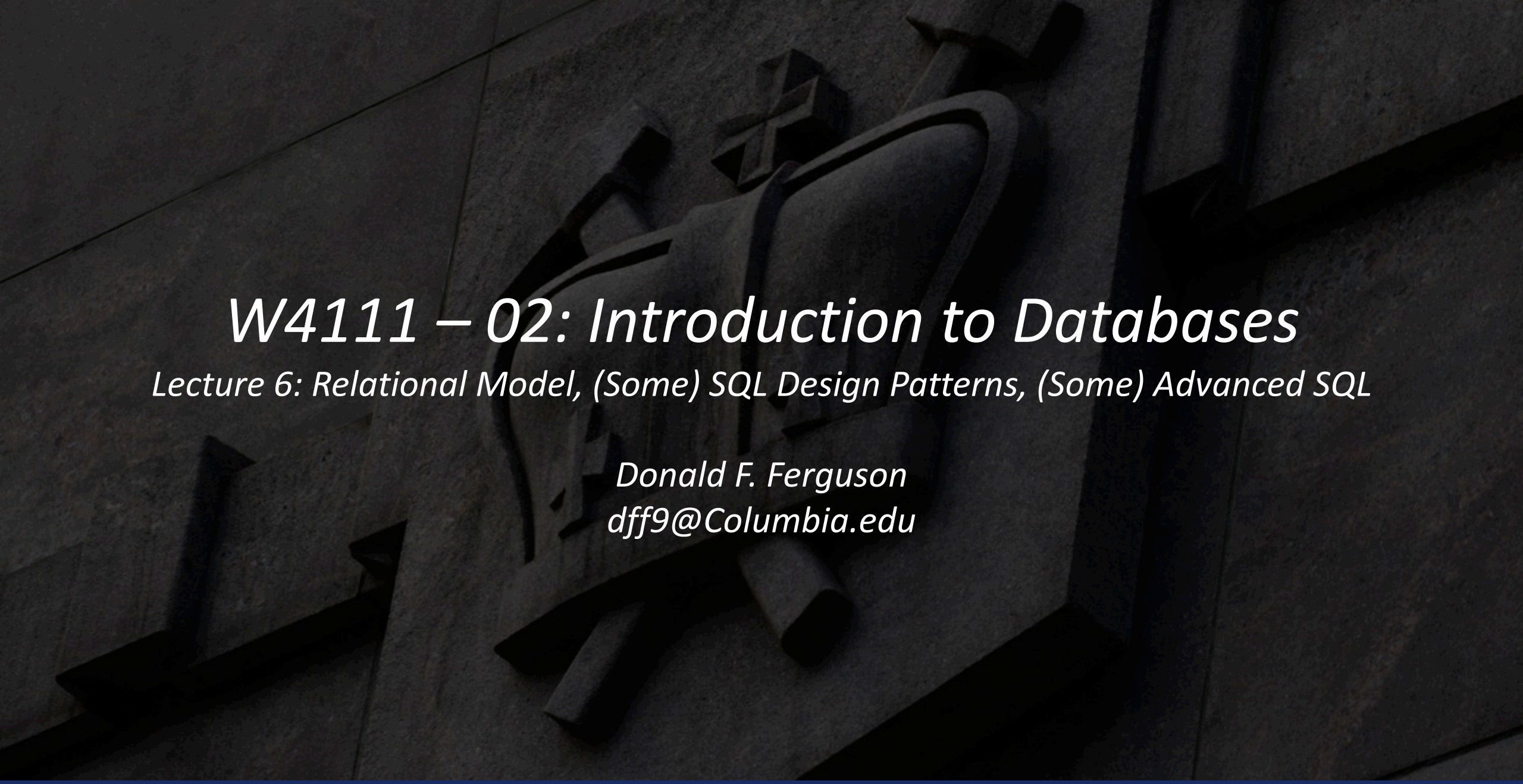


W4111 – 02: Introduction to Databases

Lecture 6: Relational Model, (Some) SQL Design Patterns, (Some) Advanced SQL

Donald F. Ferguson
dff9@Columbia.edu

Contents

Contents

- Relational Algebra
  - Almost never arises in practice.
  - Useful for understanding SQL and other query models.
  - Essential for understanding query processing and optimization (Module II)
- Entity-Relationship Modeling Patterns and Realization
  - Specialization (again) and putting some pieces together.
  - Ternary relationships.
  - Complex Attributes.
  - Aggregation.
- Some Advanced SQL
  - JOIN types.
  - Recursive queries (not kidding).
  - Security.

Relational Model

Motivation

Why Relational Model (Theory)

- RDBs and SQL implement and extend a formal model.

- The formal model almost never arises in practice;
  SQL *is the relational model* for all practical purposes.

- Why study the relational model?
  - Critical to how RDBs implement query processing and optimization.
  - Many elements of SQL (and query languages in general) become clearer when you understand the underlying algebra.
  - Advanced courses and TAing for DB courses require knowledge.

- It is important for some more advanced classes and
  may arise on job/internship interviews.

Relational Algebra

- A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
- Six basic operators
  - select: σ
  - project: Π
  - union: $\cup$
  - set difference: –
  - Cartesian product: x
  - rename: ρ

- *Wait. What? Where did JOIN go? (Hold that thought). Hint –*
  - The notation for select with a "where clause" is $\sigma_p(R)$, where p is a predicate/condition.
  - The symbol for JOIN is $\bowtie$. If θ is the ON condition, then the definition of JOIN is $R \bowtie_\theta L \equiv \sigma_\theta(R \times S)$

Why Relational Model (Theory) – Example

- Online relational algebra calculator: https://dbis-uibk.github.io/relax/calc.htm#
- Data: https://gist.github.com/woolfg/7d1871f79a8bcb4788de

Person

Person

Person.name	Person.age	Person.gender
Amy	16	female
Ben	21	male
Cal	33	male
Dan	13	male
Eli	45	male
Fay	21	female
Gus	24	male
Hil	30	female
Ian	18	male

Eats

Eats

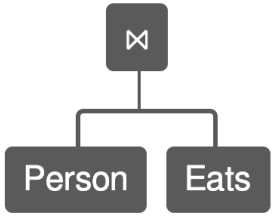
Eats.name	Eats.pizza
Amy	pepperoni
Amy	mushroom
Ben	pepperoni
Ben	cheese
Cal	supreme
Dan	pepperoni
Dan	cheese
Dan	sausage
Dan	supreme
Dan	mushroom
Eli	supreme
Eli	cheese
Fay	mushroom
Gus	mushroom
Gus	supreme
Gus	cheese
Hil	supreme
Hil	cheese
Ian	supreme
Ian	pepperoni

Why Relational Model (Theory) – Example

- Person ⋈ Eats

```
result = []

for p in Person:
    for e in Eats:

        # Only common column is 'name'
        if p['name'] == e['name']:

            # p is a row of column values. e is a row of column values
            # The result is a union of the values of the two rows.
            r = p.union(e)

            # Add to the result
            result.append(r)
```

- The computational complexity is len(Person) * len(Eats)



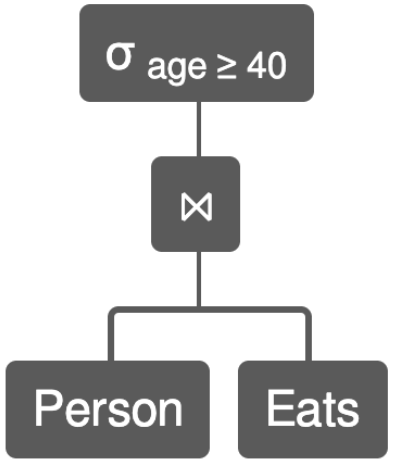
Person ⋈ Eats

Person.name	Person.age	Person.gender	Eats.pizza
Amy	16	female	pepperoni
Amy	16	female	mushroom
Ben	21	male	pepperoni
Ben	21	male	cheese
Cal	33	male	supreme
Dan	13	male	pepperoni
Dan	13	male	cheese
Dan	13	male	sausage
Dan	13	male	supreme
Dan	13	male	mushroom
Eli	45	male	supreme
Eli	45	male	cheese
Fay	21	female	mushroom
Gus	24	male	mushroom
Gus	24	male	supreme
Gus	24	male	cheese
Hil	30	female	supreme
Hil	30	female	cheese
Ian	18	male	supreme
Ian	18	male	pepperoni

Why Relational Model (Theory) – Example

- σ age>= 40 (Person ⋈ Eats)

```
result = []

for p in Person:
    for e in Eats:

        # Only common column is 'name'
        if p['name'] == e['name']:

            # p is a row of column values. e is a row of column values
            # The result is a union of the values of the two rows.
            r = p.union(e)

            # Add to the result
            result.append(r)

final_result = []

for r in result:
    if r['age'] >= 40:
        final_result.append(r)
```



σ age ≥ 40 (Person ⋈ Eats)

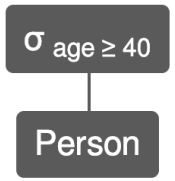
Person.name	Person.age	Person.gender	Eats.pizza
Eli	45	male	supreme
Eli	45	male	cheese

- The computational complexity is len(Person) * len(Eats) = 9 * 20 = 180

Why Relational Model (Theory) – Example

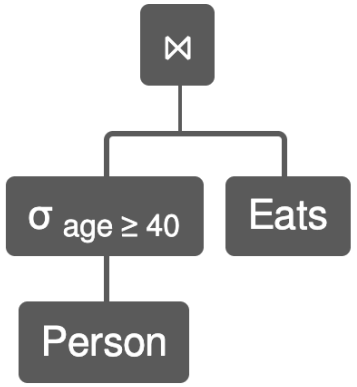
- The rule of the algebra tell us
  - σ age>= 40(Person ⋈ Eats)
  - Is equivalent to
  - σ age>= 40 (Person) ⋈ Eats

```
valid_people = []

for p in Person:
    if p['age'] >= 40:
        valid_people.append(p)

for p in valid_people:
    for e in Eats:

        # Only common column is 'name'
        if p['name'] == e['name']:

            # p is a row of column values. e is a row of column values
            # The result is a union of the values of the two rows.
            r = p.union(e)

            # Add to the result
            result.append(r)
```



σ age ≥ 40 (Person)

Person.name	Person.age	Person.gender
Eli	45	male



σ age ≥ 40 (Person) ⋈ Eats

Person.name	Person.age	Person.gender	Eats.pizza
Eli	45	male	supreme
Eli	45	male	cheese

- The complexity is len(Person) + [len(σ age>= 40 (Person)) * len(Eats)] = 20 + [1 * 20] = 40.

Why Relational Model (Theory) – Example

- Th

An algebra allows us to determine when different statements are equivalent:

- p * (x + y+ z) ≡ p*x + p*y + p*z ≡ p*(x + y) + p*z
- (x + y) + z ≡ x + (y + z)

Relational algebra defines the foundation for determining when relational statements are equivalent:

- $\sigma(A \bowtie B) \equiv \sigma(A) \bowtie \sigma(B)$
- $A \bowtie B \equiv B \bowtie A$
- $(A \bowtie B) \bowtie C \equiv A \bowtie (B \bowtie C)$

The *performance* of two equivalent queries can be astoundingly different based on sizes of tables, distinct values in columns, indexes, etc.

The algebra rules allow the *query optimizer* to convert the query you wrote into an equivalent query that you should have written and that is much more efficient.

- 20

Relational Algebra
(from the textbook)

Relational Query Languages

- Procedural versus non-procedural, or declarative
- “Pure” languages:
  - Relational algebra
  - Tuple relational calculus
  - Domain relational calculus
- The above 3 pure languages are equivalent in computing power
- We will concentrate in this chapter on relational algebra
  - Not turning-machine equivalent
  - Consists of 6 basic operations

Relational Algebra

- A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
- Six basic operators
  - select: σ
  - project: Π
  - union: $\cup$
  - set difference: –
  - Cartesian product: x
  - rename: ρ

Select Operation

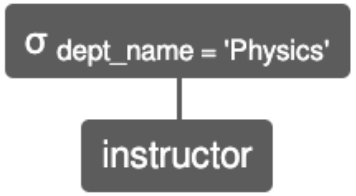
- The **select** operation selects tuples that satisfy a given predicate.
- Notation: $\sigma_p(r)$
- p is called the **selection predicate**
- Example: select those tuples of the *instructor* relation where the instructor is in the "Physics" department.
  - Query

$$\sigma_{dept\_name="Physics"}(instructor)$$

  - Result

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

From: https://dbis-uibk.github.io/relax/calc.htm#



σ dept_name = 'Physics' (instructor)

instructor.ID	instructor.name	instructor.dept_name	instructor.salary
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

Select Operation (Cont.)

- We allow comparisons using

  $=, \neq, >, \geq, <, \leq$

  in the selection predicate.
- We can combine several predicates into a larger predicate by using the connectives:

  $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
- Example: Find the instructors in Physics with a salary greater $90,000, we write:

  $$\sigma_{dept\_name=\text{“Physics”} \wedge salary > 90,000}(instructor)$$
- Then select predicate may include comparisons between two attributes.
  - Example, find all departments whose name is the same as their building name:
  - $\sigma_{dept\_name=building}(department)$

Select Operation (Cont.)

- We allow comparisons using

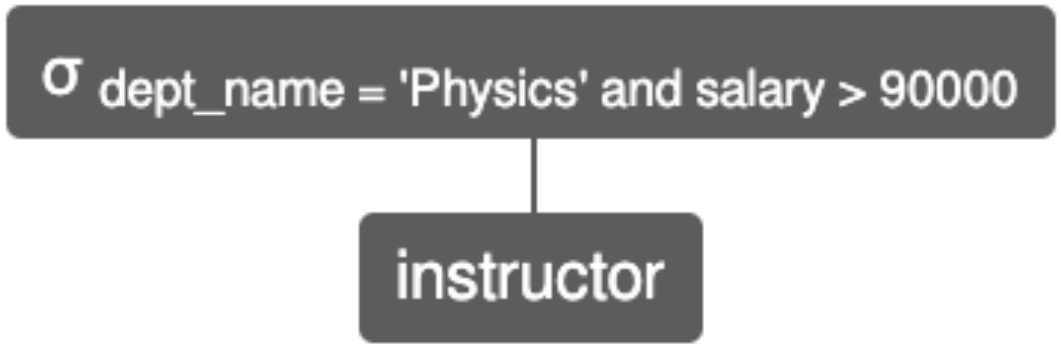
in th

- We
conn

- Exar
write

σ

- Ther
  -
  -



$\sigma_{dept\_name = 'Physics' \text{ and } salary > 90000}(instructor)$

instructor.ID	instructor.name	instructor.dept_name	instructor.salary
22222	Einstein	Physics	95000

Project Operation

- A unary operation that returns its argument relation, with certain attributes left out.
- Notation:

$$\Pi_{A_1, A_2, A_3 \ldots A_k} (r)$$

  where A_1, A_2 are attribute names and r is a relation name.
- The result is defined as the relation of k columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets (<dff>SQL does not behave this way</dff>)

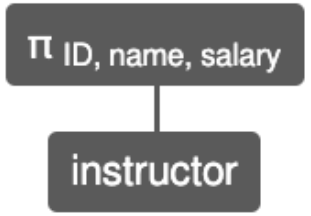
Project Operation (Cont.)

- Example: eliminate the *dept_name* attribute of *instructor*
- Query*:*

$$\Pi_{ID,\ name,\ salary}\ (instructor)$$

- Result:

ID	*name*	*salary*
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000



π ID, name, salary (instructor)

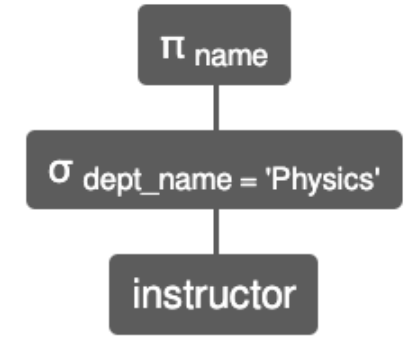
instructor.ID	instructor.name	instructor.salary
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

Composition of Relational Operations

- The result of a relational-algebra operation is relation and therefore of relational-algebra operations can be composed together into a **relational-algebra expression**.
- Consider the query -- Find the names of all instructors in the Physics department.

$$\Pi_{name}(\sigma_{dept\_name = \text{“Physics”}} (instructor))$$

- Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.



$\pi_{name}(\sigma_{dept\_name = \text{'Physics'}}(instructor))$

instructor.name
Einstein
Gold

Cartesian-Product Operation

- The Cartesian-product operation (denoted by X) allows us to combine information from any two relations.
- Example: the Cartesian product of the relations *instructor* and t*eaches* is written as:

  instructor X *teaches*

- We construct a tuple of the result out of each possible pair of tuples: one from the *instructor* relation and one from the *teaches* relation (see next slide)
- Since the instructor *ID* appears in both relations we distinguish between these attribute by attaching to the attribute the name of the relation from which the attribute originally came.
  - *instructor.ID*
  - *teaches.ID*

The *instructor* x *teaches* table

Instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2017
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2018
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2017
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…

Cartesian Product Simpler Example

appetizer

appetizer

appetizer.number	appetizer.dish
A1	Soup
A2	Salad

entree

entree

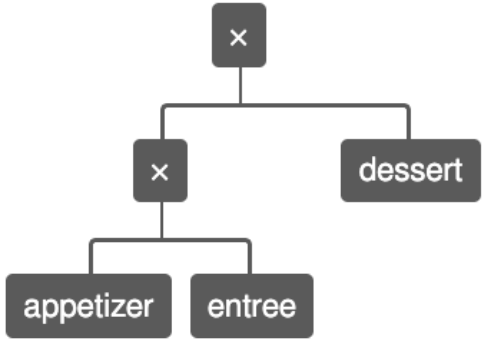
entree.number	entree.dish
E1	Fish
E2	Chicken

dessert

dessert

dessert.number	dessert.dish
D1	Ice Cream
D2	Cake

Sometimes this makes sense, …



appetizer × entree × dessert

appetizer.number	appetizer.dish	entree.number	entree.dish	dessert.number	dessert.dish
A1	Soup	E1	Fish	D1	Ice Cream
A1	Soup	E1	Fish	D2	Cake
A1	Soup	E2	Chicken	D1	Ice Cream
A1	Soup	E2	Chicken	D2	Cake
A2	Salad	E1	Fish	D1	Ice Cream
A2	Salad	E1	Fish	D2	Cake
A2	Salad	E2	Chicken	D1	Ice Cream
A2	Salad	E2	Chicken	D2	Cake

But sometimes you need to think about it.

Instructor X course

Is every possible pairing, including pairing me with

Latin GU4105 section 001
LAT LITERATURE OF THE REPUBLIC

Join Operation

- The Cartesian-Product

  instructor X *teaches*

  associates every tuple of instructor with every tuple of teaches.

  - Most of the resulting rows have information about instructors who did NOT teach a particular course.
- To get only those tuples of "*instructor* X *teaches* " that pertain to instructors and the courses that they taught, we write:

  $\sigma_{instructor.id = teaches.id}$ (*instructor* x *teaches*))

  - We get only those tuples of "*instructor* X *teaches*" that pertain to instructors and the courses that they taught.
- The result of this expression, shown in the next slide

Join Operation (Cont.)

- The table corresponding to:

$$\sigma_{instructor.id = teaches.id}(instructor \times teaches))$$

Instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	76766	BIO-301	1	Summer	2018
83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-319	2	Spring	2018
98345	Kim	Elec. Eng.	80000	98345	EE-181	1	Spring	2017

Join Operation (Cont.)

- The **join** operation allows us to combine a select operation and a Cartesian-Product operation into a single operation.
- Consider relations *r* (*R*) and *s* (*S*)
- Let "theta" be a predicate on attributes in the schema R "union" S. The join operation r $\bowtie_\theta$ s is defined as follows:

$$r \bowtie_\theta s = \sigma_\theta (r \times s)$$

- Thus

$$\sigma_{instructor.id = teaches.id} (instructor \times teaches))$$

- Can equivalently be written as

$$instructor \bowtie_{Instructor.id = teaches.id} teaches.$$

Union Operation

- The union operation allows us to combine two relations
- Notation: $r \cup s$
- For $r \cup s$ to be valid.
  1. *r, s* must have the *same* **arity** (same number of attributes)
  2. The attribute domains must be **compatible** (example: 2nd column of *r* deals with the same type of values as does the 2nd column of *s*)
- Example: to find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both

$$\Pi_{course\_id}(\sigma_{semester=\text{“Fall”} \wedge year=2017}(section)) \cup \Pi_{course\_id}(\sigma_{semester=\text{“Spring”} \wedge year=2018}(section))$$

Union Operation (Cont.)

- Result of:

$$\Pi_{course\_id}(\sigma_{semester=\text{“Fall”} \wedge year=2017}(section)) \cup \Pi_{course\_id}(\sigma_{semester=\text{“Spring”} \wedge year=2018}(section))$$

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

Set-Intersection Operation

- The set-intersection operation allows us to find tuples that are in both the input relations.
- Notation: $r \cap s$
- Assume:
  - r, s have the *same arity*
  - attributes of r and s are compatible
- Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.

$$\Pi_{course\_id}(\sigma_{semester=\text{“Fall”} \wedge year=2017}(section)) \cap \Pi_{course\_id}(\sigma_{semester=\text{“Spring”} \wedge year=2018}(section))$$

  - Result

course_id
CS-101

Set Difference Operation

- The set-difference operation allows us to find tuples that are in one relation but are not in another.
- Notation $r - s$
- Set differences must be taken between **compatible** relations.
  - r and s must have the same arity
  - attribute domains of r and s must be compatible
- Example: to find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

$$\Pi_{course\_id}(\sigma_{semester=\text{“Fall”} \wedge year=2017}(section)) - \Pi_{course\_id}(\sigma_{semester=\text{“Spring”} \wedge year=2018}(section))$$

course_id
CS-347
PHY-101

The Assignment Operation

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation is denoted by $\leftarrow$ and works like assignment in a programming language.
- Example: Find all instructor in the "Physics" and Music department.

$$Physics \leftarrow \sigma_{dept\_name="Physics"}(instructor)$$

$$Music \leftarrow \sigma_{dept\_name="Music"}(instructor)$$

$$Physics \cup Music$$

- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

The Rename Operation

- The results of relational-algebra expressions do not have a name that we can use to refer to them. The rename operator, ρ, is provided for that purpose
- The expression:

  $$\rho_x(E)$$

  returns the result of expression E under the name x
- Another form of the rename operation:

  $$\rho_{x(A1,A2,\ldots An)}(E)$$

Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department with salary greater than 90,000
- Query 1

$$\sigma_{dept\_name=\text{“Physics”} \wedge salary > 90{,}000}(instructor)$$

- Query 2

$$\sigma_{dept\_name=\text{“Physics”}}(\sigma_{salary > 90.000}(instructor))$$

- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

Equivalent Queries

- There is more than one way to write a query in relational algebra.
- Example: Find information about courses taught by instructors in the Physics department
- Query 1

$$\sigma_{dept\_name=\text{“Physics”}}(instructor \bowtie_{instructor.ID = teaches.ID} teaches)$$

- Query 2

$$(\sigma_{dept\_name=\text{“Physics”}}(instructor)) \bowtie_{instructor.ID = teaches.ID} teaches$$

- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

Generalization Projection

Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list.

$$\prod_{F_1, F_2, \ldots, F_n}(E)$$

- E is any relational-algebra expression
- Each of F_1, F_2, …, F_n are are arithmetic expressions involving constants and attributes in the schema of E.
- Given relation *instructor(ID, name, dept_name,* salary) where salary is annual salary, get the same information but with monthly salary

$$\prod_{ID,\ name,\ dept\_name,\ salary/12}(instructor)$$

http://www.cbcb.umd.edu/confcour/Spring2014/CMSC424/Relational_algebra.pdf

Aggregate Functions and Operations

- **Aggregation function** takes a collection of values and returns a single value as a result.

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

- **Aggregate operation** in relational algebra

$$_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1(A_1),F_2(A_2,\ldots,F_n(A_n)}(E)$$

  E is any relational-algebra expression
  - G_1, G_2 …, G_n is a list of attributes on which to group (can be empty)
  - Each F_i is an aggregate function
  - Each A_i is an attribute name

- Note: Some books/articles use γ instead of $\mathcal{G}$ (Calligraphic G)

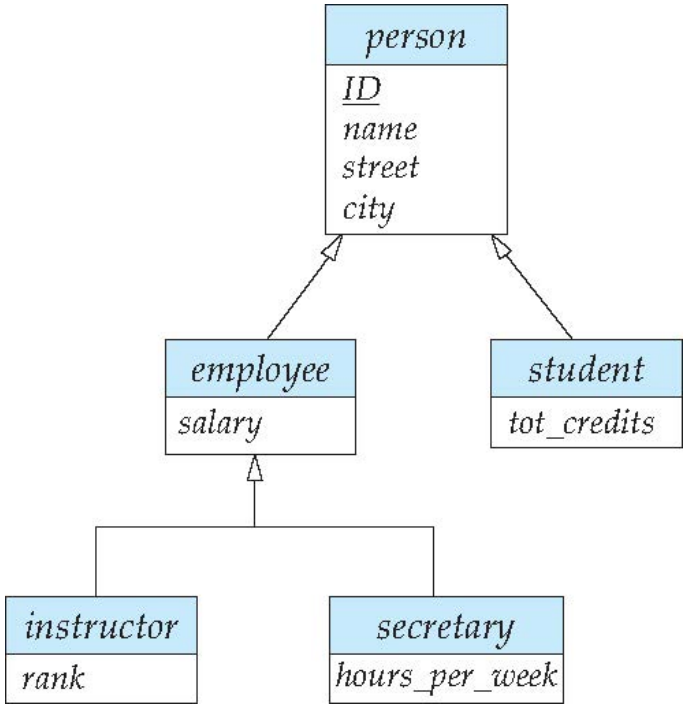
http://www.cbcb.umd.edu/confcour/Spring2014/CMSC424/Relational_algebra.pdf

Some More Advanced ER Modeling

Specialization

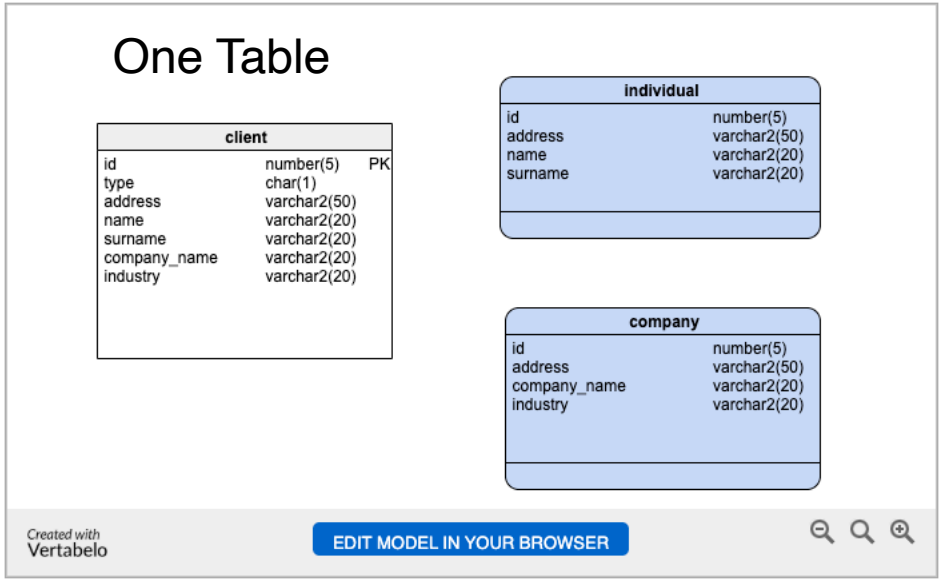
Specialization Example

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

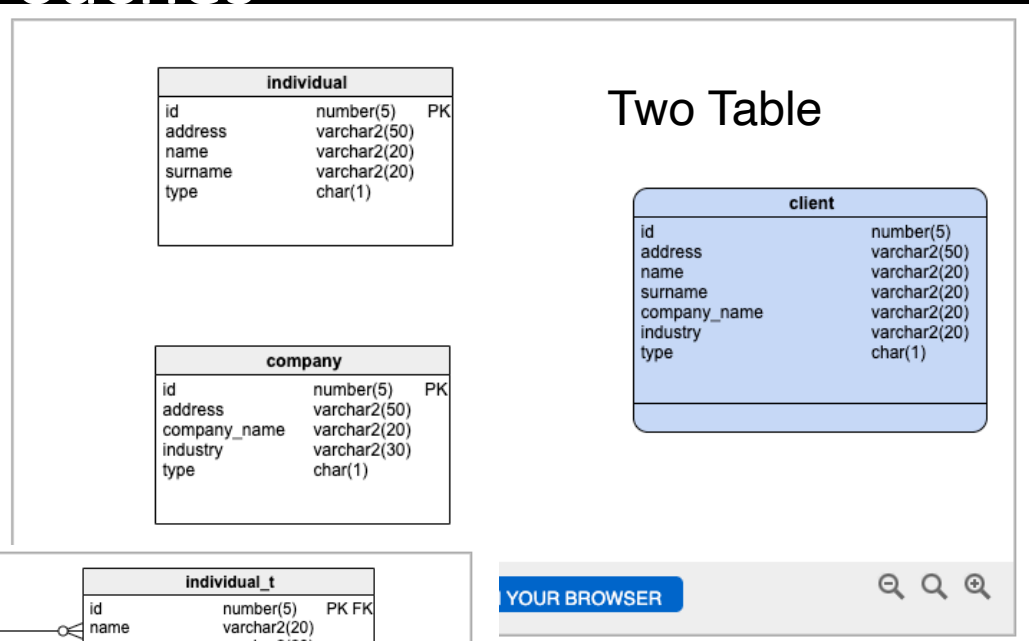


Specialization/Inheritance – Design Approaches

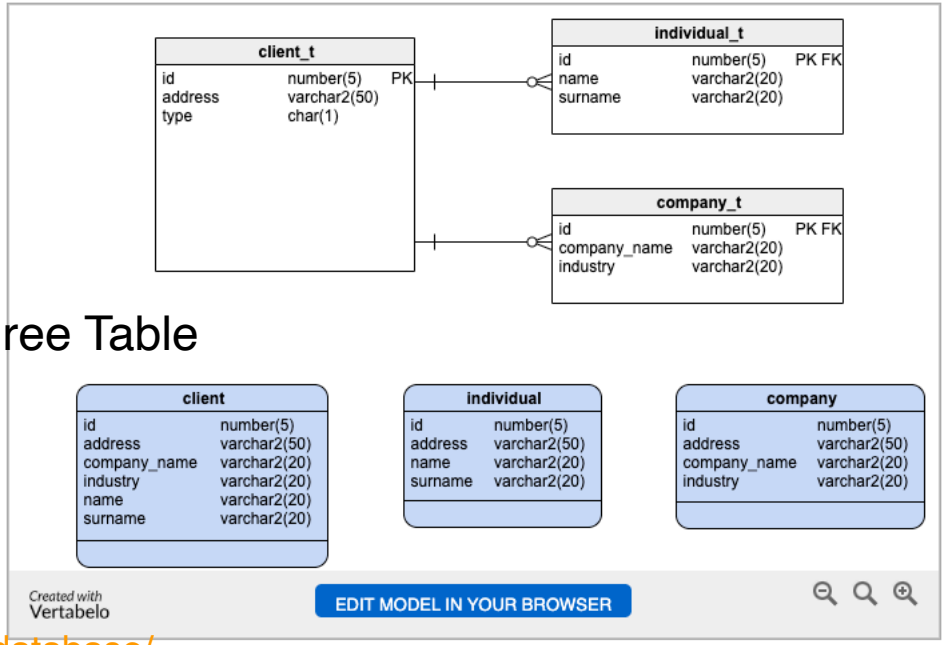
One Table



Two Table

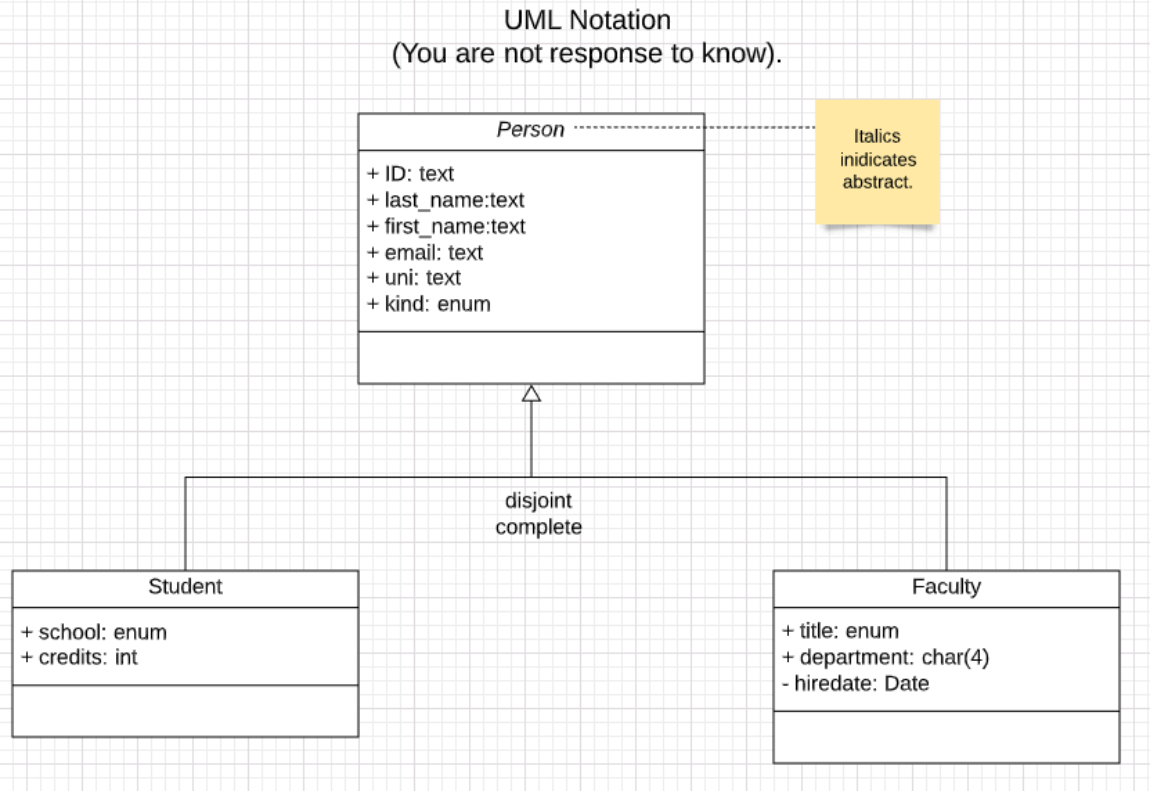


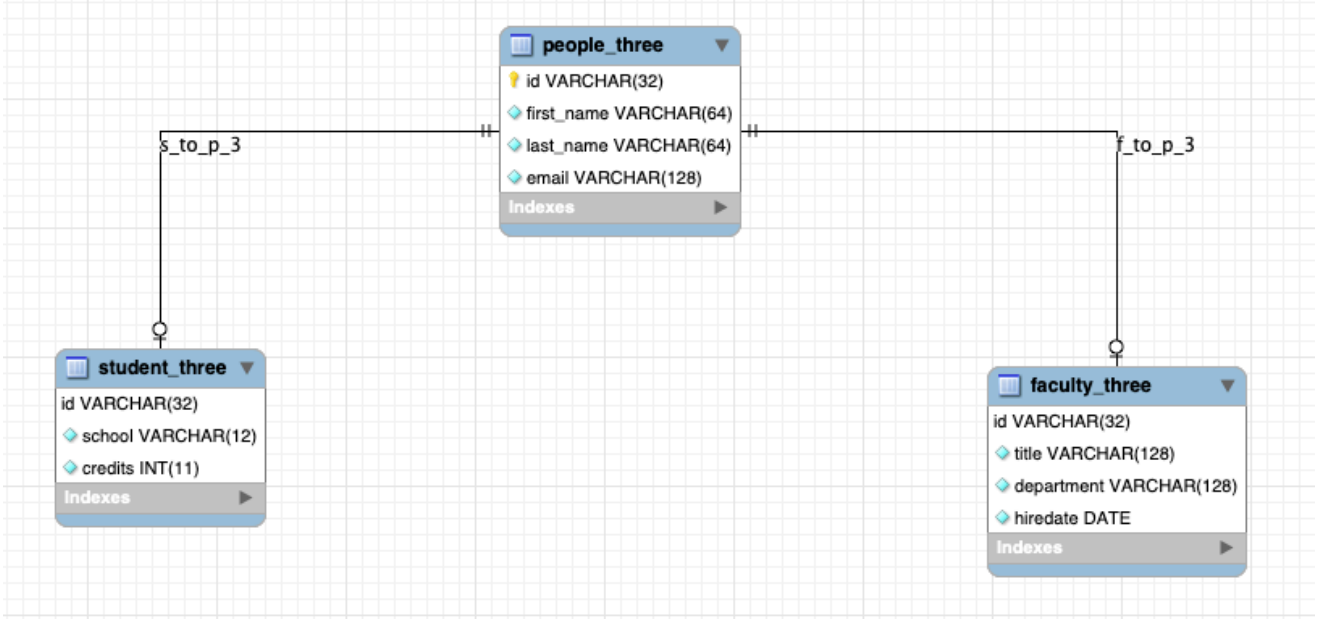
Three Table



https://www.vertabelo.com/blog/inheritance-in-a-relational-database/

Three Table Solution to University People (Reminder)





Switch to Notebook

What we have Logically Done (are Doing)

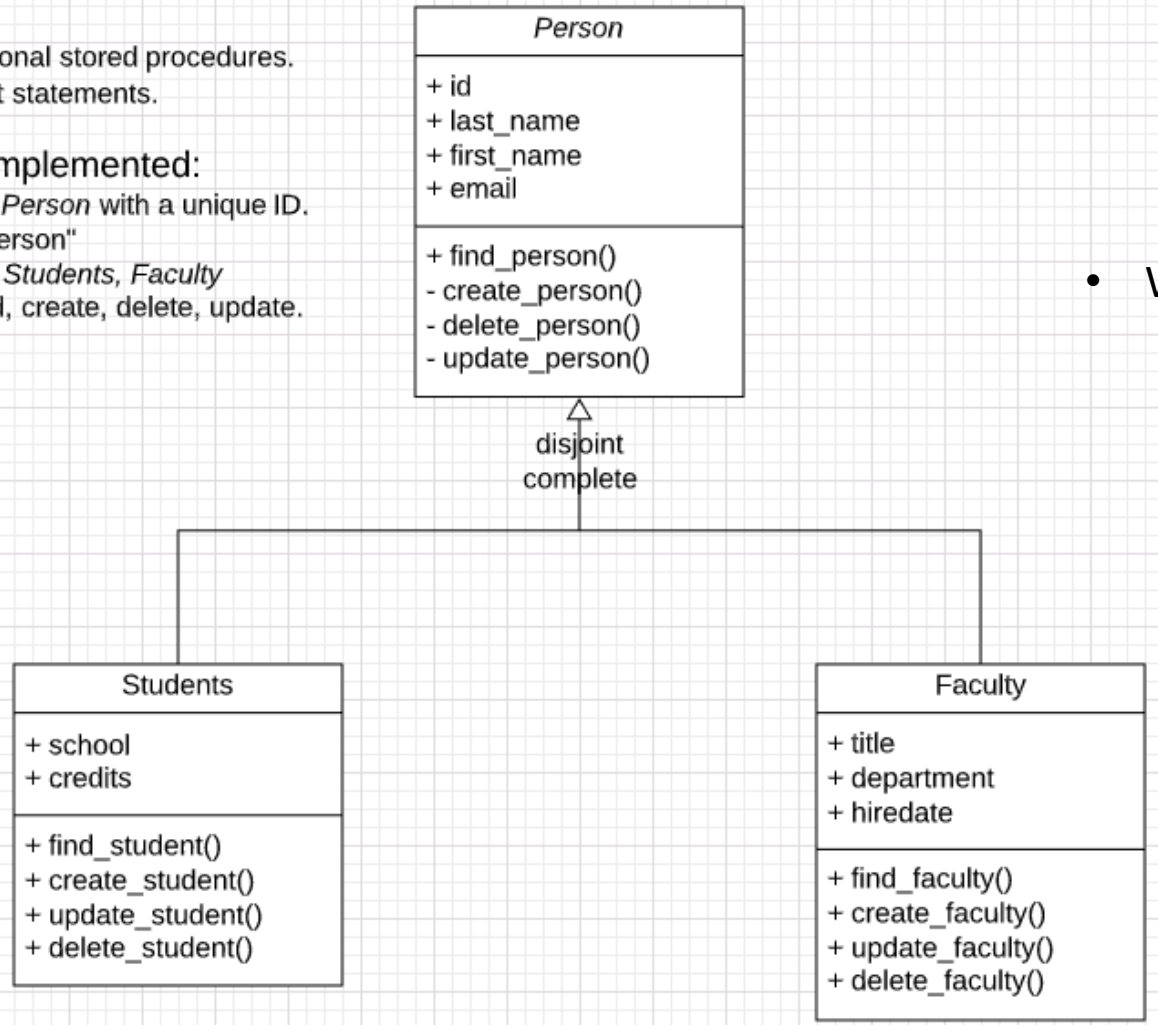
Logically what we are doing.

Assuming that:

- Implement some additional stored procedures.
- Define the correct grant statements.

We have logically implemented:

- An abstract base class *Person* with a unique ID.
- The ability to find all "Person"
- Concrete base classes *Students, Faculty*
- With full support for find, create, delete, update.

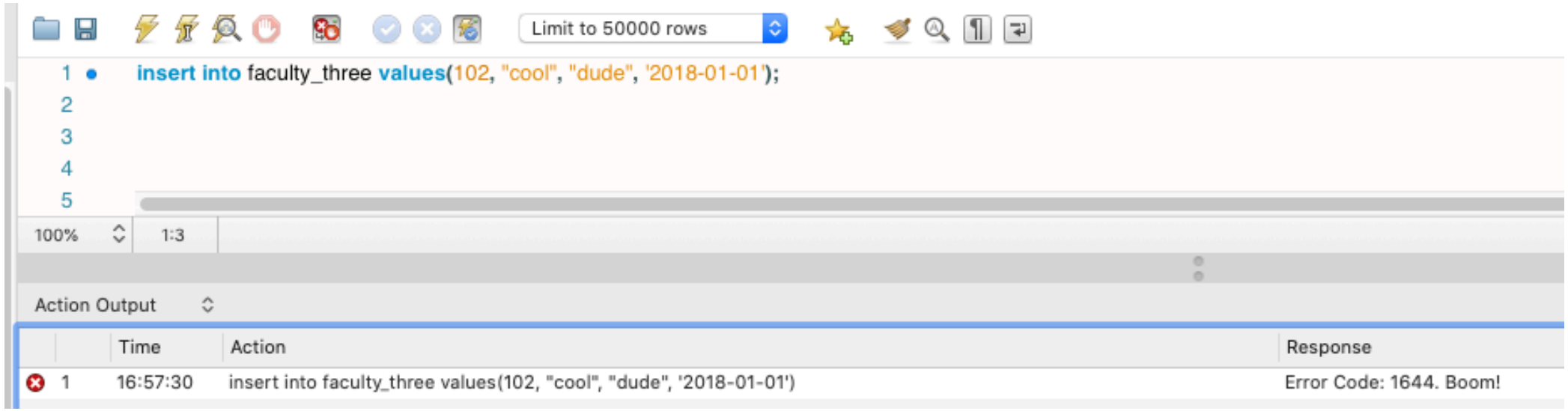


- Some additional constraints:
  - Completeness: There is no way to make a Person that is not a Student or Faculty.
  - Disjoint: It is still possible to have a Student and Faculty with the same ID.
- We can implement disjoint with
  - Check constraint, except most DBMS do not support subqueries in check constraints.
  - Triggers.

What we have Logically Done (are Doing)

```
CREATE DEFINER=`root`@`localhost` TRIGGER `faculty_three_BEFORE_INSERT`
   BEFORE INSERT ON `faculty_three` FOR EACH ROW BEGIN

   DECLARE  student_id VARCHAR(32);

   IF New.id in (SELECT ID FROM students_three) THEN
      SIGNAL SQLSTATE '50100' SET message_text = "Boom!";
   END IF;

END
```

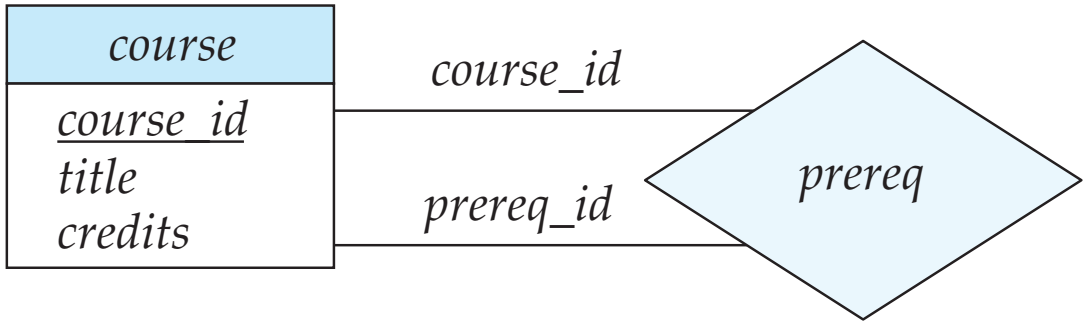
```
CREATE DEFINER=`root`@`localhost` TRIGGER `faculty_three_BEFORE_UPDATE`
   BEFORE UPDATE ON `faculty_three` FOR EACH ROW BEGIN

   DECLARE  student_id VARCHAR(32);

   IF New.id in (SELECT ID FROM students_three) THEN
      SIGNAL SQLSTATE '50100' SET message_text = "Boom!";
   END IF;

END
```



Roles

Roles

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



- DFF Note: The concept of roles applies to all relationships/associations.

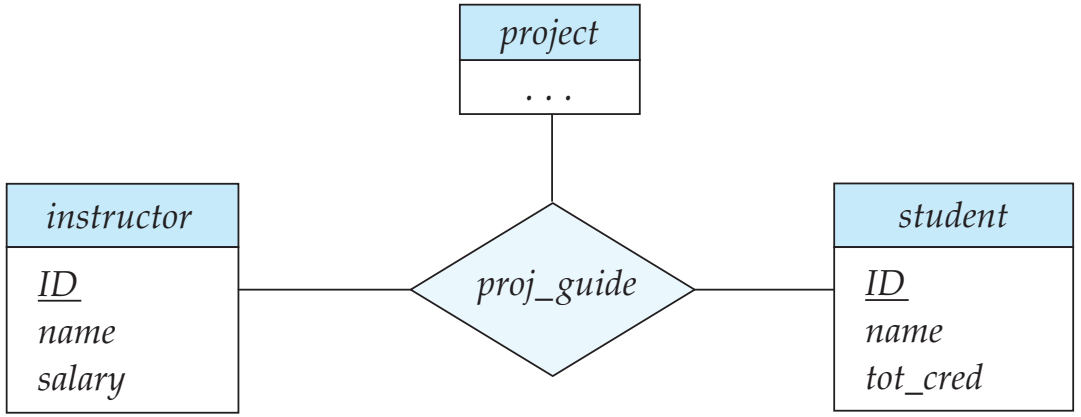
Relationships

Degree of a Relationship Set

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

Non-binary Relationship Sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary.
- E-R Diagram with a Ternary Relationship



- Note: In SQL/RDB, this requires four tables (relations)
  1. Instructor
  2. Student
  3. Project
  4. “Project Participants”

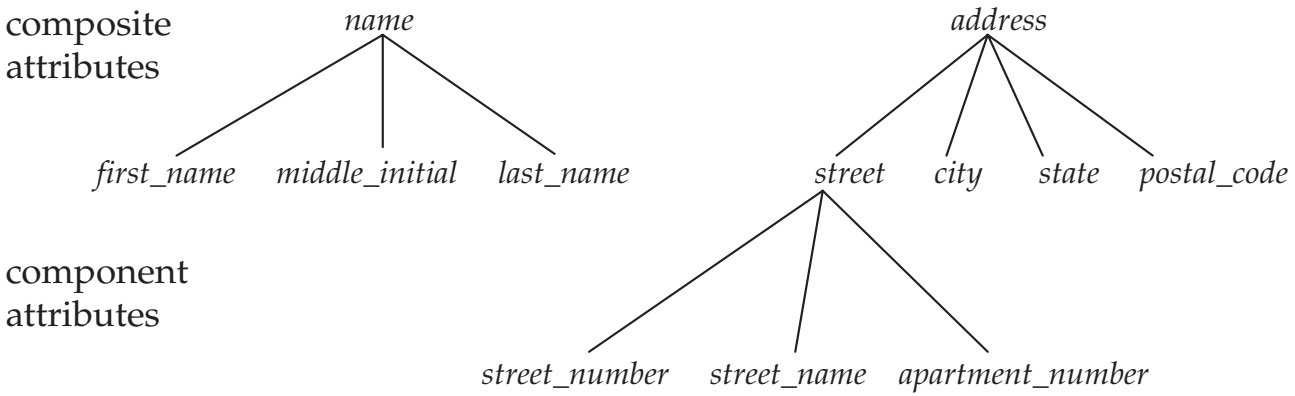
Complex Attributes

Complex Attributes

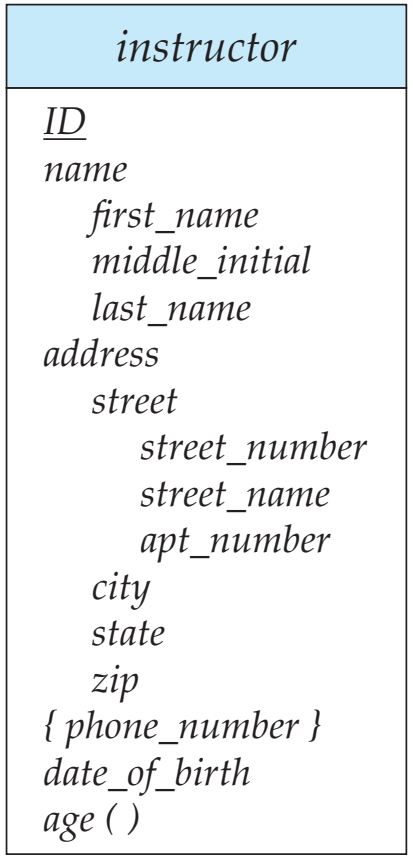
- Attribute types:
  - **Simple** and **composite** attributes.
  - **Single-valued** and **multivalued** attributes
    - Example: multivalued attribute: *phone_numbers*
  - **Derived** attributes
    - Can be computed from other attributes
    - Example: age, given date_of_birth
- **Domain** – the set of permitted values for each attribute

Composite Attributes

- Composite attributes allow us to divided attributes into subparts (other attributes).



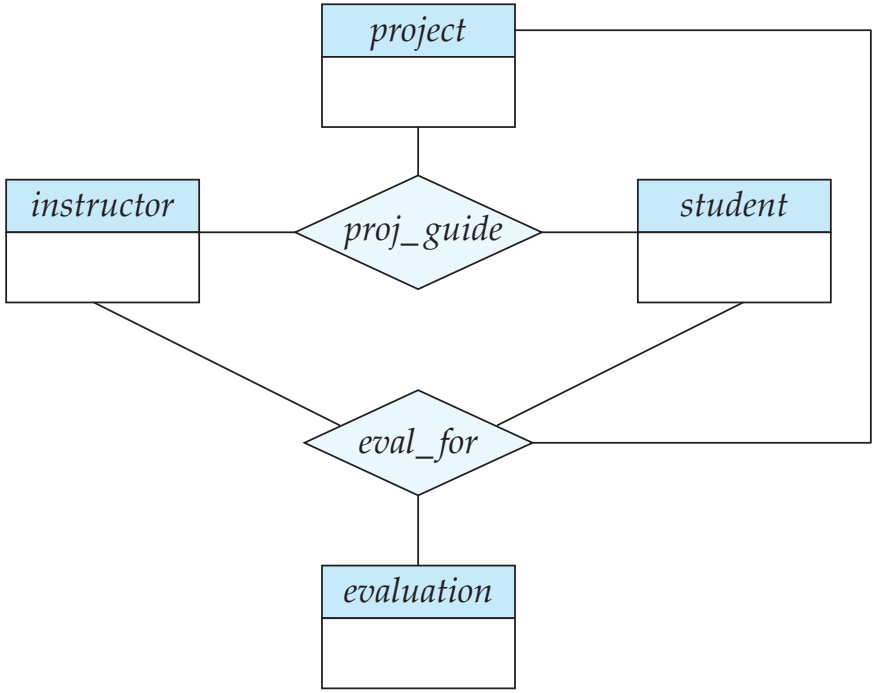
Representing Complex Attributes in ER Diagram



Aggregation

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

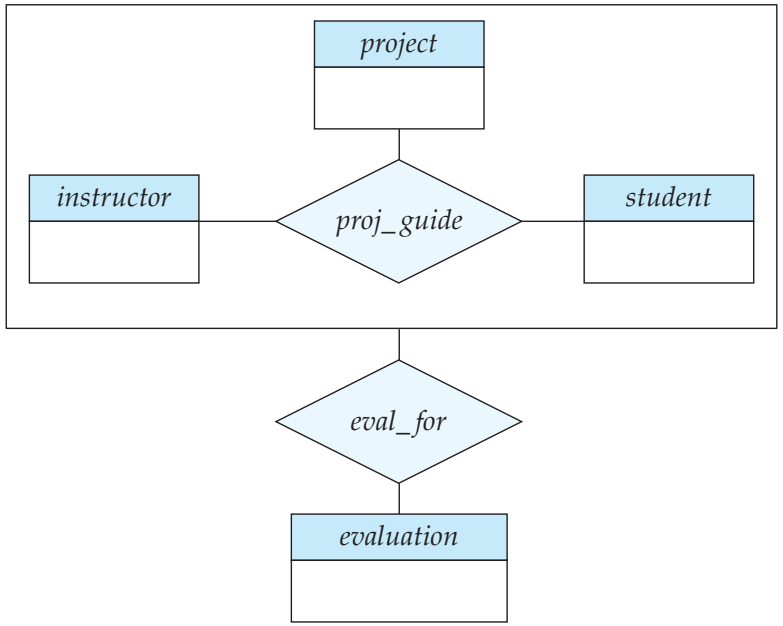


Aggregation (Cont.)

- 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

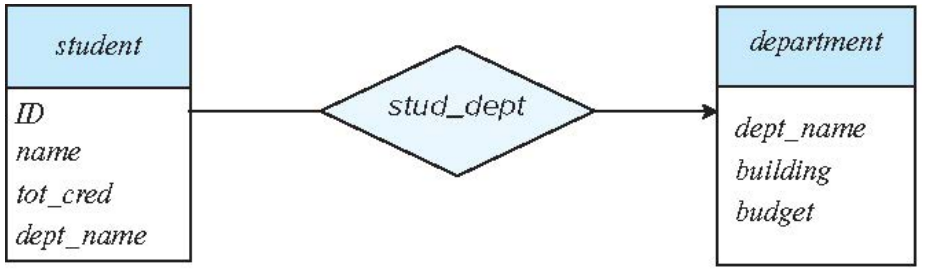
- To represent aggregation, create a schema containing
  - Primary key of the aggregated relationship,
  - The primary key of the associated entity set
  - Any descriptive attributes
- In our example:
  - The schema *eval_for* is:

    eval_for (*s_ID, project_id, i_ID, evaluation_id*)
  - The schema *proj_guide* is redundant.

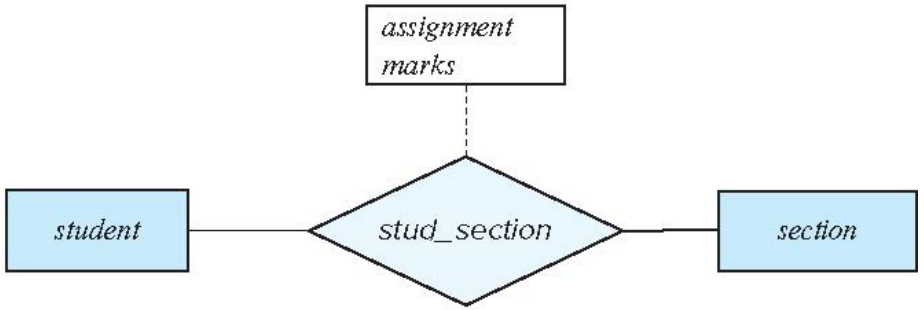
Design Issues

Common Mistakes in E-R Diagrams

- Example of erroneous E-R diagrams



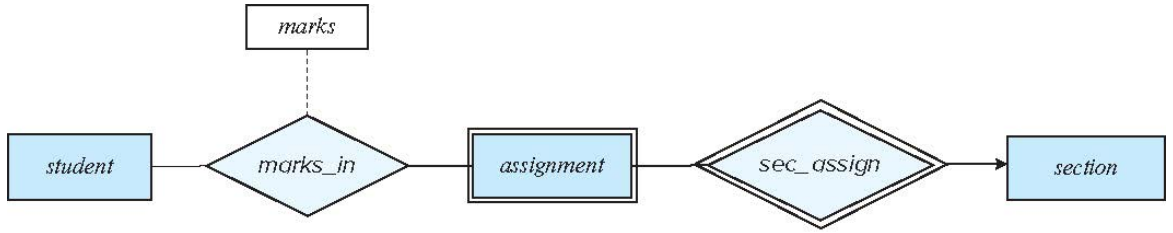
(a) Incorrect use of attribute



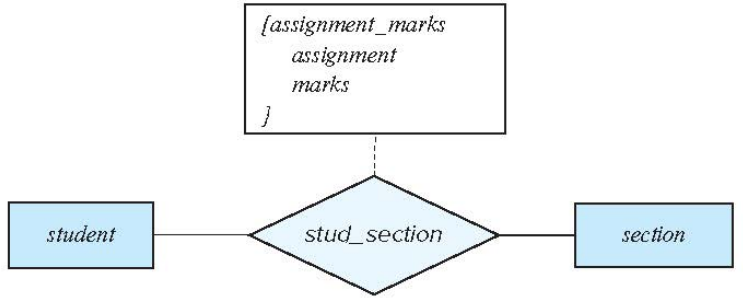
(b) Erroneous use of relationship attributes

Common Mistakes in E-R Diagrams (Cont.)

- Correct versions of the E-R diagram of previous slide



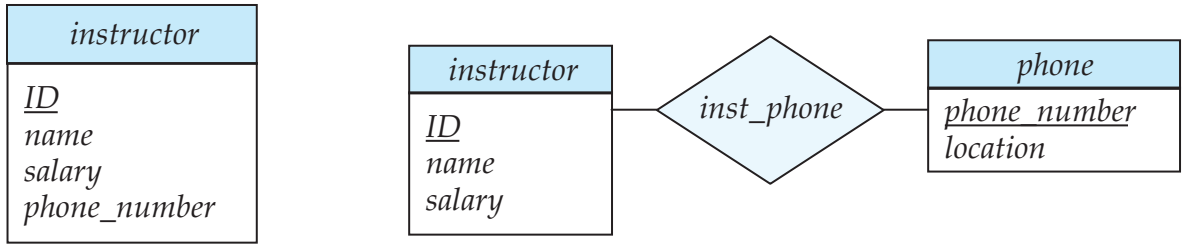
(c) Correct alternative to erroneous E-R diagram (b)



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

Entities vs. Attributes

- Use of entity sets vs. attributes

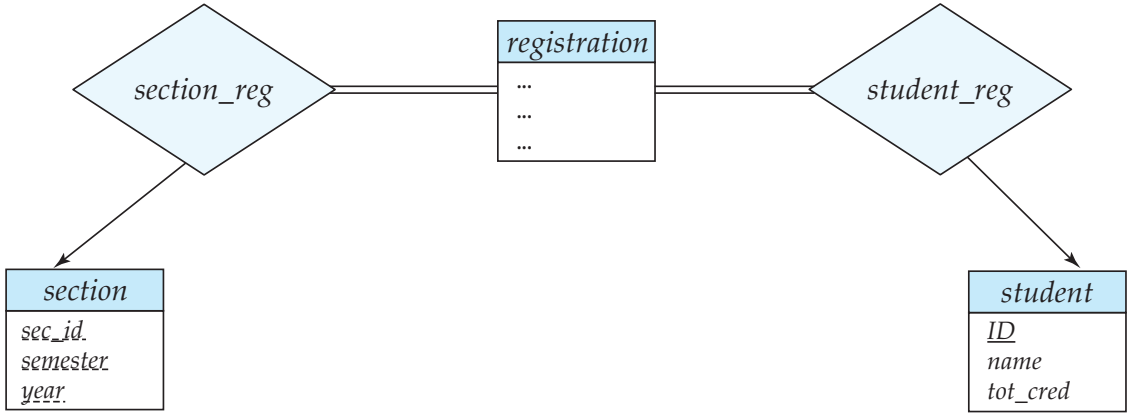


- Use of phone as an entity allows extra information about phone numbers (plus multiple phone numbers)

Entities vs. Relationship sets

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

  Possible guideline is to designate a relationship set to describe an action that occurs between entities



- **Placement of relationship attributes**

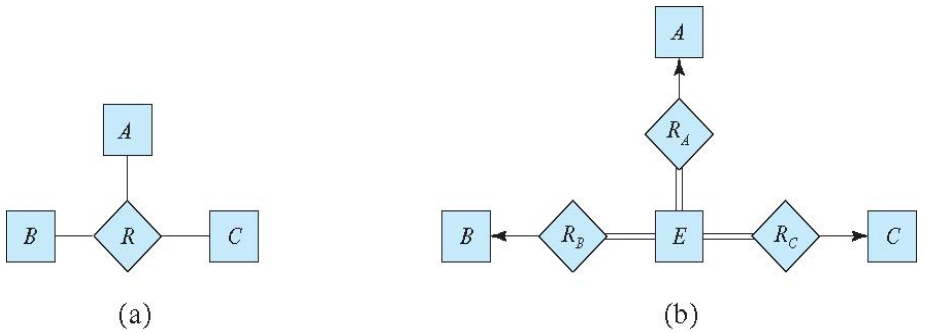
  For example, attribute date as attribute of advisor or as attribute of student

Binary Vs. Non-Binary Relationships

- Although it is possible to replace any non-binary (n-ary, for $n > 2$) relationship set by a number of distinct binary relationship sets, a n-ary relationship set shows more clearly that several entities participate in a single relationship.
- Some relationships that appear to be non-binary may be better represented using binary relationships
  - For example, a ternary relationship *parents*, relating a child to his/her father and mother, is best replaced by two binary relationships, *father* and *mother*
    - Using two binary relationships allows partial information (e.g., only mother being known)
  - But there are some relationships that are naturally non-binary
    - Example: *proj_guide*

Converting Non-Binary Relationships to Binary Form

- In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
  - Replace R between entity sets A, B and C by an entity set E, and three relationship sets:
    1. R_A, relating E and A
    2. R_B, relating E and B
    3. R_C, relating E and C
  - Create an identifying attribute for E *and* add any attributes of R to E
  - For each relationship (a_i , b_i , c_i) in R, create
    1. a new entity e_i in the entity set E
    2. add (e_i , a_i) to R_A
    3. add (e_i , b_i) to R_B
    4. add (e_i , c_i) to R_C



Converting Non-Binary Relationships (Cont.)

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

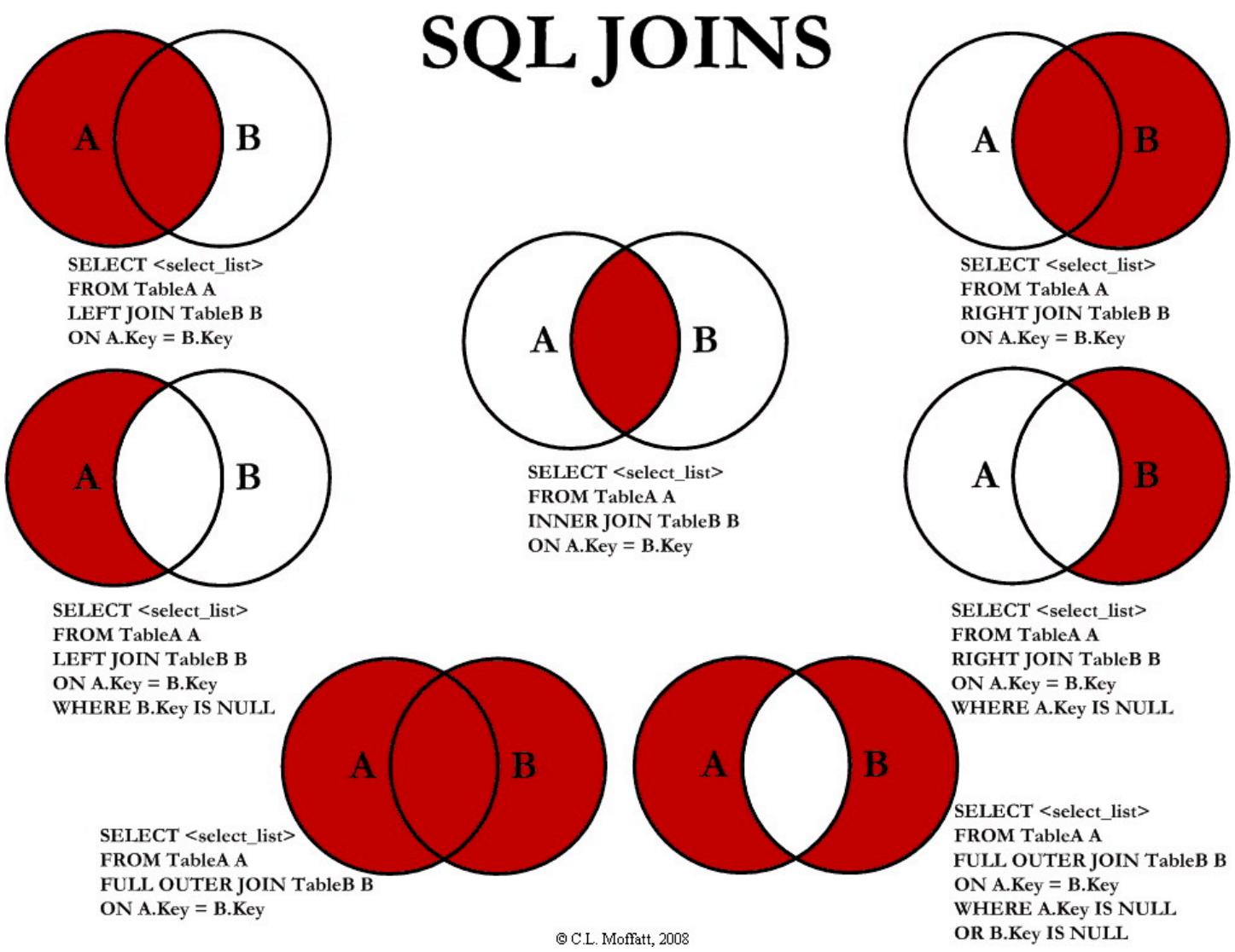
E-R Design Decisions

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

Advanced SQL

JOINs

How to Think About Joins



Remember

- The rows in the result are of the form:
  {ai, aj, am, b1,b7,b11}
- Represent a pair of rows that "matched" on JOIN condition.

- The way to think about these set diagrams is:
  - The cartesian product is all possible pairs of rows.
  - The JOINs represent the subsets of the cartesian product.

Authorization

Authorization

- We may assign a user several forms of authorizations on parts of the database.
  - **Read** - allows reading, but not modification of data.
  - **Insert** - allows insertion of new data, but not modification of existing data.
  - **Update** - allows modification, but not deletion of data.
  - **Delete** - allows deletion of data.
- Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

Authorization (Cont.)

- Forms of authorization to modify the database schema
  - **Index** - allows creation and deletion of indices.
  - **Resources** - allows creation of new relations.
  - **Alteration** - allows addition or deletion of attributes in a relation.
  - **Drop** - allows deletion of relations.

Authorization Specification in SQL

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

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

Privileges in SQL

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

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

Revoking Authorization in SQL

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

  revoke <privilege list> **on** <relation or view> **from** <user list>

- Example:

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

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

Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:

  create a role <name>
- Example:
  - **create role** instructor
- Once a role is created we can assign "users" to the role using:
  - **grant** <role> **to** <users>

Roles Example

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

Authorization on Views

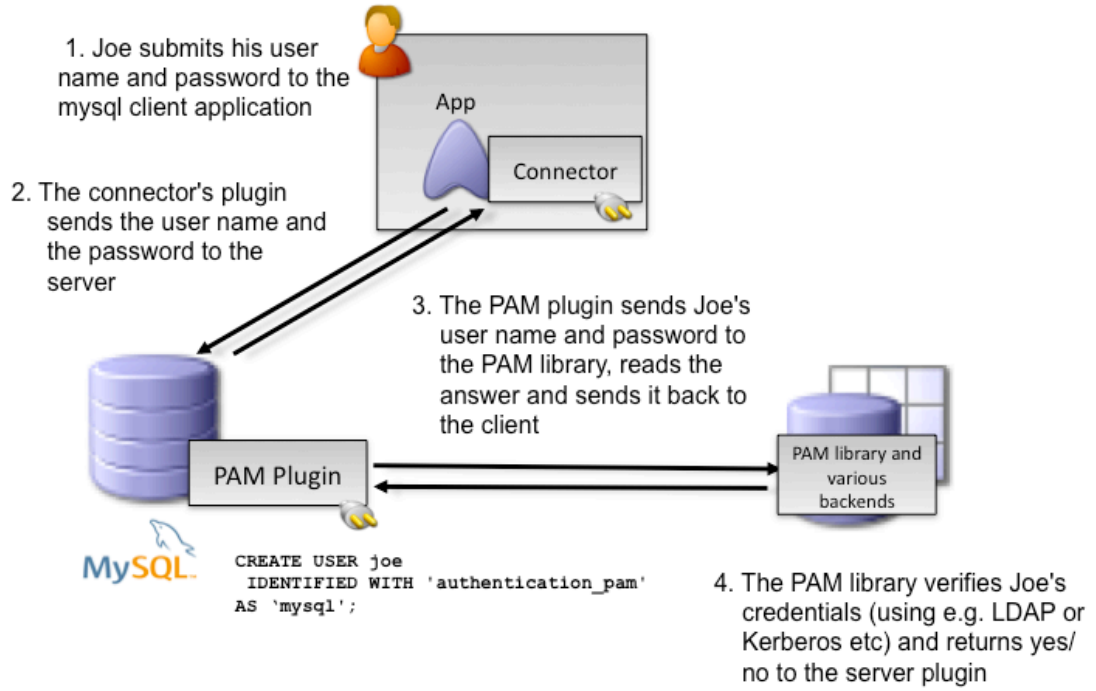
- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions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

Other Authorization Features

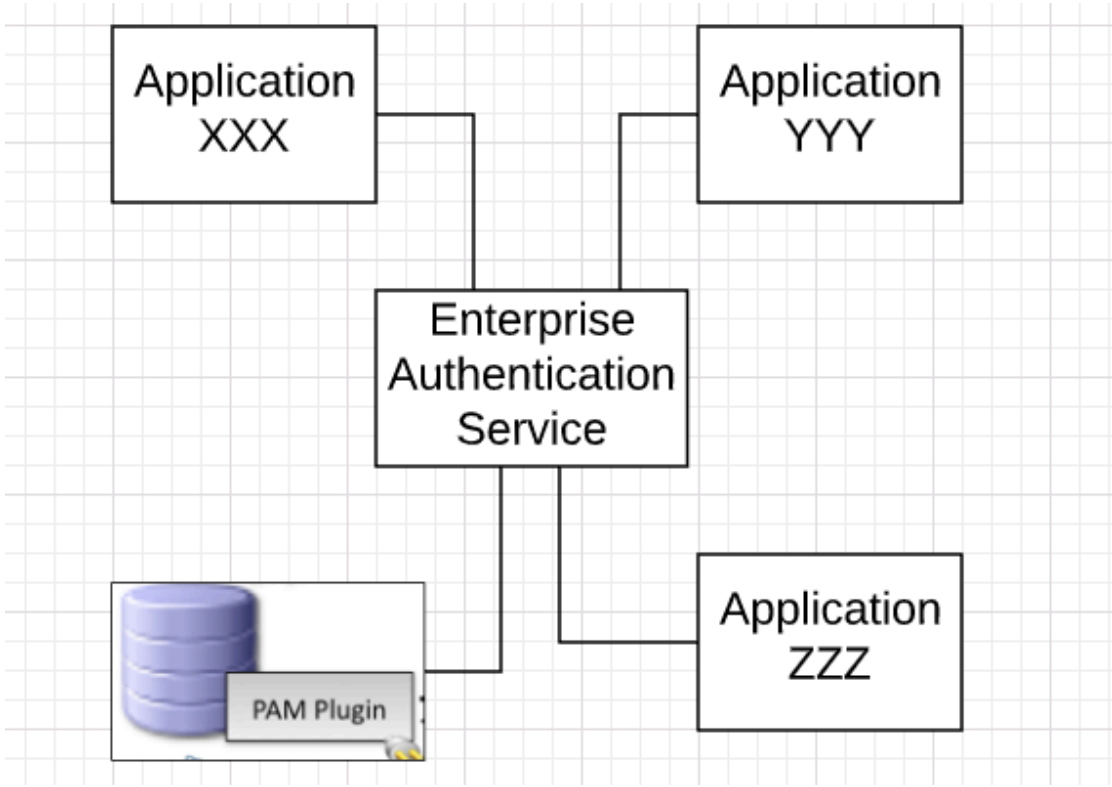
- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - Why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
  - And more!

Pluggable Authentication Module

Authentication

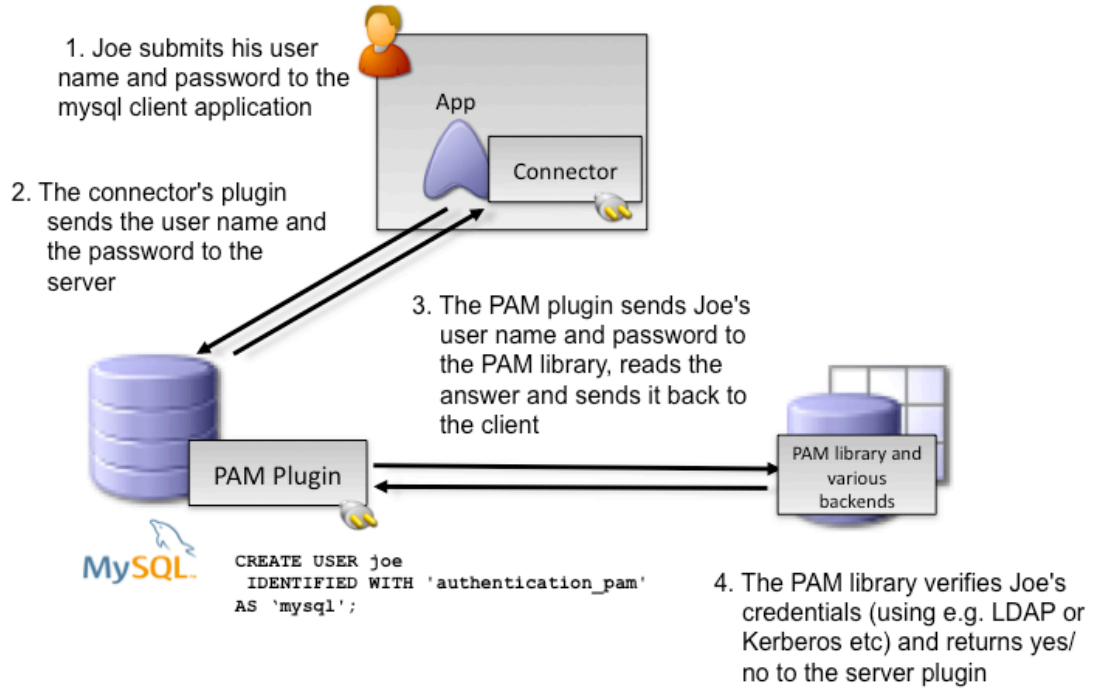


Commercial Database Management Systems support
"pluggable authentication modules."



Simplified Enterprise Authentication Concept

Authentication



Commercial Database Management Systems support
"pluggable authentication modules."

Recursive Queries

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book

The Power of Recursion

- Computing transitive closure using iteration, adding successive tuples to *rec_prereq*
  - The next slide shows a *prereq* relation
  - Each step of the iterative process constructs an extended version of *rec_prereq* from its recursive definition.
  - The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be **monotonic**. That is, if we add tuples to *prereq* the view *rec_prereq* contains all of the tuples it contained before, plus possibly more

Example of Fixed-Point Computation

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-319	CS-101
CS-319	CS-315
CS-347	CS-319

Iteration Number	*Tuples in c1*
0	
1	(CS-319)
2	(CS-319), (CS-315), (CS-101)
3	(CS-319), (CS-315), (CS-101), (CS-190)
4	(CS-319), (CS-315), (CS-101), (CS-190)
5	done