

*W4111 – Introduction to Databases  
Section 002, Spring 2023*

*Lecture 4: ER, Relational, SQL (III)  
Applications*



# *Contents*

# Contents

- Integrity constraints part 1:
  - Primary key, unique key
  - Foreign key
- Indexes: Concepts and examples.
- Subqueries: Concepts and examples. Including set membership.
- ER design patterns and advanced concepts.
- Some more relational algebra.
- Applications:
  - REST and web applications.
  - Data analysis and visualization.

# *Integrity Constraints, Part 1*



# Integrity Constraints

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

## DFF

- Without integrity constraints in the database, maintaining data correctness requires:
  - Lots of users know what to do and do not make mistakes.
  - Dozens of programs correctly implement constraints in the code and stay up to date on changes.
- Implementing the constraints as part of the schema eliminates many issues.



# Constraints on a Single Relation

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



# Not Null Constraints

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



# Unique Constraints

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

# Simple Example

- Consider a simple example of an entity class *major*:
  - *major(id, name, track)*
  - “id” is a uniquely generated ID
  - “name” is the major name, e.g. “Computer Science,” “Economics,” ... ...
  - “track” is a sub-track/specialty within the major, e.g. “Applications,” “AI/ML,” ...
    - “track” is optional
    - The combination of *(name, track)* is *unique*.
- **Note:** In many DBMS, this automatically creates indices for keys/constraints.
- Switch to Notebook.

```
create table if not exists majors
(
    id          int auto_increment
    primary key,
    major_name  varchar(64) not null,
    major_track varchar(64) null,
    constraint table_name_pk
        unique (major_name, major_track)
);
```



# The check clause

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

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

DFF:

- We could handle the *semester check* with an *enum*.
- Switch to notebook for a slightly different example.



# Referential Integrity

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



## Referential Integrity (Cont.)

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



# Cascading Actions in Referential Integrity

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

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

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

## DFF:

- I do not like using *cascade*. I think making changes should be explicit.
- Other people disagree.
- You will get some simple practice on HW or exams.

# *Indexes*

## *Concepts and Examples*



# Index Creation

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

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



# Index Creation Example

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

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

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

DFF:

- An index on (column1, column2, column3)
- Is also an index on (column1) and (column1, column2)
- But not (column2), (column3), (column2, column3).
- We will see “why” later in the semester.

# *Subqueries*

## *Concepts and Examples*

### *(Including Set Membership)*



# Nested Subqueries

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

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

as follows:

- **From clause:**  $r_i$  can be replaced by any valid subquery
- **Where clause:**  $P$  can be replaced with an expression of the form:

$B <\text{operation}> (\text{subquery})$

$B$  is an attribute and  $<\text{operation}>$  to be defined later.

- **Select clause:**

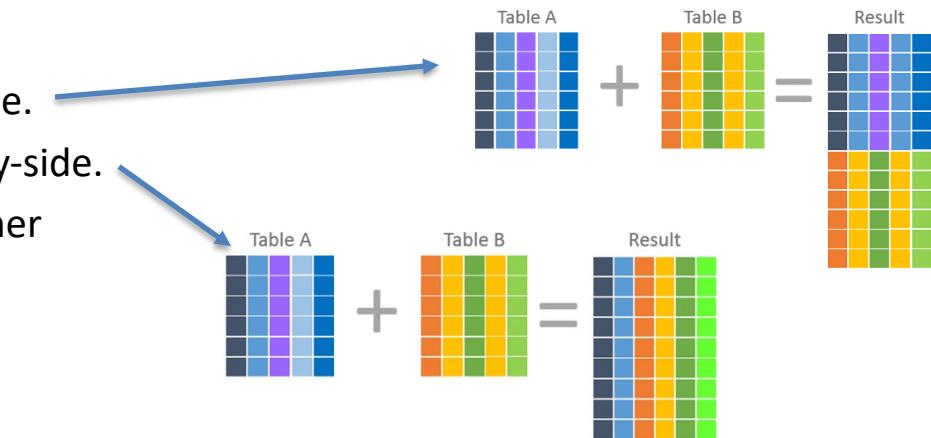
$A_i$  can be replaced by a subquery that generates a single value.

## Note:

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

# Nested Subquery

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



# Consider Some Tables

Takes

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

Student

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

# Consider a Subquery Tables

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

Takes

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

- Assume I wrote a function `find_student_name(x)`
  - Input is an `x`
  - Loops through all students and returns students with `student.ID = x`.
- The query with a subquery above is like:

`result = []`

For `t` in `takes`:

```
new_r = t + find_student_name(t.id)
result.append(new_r)
```

Switch to Notebook

Student

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



# Nested Subqueries

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

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

as follows:

- **From clause:**  $r_i$  can be replaced by any valid subquery
- **Where clause:**  $P$  can be replaced with an expression of the form:

$B <\text{operation}> (\text{subquery})$

$B$  is an attribute and  $<\text{operation}>$  to be defined later.

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

Note: Subquery MUST return

- A single scalar if in the SELECT.
- A Table if in the FROM.
- If in the WHERE:
  - Either a scalar or a table.
  - Depending on the operation.



# Set Membership



# Set Membership

- Find courses offered in Fall 2017 and in Spring 2018

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

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

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



# Set Membership (Cont.)

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

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

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

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

- Note: Above query can be written in a much simpler manner.  
The formulation above is simply to illustrate SQL features



# Set Comparison



# Set Comparison – “some” Clause

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

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

- Same query using > **some** clause

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



## Definition of “some” Clause

- $F <\text{comp}> \text{some } r \Leftrightarrow \exists t \in r \text{ such that } (F <\text{comp}> t)$   
Where  $\text{comp}$  can be:  $<$ ,  $\leq$ ,  $>$ ,  $=$ ,  $\neq$

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

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

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

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

$(= \text{some}) \equiv \text{in}$

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



## Set Comparison – “all” Clause

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

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



# Definition of “all” Clause

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

(5 < all 

0
5
6

) = false

(5 < all 

6
10

) = true

(5 = all 

4
5

) = false

(5 ≠ all 

4
6

) = true (since  $5 \neq 4$  and  $5 \neq 6$ )

$(\neq \text{all}) \equiv \text{not in}$

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



# Test for Empty Relations

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



# Use of “exists” Clause

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

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

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



# Use of “not exists” Clause

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

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

- First nested query lists all courses offered in Biology
- Second nested query lists all courses a particular student took
- Note that  $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = all and its variants



# Test for Absence of Duplicate Tuples

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

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



## Subqueries in the From Clause



# Subqueries in the Form Clause

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

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

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

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



# With Clause

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

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



# Complex Queries using With Clause

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

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



# Scalar Subquery

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

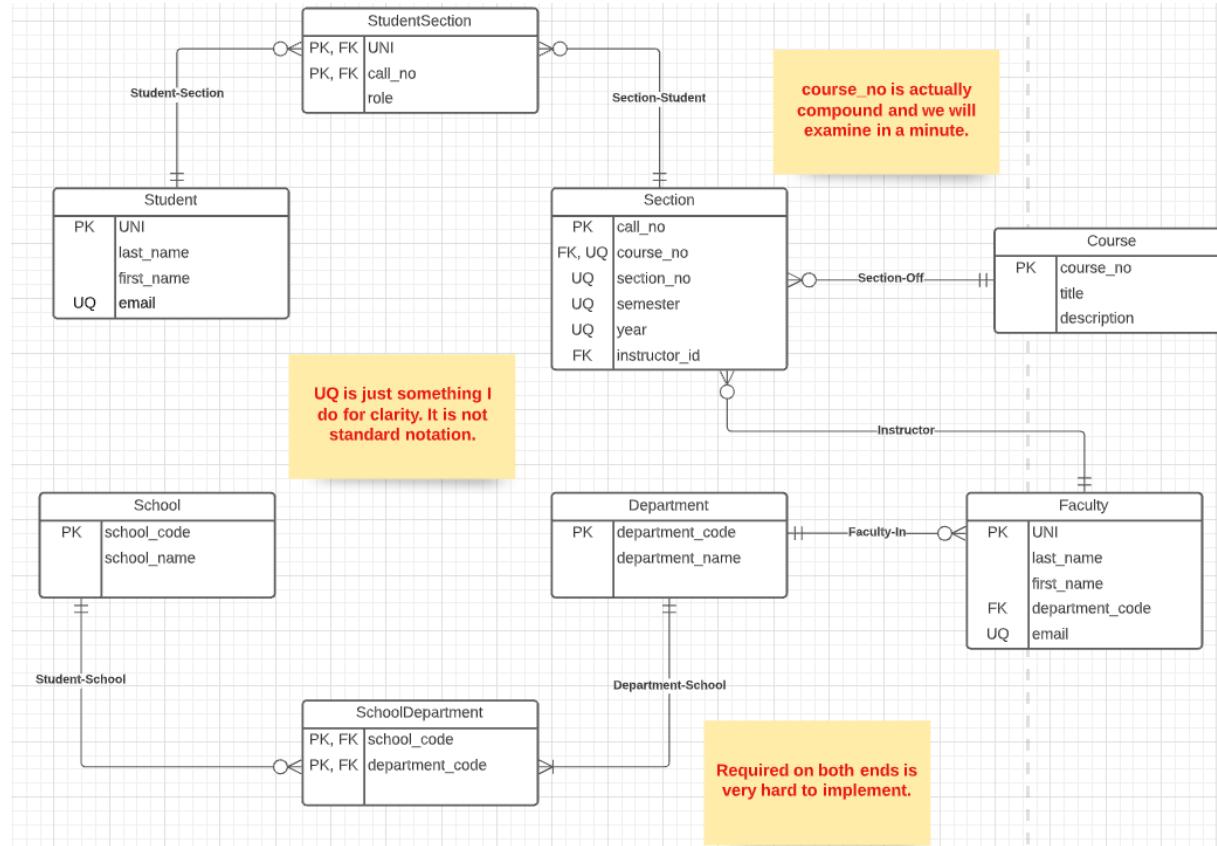
```
select dept_name,  
       ( select count(*)  
           from instructor  
          where department.dept_name = instructor.dept_name)  
      as num_instructors  
  from department;
```

- Runtime error if subquery returns more than one result tuple

# *ER Modeling*

## *Design Patterns and Advanced Concepts*

# Approximate Model



# Course Number

## Key to Columbia Course Listings

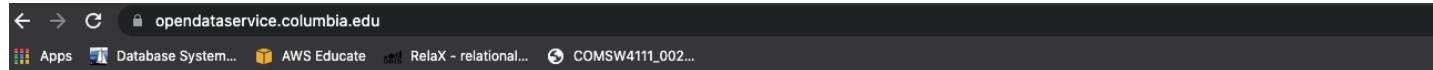
<https://www.cc-seas.columbia.edu/sites/dsa/files/handbooks/Columbia%20Key%20to%20Course%20Listing.pdf>

(Example: ECON W1105 001 Principles of Economics, 4 pts)

A	Architecture, Planning, and Preservation*
B	Business*
BC	Barnard College
C	Columbia College
D	Dentistry**
E	Engineering and Applied Science
F	General Studies
G	Graduate School of Arts and Sciences
H	Reid Hall, Paris**
I	Berlin Consortium Program**
J	Journalism*
K	Continuing Education**
L	Law**
M	Medicine**
N	Nursing**
O	Union Theological**
P	School of Public Health*
R	School of the Arts*
S	Summer Session
T	Social Work*
TA-TZ	Teachers College*
U	International and Public Affairs*
V	Interschool course with Barnard
W	Interfaculty course
X	Barnard College
Z	American Language Program(no credit)**

	Example	Description																																
Call # (5 digit number)	<b>16238</b>	This 5 digit code is assigned to individual courses and is specific to each semester.																																
Department Code (4 letter code)	<b>ECON</b>	This 4 letter code represents the Academic Department that manages the course.																																
Course Number (1 capital letter followed by 4 digit code)	<b>W</b>  <b>1105</b>	<p>The <i>capital letters</i> indicate the instructor teaching the course and their affiliation with a <b>division, school or affiliate</b> of the University.</p> <p>Unless otherwise noted, courses numbers beginning with the following letters are generally open to CC/SEAS undergraduate students:</p> <table border="1"> <tbody> <tr><td>BC</td><td>Barnard College</td></tr> <tr><td>C</td><td>Columbia College</td></tr> <tr><td>E</td><td>Engineering and Applied Science</td></tr> <tr><td>F</td><td>General Studies</td></tr> <tr><td>G</td><td>Graduate School of Arts and Sciences</td></tr> <tr><td>V</td><td>Interschool course with Barnard</td></tr> <tr><td>W</td><td>Interfaculty course</td></tr> <tr><td>X</td><td>Barnard College</td></tr> </tbody> </table> <p>The first <i>digit</i> indicates the <b>level of the course</b>. Generally, levels are indicated as:</p> <table border="1"> <tbody> <tr><td>0</td><td>Course that cannot be credited toward any degree</td></tr> <tr><td>1</td><td>Undergraduate course, introductory</td></tr> <tr><td>2</td><td>Undergraduate course, intermediate</td></tr> <tr><td>3</td><td>Undergraduate course, advanced</td></tr> <tr><td>4</td><td>Graduate course that is open to qualified undergraduates</td></tr> <tr><td>6</td><td>Graduate course</td></tr> <tr><td>8</td><td>Graduate course, advanced</td></tr> <tr><td>9</td><td>Graduate research course or seminar</td></tr> </tbody> </table>	BC	Barnard College	C	Columbia College	E	Engineering and Applied Science	F	General Studies	G	Graduate School of Arts and Sciences	V	Interschool course with Barnard	W	Interfaculty course	X	Barnard College	0	Course that cannot be credited toward any degree	1	Undergraduate course, introductory	2	Undergraduate course, intermediate	3	Undergraduate course, advanced	4	Graduate course that is open to qualified undergraduates	6	Graduate course	8	Graduate course, advanced	9	Graduate research course or seminar
BC	Barnard College																																	
C	Columbia College																																	
E	Engineering and Applied Science																																	
F	General Studies																																	
G	Graduate School of Arts and Sciences																																	
V	Interschool course with Barnard																																	
W	Interfaculty course																																	
X	Barnard College																																	
0	Course that cannot be credited toward any degree																																	
1	Undergraduate course, introductory																																	
2	Undergraduate course, intermediate																																	
3	Undergraduate course, advanced																																	
4	Graduate course that is open to qualified undergraduates																																	
6	Graduate course																																	
8	Graduate course, advanced																																	
9	Graduate research course or seminar																																	
Course Section	<b>001</b>	Based on course demand, some academic departments offer the same course during 2 or more different time slots. Each time slot is assigned a different section number.																																
Points/Credits	<b>4</b>	The term "points" and "credits" are often used interchangeably and is generally related to the number of classroom contact hours.																																

# Columbia Open Data Service



COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

## OPEN DATA SERVICE

[News and Updated](#)   [About Data Feeds](#)   [Request a New Feed](#)

### Data Feed Service

Columbia University offers data feeds in programming-friendly formats for research and academic purposes. The Open Data Feed Service currently offers the following data feeds:

- [Course Information](#)
- [Athletic Schedule](#)
- [Academic Commons](#)
- [CLIO - Library Catalog Data](#)
- [Textbooks Feed](#)

Data feed information includes the feed's refresh schedule, data diagrams, table and data element documentation, data training, and data security. More details on each feed is available on each feed's page.



Academic Commons



Athletics Schedule



CLIO-Library Catalog



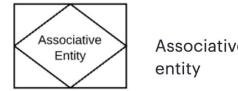
Course Information



Textbooks

# 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*.

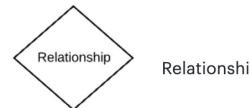


Associative entities relate the instances of several entity types. They also contain attributes specific to the relationship between those entity instances.

## 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
---------------------	------	-------------



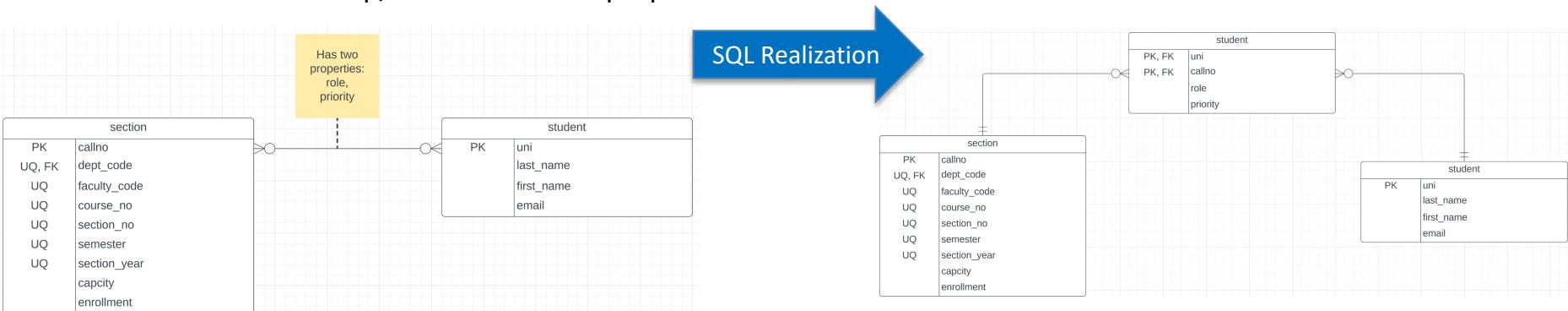
Relationships are associations between or among entities.



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](https://en.wikipedia.org/wiki/Associative_entity))
- Consider *Students – Sections*:
  - This is many-to-many. There is not way to implement in SQL. You see this in the *Advise*s table in the sample database.
  - The “relationship/association” has properties that are not attributes of the connected entities.



Switch to notebook diagram.

# *More Relational Algebra*

# What are all those other Symbols?

- $\tau$       order by
- $\gamma$       group by
- $\neg$       negation
- $\div$       set division
- $\bowtie$       natural join, theta-join
- $\bowtie_l$       left outer join
- $\bowtie_r$       right outer join
- $\bowtie_f$       full outer join
- $\bowtie_s$       left semi join
- $\bowtie_r$       right semi join
- $\triangleright$       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 operators because they may be common internships/job interview questions.

# Anti – Join

- “An anti-join is when you would like to keep all of the records in the original table except those records that match the other table.”

instructor  $\triangleright$  ID=i\_id advisor

instructor.ID	instructor.name	instructor.dept_name	instructor.salary
12121	'Wu'	'Finance'	90000
15151	'Mozart'	'Music'	40000
32343	'El Said'	'History'	60000
33456	'Gold'	'Physics'	87000
58583	'Califieri'	'History'	62000
83821	'Brandt'	'Comp. Sci.'	92000

$\sigma_{i\_id=null} (\text{instructor} \bowtie \text{ID}=i\_id \text{ advisor})$

instructor.ID	instructor.name	instructor.dept_name	instructor.salary	advisor.s_id	advisor.i_id
12121	'Wu'	'Finance'	90000	null	null
15151	'Mozart'	'Music'	40000	null	null
32343	'El Said'	'History'	60000	null	null
33456	'Gold'	'Physics'	87000	null	null
58583	'Califieri'	'History'	62000	null	null
83821	'Brandt'	'Comp. Sci.'	92000	null	null

# Group By, Order By

classroom

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

- These are very simple examples.
- We can apply them to relations created by operations on other tables.

$\tau \text{total\_seats } (\gamma \text{ building; sum(capacity)} \rightarrow \text{total\_seats} \text{ (classroom)})$

classroom.building	total_seats
'Painter'	10
'Taylor'	70
'Watson'	80
'Packard'	500

# *REST*

# Data Modeling Concepts and REST

Almost any data model has the same core concepts:

- Types and instances:
  - Entity Type: A definition of a type of thing with properties and relationships.
  - Entity Instance: A specific instantiation of the Entity Type
  - Entity Set Instance: An Entity Type that:
    - Has properties and relationships like any entity, but ...
    - Has at least one *special relationship* – ***contains***.
- Operations, minimally CRUD, that manipulate entity types and instances:
  - Create
  - Retrieve
  - Update
  - Delete
  - Reference/Identify/... ...

## What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

## HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

- **GET** – Provides a read only access to a resource.
- **POST** – Used to create a new resource.
- **DELETE** – Used to remove a resource.
- **PUT** – Used to update a existing resource or create a new resource.

## Introduction to RESTful web services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

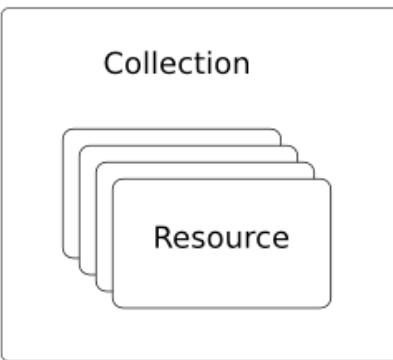
## Creating RESTful Webservice

In next chapters, we'll create a webservice say user management with following functionalities –

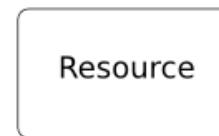
Sr.No.	URI	HTTP Method	POST body	Result
1	/UserService/users	GET	empty	Show list of all the users.
2	/UserService/addUser	POST	JSON String	Add details of new user.
3	/UserService/getUser/id	GET	empty	Show details of a user.

# REST and Resources

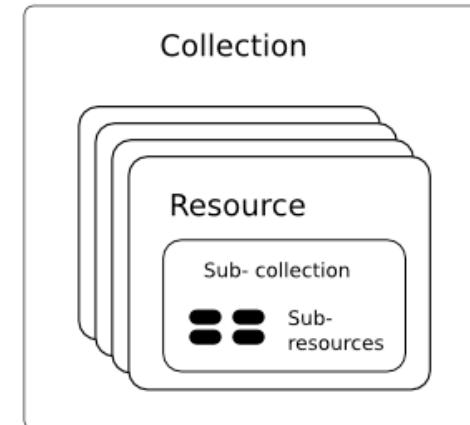
## Resource Model



A Collection with  
Resources

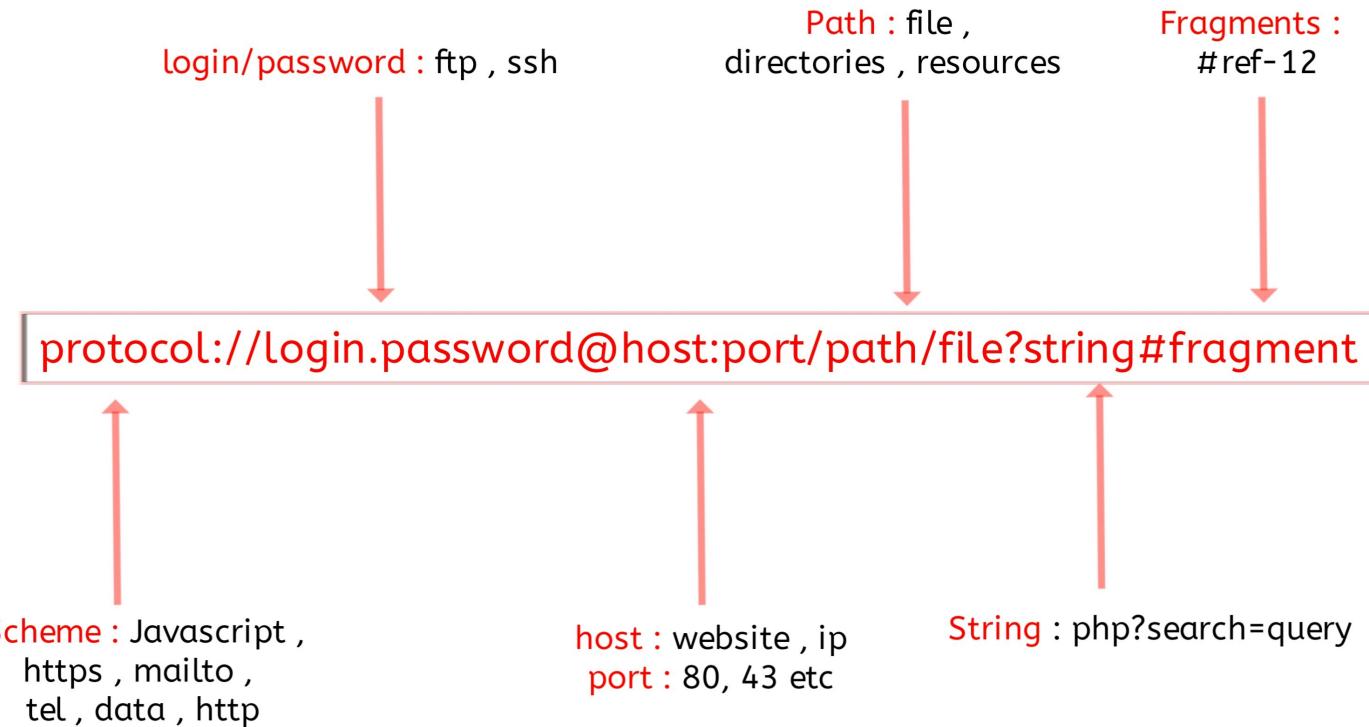


A Singleton  
Resource



Sub-collections and  
Sub-resources

# URLs



# Simplistic, Conceptual Mapping (Examples)

REST Method	Resource Path	Relational Operation	DB Resource
DELETE	/people	DROP TABLE	people table
POST	/people	INSERT INTO PEOPLE (...) VALUES(...)	people table people row
GET	/people/21	SHOW KEYS FROM people ...;  SELECT * FROM people WHERE playerID= 21	people row
GET	/people/21/batting	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21	
GET	/people/21/batting/2004_1	SELECT batting.* FROM people JOIN batting USING(playerID) WHERE playerID=21 AND yearID=2004 AND stint=1	

# Application Architecture

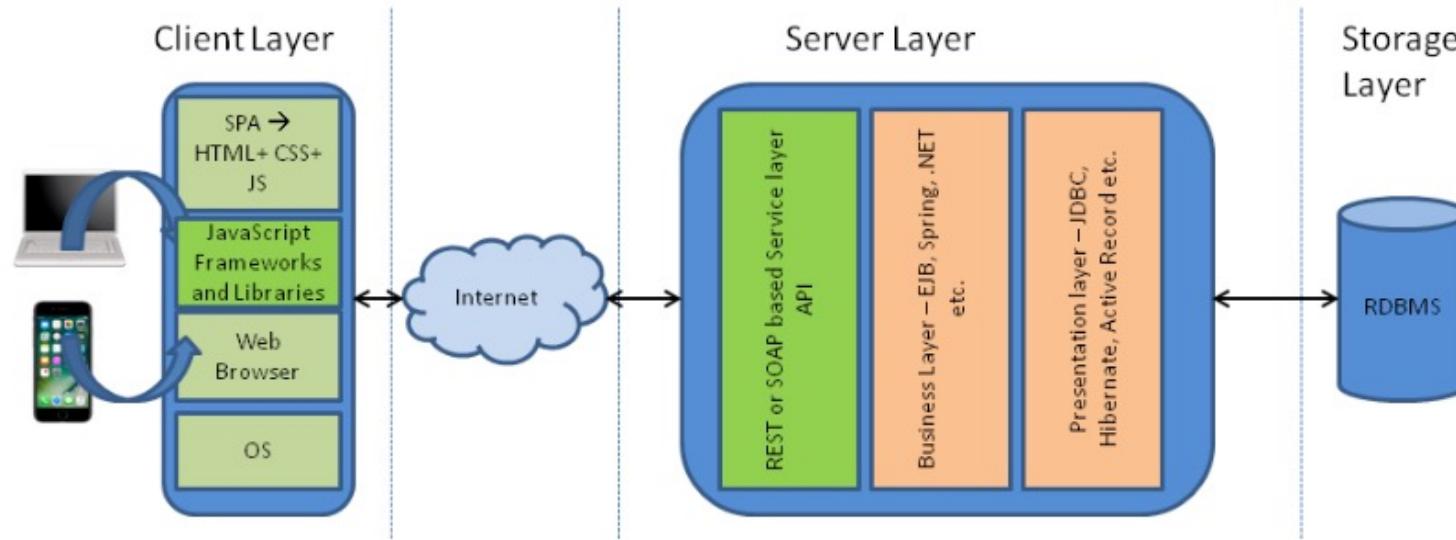


Diagram 2: The moving of the Web Layer from the Server to the Client

# Walkthrough

- Simple web application template.
- Calling some cloud APIs.

# *Data and Visualization*

## *(Walk through the data sets and visualizations)*