W4111 – Introduction to Databases
Section 002, Fall 2021

Lecture 2: ER, Relational, SQL (I)

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

W4111 – Introduction to Databases
Section 002, Fall 2021

Lecture 2: ER, Relational, SQL (I)

We will start in a couple of minutes.

Contents

Contents

- Introduction: comments, questions and answers
- Motivating Sample Problems:
  - Bottom-Up: IMDB, Game of Thrones
  - Top-Down: Columbia University/Sample University
- ER (Diagram) Modeling (Chapter 6)
- The *Relational Model* and *Algebra* (Chapter 2)
- SQL (Chapter 3)

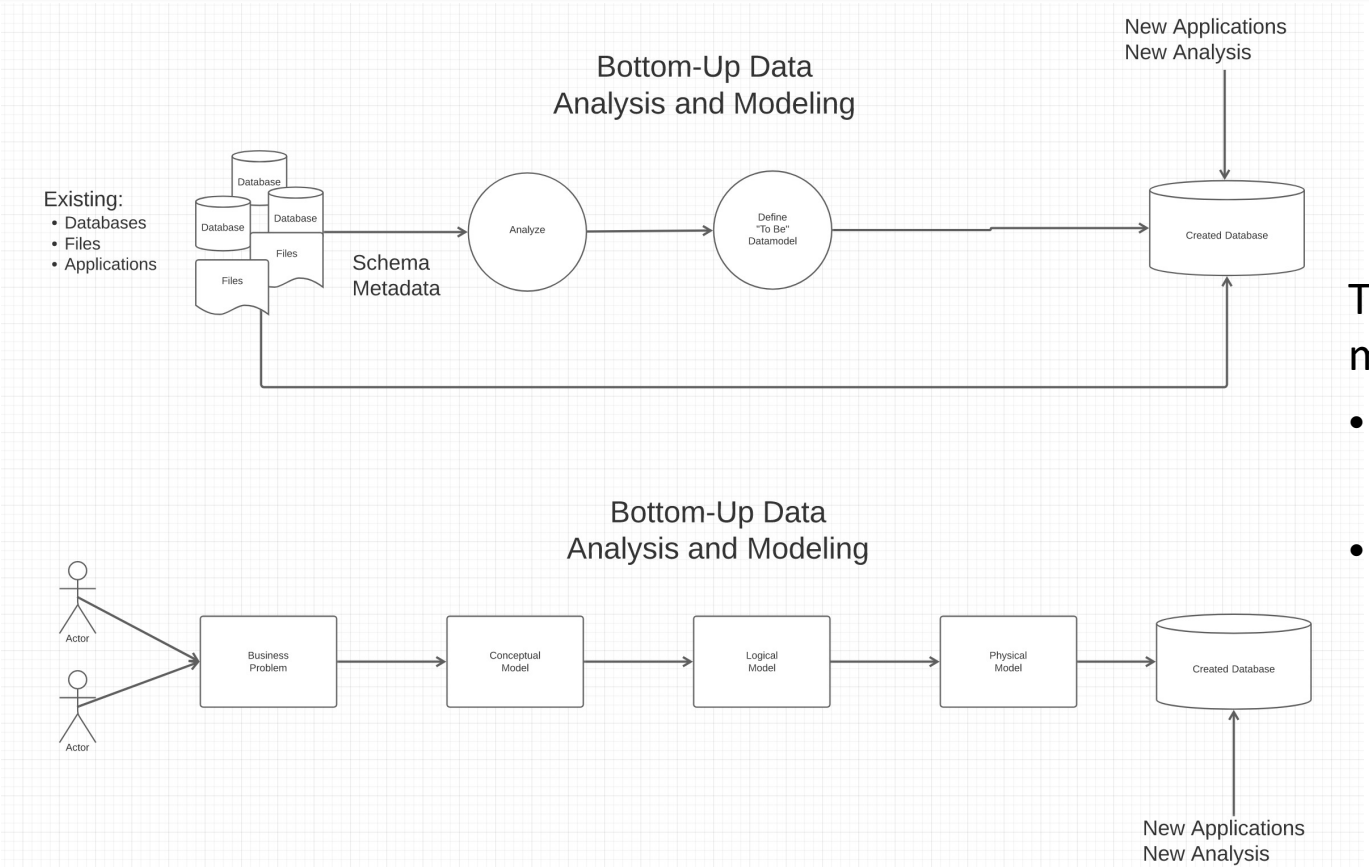
Questions, Answers, Comments

TBD

- Answer interesting questions from discussion forum.
- Answer questions from the class.

Motivating Problems

Top-Down and Bottom-Up Data Modeling



The most common is meet-in-the-middle:

- Top-down defines the end state.
- Bottom-up defines how to get there.

Bottom-Up: Game of Thrones and IMDB

Several files from IMDB:

- Titles Basics
- Titles AKAS
- Titles Episodes
- Titles Crews
- Title Principals
- Title Ratings
- Name Basics

Tab Separated Value

Game of Thrones:
(https://jeffreylancaster.github.io/game-of-thrones/)

JSON

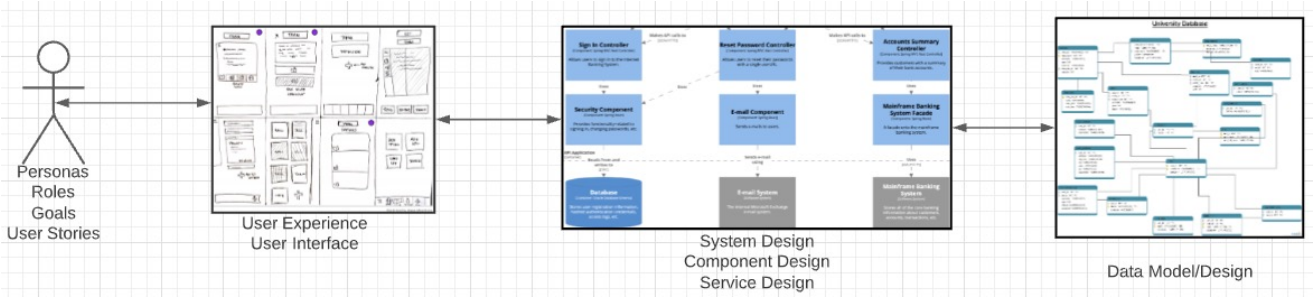
- Characters, Genders, Groups
- Episodes
- Locations, Opening Locations
- Words: Script, Bag, ...
- Geography
- ... ...

Take a Look

- Open Jupyter Notebook

Top-Down: University Management

- We must build a system that supports academic operations in a university.
- There are two major domains:
  - Interactive operations, e.g. register, choose class, assign grade, ...
  - Insight and reporting, e.g. enrollment trends, overloaded resources, ...
- We will design, develop, test and deploy the system iteratively and continuously.
- There are four core domains.



- In this course,
- We focus on the data dimension.
- We will get some insight into the other dimensions.

- The processes are iterative, with continuous extension and details.
  We will obviously focus on the data aspects in a database course.

Database design, Entity-Relationship Model (Part 1)

Concepts

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.

DFF Comments:

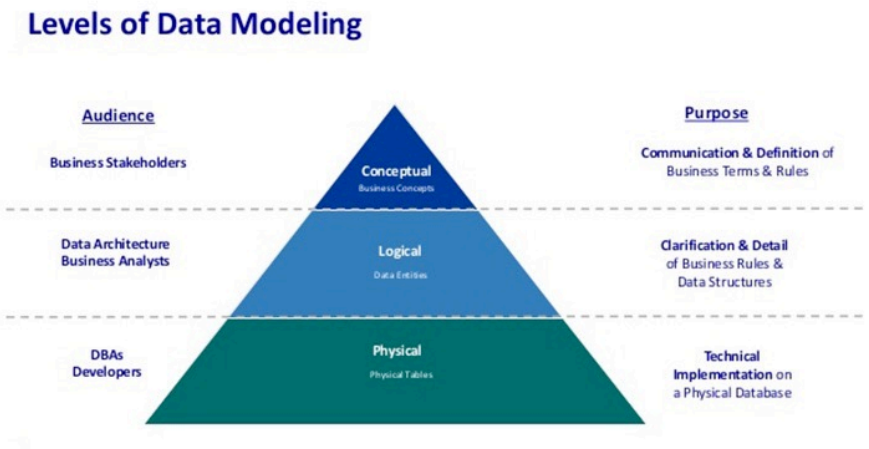
- We you see slides with this formatting, the come directly from the presentations associated with the textbook. (https://www.db-book.com/db7/slides-dir/index.html)
- The number at the bottom is of the form chapter.slide_no.
- I try to put my comments, modifications and annotations in red text, or inside a red rectangle/callout.

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

A Common and my Approach: Conceptual → Logical → Physical

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

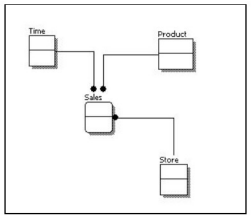


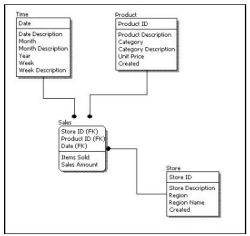
- It is easy to get carried away with modeling. You can spend all your time modeling and not actually build the schema.
- We will use the approaches in class.
- Mostly to understand concepts and patterns.

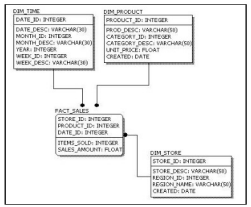
https://www.1keydata.com/datawarehousing/data-modeling-levels.html

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

Conceptual Model Design | Logical Model Design | Physical Model Design







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

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.

DFF Comments:

- Some of these statements apply primarily to OO systems and the relational/SQL models.
- A motivation for "No SQL" is to relax the constraints.

Entity Sets -- *instructor* and *student*

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

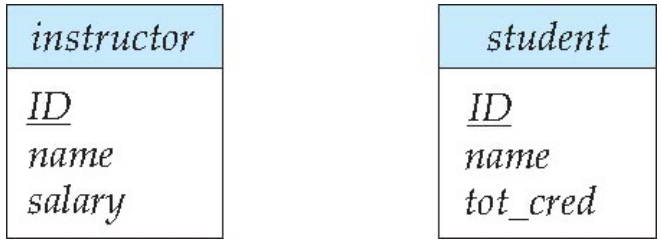
instructor

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

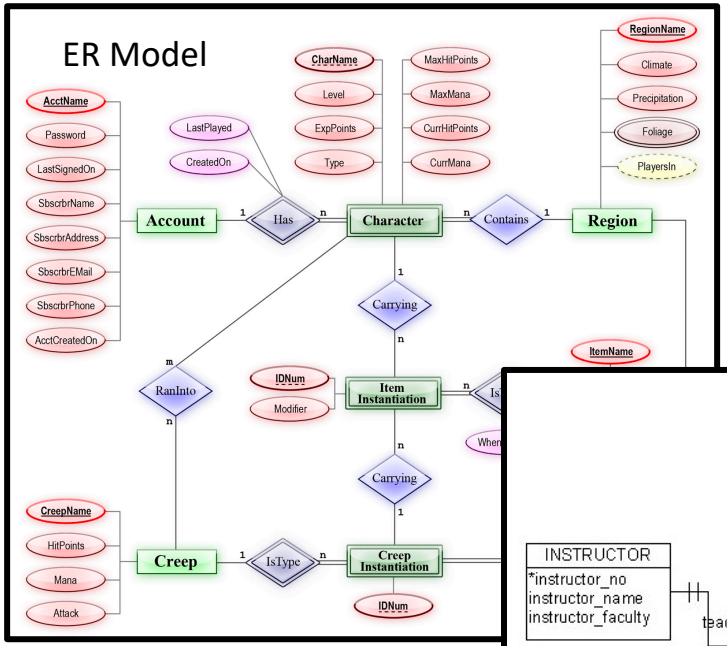
student

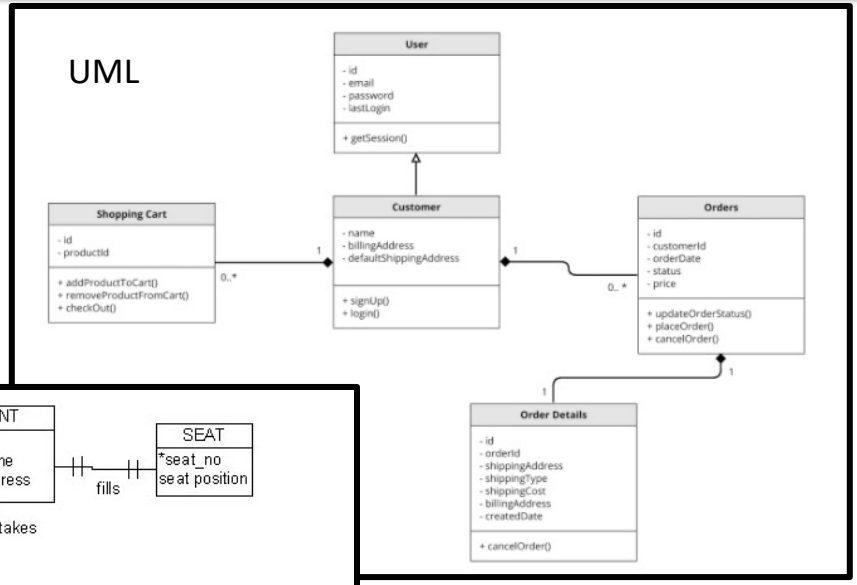
Representing Entity sets in ER Diagram

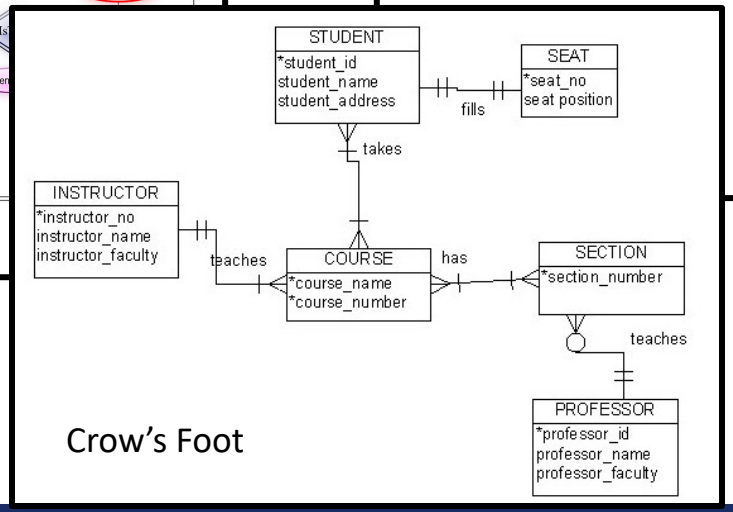
- Entity sets can be represented graphically as follows:
  - Rectangles represent entity sets.
  - Attributes listed inside entity rectangle
  - Underline indicates primary key attributes



Visual Notation – Many Notations



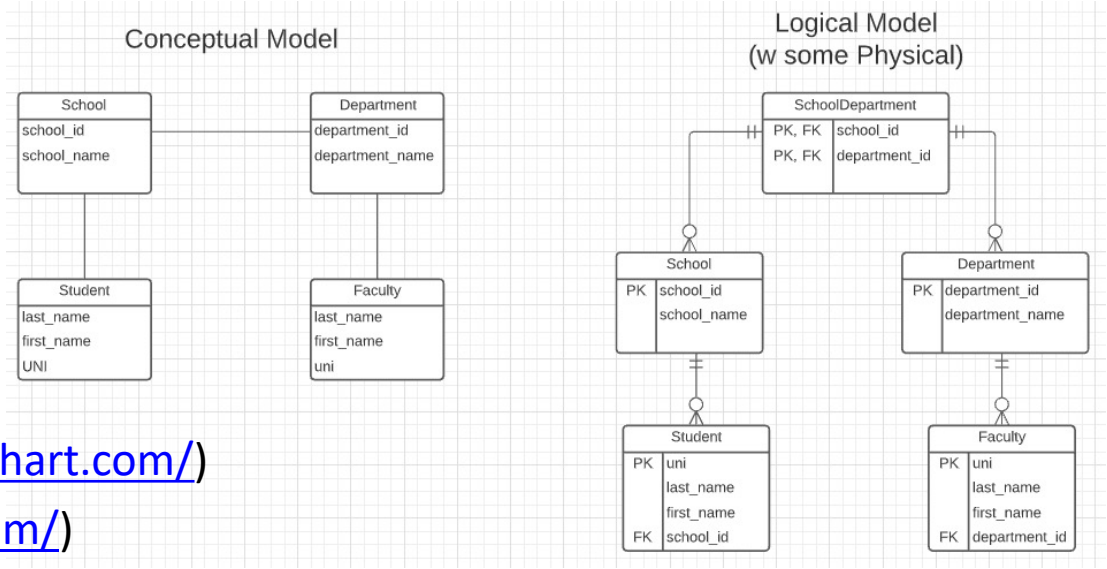




- "Other," i.e. PowerPoint is the most common modeling notation.
- It is easy to get "carried away."
- The trick is to do "just enough modeling."
- I mostly use Crow's Foot
  - It is "just enough"
  - But lacks some capabilities.
- The book uses ER notation.

First Iteration/Step for University Data Model

- This is the level of detail I want when I ask for:
  - Conceptual Model diagram.
  - Logical Model diagram.
- Some online tools with "free," constrained usage.
  - Lucidchart (https://www.lucidchart.com/)
  - Vertabelo (https://vertabelo.com/)
- The model has:
  - Four entity sets: *School, Department, Student, Faculty*
  - Three relationship sets: *Student-School, School-Department, Faculty-Department.*



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

DFF Comments: Just a reminder of a previous slide.

Relationship Sets

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

  Example:

  44553 (Peltier) — *student* entity; *advisor* — relationship set; 22222 (Einstein) — *instructor* entity

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

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

  where $(e_1, e_2, \ldots, e_n)$ is a relationship

  - Example:

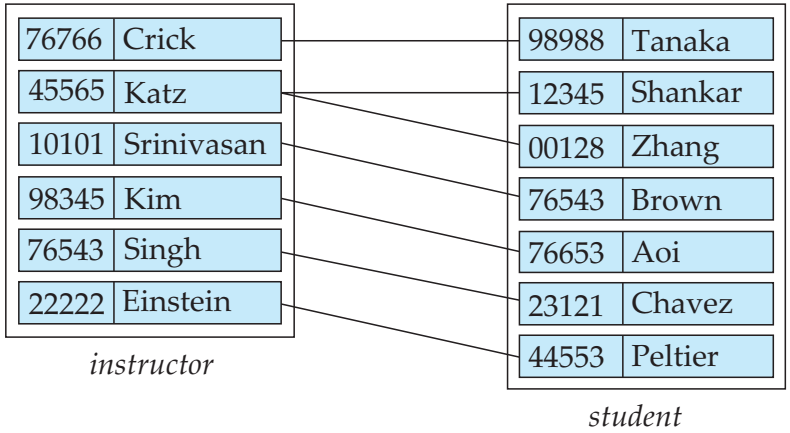
    $(44553,22222) \in advisor$

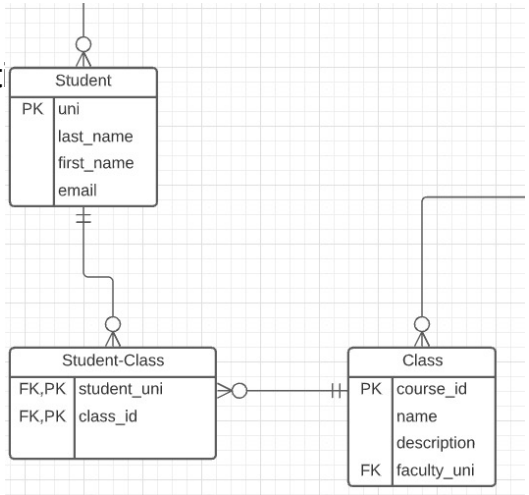
DFF Comments:

- Nobody thinks about relationships this way.
- There is no idea so simple that a DB professor cannot make it confusing, usually by using math.

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 ent





(dff9, W4111)

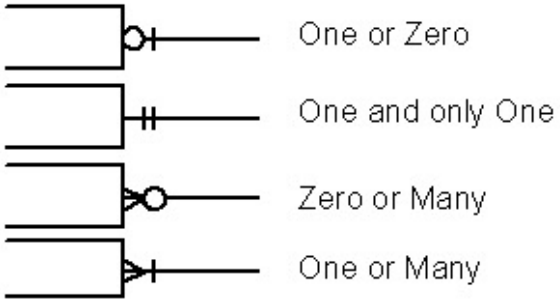
DFF Comments:

- Nobody draws the diagrams this way, but ...
- Sometimes thinking this way helps understand other ways to depict the concept.

Notation has Precise Meaning

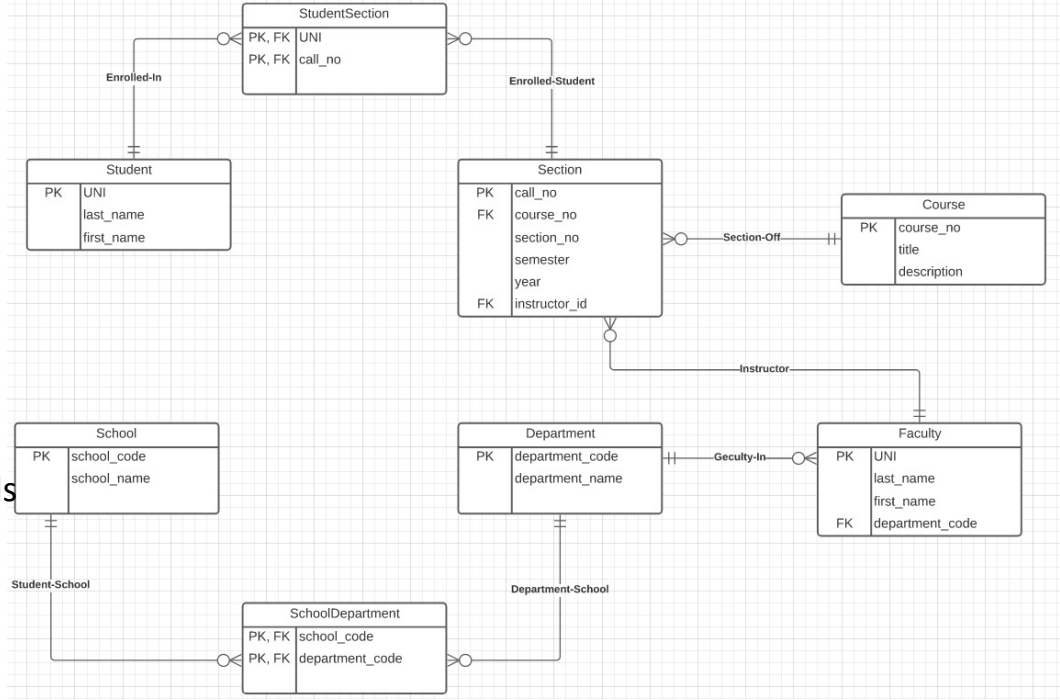
- Attribute annotations:
  - PK = Primary Key
  - FK = Foreign Key
- Line annotations:

Summary of Crow's Foot Notation



- We will learn over time and there are good tutorials (https://www.lucidchart.com/pages/er-diagrams) to help study and refresh.

- We will spend a lot of time discussing keys.
- We will start in a couple of slides.



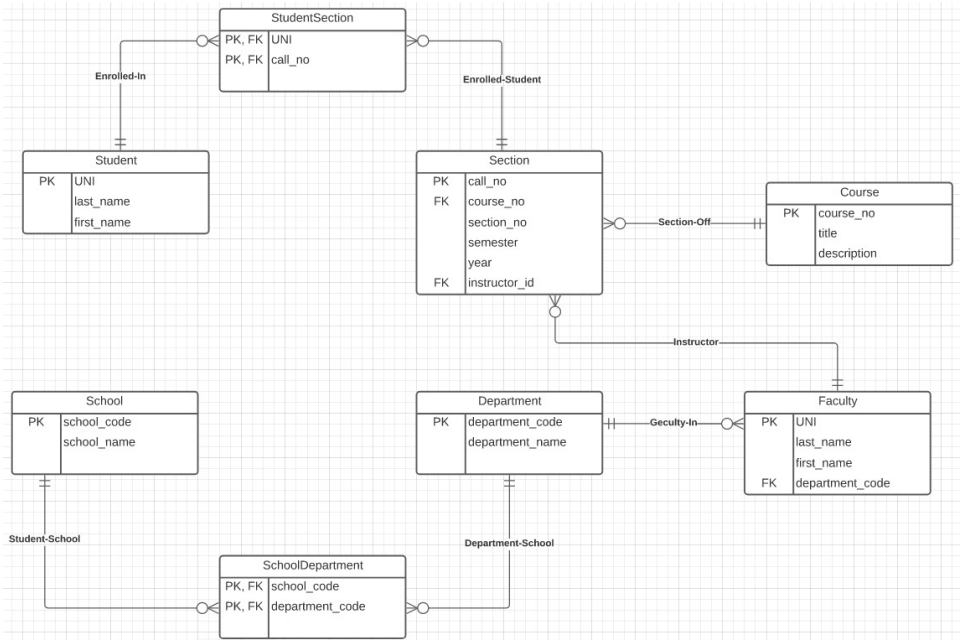
What Does this Mean? Let's Get Started

Primary Key means that the value occurs **at most once.**

School Code	School Name
CC	Columbia College
SEAS	Fu Foundation School of Engineering and Applied Science
GSAS	Graduate School of Arts and Sciences
GS	General Studies
… …	… …

Foreign Key means that if a value occurs in school_id for any row, there must be a row in School with that key.

UNI	Last name	First name	school_id
dff9	Ferguson	Donald	CC
js11	Smith	John	GS
jp9	Public	James	CC
bb101	Baggins	Bilbo	CC
… …	… …	… …	… …



The line notations mean:

- A student is related to EXACTLY ONE school.
- A School may be related to 0, 1 or many students.

ER model -- Database Modeling

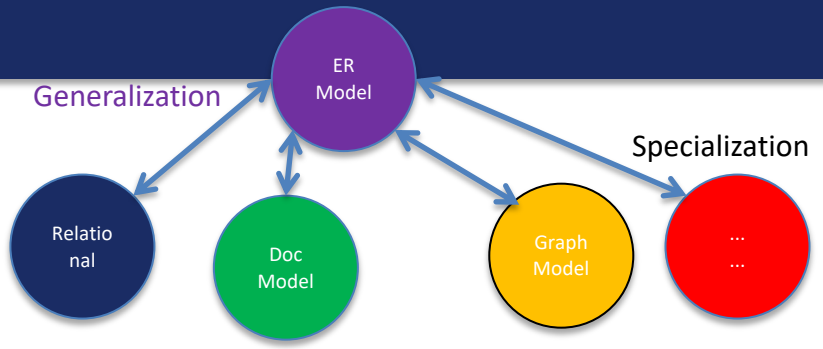
- The ER data mode was developed to facilitate database design by allowing specification of an **enterprise schema** that represents the overall logical structure of a database.
- The ER data model employs three basic concepts:
  - entity sets,
  - relationship sets,
  - attributes.
- The ER model also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

DFF Comments:

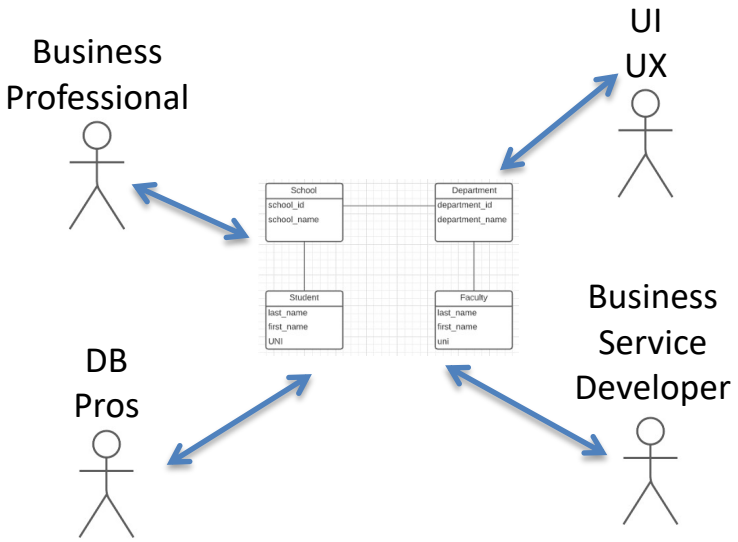
- The book and slides do not do a great job of motivating the ER model or ER diagrams.
- Why do people and teams think about or use the ER model and modeling?

ER Model and ER Modeling

- ER Model: Agility, Separation of Concerns
  - ER model is a generalization that most DB models implement in some form.
  - Using the ER model enables:
    - Thinking about and collaborating on design with getting bogged down in details.
    - Enable flexible choices about how to realize/Implement data.



- ER Diagrams: Communication, Quality, Precision
  - With a little experience, everyone can understand and ER diagram.
  - Easier to discuss and collaborate on application's data than showing SQL table definitions, JSON, ...
  - People think visually. That is why we have whiteboards. ER diagrams are precise and unambiguous.
  - Guides you to think about relationships, keys, ... And prevents "re-dos" later in the process. It is easier to fix a diagram than a database schema.



ER Modeling – Reasonably Good Summary

Advantages of ER Model

Conceptually it is very simple: ER model is very simple because if we know relationship between entities and attributes, then we can easily draw an ER diagram.

Better visual representation: ER model is a diagrammatic representation of any logical structure of database. By seeing ER diagram, we can easily understand relationship among entities and relationship.

Effective communication tool: It is an effective communication tool for database designer.

Highly integrated with relational model: ER model can be easily converted into relational model by simply converting ER model into tables.

Easy conversion to any data model: ER model can be easily converted into another data model like hierarchical data model, network data model and so on.

Disadvantages of ER Model

Limited constraints and specification

Loss of information content: Some information be lost or hidden in ER model

Limited relationship representation: ER model represents limited relationship as compared to another data models like relational model etc.

No representation of data manipulation: It is difficult to show data manipulation in ER model.

Popular for high level design: ER model is very popular for designing high level design

No industry standard for notation

https://pctechnicalpro.blogspot.com/2017/04/advantages-disadvantages-er-model-dbms.html

Note:

- If you get to use Google to help with take home exams, HW, etc.
- I get to use Google to help with slides.

Apply to –
Game of Thrones
University Data Model

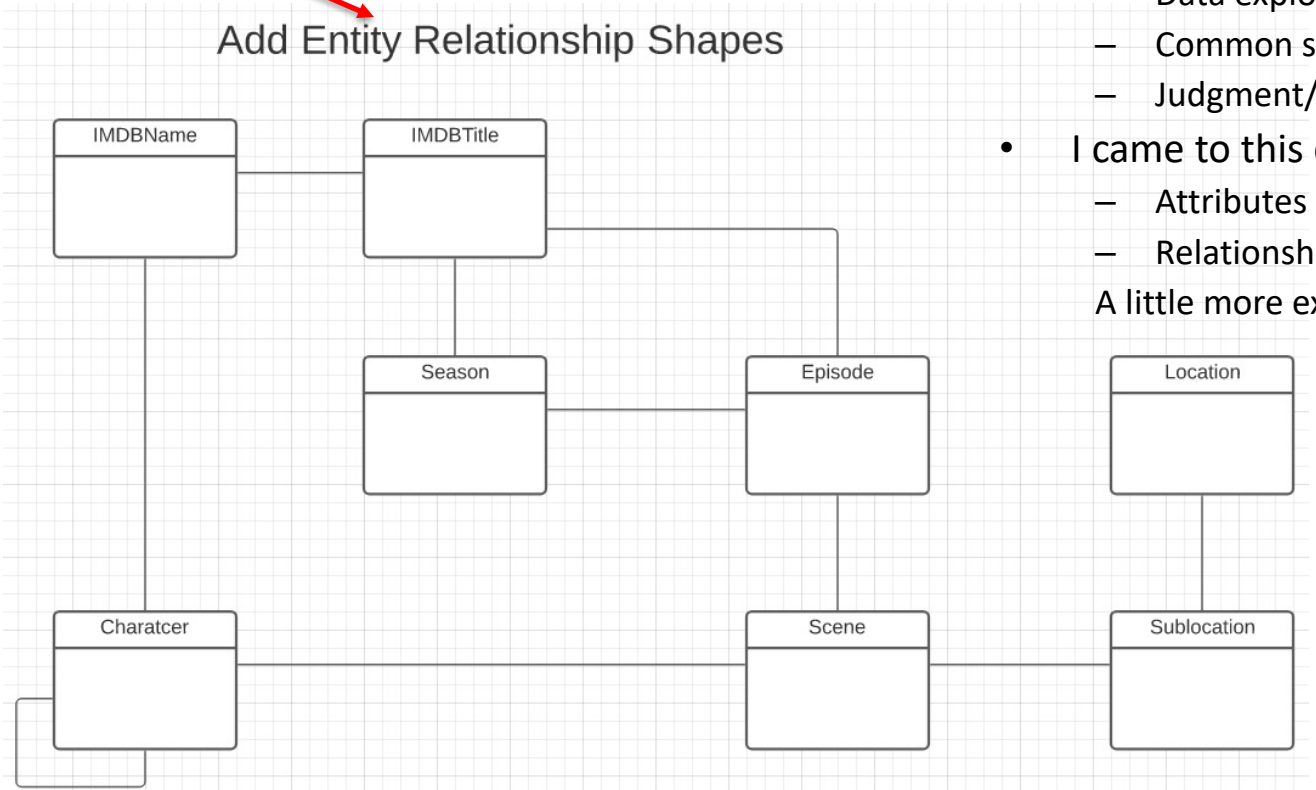
Game of Thrones

- Bottom-Up Data Mapping:
  - "Nouns" usually map to Entity/Entity Set.
  - Nouns inside other nouns often map to:
    - Attribute
    - Relationship
  - Verbs often map to relationships.
  - Adjectives usually map to properties.
- We will start with a subset of the information:
  - Game of Thrones:
    - Episodes
    - Characters
  - IMDB:
    - names_basics
    - title_basics

- Entities
  - Character
  - Season
  - Episode
  - Scene
  - Location, Sublocation
  - … …
- Relationships
  - Character – Scene
  - Character – Character (e.g. KilledBy)
  - Season – IMDB Title
  - Actor – IMDB Name
  - … …

Game of Thrones – Conceptual Model

Add shapes in Lucidchart ...

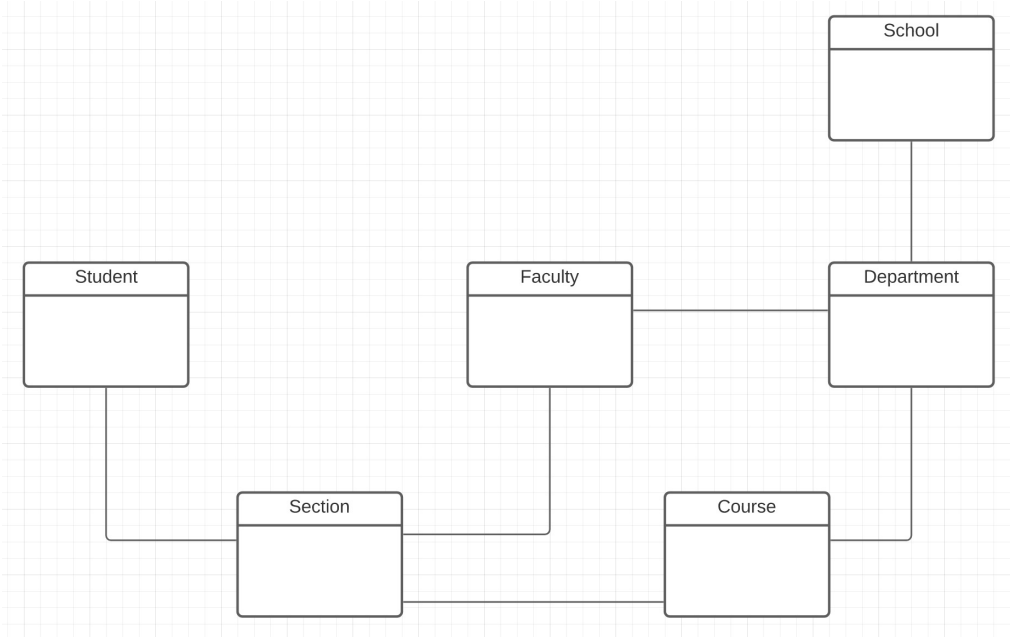


- With a little
  - Data exploration
  - Common sense
  - Judgment/experience
- I came to this conceptual model.
  - Attributes unspecified
  - Relationship required/cardinality unspecified.

  A little more exploration is needed.

Columbia University – Conceptual Model

- A good first pass at a subset of entity types is:
  - Student
  - Faculty
  - Class
  - Section
  - Department
  - School



- And there are some obvious relationships

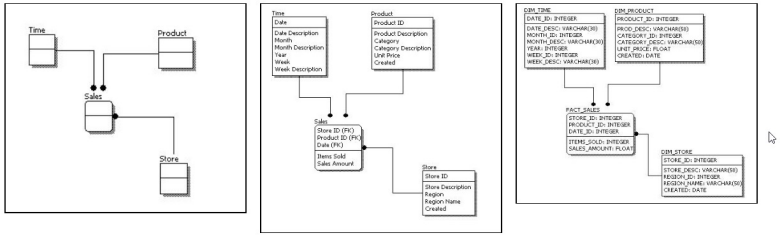
HW1: Conceptual to Logical

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

- The preceding material provided a conceptual model for a question on HW1
- HW 1 requests a *logical model.* We must add:
  - Attributes
  - Primary Keys
  - Foreign Keys
- HW 1 reinforces logical modeling concepts:
  - It is not trying to accurately model a university data model.
  - Do not worry about perfect semantics.
  - Just show you understand the concepts.
- To understand logical models, let's look at the Lahman's Baseball DB.

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

Conceptual Model Design | Logical Model Design | Physical Model Design

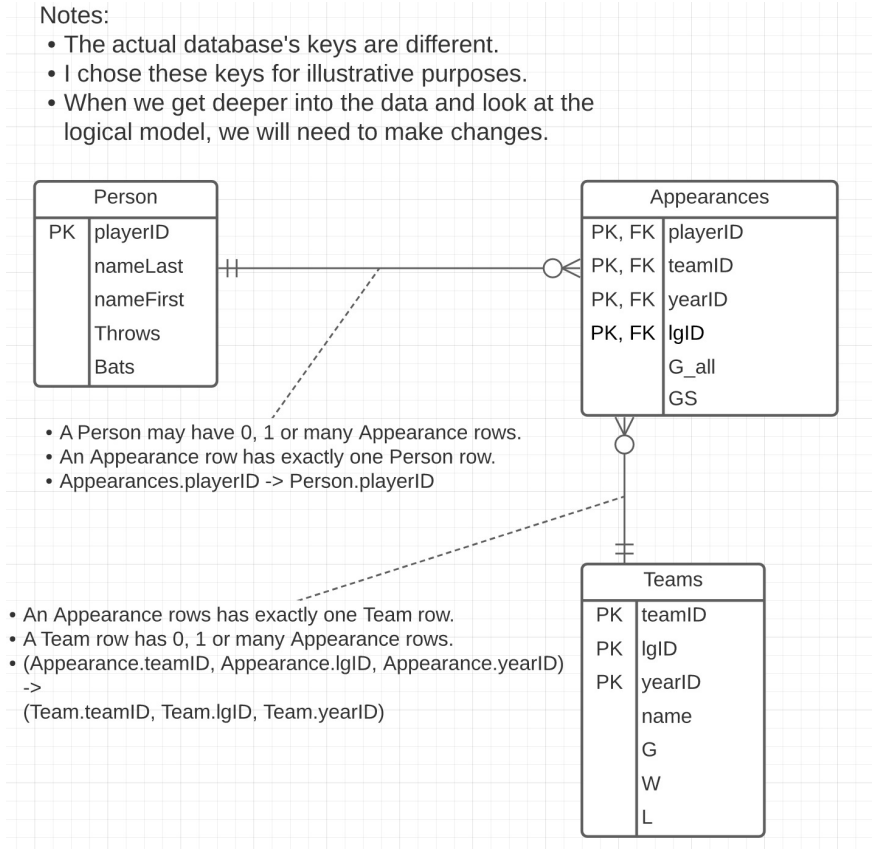


https://www.1keydata.com/datawarehousing/data-modeling-levels.html

Lahman's Baseball DB

- Some core entity types are:
  - Person
  - Team
  - Appearances
- The relationship is simple:
  - Person – Appearance
  - Appearance – Team
- We will only do a subset of the attributes.

Switch to DataGrip and run queries.



Some Interesting Queries

- A super key “test:”
  - **select playerID, teamID, lgID, yearID,** *count*(*) **as count from** appearances **group by playerID, teamID, yearID order by count desc limit** 10;
- A candidate key “test:”
  - **select playerID, teamID, yearID,** *count*(*) **as count from** appearances **group by playerID, teamID, yearID order by count desc limit** 10;
- A foreign key ”test:”
  - **select** * **from** appearances **where playerID not in** (**select playerID from** people)
  - **select** * **from** appearances **where not** *exists* (**select** * **from** teams **where** appearances.**teamID**=teams.**teamID and** appearances.**yearID**=teams.**yearID and** appearances.**lgID**=teams.**lgID**)

Relation Model and Algebra

Schema Definition

Notation

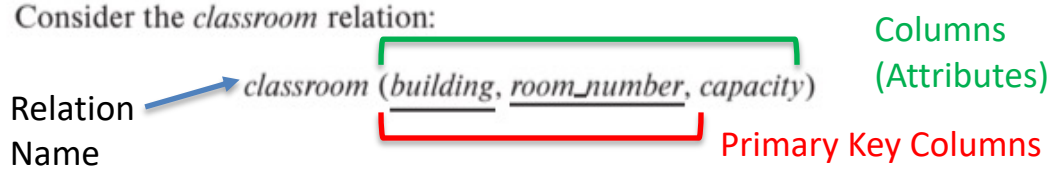
Classroom relation

building	room_number	capacity
Packard	101	500
Painter	100	125
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

classroom schema

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept_name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined.

Consider the *classroom* relation:



- The primary key is a *composite key.* Neither column is a key (unique) by itself.
- Keys are statements about all possible, valid tuples and not just the ones in the relation.
  - Capacity is unique in this specific data, but clearly not unique for all possible data.
  - In this domain, there cannot be two classrooms with the same building and room number.
- Relation schema:
  - Underline indicates a primary key column. There is no standard way to indicate other types of key.
  - We will use **bold** to indicate foreign keys.
  - You will sometimes see things like *classroom(building:string, room_number:number, capacity:number)*

Some More Relational Algebra and The Dreaded RelaX Calculator

Join and HW 1

Homework 1 asks for a statement equivalent to an anti-join.

- "An anti-join is a form of join with reverse logic. Instead of returning rows when there is a match (according to the join predicate) between the left and right side, an anti-join returns those rows from the left side of the predicate for which there is no match on the right."
  - X = σ building='Watson' ∨ building='Taylor' (classroom)
  - Y = (department ▷ X)
  - Y
- Find an alternate expression to (2) that computes the correct answer given X. Display the execution of your query below."

- I should probably team you about JOIN.

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 last lecture
  - ~~select: σ~~
  - ~~project: Π~~
  - union: $\cup$
  - ~~set difference: $-$~~
  - Cartesian product: x
  - ~~rename: ρ~~
  - ~~intersect~~
  - ~~$\leftarrow$ assignment~~

Another lecture.

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.

Why highlight composition before completing coverage of the operations?

- The next core relational operator most people consider is JOIN.
- The definition of JOIN relies on operation composition.

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

- This only sort of makes sense. The result is:
  - Every possible combination of the form
  - (instructor, teaches)
  - Even if the instructor is NOT the instructor in the teaches row.

- Examining in MySQL makes a little clearer.
  - Let's look in the lecture examples notebook.
  - Confusingly, in SQL Cartesian Product is **JOIN**

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
…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…

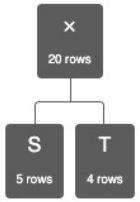
Simpler Example

T

T.b	T.d
'a'	100
'd'	200
'f'	400
'g'	120

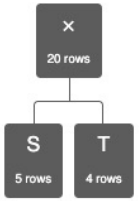
S

S.b	S.d
'a'	100
'b'	300
'c'	400
'd'	200
'e'	150



S × T

S.b	S.d	T.b	T.d
'a'	100	'a'	100
'a'	100	'd'	200
'a'	100	'f'	400
'a'	100	'g'	120
'b'	300	'a'	100
'b'	300	'd'	200
'b'	300	'f'	400
'b'	300	'g'	120
'c'	400	'a'	100
'c'	400	'd'	200



S × T

S.b	S.d	T.b	T.d
'c'	400	'f'	400
'c'	400	'g'	120
'd'	200	'a'	100
'd'	200	'd'	200
'd'	200	'f'	400
'd'	200	'g'	120
'e'	150	'a'	100
'e'	150	'd'	200
'e'	150	'f'	400
'e'	150	'g'	120

- Assume we have two tables
  - S has two columns, 5 rows.
  - T has two columns, 4 rows.
- S x T has
  - 4 columns.
  - 20 rows.
- Cartesian product does not come up a lot in applications.
- There are cases in optimization in which:
  - You want to generate all possible combinations.
  - Score, rate, rank etc. to determine best choices.

Join Operation

- The Cartesian-Product

  instructor X *teaches*

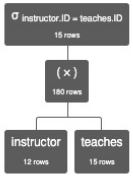
  associates every tuple of instructor with every tuple of teaches.
  - Most of the resulting rows have information about instructors who did NOT teach a particular course.
- To get only those tuples of "*instructor* X *teaches* " that pertain to instructors and the courses that they taught, we write:

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

A fundamental definition:

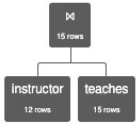
- $\sigma_{instructor.ID=teaches.ID}$ (instructor × teaches) = instructor ⋈ teaches
- ⋈ is the JOIN operations.

JOIN Definition



σ instructor.ID = teaches.ID (instructor × teaches)

instructor.ID	instructor.name	instructor.dept_name	instructor.salary	teaches.ID	teaches.course_id	teaches.sec_id	teaches.semester	teaches.year
10101	'Srinivasan'	'Comp. Sci.'	65000	10101	'CS-101'	1	'Fall'	2009
10101	'Srinivasan'	'Comp. Sci.'	65000	10101	'CS-315'	1	'Spring'	2010
10101	'Srinivasan'	'Comp. Sci.'	65000	10101	'CS-347'	1	'Fall'	2009
12121	'Wu'	'Finance'	90000	12121	'FIN-201'	1	'Spring'	2010
15151	'Mozart'	'Music'	40000	15151	'MU-199'	1	'Spring'	2010
22222	'Einstein'	'Physics'	95000	22222	'PHY-101'	1	'Fall'	2009
32343	'El Said'	'History'	60000	32343	'HIS-351'	1	'Spring'	2010
45565	'Katz'	'Comp. Sci.'	75000	45565	'CS-101'	1	'Spring'	2010
45565	'Katz'	'Comp. Sci.'	75000	45565	'CS-319'	1	'Spring'	2010
76766	'Crick'	'Biology'	72000	76766	'BIO-101'	1	'Summer'	2009



instructor ⋈ teaches

instructor.ID	instructor.name	instructor.dept_name	instructor.salary	teaches.course_id	teaches.sec_id	teaches.semester	teaches.year
10101	'Srinivasan'	'Comp. Sci.'	65000	'CS-101'	1	'Fall'	2009
10101	'Srinivasan'	'Comp. Sci.'	65000	'CS-315'	1	'Spring'	2010
10101	'Srinivasan'	'Comp. Sci.'	65000	'CS-347'	1	'Fall'	2009
12121	'Wu'	'Finance'	90000	'FIN-201'	1	'Spring'	2010
15151	'Mozart'	'Music'	40000	'MU-199'	1	'Spring'	2010
22222	'Einstein'	'Physics'	95000	'PHY-101'	1	'Fall'	2009
32343	'El Said'	'History'	60000	'HIS-351'	1	'Spring'	2010
45565	'Katz'	'Comp. Sci.'	75000	'CS-101'	1	'Spring'	2010
45565	'Katz'	'Comp. Sci.'	75000	'CS-319'	1	'Spring'	2010
76766	'Crick'	'Biology'	72000	'BIO-101'	1	'Summer'	2009

$\sigma_{\text{instructor.ID=teaches.ID}}$ (instructor × teaches)

instructor $\bowtie$ teaches

instructor $\bowtie_{\text{instructor.ID>teaches.ID}}$ teaches

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

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

(a,b,c,d)

(x,y)

Project(a,

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

Note: The preloaded dataset on the RelaX calculator is different from the most recent data referenced in the book. It is from a previous edition.

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

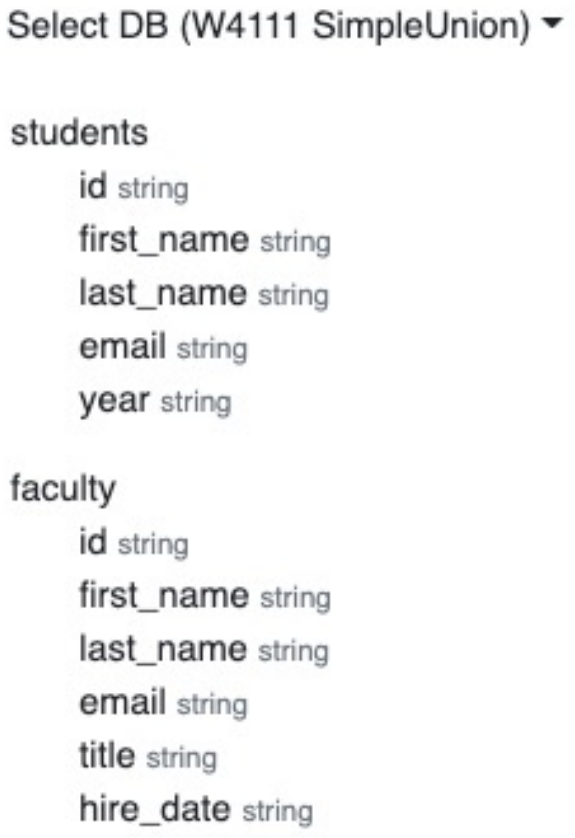
Set Difference Operation

- 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

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

course_id
CS-347
PHY-101

Same “arity”



- Same “arity”
  - Same number of columns.
  - Compatible types.
  - The i-th column in each table is from a compatible domain.
    - Student 5th column is “year.”
    - Faculty 5th column is “title”
    - Both are strings but combing them does not make sense.
- You can shape two incompatible tables using *project operations.* For example
  - π first_name,last_name,email (students)
    ∩
    π first_name,last_name,email (faculty)
  - π last_name,email, title←'Student' (students)
    ∪
    π last_name,email, title (faculty)

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 ← 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.

What are all those other Symbols?

- τ order by
- γ group by
- ¬ negation
- ÷ set division
- ⋈ natural join, theta-join
- ⟕ left outer join
- ⟖ right outer join
- ⟗ full outer join
- ⋉ left semi join
- ⋊ right semi join
- ▷ anti-join

- Some of these are pretty obscure
  - Division
  - Anti-Join
  - Left semi-join
  - Right semi-join
- Most SQL engines do not support them.
  - You can implement them using combinations of JOIN, SELECT, WHERE, ...
  - But, I cannot every remember using them in applications I have developed.
- Outer JOIN is very useful, but less common. We will cover.
- There are also some “patterns” or “terms”
  - Equijoin
  - Non-equi join
  - Natural join
  - Theta join
  - ... ...
- I may ask you to define these terms on some exams because they may be common internships/job interview questions.

HW Question – Join and HW 1

Homework 1 asks for a statement equivalent to an anti-join.

- "An anti-join is a form of join with reverse logic. Instead of returning rows when there is a match (according to the join predicate) between the left and right side, an anti-join returns those rows from the left side of the predicate for which there is no match on the right."
  - X = σ building='Watson' ∨ building='Taylor' (classroom)
  - Y = (department ▷ X)
  - Y
- Find an alternate expression to (2) that computes the correct answer given X. Display the execution of your query below."

- I should probably team you about JOIN.

Some Terms

- "A NATURAL JOIN is **a JOIN operation that creates an implicit join clause for you based on the common columns in the two tables being joined**. Common columns are columns that have the same name in both tables." (https://docs.oracle.com/javadb/10.8.3.0/ref/rrefsqljnaturaljoin.html)

- ⋈ ➔ Natural Join in relational algebra.

- So, think about it …
  - I showed you how to produce all possible pairs.
  - I showed you how to produce all naturally matching pairs.
  - Some simple set operations gives the anti-join.

SQL

Core Concepts and Operations

You can think of SQL being
an extended, usable, useful version
of the relational model.

Chapter 3: Introduction to SQL

Outline

- Overview of The SQL Query Language
- SQL Data Definition
- Basic Query Structure of SQL Queries
- ~~Additional Basic Operations~~
- ~~Set Operations~~
- Null Values
- ~~Aggregate Functions~~
- ~~Nested Subqueries~~
- Modification of the Database

Will cover in next lecture and/or tutorial.

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.

SQL Parts

- DML -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- integrity – the DDL includes commands for specifying integrity constraints.
- View definition -- The DDL includes commands for defining views.
- Transaction control –includes commands for specifying the beginning and ending of transactions.
- Embedded SQL and dynamic SQL -- define how SQL statements can be embedded within general-purpose programming languages.
- Authorization – includes commands for specifying access rights to relations and views.

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length *n.*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n.*
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point. (ex., **numeric**(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.
- More are covered in Chapter 4.

Create Table Construct

- An SQL relation is defined using the **create table** command:

  create table *r*

  (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - *r* is the name of the relation
  - each A_i is an attribute name in the schema of relation *r*
  - D_i is the data type of values in the domain of attribute A_i

- Example:

  create table *instructor* (
  ID **char**(5),
  name **varchar**(20),
  dept_name **varchar**(20),
  salary **numeric**(8,2))

Integrity Constraints in Create Table

- Types of integrity constraints
  - **primary key** (A_1, ..., A_n)
  - **foreign key** (A_m, ..., A_n) **references** *r*
  - **not null**
- SQL prevents any update to the database that violates an integrity constraint.
- Example:

```
create table instructor (
        ID          char(5),
        name        varchar(20) not null,
        dept_name   varchar(20),
        salary      numeric(8,2),
        primary key (ID),
        foreign key (dept_name) references department);
```

And a Few More Relation Definitions

- **create table** *student* (
  ID **varchar**(5),
  name **varchar**(20) not null,
  dept_name **varchar**(20),
  tot_cred **numeric**(3,0),
  primary key *(ID),*
  foreign key *(dept_name*) **references** *department*);

- **create table** *takes* (
  ID **varchar**(5),
  course_id **varchar**(8),
  sec_id **varchar**(8),
  semester **varchar**(6),
  year **numeric**(4,0),
  grade **varchar**(2),
  primary key *(ID, course_id, sec_id, semester, year)* ,
  foreign key (*ID*) **references** *student,*
  foreign key (*course_id, sec_id, semester, year*) **references** *section*);

And more still

- **create table** *course* (
  course_id **varchar**(8),
  title **varchar(**50),
  dept_name **varchar**(20),
  credits **numeric**(2,0),
  primary key *(course_id),*
  foreign key *(dept_name*) **references** *department*);

Updates to tables

- **Insert**
  - **insert into** *instructor* **values** ('10211', 'Smith', 'Biology', 66000);
- **Delete**
  - Remove all tuples from the *student* relation
    - **delete from** *student*
- **Drop Table**
  - **drop table** *r*
- **Alter**
  - **alter table** *r* **add** *A D*
    - where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
    - All exiting tuples in the relation are assigned *null* as the value for the new attribute.
  - **alter table** *r* **drop** *A*
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases.

Basic Query Structure

- A typical SQL query has the form:

  select A_1, A_2, ..., A_n
  from r_1, r_2, ..., r_m
  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.

Note:

- The SELECT ... FROM ... WHERE ... Combines two relational operators, σ and $\prod$.
- Actually, it also combines other operators, e.g. x

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* ≡ *NAME* ≡ *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 "s*alary/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* ≡ *instructor T*

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 + **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.

Note:

- NULL is an extremely important concept.
- You will find it hard to understand for a while.

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* < **null** or **null** <> **null** or **null** = **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** : *(true* **and** *unknown) = unknown,*
    (false **and** *unknown) = false,*
    (unknown **and** *unknown) = unknown*
  - **or:** (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown) = unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

Deletion

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

  delete from *instructor*
  where *dept_name*= 'Finance';

- *Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.*

  delete from *instructor*
  where *dept name* **in** (**select** *dept name*
  from *department*
  where *building* = 'Watson');

Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

  - Problem: as we delete tuples from *instructor*, the average salary changes
  - Solution used in SQL:
    1. First, compute **avg** (salary) and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Insertion

- Add a new tuple to *course*

  insert into *course*
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- or equivalently

  insert into *course* (*course_id*, *title*, *dept_name*, *credits*)
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- Add a new tuple to *student* with *tot_creds* set to null

  insert into *student*
  values ('3003', 'Green', 'Finance', *null*);

Insertion (Cont.)

- Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

```
insert into instructor
    select ID, name, dept_name, 18000
    from   student
    where  dept_name = 'Music' and total_cred > 144;
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

Otherwise queries like

```
insert into table1 select * from table1
```

would cause problem

Updates

- Give a 5% salary raise to all instructors

```
update instructor
  set salary = salary * 1.05
```

- Give a 5% salary raise to those instructors who earn less than 70000

```
update instructor
  set salary = salary * 1.05
 where salary < 70000;
```

- Give a 5% salary raise to instructors whose salary is less than average

```
update instructor
set salary = salary * 1.05
where salary < (select avg (salary)
                from instructor);
```

Updates (Cont.)

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

```
update instructor
   set salary = salary * 1.03
   where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

  - The order is important
  - Can be done better using the **case** statement (next slide)

Aggregate Functions

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

Note: Some database implementations have additional aggregate functions.

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*;

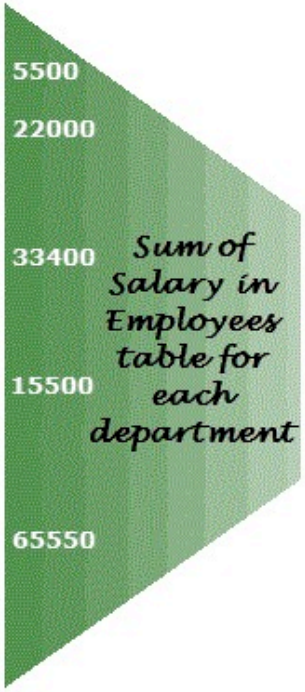
ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music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

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Another View

Employees

DEPARTMENT_ID	SALARY
10	5500
20	15000
20	7000
30	12000
30	5100
30	4900
30	5800
30	5600
40	7500
40	8000
50	9000
50	8500
50	9500
50	8500
50	10500
50	10000
50	9500



DEPARTMENT_ID	SUM(SALARY)
10	5500
20	22000
30	33400
40	15500
50	65550

- GROUP BY column list
  - Forms partitions containing multiple rows.
  - All rows in a partition have the same values for the GROUP BY columns.
- The aggregate functions
  - Merge the non-group by attributes, which may differ from row to row.
  - Into a single value for each attribute.
- The result is one row per distinct set of GROUP BY values.
- There may be multiple non-GROUP BY COLUMNS, each with its own aggregate function.
- You can use HAVING in place of WHERE on the GROUP BY result.

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*;

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