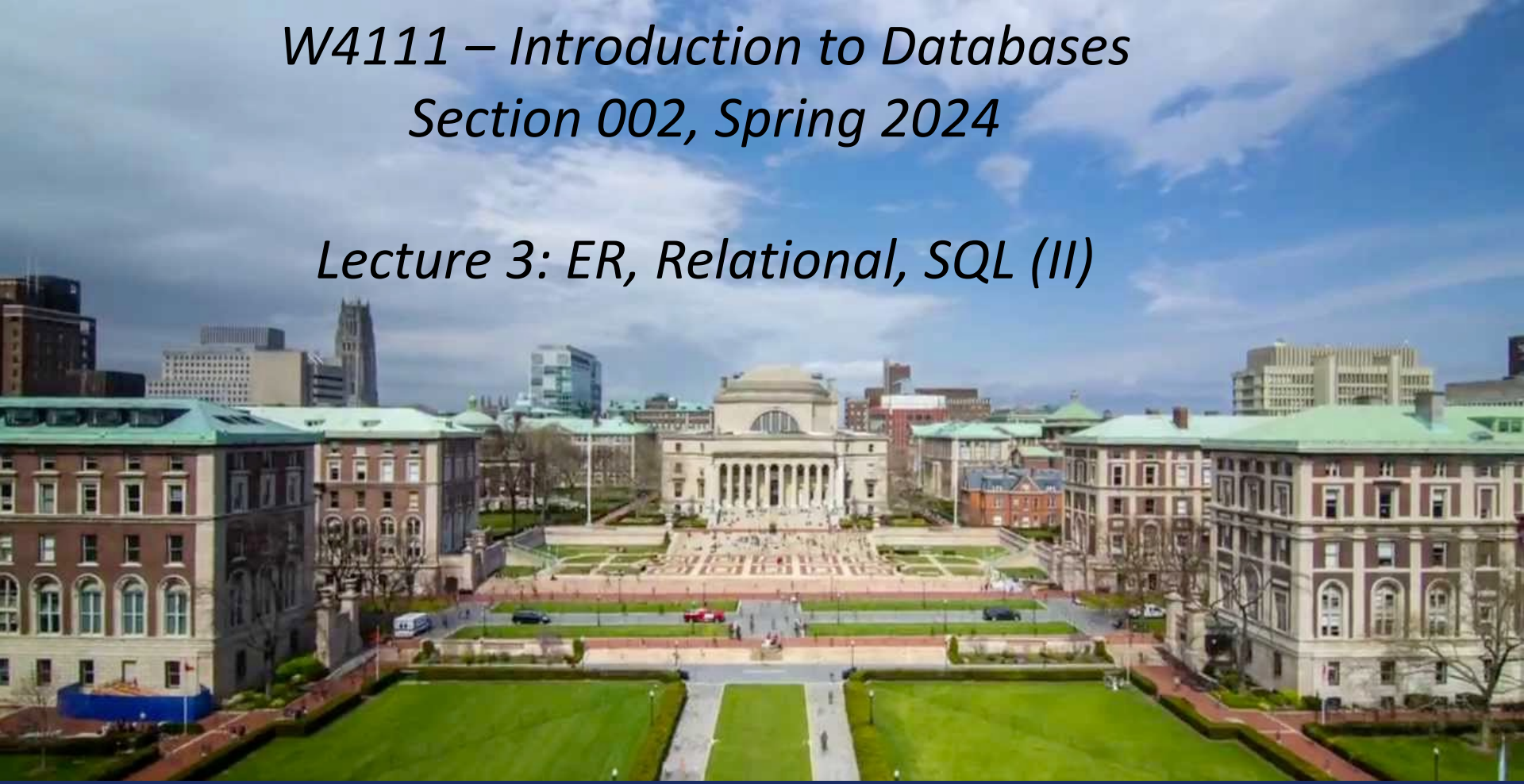
W4111 – Introduction to Databases
Section 002, Spring 2024

Lecture 3: ER, Relational, SQL (II)



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

W4111 – Introduction to Databases
Section 002, Spring 2024

Lecture 3: ER, Relational, SQL (II)

We will start in a couple of minutes.

Contents

Contents

- Introduction: Status updates
- ER (Diagram) Modeling – More Complex Scenarios
- The Relational Model and Algebra Continued (Chapter 2)
- SQL Continued (Chapter 3, 4, 5)
- Project Examples:
  - Web Applications and REST
  - Data Engineering and Visualization

Introduction

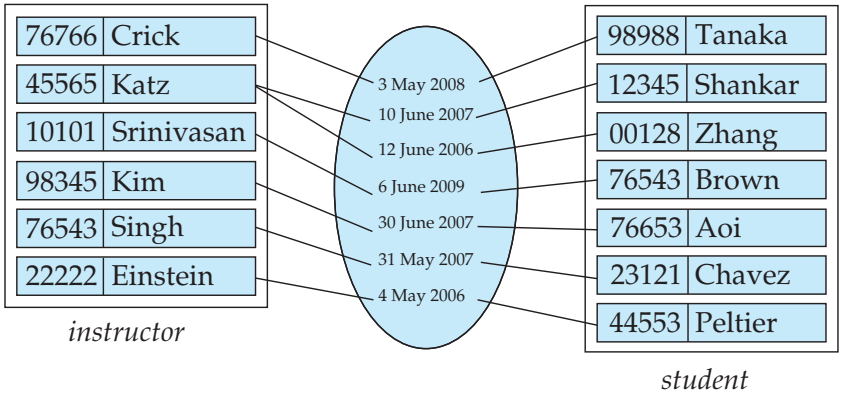
Status Updates

Database design, Entity-Relationship Model (Part 2)

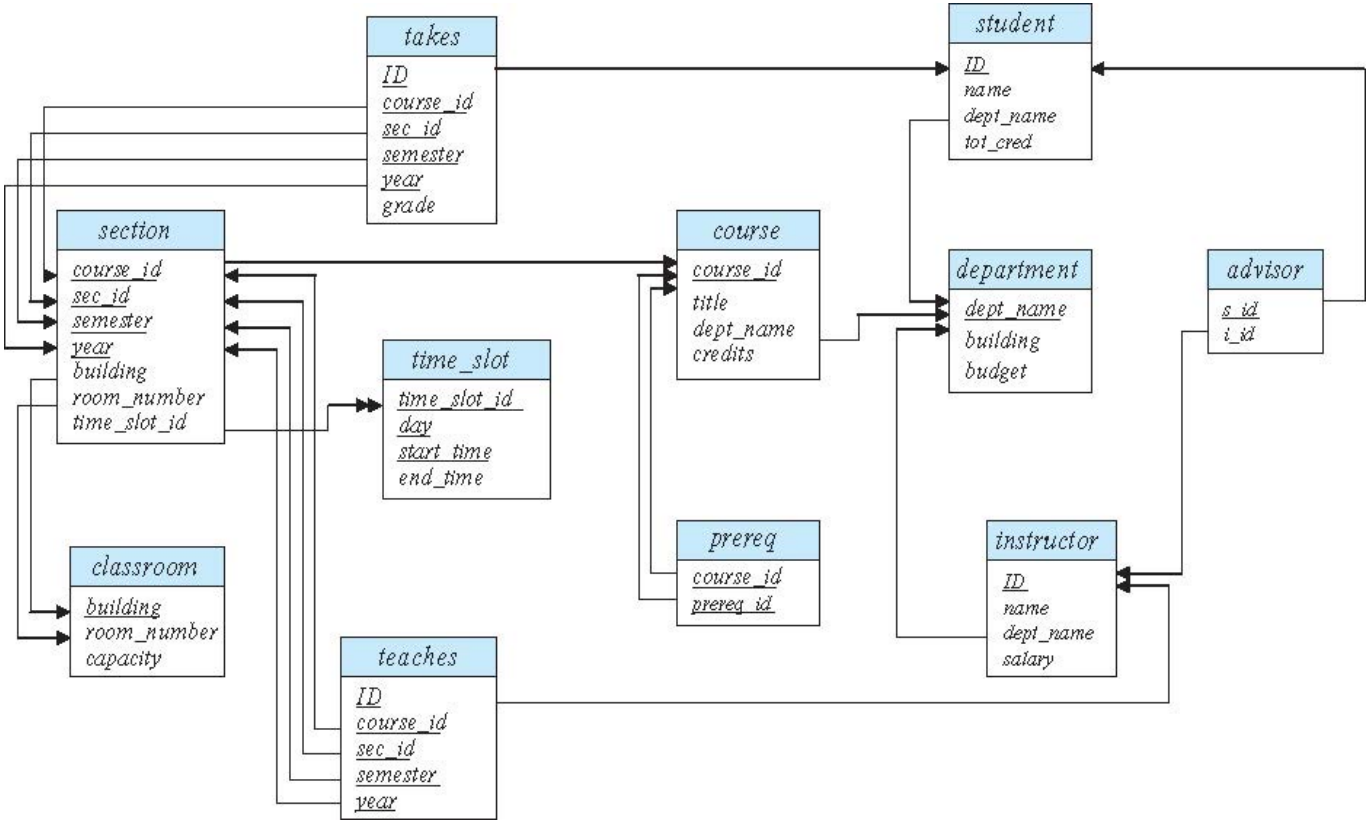
More Complex Relationships

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

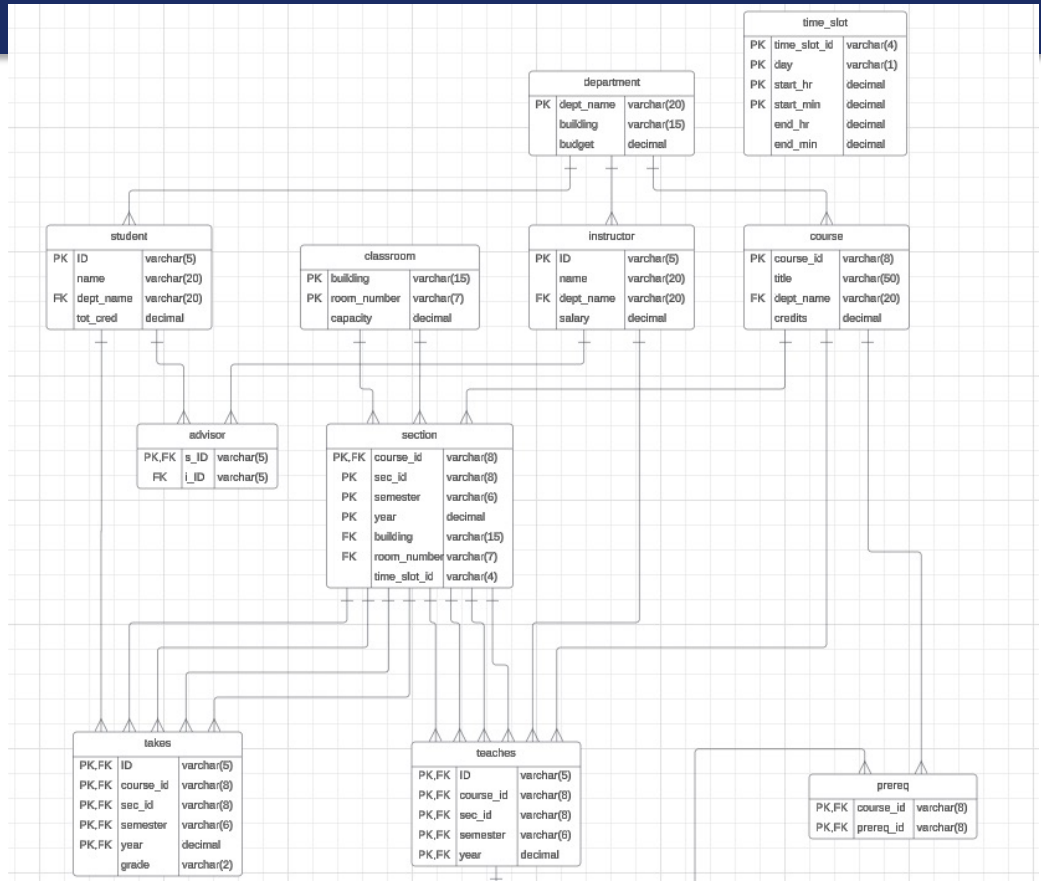


Schema Diagram for University Database



Some Observations

- Reverse engineering seems to have made some mistakes.
- But, notice that sometimes:
  - Relationship are foreign keys, e.g. Section-Classroom
  - Are something else, e.g. Advisor
- This something else is called and Associative Entity.
- SQL requires and Associative Entity if
  - The relationship is many-to-many.
  - There are properties on the relationships.



Associative Entity

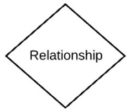
- The ER model represents “associations/relationships” as
  - First class “things”
  - That are different from “entities.”
- The SQL model does not have “relationships” or associations as first-class types. You have
  - Tables
  - Columns
  - Keys
  - Constraints
  - … …
- You can implement some “relationships” using foreign keys. Others require something more complex – an *associative entity.*

Associative entity

Associative entities relate the instances of several entity types. They also contain attributes specific to the relationship between those entity instances.

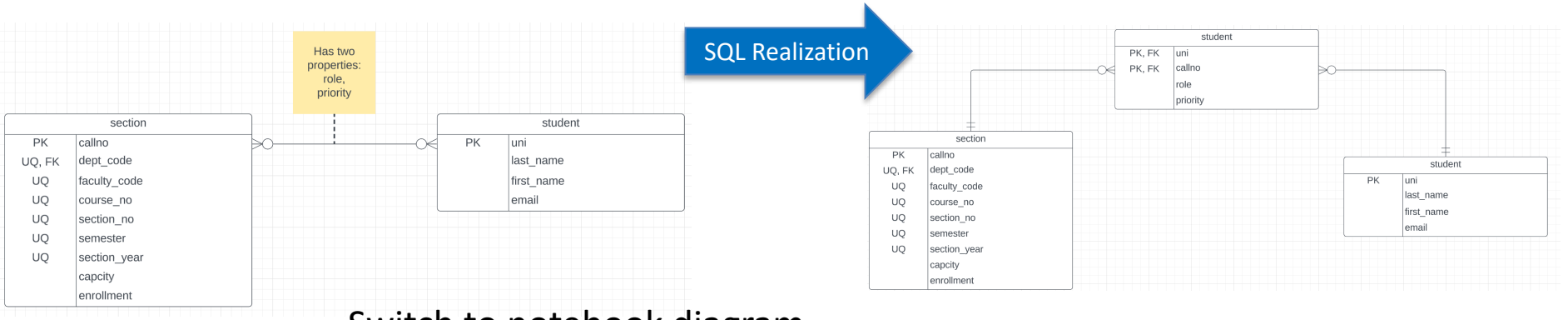
ERD relationship symbols

Within entity-relationship diagrams, relationships are used to document the interaction between two entities. Relationships are usually verbs such as assign, associate, or track and provide useful information that could not be discerned with just the entity types.

Relationship Symbol	Name	Description
	Relationship	Relationships are associations between or among entities.
	Weak relationship	Weak Relationships are connections between a weak entity and its owner.

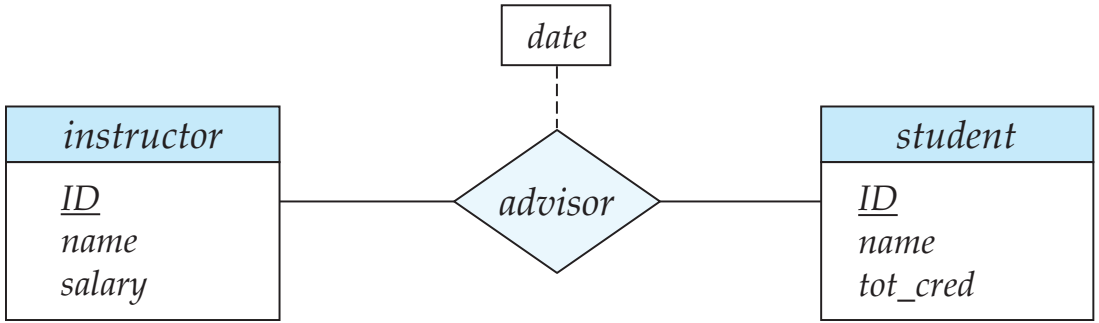
Associative Entity

- "An associative entity is a term used in relational and entity–relationship theory. A relational database requires the implementation of a base relation (or base table) to resolve many-to-many relationships. A base relation representing this kind of entity is called, informally, an associative table." (https://en.wikipedia.org/wiki/Associative_entity)
- Consider *Students – Sections:*
  - This is many-to-many. There is no way to implement in SQL. You see this in the *Advises* table in the sample database.
  - The "relationship/association" has properties that are not attributes of the connected entities.



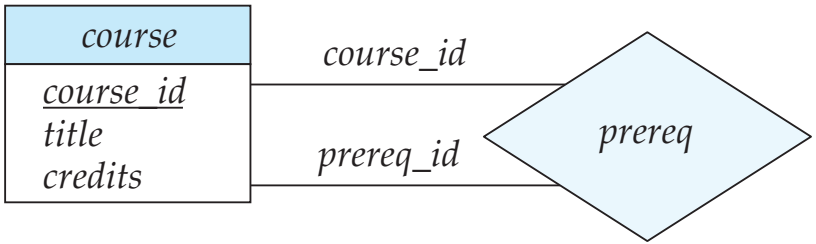
Switch to notebook diagram.

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**.

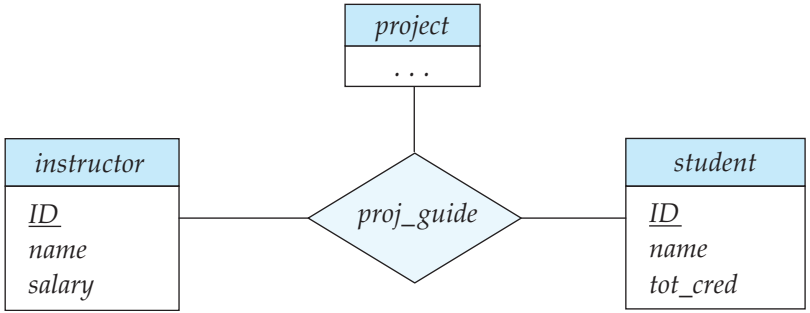


Degree of a Relationship Set

- 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



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

Apply to –

Game of Thrones

https://github.com/jeffreylancaster/game-of-thrones

~/Dropbox/000/000-Data/GoT

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

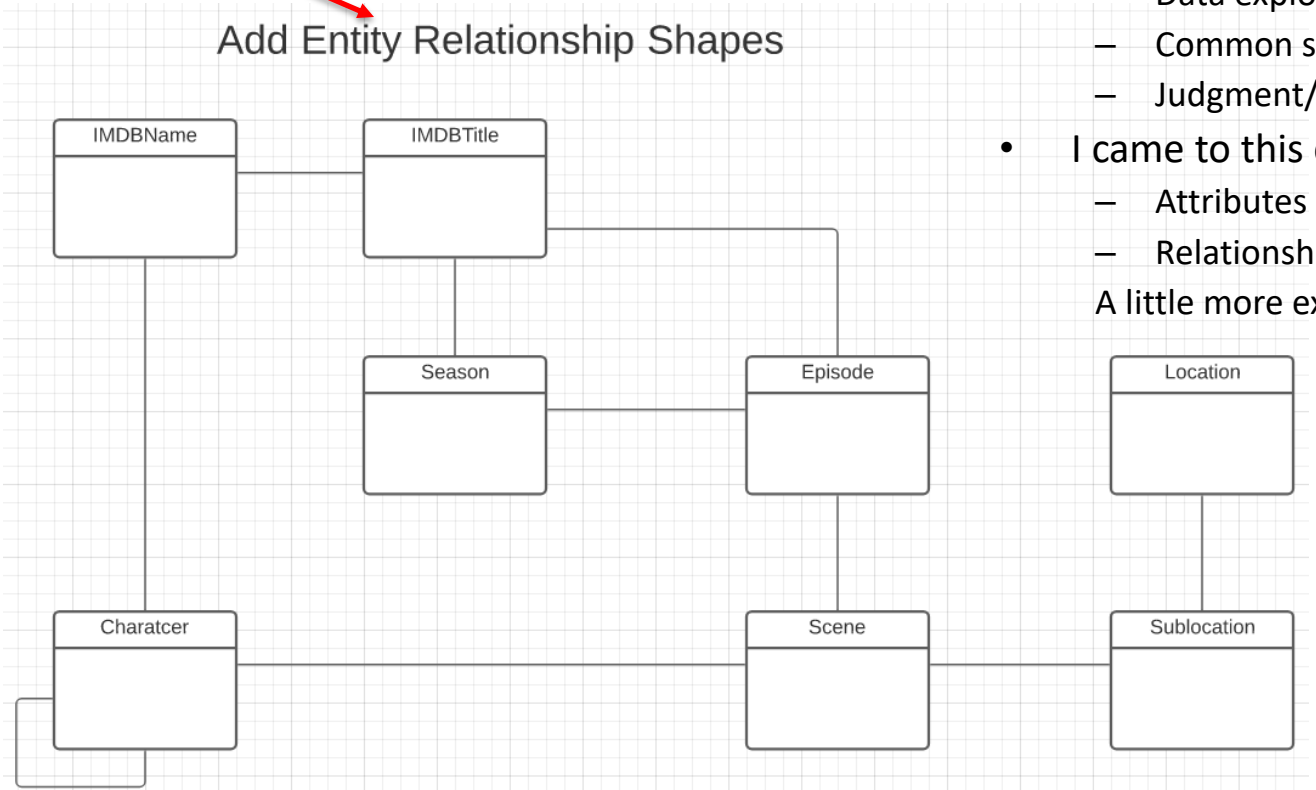
Game of Thrones

- Bottom-Up Data Mapping:
  - –
  - –
  - –
  - –
- We
  info
  - –
  - – IM
    - names_basics
    - title_basics
- Entities Sets
  - – Character – IMDB Name
  - – … …

1. IMDB: https://developer.imdb.com/non-commercial-datasets/
   - Do not download.
   - Despite being “tiny” compared to the reall world.
   - The datasets are too big for most laptops.
2. Game of Thrones: https://github.com/jeffreylancaster/game-of-thrones

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.

Game of Thrones – Conceptual Model

- Show JSON files and explain.
- Draw diagrams in Lucidchart for
  - GoT: character, season, episode, scene and the relationships.
  - IMDB: namebasic, titles, knownFor corrected, titlePrincipals, etc.
- Loading this data into relational is a bear.
  - GoT:
    - The data is nested document format.
    - Relational is tables.
    - It is doable but requires code and pipelines.
    - I will give you the uncleaned up relational for HW2
  - IMDB:
    - Datasets are very large for a personal/laptop.
    - Many non-atomic columns.
    - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/IMDB_GOT_Processing

Relation Model and Algebra

Assignment
Rename

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.

Other Operators
Fun with JOIN

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.

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.

Equivalent Queries

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

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

- Query 2

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

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

Equivalent Queries

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

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

- Query 2

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

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

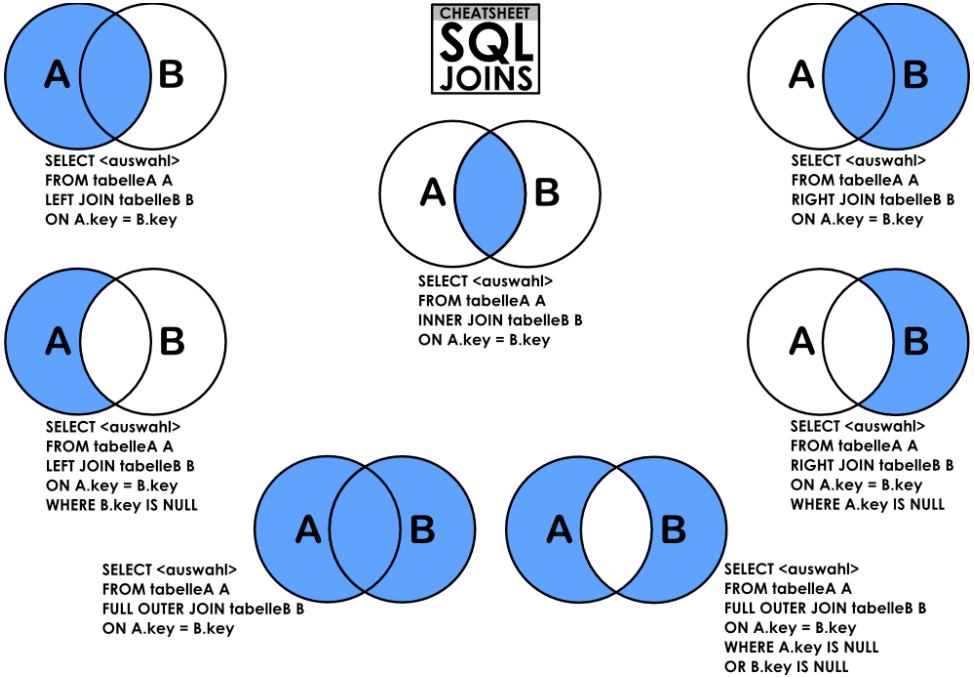
What are all those other Symbols?

- 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 the operators are useful and "common," but not always considered part of the core algebra.
- 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 or the obscure operatorsbecause they may be common internships/job interview questions.

Thinking about JOINs

- Some terms:
  - Natural Join
    - Equality of A and B columns
    - With the same name.
  - Equijoin
    - Explicitly specify columns that must have the same value.
    - A.x=B.z AND A.q=B.w
  - Theta Join: Arbitrary predicate.
- Inner Join
  - JOIN “matches” rows in A and B.
  - Result contains ONLY the matched pairs.
- What is I want:
  - All the rows that matched.
  - And the rows from A that did not match?
  - OUTER JOIN (⟕, ⟖)



Do Some Joins

- prereq is interesting
  - All courses with prereqs.
  - All courses and prereqs even if it does not have a prereq.
  - All courses and ones that are prereqs
  - Courses that do not have prereqs
  - Etc.
- See notebook with examples.

Some Examples

- **A course and its prerequisites**

π course_id←course.course_id,
course_title←course.title,
prerequisite_id←prereq.prereq_id,
prerequisite_title←p.course_id
(
(course ⋈ prereq)
⋈ prereq.prereq_id=p.course_id
ρ p(course))

- Join c
  -
  -
- Join v
  -
  - To get the prereq info
- But,
  - course.course_id is ambiguous
  - Because the table appears twice.
  - So, I have to “alias” the second use.
- Also, note the use of renaming in the project for column names.

Skip in class
But interesting

Some Examples

- **Courses-prerequisites and courses without prereqs.**

τ prerequisite_id
(π course_id←course.course_id,
course_title←course.title,
prerequisite_id←prereq.prereq_id,
prerequisite_title←p.course_id
(
(course ⟗ prereq)
⟗ prereq.prereq_id=p.course_id
ρ p(course))
)

Skip in class
But interesting

Some Examples

- **Courses that are not prerequisites**

```
τ prereq_course_id
(
        π prereq_course_id←prereq.course_id,
                course_id←course.course_id,
                course_title←course.title
                (
                        prereq ⋈ prereq_id=course.course_id course
                )
        )
```

Skip in class
But interesting

Relational Algebra

- We will do more examples in lectures, HW assignments and exams.
- The language *is interesting but* can be tedious and confusing.
- Useful in some advanced scenarios:
  - Designing query languages for new databases and data models.
  - Understanding how DBMS implement query processing and optimization.
- Also cover because it does come up in some advanced courses, and may come up in job interviews.
- I give take home exams and homework assignments:
  - I cannot remember some of the more obscure operators.
  - You have to tinker with the expressions to get an answer.

SQL

Assignment Rename

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

Codd's 12 Rules

Rule 1: Information Rule

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

Rule 2: Guaranteed Access Rule

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

Rule 3: Systematic Treatment of NULL Values

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

Rule 4: Active Online Catalog

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

Rule 5: Comprehensive Data Sub-Language Rule

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

Rule 6: View Updating Rule

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

Codd's 12 Rules

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

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

Rule 8: Physical Data Independence

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

Rule 9: Logical Data Independence

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

Rule 10: Integrity Independence

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

Rule 11: Distribution Independence

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

Rule 12: Non-Subversion Rule

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

Codd's 12 Rules

Rule 3: Systematic treatment of null values:

- "Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing **missing information** and **inapplicable** information in a systematic way, independent of data type."

- Sometimes programmers and database designers are tempted to use "special values" to indicate unknow, missing or inapplicable values.
  - String: "", "NA", "UNKNOWN", …
  - Numbers: -1, 0, -9999

- Indicators can cause confusion because you have to carefully code some SQL statements to the specific, varying choices programmers made.

NULL and Correct Answers

```
In [4]: 1 %sql describe aaaaS21Examples.null_examples;
```

```
 * mysql+pymysql://dbuser:***@localhost
3 rows affected.
```

Out[4]:

Field	Type	Null	Key	Default	Extra
name	varchar(32)	NO	PRI	None	
weight	int	YES		None	
net_worth	int	YES		None	

```
In [5]: 1 %sql select * from aaaaS21Examples.null_examples;
```

```
 * mysql+pymysql://dbuser:***@localhost
4 rows affected.
```

Out[5]:

name	weight	net_worth
Joe	100	100
Larry	0	0
Pete	None	None
Tim	200	200

Without NULL, to get a correct answer:

- I must understand the domain to determine “unknown” values or know what choice a developer made.
- Explicitly include “where weight != 0” in all statements.
- And this varies from column to column, table to table, schema to schema, etc.

```
In [7]: 1 %%sql select avg(weight) as avg_weight, avg(net_worth) as avg_net_worth
        2     from aaaaS21Examples.null_examples where name in ('Joe', 'Larry', 'Tim')
```

```
 * mysql+pymysql://dbuser:***@localhost
1 rows affected.
```

Out[7]:

avg_weight	avg_net_worth
100.0000	100.0000

```
In [9]: 1 %%sql select avg(weight) as avg_weight, avg(net_worth) as avg_net_worth
        2     from aaaaS21Examples.null_examples where name in ('Joe', 'Pete', 'Tim')
```

```
 * mysql+pymysql://dbuser:***@localhost
1 rows affected.
```

Out[9]:

avg_weight	avg_net_worth
150.0000	150.0000

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*

Practice with Modifications

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*;

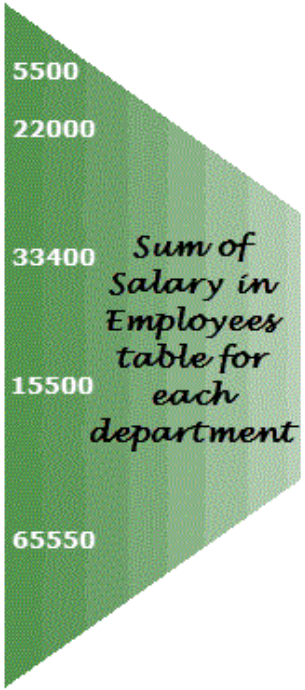
ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music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

Fun with JOINs

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

Notes:

- You will also hear terms like equi-join, non-equi-join, theta join, semi-join, … …
- I ask for definitions on exams, but you can just look them up.

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 $A_1, A_2, \ldots A_n$
  from r_1 **natural join** r_2 **natural join .. natural join** r_n
  where P ;

Student Relation

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

Takes Relation

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	*null*

student **natural join** *takes*

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
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	*null*

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.)

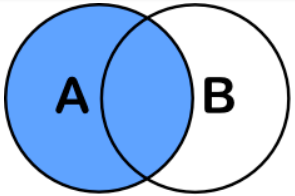
- The **on** condition allows a general predicate over the relations being joined.
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**.
- Query example

  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:

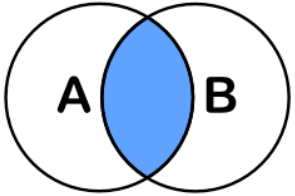
  select *
  from *student , takes*
  where *student_ID = takes_ID*

One Way to Think About Joins

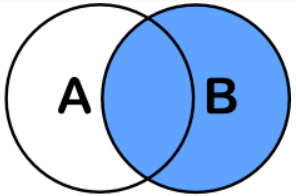


SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key

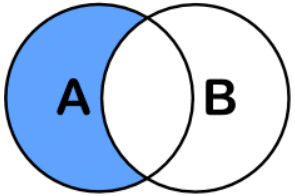
CHEATSHEET
SQL
JOINS



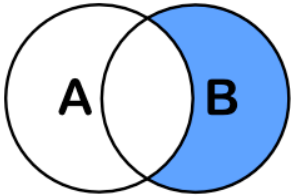
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key



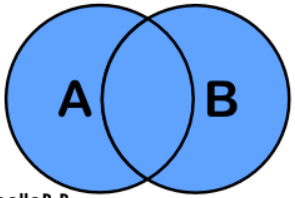
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key



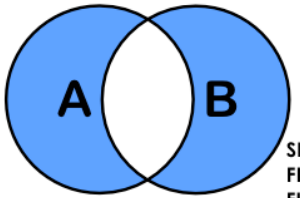
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL



SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL



SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key



SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL

Examples in Notebook

Switch to Notebook

Examples
Web Application
Data Engineering Notebook

Sample Projects

Projects

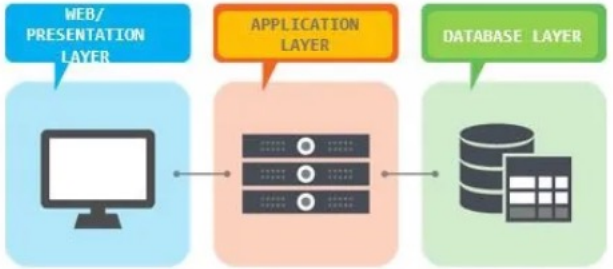
- The programming track will implement a simple, full stack web application.

Full-stack Web Developer

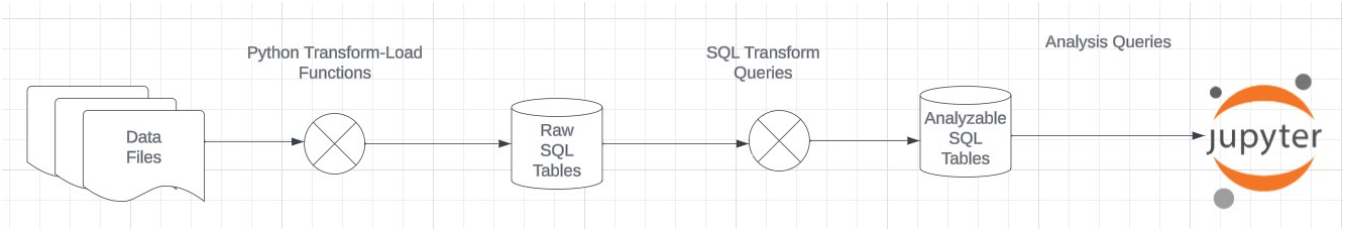
A full-stack web developer is a person who can develop both **client** and **server** software.

In addition to mastering HTML and CSS, he/she also knows how to:

- Program a **browser** (e.g. using JavaScript, jQuery, Angular, or Vue)
- Program a **server** (e.g. using PHP, ASP, Python, or Node)
- Program a **database** (e.g. using SQL, SQLite, or MongoDB)



- The non-programming track will implement a data engineering and visualization process in a Jupyter notebook.



Sample Projects

- Walkthrough of Web Apps:
  - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/W4111-FastAPI-IMDB-Solution
  - /Users/donaldferguson/Dropbox/000/000-A-Current-Examples/current-dashboard

- Data Engineering:
  - /Users/donaldferguson/Dropbox/000/000-Columbia/2024-Spring/W4111-Intro-to-Databases-Spring-2024/Lectures/s-2024-Lecture-3/More-Data-Engineering.ipynb
  - And others.

The End for Today