

W4111 – 02: Introduction to Databases

Lecture 3,4: ER Modeling, SQL, Relational

*Donald F. Ferguson
dff9@Columbia.edu*

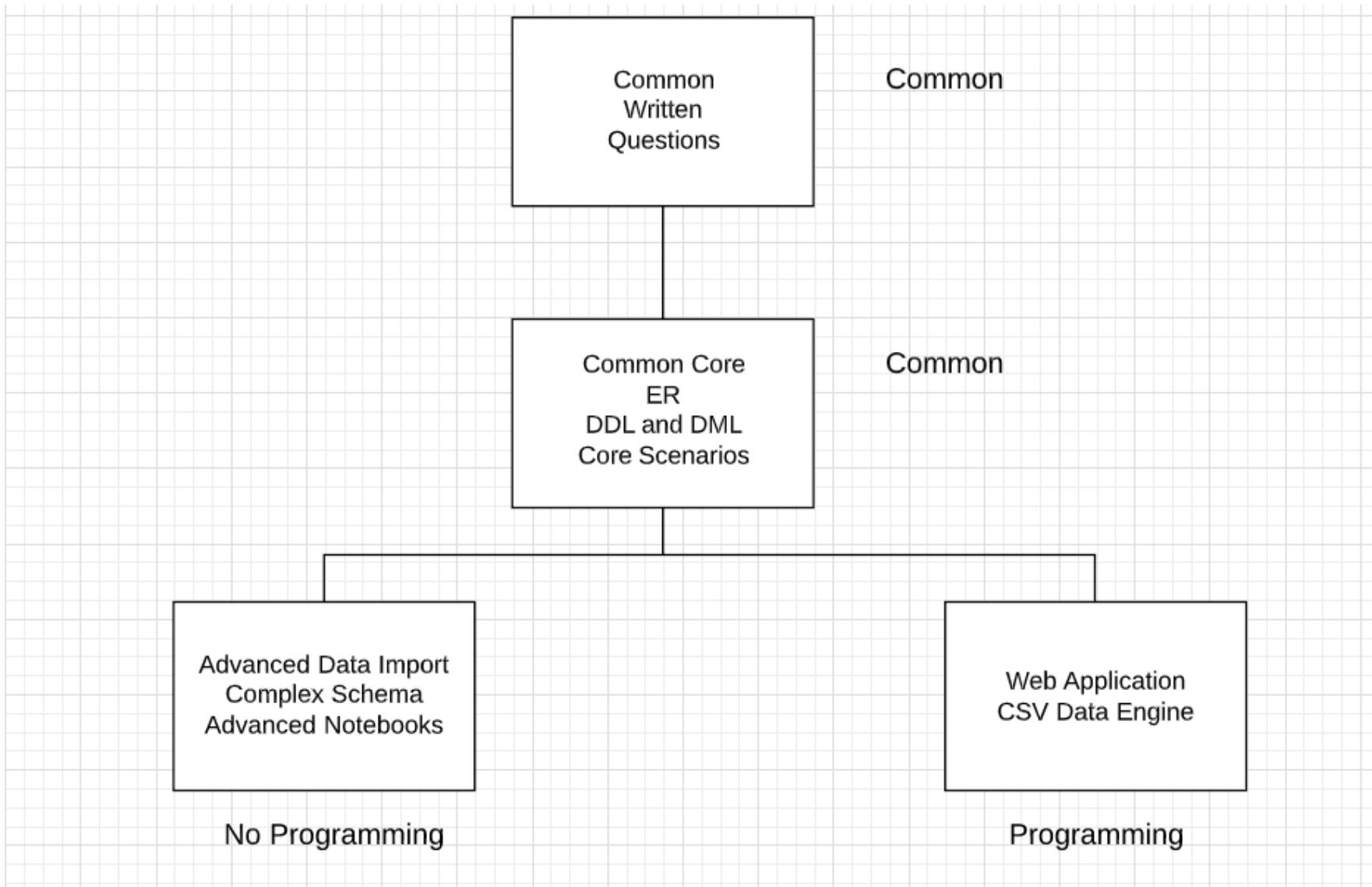
TRANSCENDING DISCIPLINES, TRANSFORMING LIVES

Today's Lecture – What You Need for HW 1

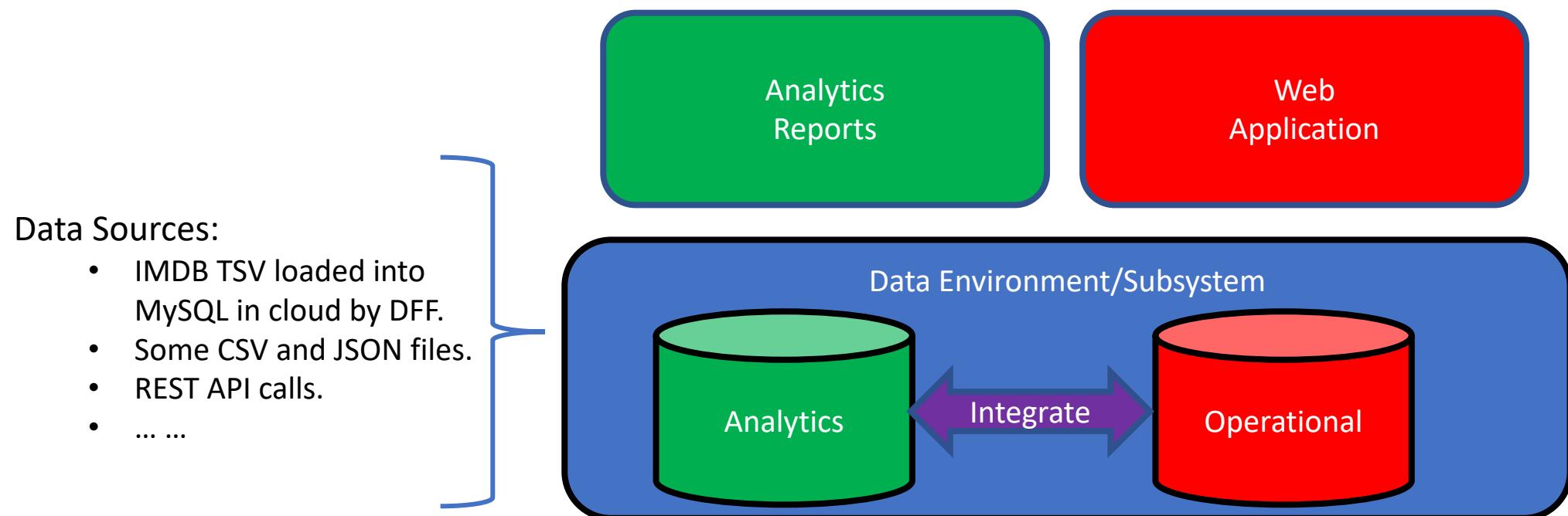
- Programming and No-Programming HW/Projects Direction
- Entity-Relationship Modeling:
 - Subset of Chapter 6 from recommended textbook.
 - Slides should be enough and there is a lot of online material.
- Crow's Foot Notation:
 - Simpler and more compact.
 - Very similar to what MySQL uses.
 - Demo of MySQL tools.
- Introduction to Some Patterns in SQL DDL
- Some Details on SQL DML SELECT.
- Interwoven with Some Demos.

Programming and No-Programming Homework Directions

Homework/Project Directions: Programming, No-Programming



Homework/Project Directions: Programming, No-Programming



- Homework/Project Approach:
 - DFF and TAs will provide helper code, some data/databases, helper code/libraries, implementation template.
 - Both tracks will cover the same core: databases, data models, queries, data transformation,
 - **Programming Track: REST API, web application, code to manage/transform data, ...**
 - **No-Programming: More complex data models/schema, using DML to load and transform, ...**
- Comments:
 - We will have to figure this out as we go. I have never done two tracks before
 - My sections of this course will continue to emphasize practical, hands-on,, but will cover theory.

Entity-Relationship Modeling



Chapter 6: Database Design Using the E-R Model

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Outline

- Overview of the Design Process
- The Entity-Relationship Model
- Complex Attributes
- Mapping Cardinalities
- Primary Key
- ~~Removing Redundant Attributes in Entity Sets~~
- ~~Reducing ER Diagrams to Relational Schemas~~
- ~~Extended E-R Features~~
- ~~Entity-Relationship Design Issues~~
- ~~Alternative Notations for Modeling Data~~
- ~~Other Aspects of Database Design~~

Will cover:

- In other lectures.
- Or required reading.



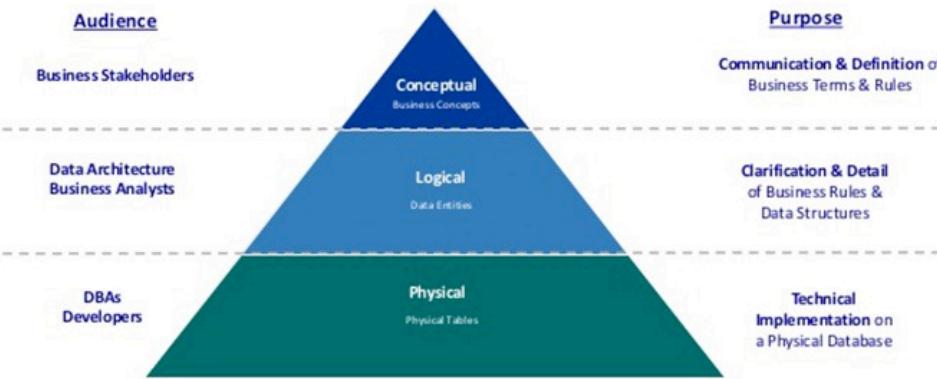
Outline

- Extended E-R Features
- Entity-Relationship Design Issues
- Alternative Notations for Modeling Data
- Other Aspects of Database Design

Logical – Physical – Conceptual

<https://ehikioya.com/conceptual-logical-physical-database-modeling/>

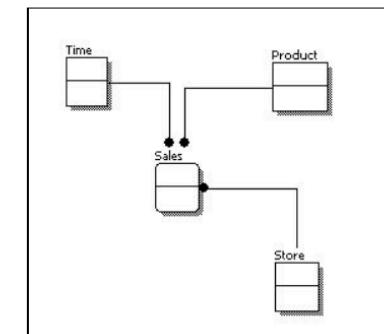
Levels of Data Modeling



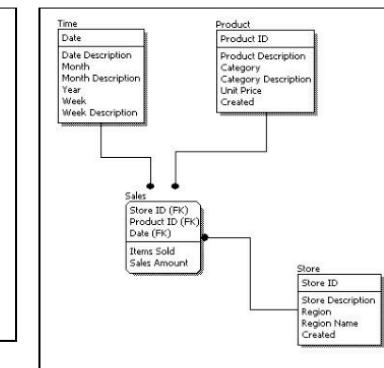
Feature	Conceptual	Logical	Physical
Entity Names	✓	✓	
Entity Relationships	✓	✓	
Attributes		✓	
Primary Keys		✓	✓
Foreign Keys		✓	✓
Table Names			✓
Column Names			✓
Column Data Types			✓

<https://www.1keydata.com/datawarehousing/data-modeling-levels.html>

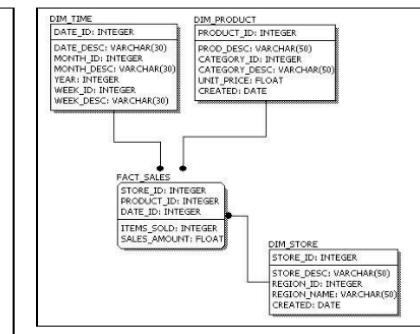
Conceptual Model Design



Logical Model Design



Physical Model Design



<https://www.1keydata.com/datawarehousing/data-modeling-levels.html>



Design Phases

- Initial phase -- characterize fully the data needs of the prospective database users.
- Second phase -- choosing a data model
 - Applying the concepts of the chosen data model
 - Translating these requirements into a conceptual schema of the database.
 - A fully developed conceptual schema indicates the functional requirements of the enterprise.
 - Describe the kinds of operations (or transactions) that will be performed on the data.



Design Phases (Cont.)

- Final Phase -- Moving from an abstract data model to the implementation of the database
 - Logical Design – Deciding on the database schema. Database design requires that we find a “good” collection of relation schemas.
 - Business decision – What attributes should we record in the database?
 - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
 - Physical Design – Deciding on the physical layout of the database



Design Alternatives

Emphasis
Added

- 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.



Design Approaches

- Entity Relationship Model (covered in this chapter)
 - Models an enterprise as a collection of *entities* and *relationships*
 - Entity: a “thing” or “object” in the enterprise that is distinguishable from other objects
 - Described by a set of *attributes*
 - Relationship: an association among several entities
 - Represented diagrammatically by an *entity-relationship diagram*:
- Normalization Theory (Chapter 7)
 - Formalize what designs are bad, and test for them



Outline of the ER Model



ER model -- Database Modeling

- The ER data mode was developed to facilitate database design by allowing specification of an **enterprise schema** that represents the overall logical structure of a database.
- The ER data model employs three basic concepts:
 - entity sets,
 - relationship sets,
 - attributes.
- The ER model also has an associated diagrammatic representation, the **ER diagram**, which can express the overall logical structure of a database graphically.



Entity Sets

- An **entity** is an object that exists and is distinguishable from other objects.
 - Example: specific person, company, event, plant
- An **entity set** is a set of entities of the same type that share the same properties.
 - Example: set of all persons, companies, trees, holidays
- An entity is represented by a set of attributes; i.e., descriptive properties possessed by all members of an entity set.
 - Example:
instructor = (ID, name, salary)
course= (course_id, title, credits)
- A subset of the attributes form a **primary key** of the entity set; i.e., uniquely identifying each member of the set.



Entity Sets -- *instructor* and *student*

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

instructor

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

student



Representing Entity sets in ER Diagram

- Entity sets can be represented graphically as follows:
 - Rectangles represent entity sets.
 - Attributes listed inside entity rectangle
 - Underline indicates primary key attributes

<i>instructor</i>
<u>ID</u>
<i>name</i>
<i>salary</i>

<i>student</i>
<u>ID</u>
<i>name</i>
<i>tot_cred</i>



Relationship Sets

- A **relationship** is an association among several entities

Example:

44553 (Peltier)	<u>advisor</u>	22222 (<u>Einstein</u>)
student entity	relationship set	instructor entity

- A **relationship set** is a mathematical relation among $n \geq 2$ entities, each taken from entity sets

$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

where (e_1, e_2, \dots, e_n) is a relationship

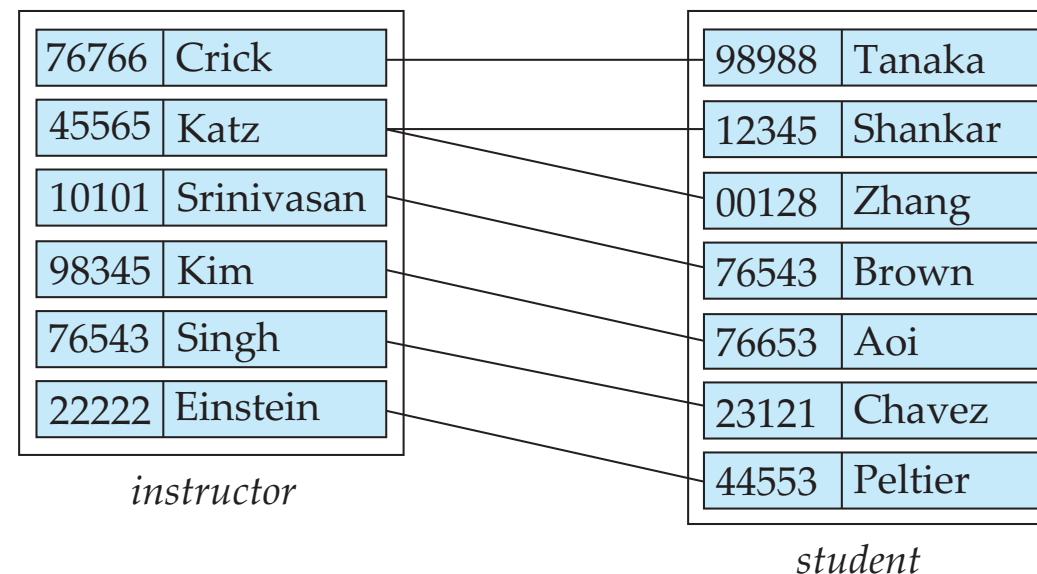
- Example:

$$(44553, 22222) \in \text{advisor}$$



Relationship Sets (Cont.)

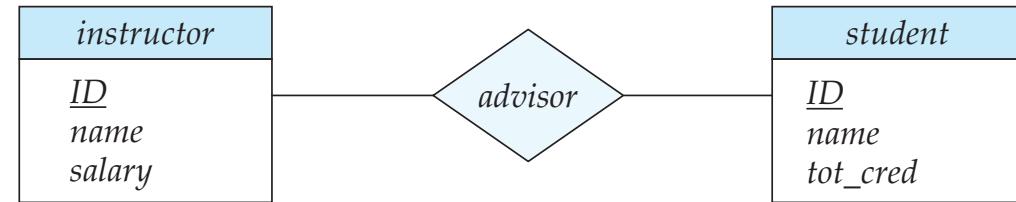
- Example: we define the relationship set *advisor* to denote the associations between students and the instructors who act as their advisors.
- Pictorially, we draw a line between related entities.





Representing Relationship Sets via ER Diagrams

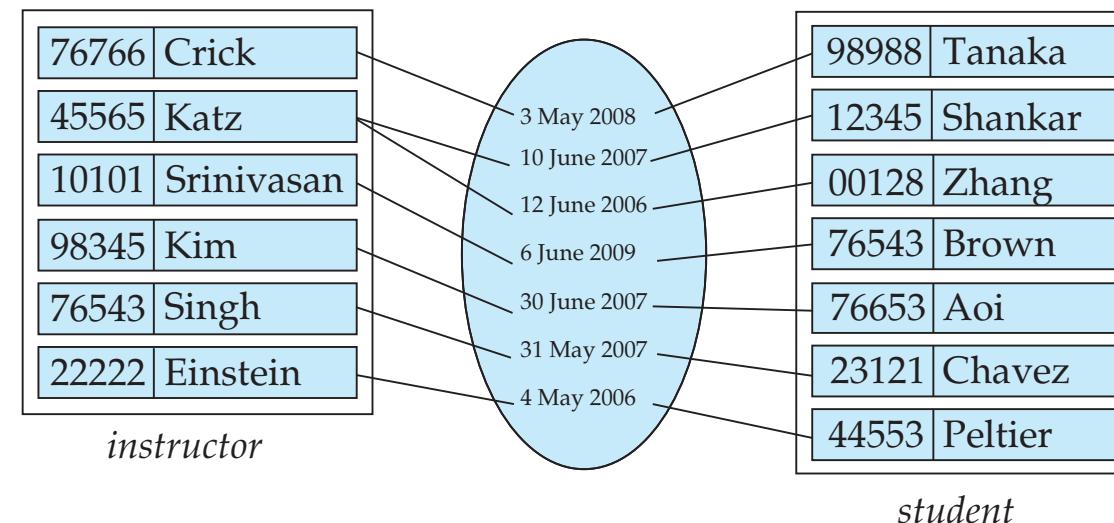
- Diamonds represent relationship sets.





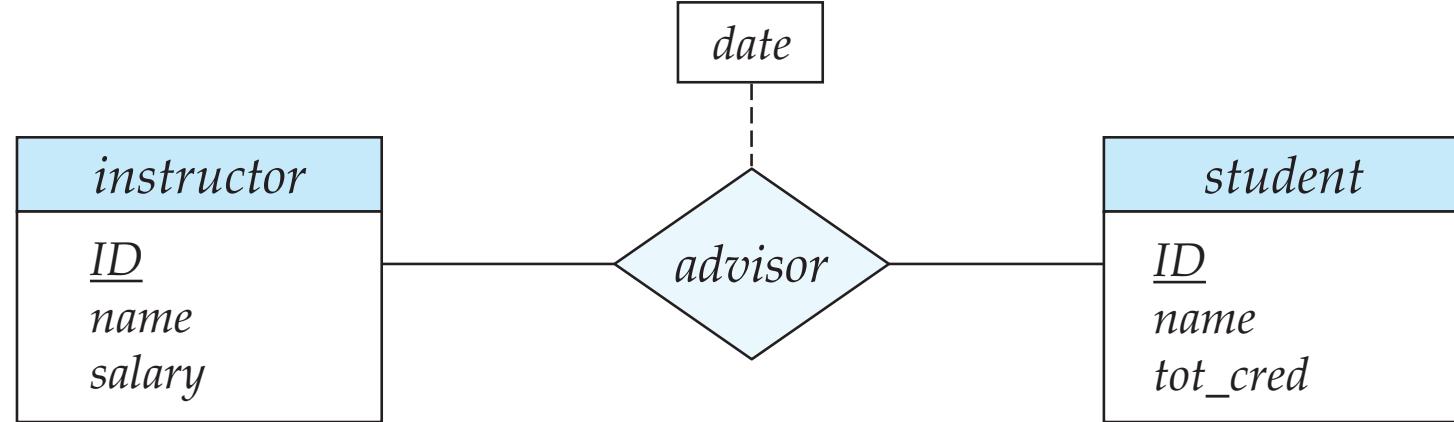
Relationship Sets (Cont.)

- An attribute can also be associated with a relationship set.
- For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have the attribute *date* which tracks when the student started being associated with the advisor





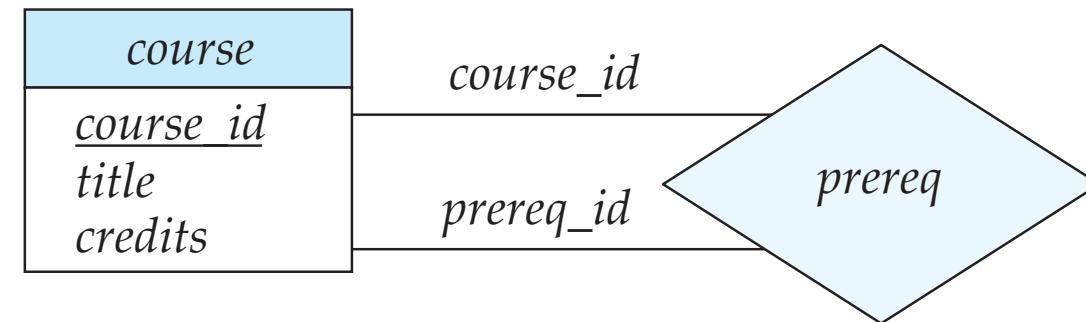
Relationship Sets with Attributes





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**.





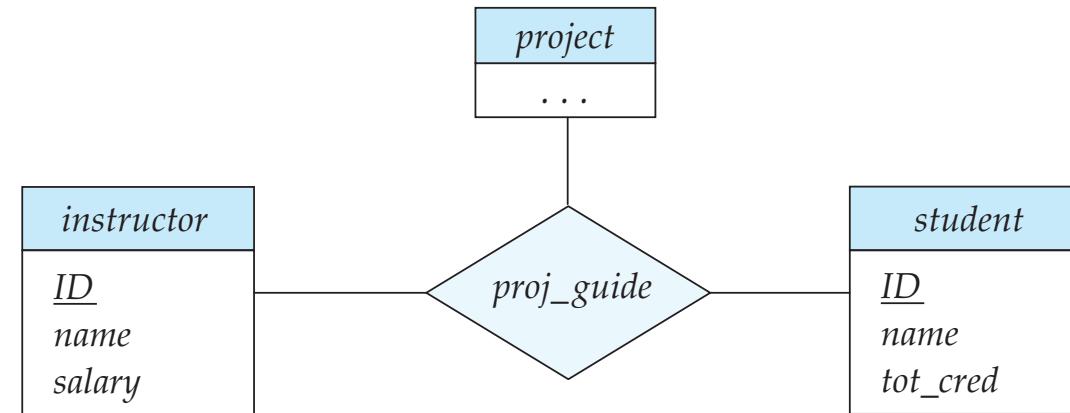
Degree of a Relationship Set

- 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





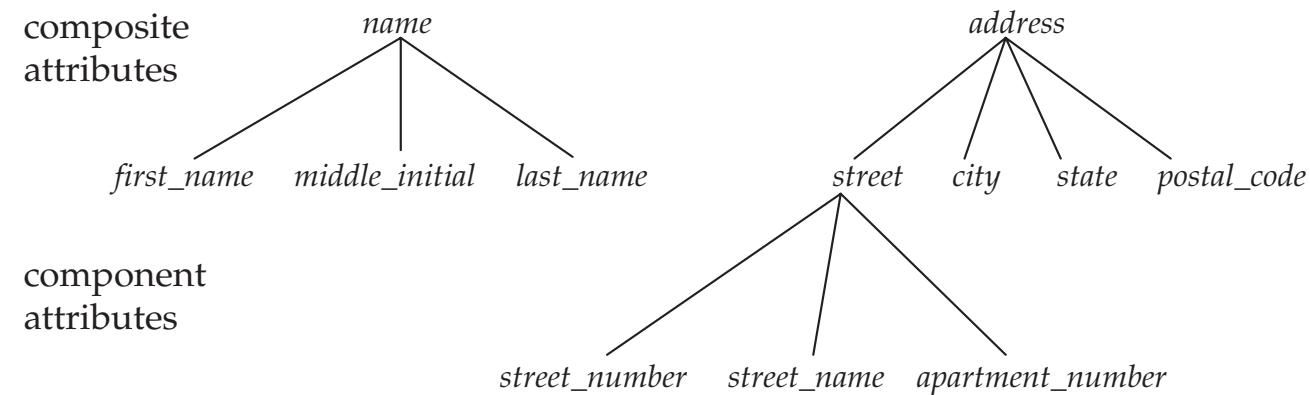
Complex Attributes

- 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 divide attributes into subparts (other attributes).





Representing Complex Attributes in ER Diagram

<i>instructor</i>
<u>ID</u>
<i>name</i>
<i>first_name</i>
<i>middle_initial</i>
<i>last_name</i>
<i>address</i>
<i>street</i>
<i>street_number</i>
<i>street_name</i>
<i>apt_number</i>
<i>city</i>
<i>state</i>
<i>zip</i>
{ <i>phone_number</i> }
<i>date_of_birth</i>
<i>age ()</i>

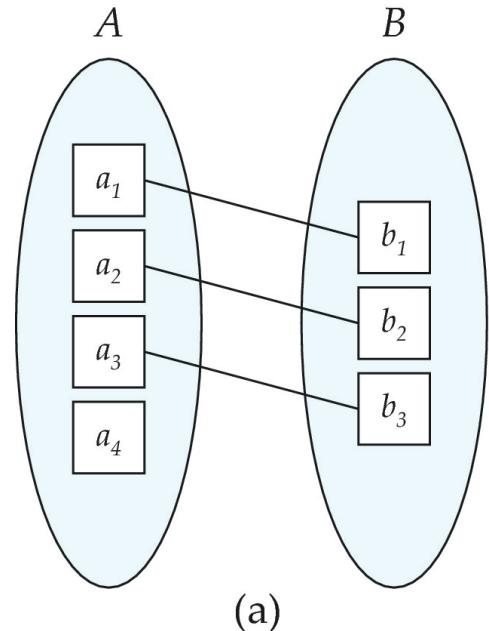


Mapping Cardinality Constraints

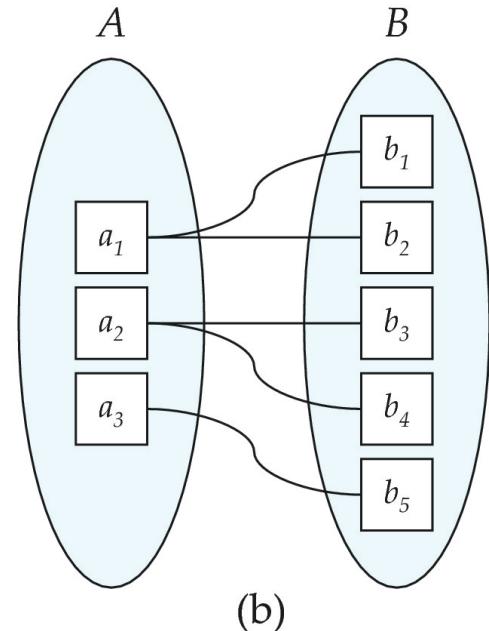
- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
 - One to one
 - One to many
 - Many to one
 - Many to many



Mapping Cardinalities



One to one

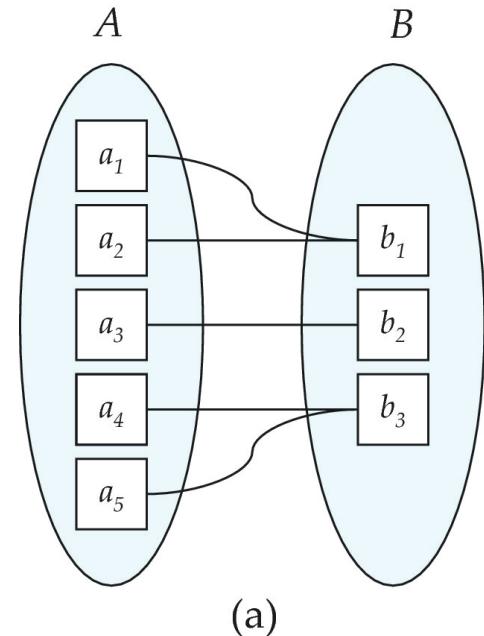


One to many

Note: Some elements in A and B may not be mapped to any elements in the other set

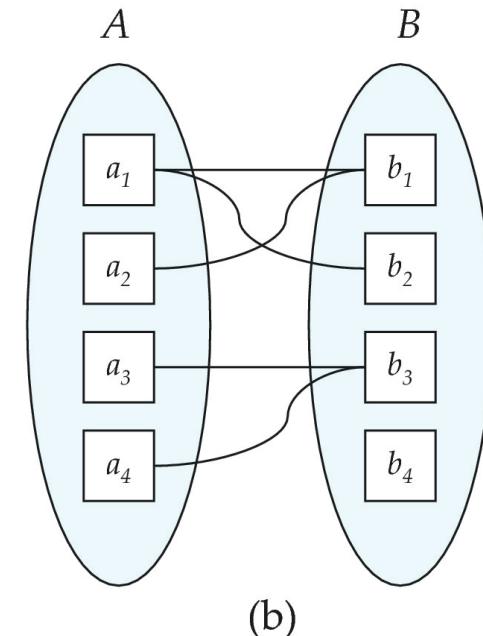


Mapping Cardinalities



(a)

Many to one



(b)

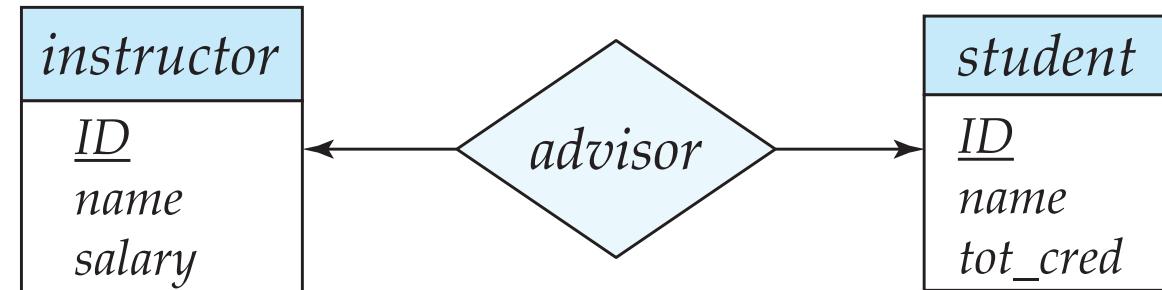
Many to many

Note: Some elements in A and B may not be mapped to any elements in the other set



Representing Cardinality Constraints in ER Diagram

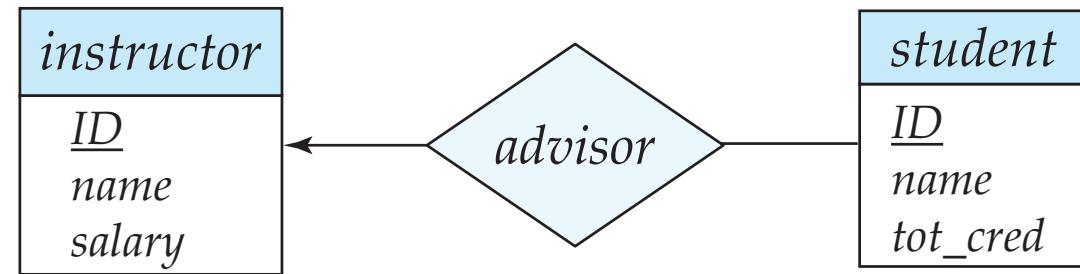
- We express cardinality constraints by drawing either a directed line (\rightarrow), signifying “one,” or an undirected line ($-$), signifying “many,” between the relationship set and the entity set.
- One-to-one relationship between an *instructor* and a *student* :
 - A student is associated with at most one *instructor* via the relationship *advisor*
 - A *student* is associated with at most one *department* via *stud_dept*





One-to-Many Relationship

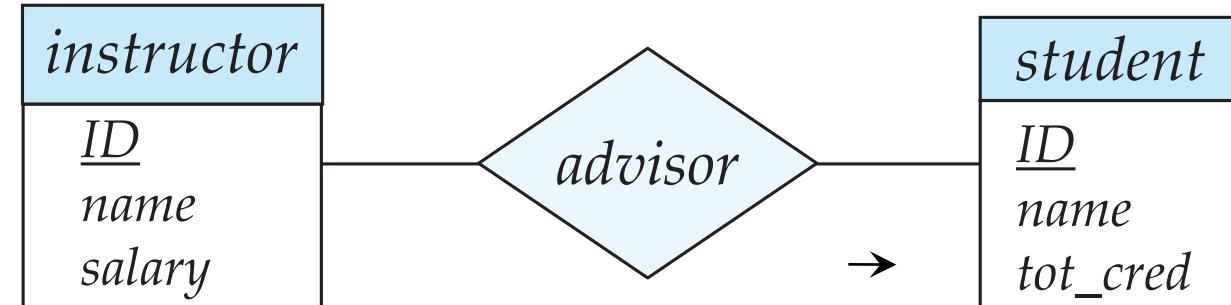
- one-to-many relationship between an *instructor* and a *student*
 - an instructor is associated with several (including 0) students via *advisor*
 - a student is associated with at most one instructor via advisor,





Many-to-One Relationships

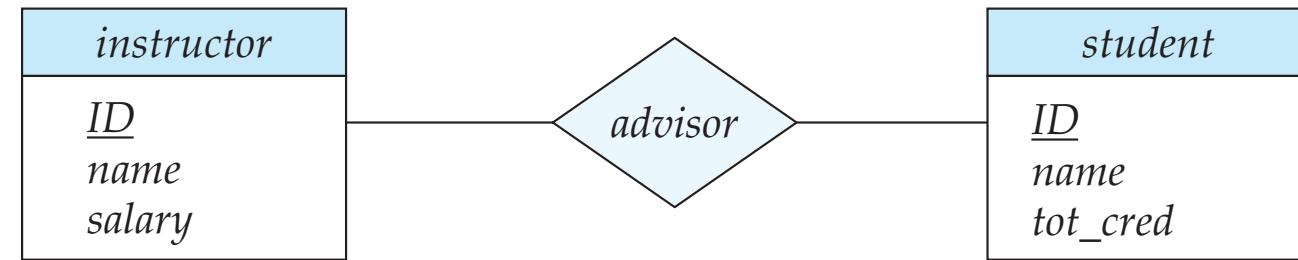
- In a many-to-one relationship between an *instructor* and a *student*,
 - an *instructor* is associated with at most one *student* via *advisor*,
 - and a *student* is associated with several (including 0) *instructors* via *advisor*





Many-to-Many Relationship

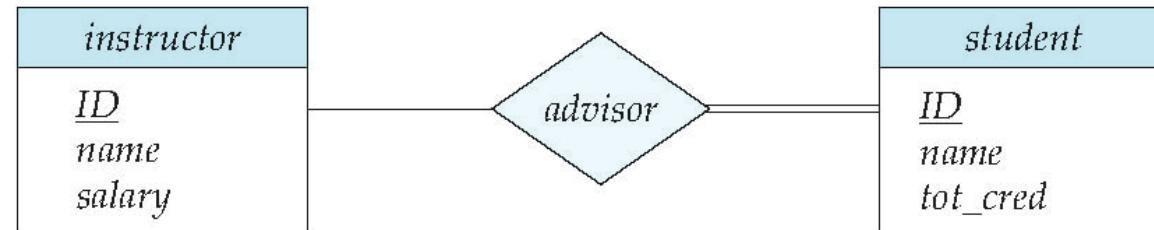
- An instructor is associated with several (possibly 0) students via *advisor*
- A student is associated with several (possibly 0) instructors via *advisor*





Total and Partial Participation

- **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set



participation of *student* in *advisor* relation is total

- every *student* must have an associated *instructor*
- **Partial participation:** some entities may not participate in any relationship in the relationship set
 - Example: participation of *instructor* in *advisor* is partial



Primary Key

- Primary keys provide a way to specify how entities and relations are distinguished. We will consider:
 - Entity sets
 - Relationship sets.
 - Weak entity sets



Primary key for Entity Sets

- By definition, individual entities are distinct.
- From database perspective, the differences among them must be expressed in terms of their attributes.
- The values of the attribute values of an entity must be such that they can uniquely identify the entity.
 - No two entities in an entity set are allowed to have exactly the same value for all attributes.
- A key for an entity is a set of attributes that suffice to distinguish entities from each other



Primary Key for Relationship Sets

- To distinguish among the various relationships of a relationship set we use the individual primary keys of the entities in the relationship set.
 - Let R be a relationship set involving entity sets E_1, E_2, \dots, E_n
 - The primary key for R consists of the union of the primary keys of entity sets E_1, E_2, \dots, E_n
 - If the relationship set R has attributes a_1, a_2, \dots, a_m associated with it, then the primary key of R also includes the attributes a_1, a_2, \dots, a_m
- Example: relationship set “advisor”.
 - The primary key consists of $instructor.ID$ and $student.ID$
- The choice of the primary key for a relationship set depends on the mapping cardinality of the relationship set.



Choice of Primary key for Binary Relationship

- Many-to-Many relationships. The preceding union of the primary keys is a minimal superkey and is chosen as the primary key.
- One-to-Many relationships . The primary key of the “Many” side is a minimal superkey and is used as the primary key.
- Many-to-one relationships. The primary key of the “Many” side is a minimal superkey and is used as the primary key.
- One-to-one relationships. The primary key of either one of the participating entity sets forms a minimal superkey, and either one can be chosen as the primary key.



Choice of Primary key for Nonbinary Relationship

- If no cardinality constraints are present, the superkey is formed as described earlier. and it is chosen as the primary key.
- If there are cardinality constraints are present:
 - Recall that we permit at most one arrow out of a relationship set.
 - AVI

Crow's Foot Notation

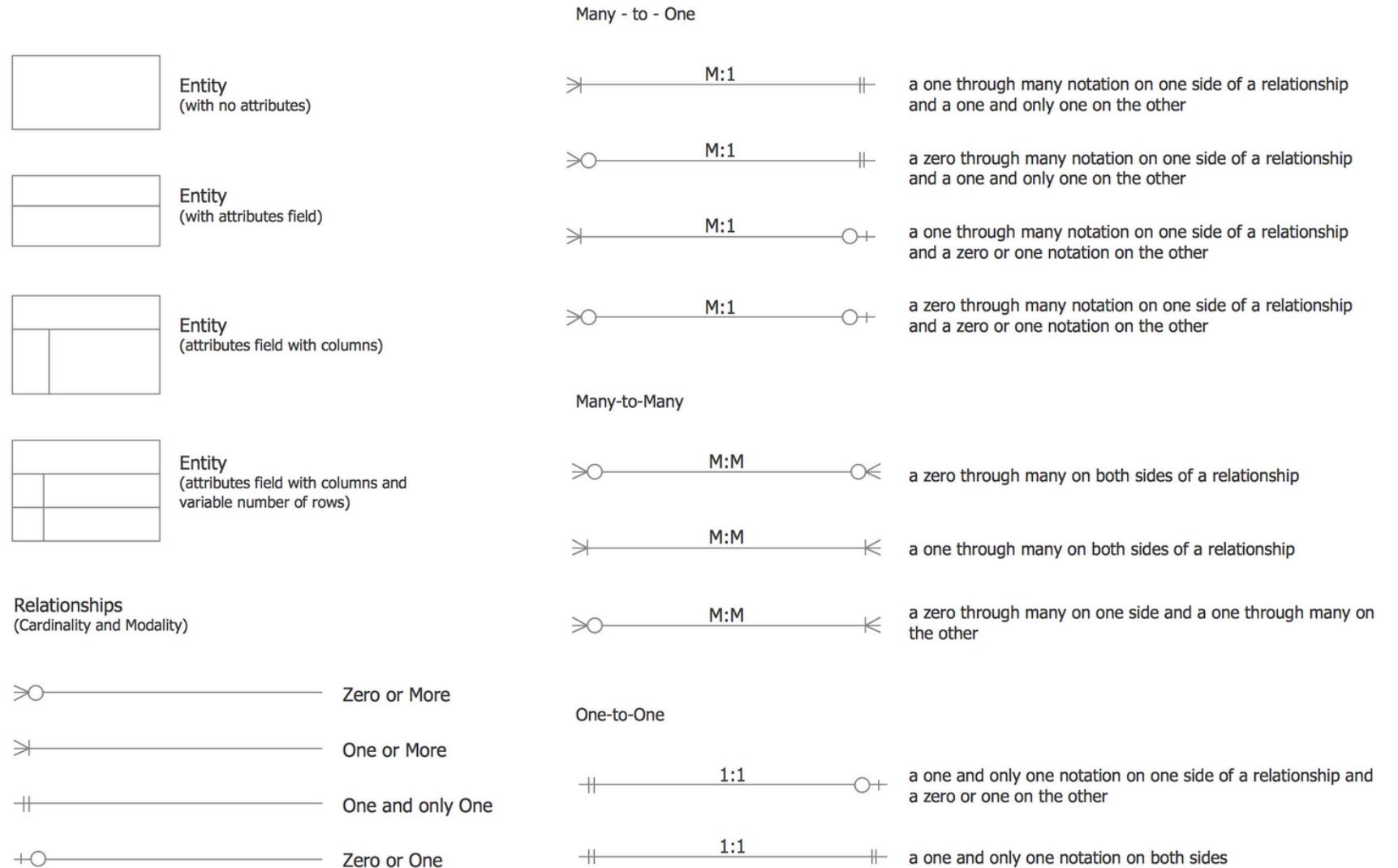
Crow's Foot Notation

Good intro: <https://www.vertabelo.com/blog/crow-s-foot-notation/>

Concepts

- Levels:
 - Conceptual
 - Logical
 - Physical
- Two directions:
 - To-Down
 - Bottom-Up
 - Meet-in-the-Middle

The concepts are general
and apply to all modeling,
not just Crow's Feet.



Switch to Demos

- MySQL: Bottom-Up applied to New Book schema.
- Lucidchart:
 - Conceptual → Logical → Physical
 - Applied to: Game of Thrones Data
 - The No-Programming Track will likely use this data.
 - We will decide after completion of homework 1.
- Write the corresponding SQL.

Example I – IMDB

SQL Details 1



Basic Query Structure

- A typical SQL query has the form:

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

- A_i represents an attribute
- R_i represents a relation
- P is a predicate.
- The result of an SQL query is a relation.



The select Clause

- The **select** clause lists the attributes desired in the result of a query
 - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:

```
select name  
      from instructor
```
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
 - E.g., *Name* \equiv *NAME* \equiv *name*
 - Some people use upper case wherever we use bold font.



The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select.
- Find the department names of all instructors, and remove duplicates

```
select distinct dept_name  
from instructor
```

- The keyword **all** specifies that duplicates should not be removed.

```
select all dept_name  
from instructor
```



The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

```
select *  
from instructor
```

- An attribute can be a literal with no **from** clause

```
select '437'
```

- Results is a table with one column and a single row with value “437”
- Can give the column a name using:

```
select '437' as FOO
```

- An attribute can be a literal with **from** clause

```
select 'A'  
from instructor
```

- Result is a table with one column and N rows (number of tuples in the *instructors* table), each row with value “A”



The select Clause (Cont.)

- The **select** clause can contain arithmetic expressions involving the operation, +, -, *, and /, and operating on constants or attributes of tuples.

- The query:

```
select ID, name, salary/12  
from instructor
```

would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

- Can rename “*salary/12*” using the **as** clause:

```
select ID, name, salary/12 as monthly_salary
```



The where Clause

- The **where** clause specifies conditions that the result must satisfy
 - Corresponds to the selection predicate of the relational algebra.

- To find all instructors in Comp. Sci. dept

```
select name  
from instructor  
where dept_name = 'Comp. Sci.'
```

- SQL allows the use of the logical connectives **and**, **or**, and **not**
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
- Comparisons can be applied to results of arithmetic expressions
- To find all instructors in Comp. Sci. dept with salary > 80000

```
select name  
from instructor  
where dept_name = 'Comp. Sci.' and salary > 80000
```



The from Clause

- The **from** clause lists the relations involved in the query
 - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

```
select *
  from instructor, teaches
```

 - generates every possible instructor – teaches pair, with all attributes from both relations.
 - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).



Examples

- Find the names of all instructors who have taught some course and the course_id
 - **select name, course_id
from instructor , teaches
where instructor.ID = teaches.ID**
- Find the names of all instructors in the Art department who have taught some course and the course_id
 - **select name, course_id
from instructor , teaches
where instructor.ID = teaches.ID **and** instructor.dept_name = 'Art'**



The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

old-name as new-name

- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.

- **select distinct** *T.name*
from *instructor as T, instructor as S*
where *T.salary > S.salary and S.dept_name = 'Comp. Sci.'*

- Keyword **as** is optional and may be omitted

instructor as T \equiv *instructor T*



Self Join Example

- Relation *emp-super*

<i>person</i>	<i>supervisor</i>
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Find the supervisor of “Bob”
- Find the supervisor of the supervisor of “Bob”
- Can you find ALL the supervisors (direct and indirect) of “Bob”?



String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:
 - percent (%). The % character matches any substring.
 - underscore (_). The _ character matches any character.
- Find the names of all instructors whose name includes the substring “dar”.

```
select name  
from instructor  
where name like '%dar%'
```

- Match the string “100%”

```
like '100 \%' escape '\'
```

in that above we use backslash (\) as the escape character.



String Operations (Cont.)

- Patterns are case sensitive.
- Pattern matching examples:
 - 'Intro%' matches any string beginning with “Intro”.
 - '%Comp%' matches any string containing “Comp” as a substring.
 - '___' matches any string of exactly three characters.
 - '___%' matches any string of at least three characters.
- SQL supports a variety of string operations such as
 - concatenation (using “||”)
 - converting from upper to lower case (and vice versa)
 - finding string length, extracting substrings, etc.



Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

```
select distinct name
  from instructor
 order by name
```

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
 - Example: **order by name desc**
- Can sort on multiple attributes
 - Example: **order by dept_name, name**



Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, $\geq \$90,000$ and $\leq \$100,000$)
 - **select name
from instructor
where salary between 90000 and 100000**
- Tuple comparison
 - **select name, course_id
from instructor, teaches
where (instructor.ID, dept_name) = (teaches.ID, 'Biology');**



Null Values

- It is possible for tuples to have a null value, denoted by **null**, for some of their attributes
- **null** signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving **null** is **null**
 - Example: $5 + \text{null}$ returns **null**
- The predicate **is null** can be used to check for null values.
 - Example: Find all instructors whose salary is null.

```
select name  
from instructor  
where salary is null
```

- The predicate **is not null** succeeds if the value on which it is applied is not null.



Null Values (Cont.)

- SQL treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**).
 - Example: $5 < \text{null}$ or $\text{null} \neq \text{null}$ or $\text{null} = \text{null}$
- The predicate in a **where** clause can involve Boolean operations (**and**, **or**, **not**); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.
 - **and** : $(\text{true and unknown}) = \text{unknown}$,
 $(\text{false and unknown}) = \text{false}$,
 $(\text{unknown and unknown}) = \text{unknown}$
 - **or**: $(\text{unknown or true}) = \text{true}$,
 $(\text{unknown or false}) = \text{unknown}$
 $(\text{unknown or unknown}) = \text{unknown}$
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*



Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value

min: minimum value

max: maximum value

sum: sum of values

count: number of values



Aggregate Functions Examples

- Find the average salary of instructors in the Computer Science department
 - **select avg (salary)
from instructor
where dept_name= 'Comp. Sci.';**
- Find the total number of instructors who teach a course in the Spring 2018 semester
 - **select count (distinct ID)
from teaches
where semester = 'Spring' **and** year = 2018;**
- Find the number of tuples in the *course* relation
 - **select count (*)
from course;**



Aggregate Functions – Group By

- Find the average salary of instructors in each department
 - `select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name;`

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

<i>dept_name</i>	<i>avg_salary</i>
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000



Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list

- /* erroneous query */
select dept_name, ID, avg (salary)
from instructor
group by dept_name;



Aggregate Functions – Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

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

- Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups



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  $A_1, A_2, \dots, A_n$ 
  from  $r_1, r_2, \dots, r_m$ 
    where  $P$ 
```

as follows:

- **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 $<\text{operation}>$ to be defined later.

- **Select clause:**
 A_i can be replaced by a subquery that generates a single value.



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 From Clause



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



Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause
- Three types of joins:
 - Natural join
 - Inner join
 - Outer join



Natural Join in SQL

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column.
- List the names of instructors along with the course ID of the courses that they taught
 - **`select name, course_id
from students, takes
where student.ID = takes.ID;`**
- Same query in SQL with “natural join” construct
 - **`select name, course_id
from student natural join takes;`**



Natural Join in SQL (Cont.)

- The **from** clause can have multiple relations combined using natural join:

```
select A1, A2, ... An  
from r1 natural join r2 natural join .. natural join rn  
where P ;
```



Student Relation

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>
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



Takes Relation

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
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	<i>null</i>



student natural join takes

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	<i>null</i>



Dangerous in Natural Join

- Beware of unrelated attributes with same name which get equated incorrectly
- Example -- List the names of students instructors along with the titles of courses that they have taken
 - Correct version

```
select name, title  
from student natural join takes, course  
where takes.course_id = course.course_id;
```

- Incorrect version
 - **select** name, title
from student **natural join** takes **natural join** course;
 - This query omits all (student name, course title) pairs where the student takes a course in a department other than the student's own department.
 - The correct version (above), correctly outputs such pairs.



Natural Join with Using Clause

- To avoid the danger of equating attributes erroneously, we can use the “**using**” construct that allows us to specify exactly which columns should be equated.
- Query example

```
select name, title  
from (student natural join takes) join course using (course_id)
```



Join Condition

- The **on** condition allows a general predicate over the relations being joined
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**
- Query example

```
select *  
from student join takes on student_ID = takes_ID
```

- The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.
- Equivalent to:

```
select *  
from student , takes  
where student_ID = takes_ID
```



Join Condition (Cont.)

- The **on** condition allows a general predicate over the relations being joined.
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**.
- Query example

```
select *  
from student join takes on student_ID = takes_ID
```

- The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.

- Equivalent to:

```
select *  
from student , takes  
where student_ID = takes_ID
```



Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.
- Three forms of outer join:
 - left outer join
 - right outer join
 - full outer join



Outer Join Examples

- Relation *course*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that

course information is missing CS-437

prereq information is missing CS-315

- \times



Left Outer Join

- *course natural left outer join prereq*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	<i>null</i>

- In relational algebra: *course* \bowtie *prereq*



Right Outer Join

- *course natural right outer join prereq*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	<i>null</i>	<i>null</i>	<i>null</i>	CS-101

- In relational algebra: *course* \bowtie *prereq*



Full Outer Join

- *course natural full outer join prereq*

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>	<i>prereq_id</i>
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	<i>null</i>
CS-347	<i>null</i>	<i>null</i>	<i>null</i>	CS-101

- In relational algebra: *course ⋗ prereq*



Joined Types and Conditions

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

<i>Join types</i>
inner join
left outer join
right outer join
full outer join

<i>Join conditions</i>
natural
on < predicate >
using (A₁, A₂, ..., A_n)



Joined Relations – Examples

- *course natural right outer join prereq*

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	null	null	null	CS-101

- *course full outer join prereq using (course_id)*

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	null
CS-347	null	null	null	CS-101



Joined Relations – Examples

- **course inner join prereq on**
 $course.course_id = prereq.course_id$

course_id	title	dept_name	credits	prereq_id	course_id
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- What is the difference between the above, and a natural join?
- **course left outer join prereq on**
 $course.course_id = prereq.course_id$

course_id	title	dept_name	credits	prereq_id	course_id
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	null	null



Joined Relations – Examples

- *course natural right outer join prereq*

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	null	null	null	CS-101

- *course full outer join prereq using (course_id)*

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	null
CS-347	null	null	null	CS-101