructor\_info*  
**values** ('69987', 'White', 'Taylor');
- Issues
  - Which department, if multiple departments in Taylor?
  - What if no department is in Taylor?



## And Some Not at All

- ```
create view history_instructors as
    select *
      from instructor
     where dept_name= 'History';
```
- What happens if we insert  
('25566', 'Brown', 'Biology', 100000)  
into *history\_instructors*?



# View Updates in SQL

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group by** or **having** clause.



# Practice

- Switch to Jupyter notebook

# *Integrity Constraints*



# Integrity Constraints

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



# Constraints on a Single Relation

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



# Not Null Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**  
*name varchar(20) not null*  
*budget numeric(12,2) not null*



# Unique Constraints

- **unique** ( $A_1, A_2, \dots, A_m$ )
  - The unique specification states that the attributes  $A_1, A_2, \dots, A_m$  form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).



# The check clause

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of fall, winter, spring or summer

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



# Referential Integrity

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



## Referential Integrity (Cont.)

- Foreign keys can be specified as part of the SQL **create table** statement  
**foreign key (*dept\_name*) references *department***
- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.  
**foreign key (*dept\_name*) references *department* (*dept\_name*)**



# Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

```
create table course (
    ...
    dept_name varchar(20),
    foreign key (dept_name) references department
        on delete cascade
        on update cascade,
    ...)
```

- Instead of cascade we can use :
  - **set null**,
  - **set default**



# Integrity Constraint Violation During Transactions

- Consider:

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

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



# Complex Check Conditions

- The predicate in the check clause can be an arbitrary predicate that can include a subquery.

```
check (time_slot_id in (select time_slot_id from time_slot))
```

The check condition states that the `time_slot_id` in each tuple in the `section` relation is actually the identifier of a time slot in the `time_slot` relation.

- The condition has to be checked not only when a tuple is inserted or modified in `section`, but also when the relation `time_slot` changes



# Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- The following constraints, can be expressed using assertions:
- For each tuple in the *student* relation, the value of the attribute *tot\_cred* must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot
- An assertion in SQL takes the form:  
**create assertion <assertion-name> check (<predicate>);**



# Built-in Data Types in SQL

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



# Large-Object Types

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



# User-Defined Types

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

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

- Example:

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



# Domains

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

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

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- Example:

```
create domain degree_level varchar(10)  
constraint degree_level_test  
check (value in ('Bachelors', 'Masters', 'Doctorate'));
```

# Let's Practice

- Jupyter notebook
- DataGrip

# *Indexes*



# Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **create index** command

```
create index <name> on <relation-name> (attribute);
```



# Index Creation Example

- **create table student**  
*(ID varchar (5),  
name varchar (20) not null,  
dept\_name varchar (20),  
tot\_cred numeric (3,0) default 0,  
primary key (ID))*
- **create index studentID\_index on student(ID)**
- The query:

```
select *  
from student  
where ID = '12345'
```

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



# Let's Practice

- Jupyter notebook
- DataGrip

# *Some Advanced ER Concepts*



# Weak Entity Sets

- Consider a *section* entity, which is uniquely identified by a *course\_id*, *semester*, *year*, and *sec\_id*.
- Clearly, section entities are related to course entities. Suppose we create a relationship set *sec\_course* between entity sets *section* and *course*.
- Note that the information in *sec\_course* is redundant, since *section* already has an attribute *course\_id*, which identifies the course with which the section is related.
- One option to deal with this redundancy is to get rid of the relationship *sec\_course*; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.



## Weak Entity Sets (Cont.)

- An alternative way to deal with this redundancy is to not store the attribute *course\_id* in the *section* entity and to only store the remaining attributes *section\_id*, *year*, and *semester*.
  - However, the entity set *section* then does not have enough attributes to identify a particular *section* entity uniquely
- To deal with this problem, we treat the relationship *sec\_course* as a special relationship that provides extra information, in this case, the *course\_id*, required to identify *section* entities uniquely.
- A **weak entity set** is one whose existence is dependent on another entity, called its **identifying entity**
- Instead of associating a primary key with a weak entity, we use the identifying entity, along with extra attributes called **discriminator** to uniquely identify a weak entity.



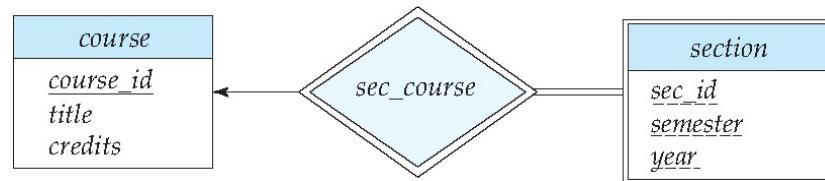
## Weak Entity Sets (Cont.)

- An entity set that is not a weak entity set is termed a **strong entity set**.
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set.
- The identifying entity set is said to **own** the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.
- Note that the relational schema we eventually create from the entity set *section* does have the attribute *course\_id*, for reasons that will become clear later, even though we have dropped the attribute *course\_id* from the entity set *section*.

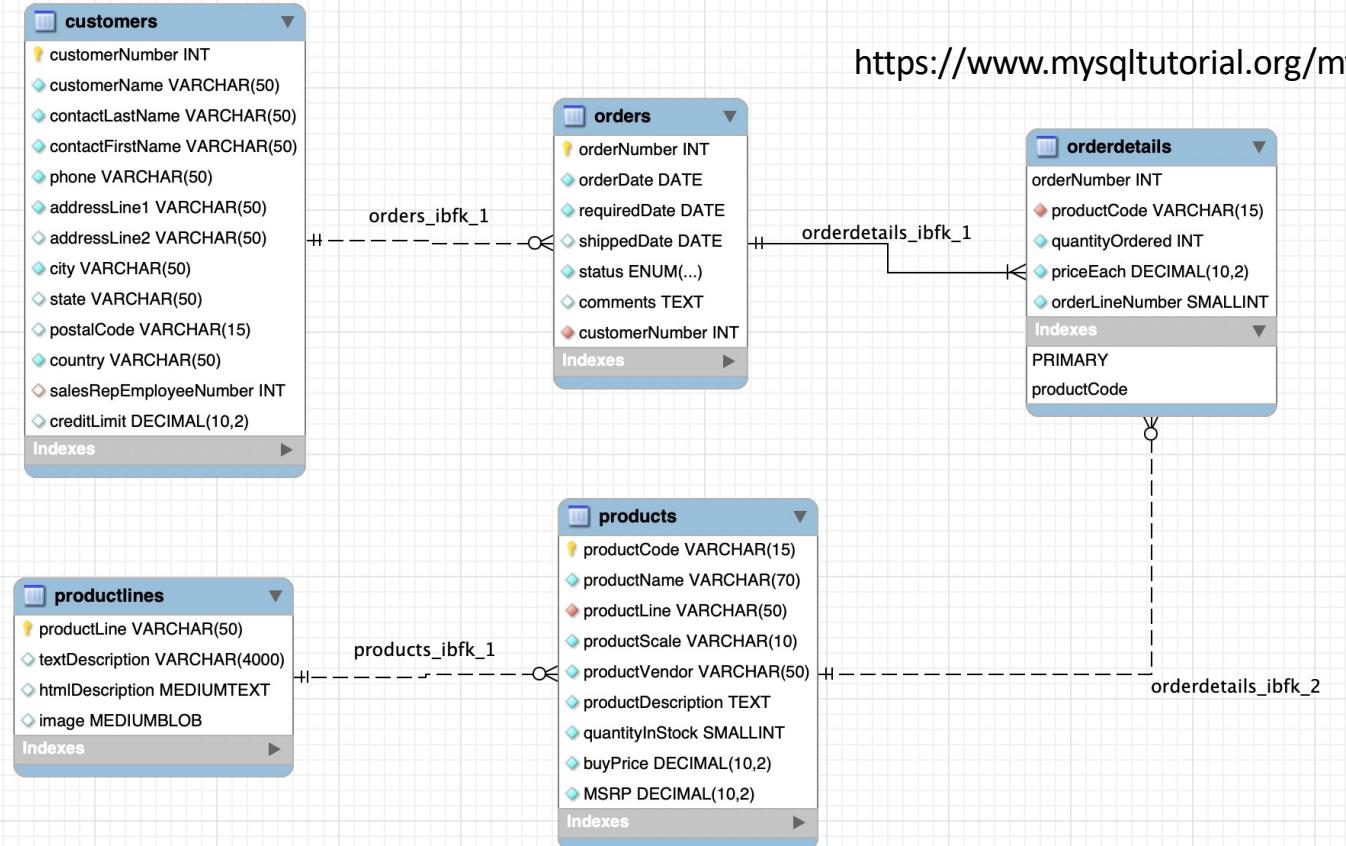


# Expressing Weak Entity Sets

- In E-R diagrams, a weak entity set is depicted via a double rectangle.
- We underline the discriminator of a weak entity set with a dashed line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.
- Primary key for *section* – (*course\_id*, *sec\_id*, *semester*, *year*)



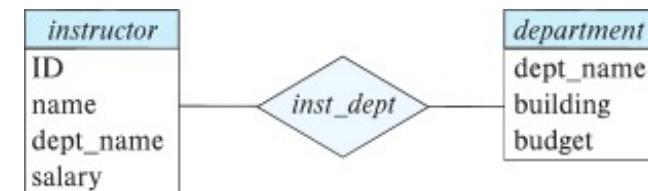
# An Example – Classic Models



<https://www.mysqltutorial.org/mysql-sample-database.aspx/>

# Redundant Attributes

- Suppose we have entity sets:
  - *instructor*, with attributes: *ID, name, dept\_name, salary*
  - *department*, with attributes: *dept\_name, building, budget*
- We model the fact that each instructor has an associated department using a relationship set *inst\_dept*
- The attribute *dept\_name* in *instructor* replicates information present in the relationship and is therefore redundant
  - and needs to be removed.
- BUT: when converting back to tables, in some cases the attribute gets reintroduced, as we will see later.



# Design Alternatives

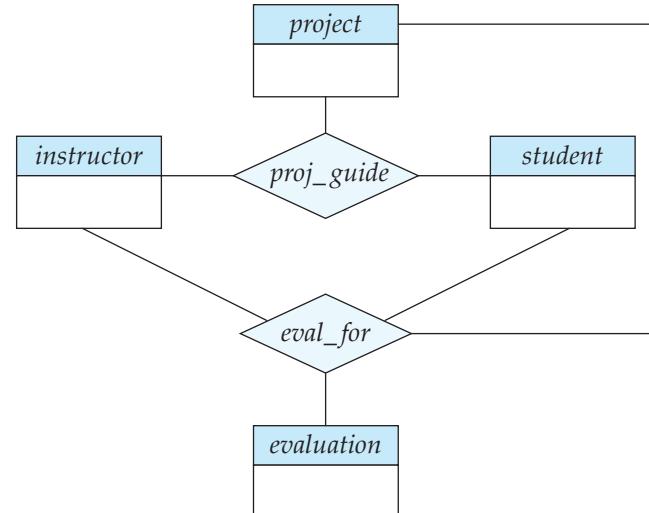
- In designing a database schema, we must ensure that we avoid two major pitfalls:
  - Redundancy: a bad design may result in repeat information.
    - **Redundant representation of information may lead to data inconsistency among the various copies of information**
  - Incompleteness: a bad design may make certain aspects of the enterprise difficult or impossible to model.
- Avoiding bad designs is not enough. There may be a large number of good designs from which we must choose.

**Emphasis  
Added**



# Aggregation

- Consider the ternary relationship *proj\_guide*, which we saw earlier
- Suppose we want to record evaluations of a student by a guide on a project





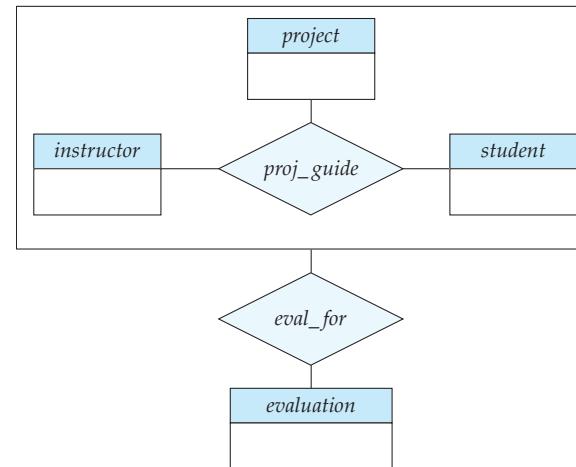
# Aggregation (Cont.)

- Relationship sets *eval\_for* and *proj\_guide* represent overlapping information
  - Every *eval\_for* relationship corresponds to a *proj\_guide* relationship
  - However, some *proj\_guide* relationships may not correspond to any *eval\_for* relationships
    - So we can't discard the *proj\_guide* relationship
- Eliminate this redundancy via *aggregation*
  - Treat relationship as an abstract entity
  - Allows relationships between relationships
  - Abstraction of relationship into new entity



# Aggregation (Cont.)

- Eliminate this redundancy via *aggregation* without introducing redundancy, the following diagram represents:
  - A student is guided by a particular instructor on a particular project
  - A student, instructor, project combination may have an associated evaluation





# Reduction to Relational Schemas

- To represent aggregation, create a schema containing
  - Primary key of the aggregated relationship,
  - The primary key of the associated entity set
  - Any descriptive attributes
- In our example:
  - The schema *eval\_for* is:  
$$\text{eval\_for} (s\_ID, project\_id, i\_ID, evaluation\_id)$$
  - The schema *proj\_guide* is redundant.



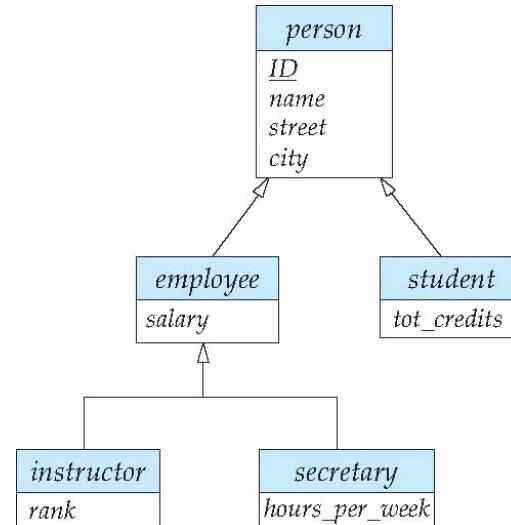
# Specialization

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



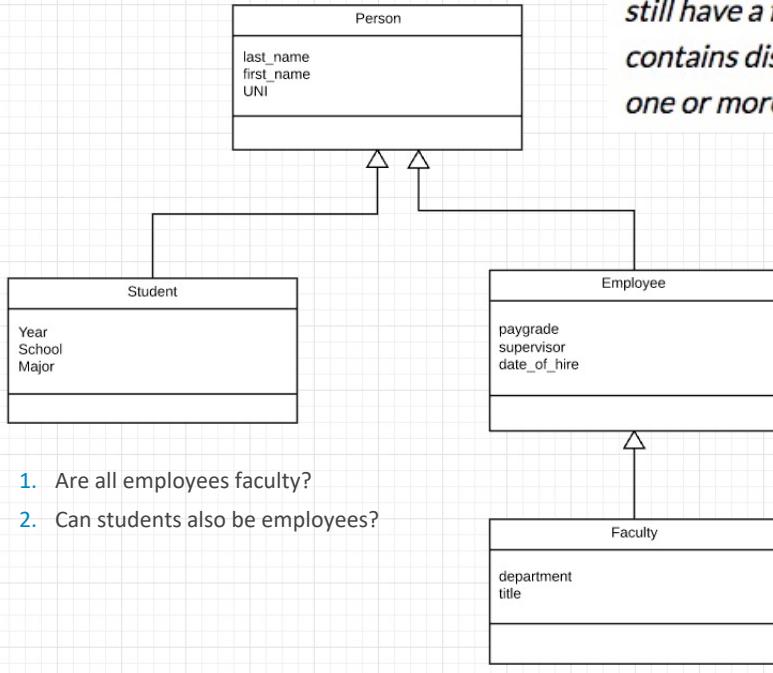
# Specialization Example

- **Overlapping** – *employee* and *student*
- **Disjoint** – *instructor* and *secretary*
- Total and partial



# Inheritance, IsA, Specialization

*In the process of designing our entity relationship diagram for a database, we may find that attributes of two or more entities overlap, meaning that these entities seem very similar but still have a few differences. In this case, we may create a subtype of the parent entity that contains distinct attributes. A parent entity becomes a supertype that has a relationship with one or more subtypes.*



1. Are all employees faculty?
2. Can students also be employees?

The subclass association line is labeled with specialization constraints. Constraints are described along two dimensions:

#### 1 incomplete/complete

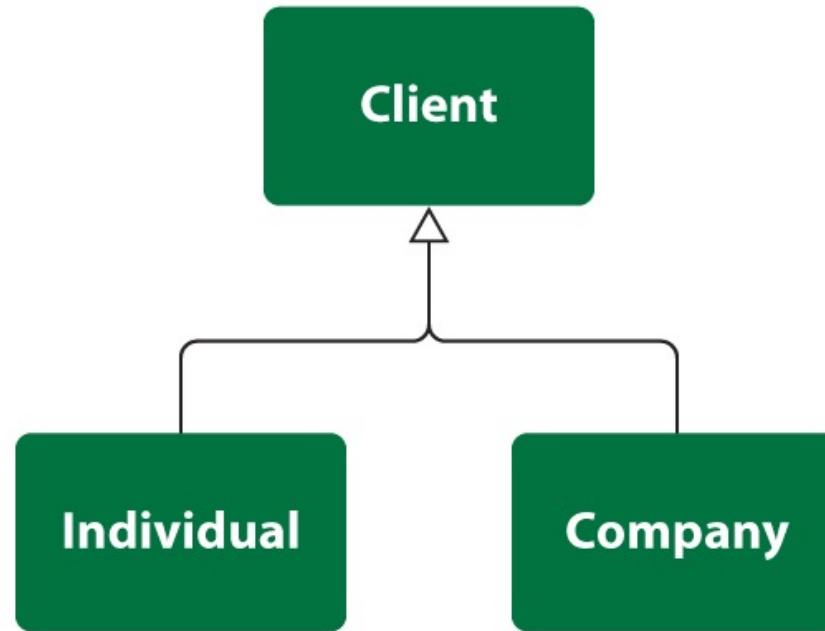
- In an **incomplete** specialization only some instances of the parent class are specialized (have unique attributes). Other instances of the parent class have only the common attributes.
- In a **complete** specialization, every instance of the parent class has one or more unique attributes that are not common to the parent class.

#### 2 disjoint/overlapping

- In a **disjoint** specialization, an object could be a member of only one specialized subclass.
- In an **overlapping** specialization, an object could be a member of more than one specialized subclass.

# Simpler Example

In class Client we distinguish two subtypes: Individual and Company. This specialization is disjoint (client can be an individual or a company) and complete (these are all possible subtypes for supertype).

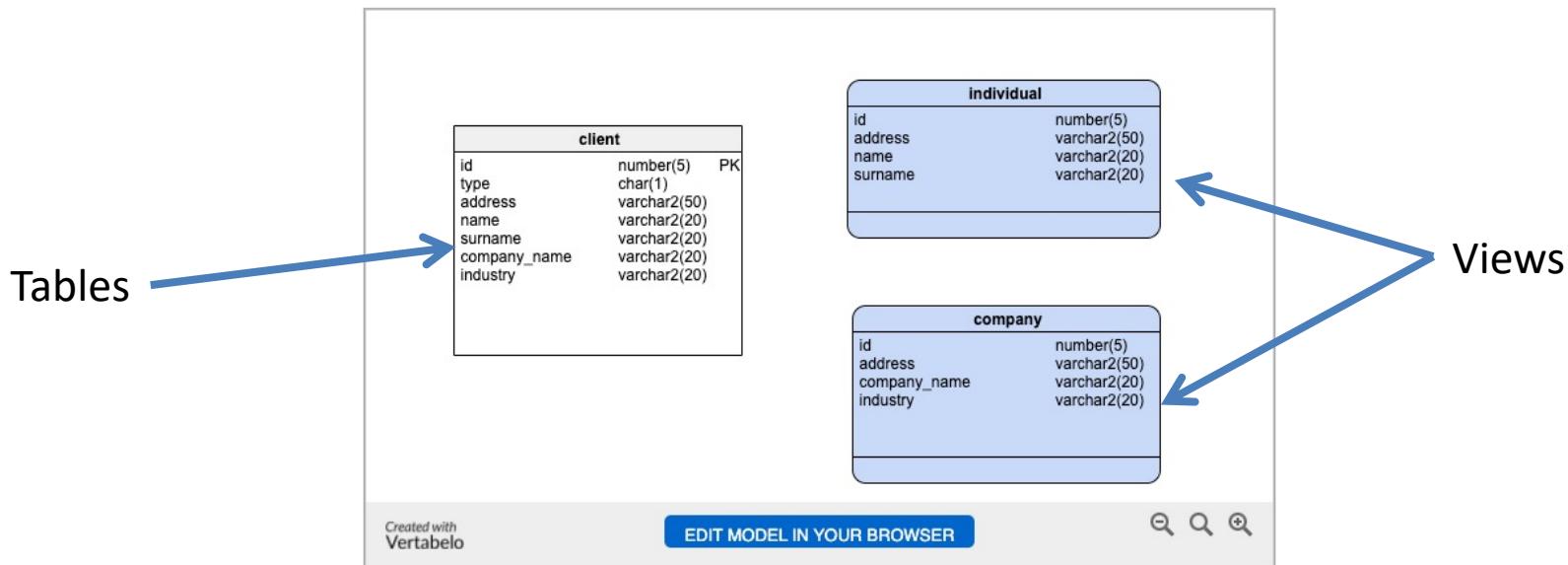


# One Table Implementation

## One table implementation

In a one table implementation, table `client` has attributes of both types.

The diagram below shows the table `client` and two views: `individual` and `company`:



# Two Table Implementation



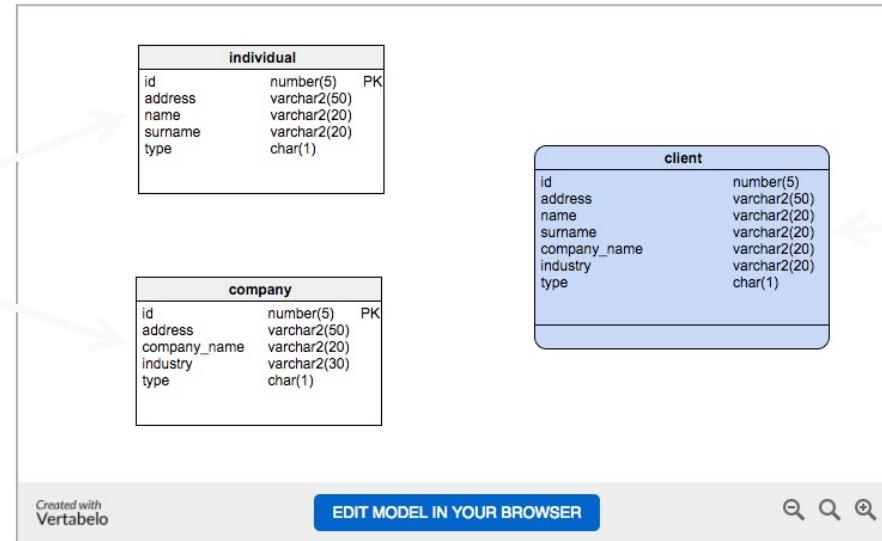
## Two-table implementation

In a two-table implementation, we create a table for each of the subtypes. Each table gets a column for all attributes of the supertype and also a column for each attribute belonging to the subtype. Access to information in this situation is limited, that's why it is important to create a view that is the union of the tables. We can add an additional attribute called 'type' that describes the subtype.

The diagram below presents two tables, `individual` and `company`, and a view (the blue one) called `client`.

Tables

View



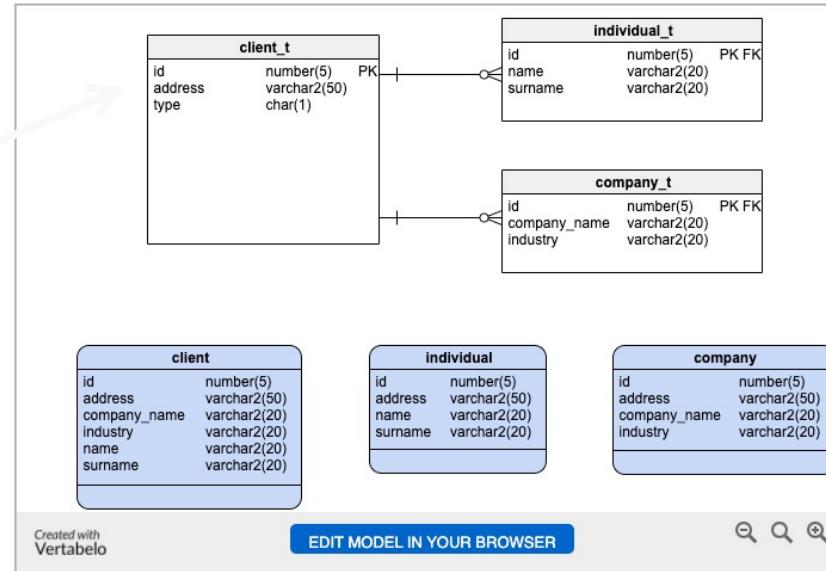
# Two Table Implementation



## Three-table implementation

In a third solution we create a single table `client_t` for the parent table, containing common attributes for all subtypes, and tables for each subtype (`individual_t` and `company_t`) where the primary key in `client_t` (base table) determines foreign keys in dependent tables. There are three views: `client`, `individual` and `company`.

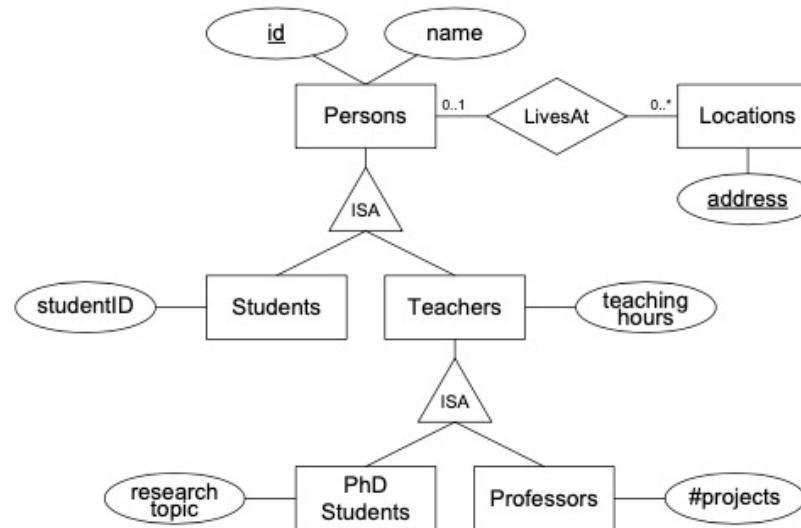
Tables



Views



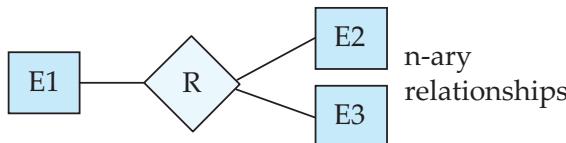
# ISA Relationship





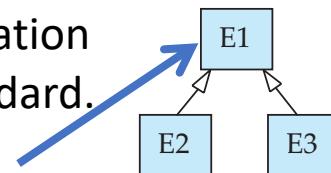
# ER vs. UML Class Diagrams

## ER Diagram Notation

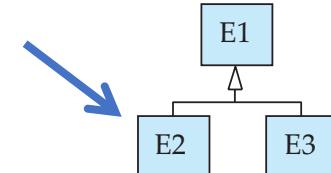


n-ary relationships

I use this approach  
in Crow's Foot Notation  
but that is not standard.

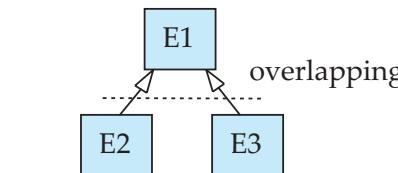
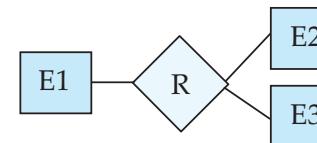


overlapping  
generalization

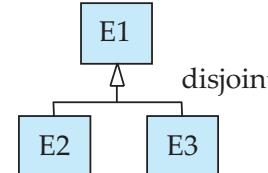


disjoint  
generalization

## Equivalent in UML



overlapping



disjoint

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

# Let's Practice – Specialization

- Jupyter notebook
- DataGrip

## *Worked Example:*

1. *ER Model from scratch.*
2. *Schema definition, DDL, ... and some new concepts.*
3. *Where's the data?*

# Worked Example – Top Down

- Scenario
  - Course with complex course ID.
  - Section
  - Faculty
  - Department
  - Instructor – Department Assoc. entity with properties (role, date).
  - Instructor – Section
  - Student - Section
- Do ER (live) diagram in Lucidchart.
- Do schema creation
  - In DataGrip
  - Show copying statements int the notebook.



# Bottom-Up: Some Sources of Information

- Course Codes:  
<https://www.cc-seas.columbia.edu/sites/dsa/files/handbooks/Columbia%20Key%20to%20Course%20Listing.pdf>
- Course/Section Properties:  
<http://www.columbia.edu/cu/bulletin/uwb/>
- Common Search Criteria → Indexes, Query Parameters, Data Links  
<https://doc.search.columbia.edu/classes/Ferguson?instr=&name=&days=&semes=&hour=&moi=>
- Department Information:
  - Script: <https://www.columbia.edu/content/academics/departments>
  - Codes: [https://academic-admin.cuit.columbia.edu/dept\\_code](https://academic-admin.cuit.columbia.edu/dept_code)
- Specific Information:
  - Course and Instructor: <https://opendataservice.columbia.edu/api/9/json>
  - Vergil Data: <https://vergil.registrar.columbia.edu/feeds/cw.js>
  - Vergil Search: <https://vergil.registrar.columbia.edu/doc-adv-queries.php?key=ferguson&moreresults=2>

# Meet in the Middle

- Show cu\_info project with data and processing.
- Show notebook for loading the data.
- Show getting part of the way through parsing HTML in project