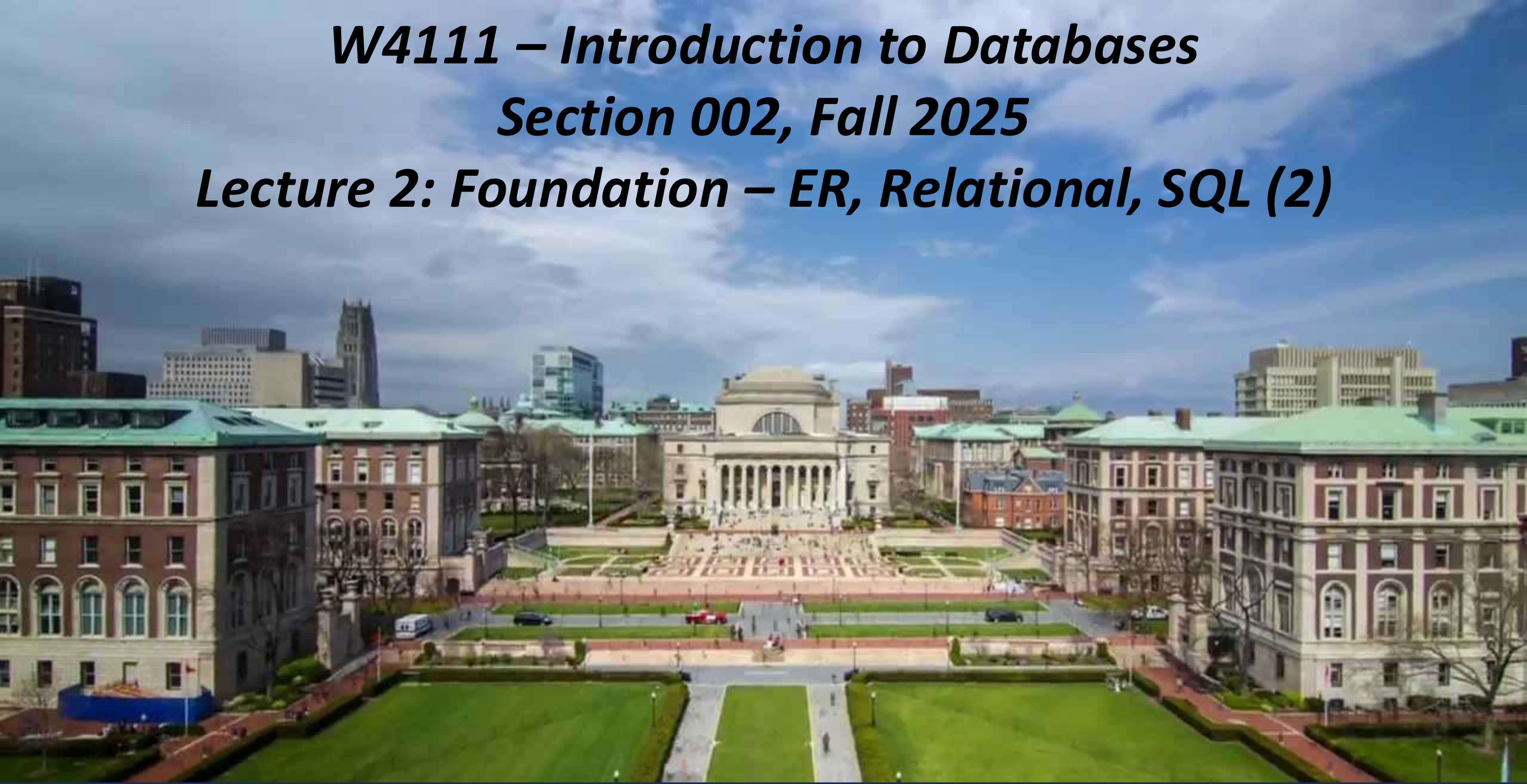
W4111 – Introduction to Databases
Section 002, Fall 2025
Lecture 2: Foundation – ER, Relational, SQL (2)



Contents

Notes and Content

- ER Model/Modeling:
  - Additional concepts: associative entity, weak entity, multi-valued attributes
  - Translating an ER diagram into SQL – some examples.
- Relational algebra
  - Additional operators: union, intersect, set difference, renaming
  - Some examples
- SQL
  - Integrity constraints
  - Some data types
  - WITH, common table expressions
  - Operators: Union, intersect, set difference, renaming
  - A first look at SUBQUERY
- Project concepts: (Web) applications, REST, data engineering

ER Model/ER Modeling

Relationship Sets

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

  Example:

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

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

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

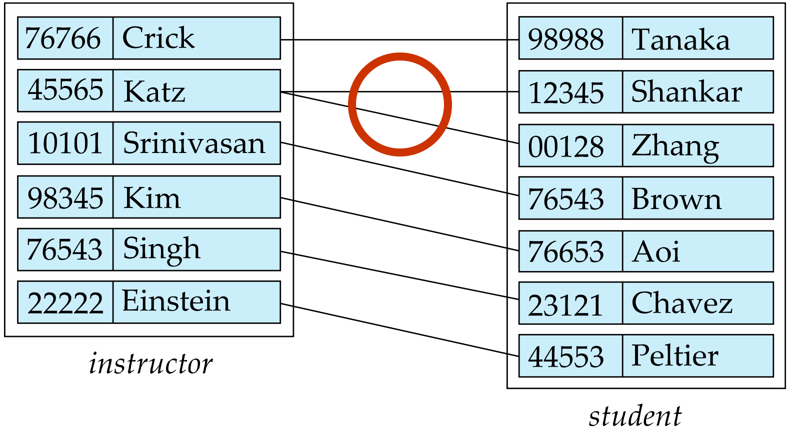
  where $(e_1, e_2, \ldots, e_n)$ is a relationship

  - Example:

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

Relationship Sets (Cont.)

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



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

instructor

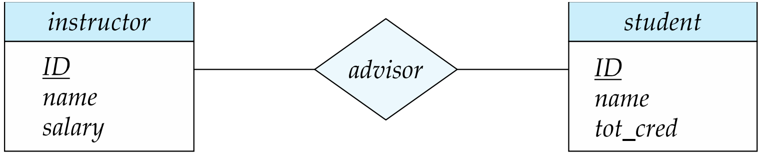
98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

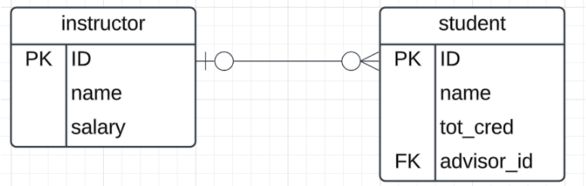
student

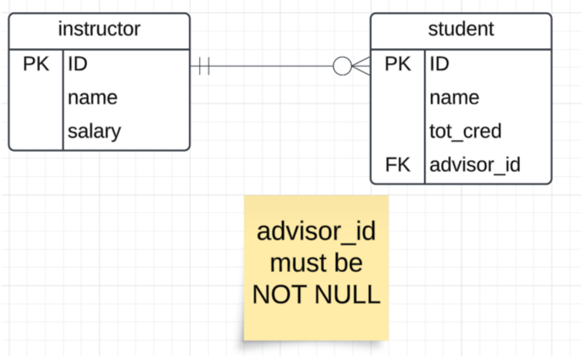
The fact that Katz advises 2 students is important ➔
The instructor – advises -> student relationship is one-to-many ➔
The foreign key must be from student to instructor.

Representing Relationship Sets via ER Diagrams

- Diamonds represent relationship sets.





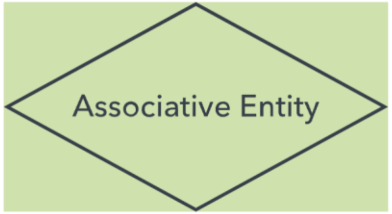


advisor_id must be NOT NULL

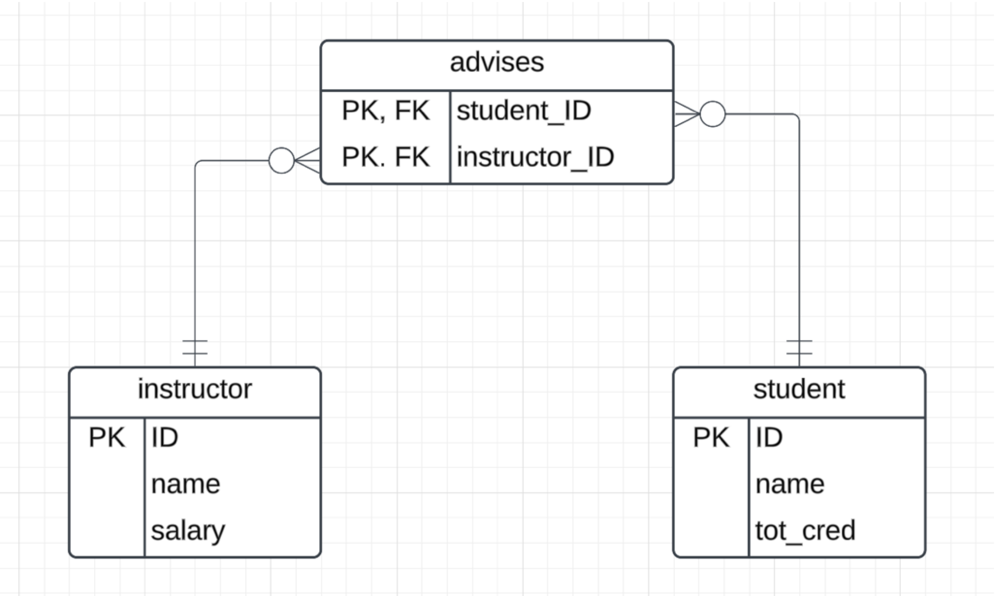
Many-to-Many

- Let's consider *a* modification and assume that
  - A student may have more than one advisor
  - An instructor may advise several students ➔
  - The relationship is many-to-many ➔
  - We cannot implement this in SQL using foreign keys ➔
  - We need an *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)

Associative Entity



This symbol is quasi-standard, but then again almost everything in ER diagrams is quasi-standard.



advises

PK, FK	student_ID
PK. FK	instructor_ID

instructor

PK	ID
	name
	salary

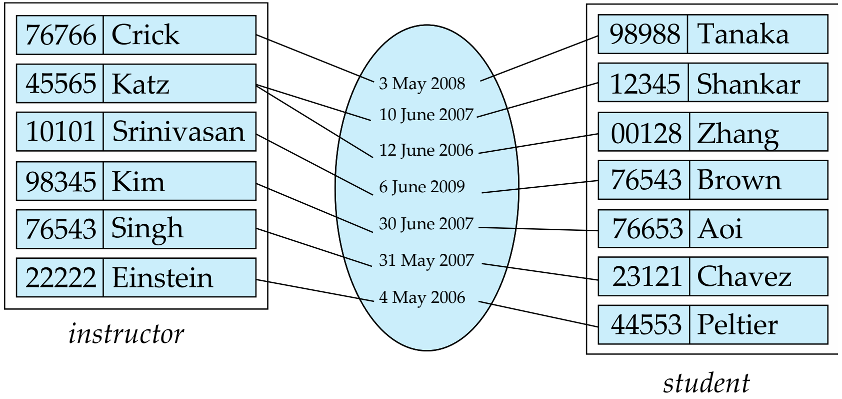
student

PK	ID
	name
	tot_cred

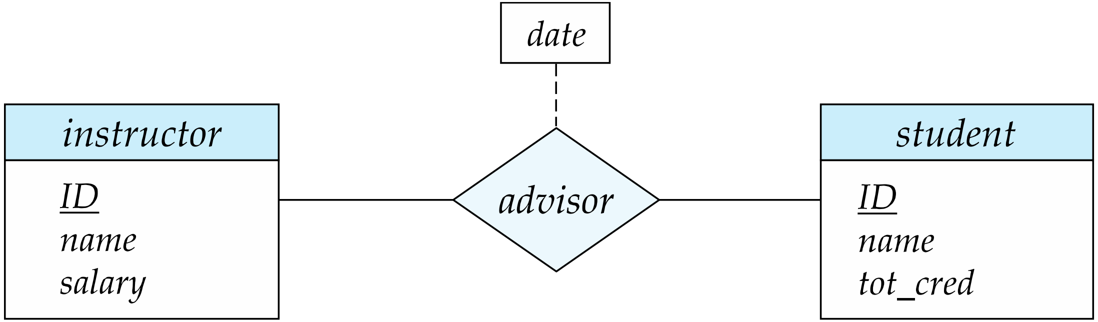
ER diagram rules –
“Tis a custom more honored in the breach than the observance”

Relationship Sets (Cont.)

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



Relationship Sets with Attributes



date

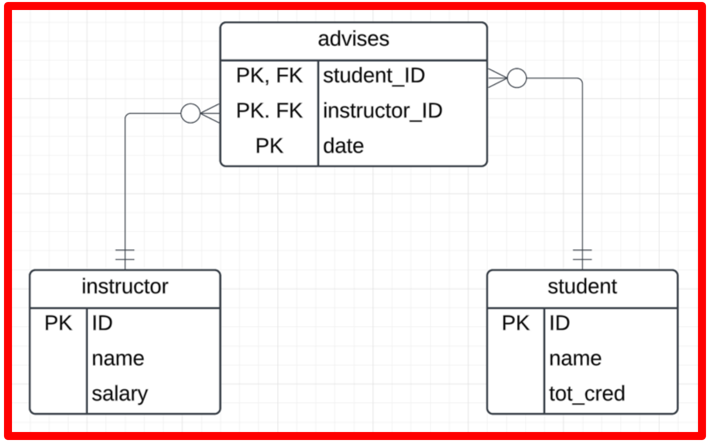
advisor

instructor

ID
name
salary

student

ID
name
tot_cred



advises	
PK, FK	student_ID
PK. FK	instructor_ID
PK	date

instructor	
PK	ID
	name
	salary

student	
PK	ID
	name
	tot_cred

Associative Entity

- When translating an ER diagram into SQL, there are two reasons to use an associative entity.
  - The relationship is many-to-many OR
  - The relationship has properties/attributes.
  - Despite being a “relational database,” SQL only has entities.

- This is not true for some NoSQL databases.
  We will see that relationship (edge) with properties is a first-class concept in graph databases (https://en.wikipedia.org/wiki/Graph_database).

Weak Entity Sets

- Consider a *section* entity, which is uniquely identified by a *course_id*, *semester, year*, and *sec_id*.
- Clearly, section entities are related to course entities. Suppose we create a relationship set *sec_course* between entity sets *section* and *course*.
- Note that the information in *sec_course* is redundant, since *section* already has an attribute *course_id*, which identifies the course with which the section is related.
- One option to deal with this redundancy is to get rid of the relationship *sec_course*; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.

Weak Entity Sets (Cont.)

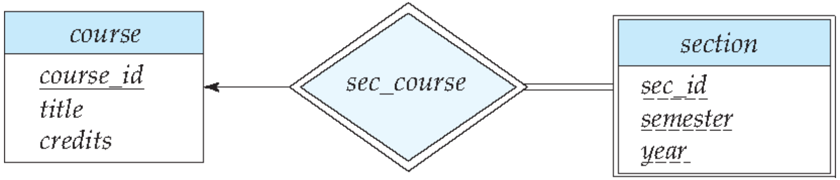
- An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *section_id*, *year*, and *semester.*
  - However, the entity set *section* then does not have enough attributes to identify a particular *section* entity uniquely
- To deal with this problem, we treat the relationship *sec_course* as a special relationship that provides extra information, in this case, the *course_id*, required to identify *section* entities uniquely.
- A **weak entity set** is one whose existence is dependent on another entity, called its **identifying entity**
- Instead of associating a primary key with a weak entity, we use the identifying entity, along with extra attributes called **discriminator** to uniquely identify a weak entity.

Weak Entity Sets (Cont.)

- An entity set that is not a weak entity set is termed a **strong entity set**.
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set.
- The identifying entity set is said to **own** the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.
- Note that the relational schema we eventually create from the entity set *section* does have the attribute *course_id*, for reasons that will become clear later, even though we have dropped the attribute *course_id* from the entity set *section.*

Expressing Weak Entity Sets

- In E-R diagrams, a weak entity set is depicted via a double rectangle.
- We underline the discriminator of a weak entity set with a dashed line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.
- Primary key for *section* – (*course_id, sec_id, semester, year*)



Weak Entity Set

- This is a somewhat obscure and pedantic concept.
  But, it may come up in interviews, discussions, etc.

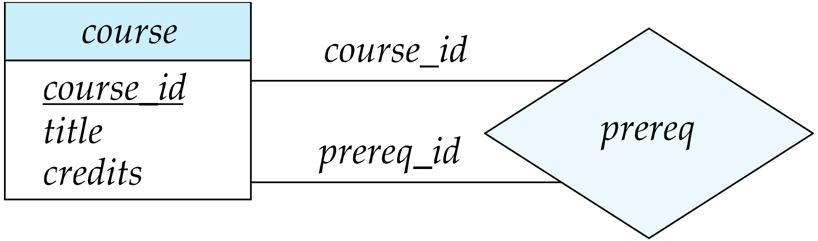
- In Crow's Foot notation, you can *infer* weak entities if an entity has an attribute that is part of a primary key and a foreign key.

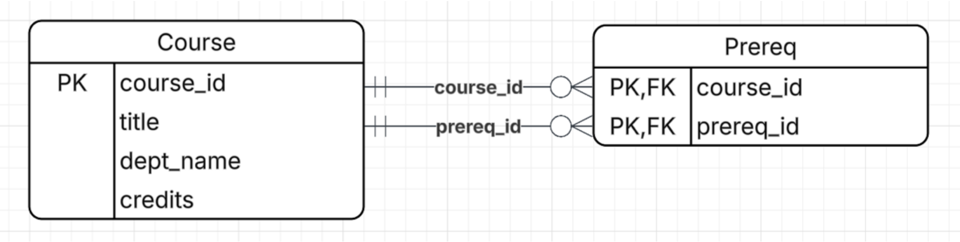
- Summary:
  - I will ask you questions about the weak entity concept.
  - You normally due not worry about this term when modeling and converting models into SQL DDL.
  - You implement the concept with primary key, foreign key constraints.

Roles

Restart lecture here.

- 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**.



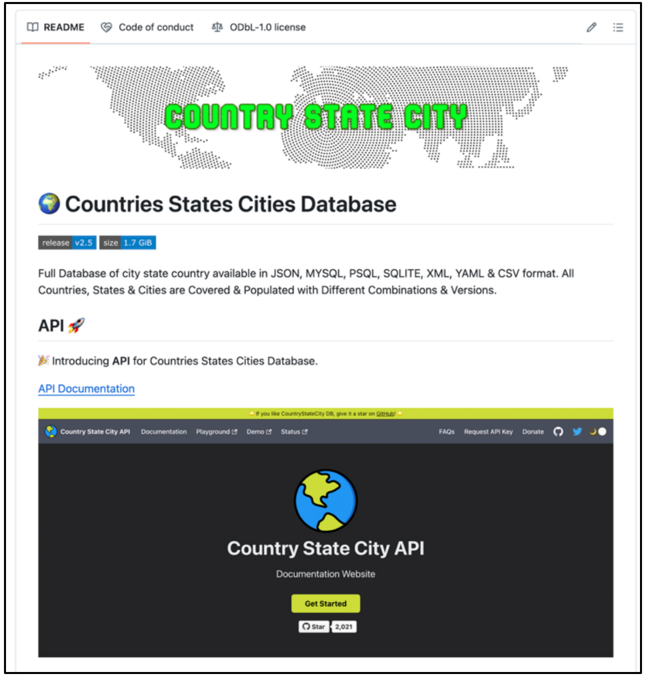


- The associative entity is only needed because course – prereq -> course is many-to-many.
- If it each course had at most one prereq, you could use a foreign key.

Complex Attributes

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

Data Cleanup/Validation



🌍 Countries States Cities Database

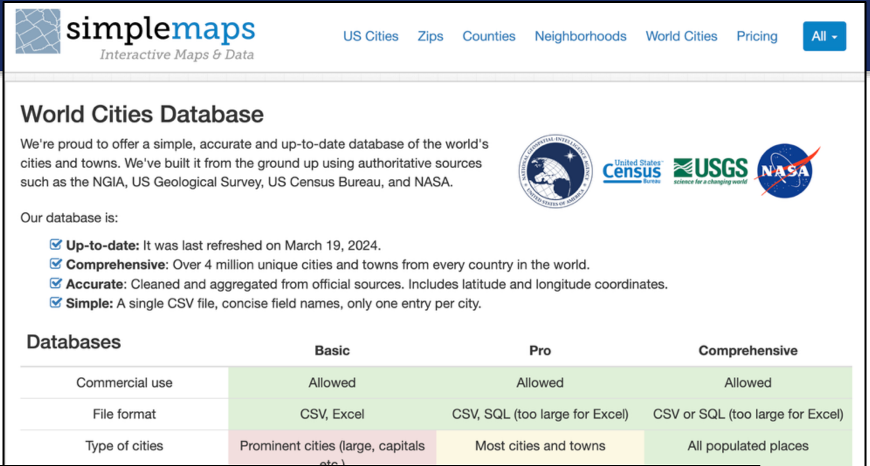
release v2.5 size 1.7 GiB

Full Database of city state country available in JSON, MYSQL, PSQL, SQLITE, XML, YAML & CSV format. All Countries, States & Cities are Covered & Populated with Different Combinations & Versions.

API 🚀

🎉 Introducing **API** for Countries States Cities Database.

API Documentation



World Cities Database

We're proud to offer a simple, accurate and up-to-date database of the world's cities and towns. We've built it from the ground up using authoritative sources such as the NGIA, US Geological Survey, US Census Bureau, and NASA.

Our database is:

- ☑ **Up-to-date:** It was last refreshed on March 19, 2024.
- ☑ **Comprehensive:** Over 4 million unique cities and towns from every country in the world.
- ☑ **Accurate:** Cleaned and aggregated from official sources. Includes latitude and longitude coordinates.
- ☑ **Simple:** A single CSV file, concise field names, only one entry per city.

Databases

	Basic	Pro	Comprehensive
Commercial use	Allowed	Allowed	Allowed
File format	CSV, Excel	CSV, SQL (too large for Excel)	CSV or SQL (too large for Excel)
Type of cities	Prominent cities (large, capitals etc.)	Most cities and towns	All populated places

million

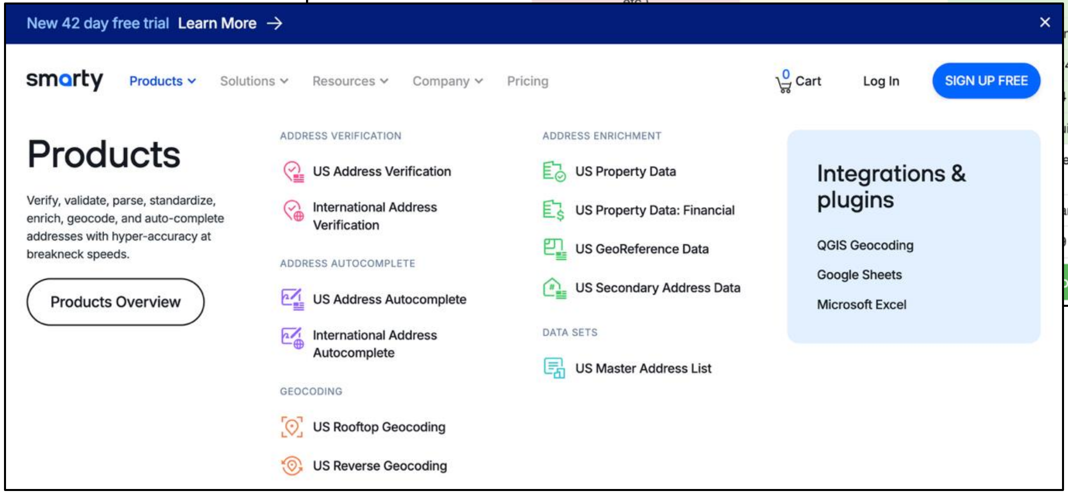
4 months

ired

edistribution

rantee

ow!



Products

Verify, validate, parse, standardize, enrich, geocode, and auto-complete addresses with hyper-accuracy at breakneck speeds.

Products Overview

ADDRESS VERIFICATION
- US Address Verification
- International Address Verification

ADDRESS AUTOCOMPLETE
- US Address Autocomplete
- International Address Autocomplete

GEOCODING
- US Rooftop Geocoding
- US Reverse Geocoding

ADDRESS ENRICHMENT
- US Property Data
- US Property Data: Financial
- US GeoReference Data
- US Secondary Address Data

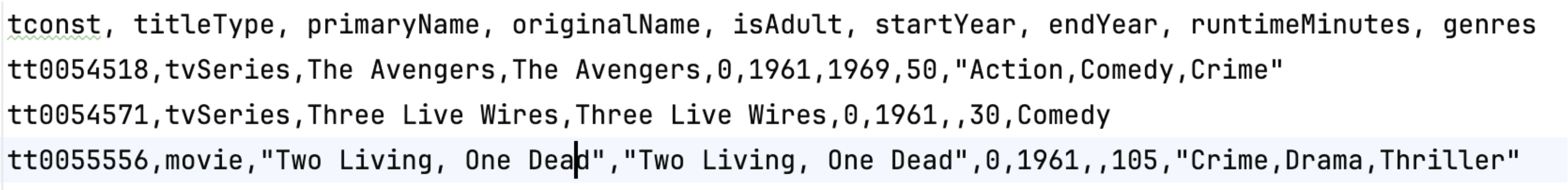
DATA SETS
- US Master Address List

Integrations & plugins

- QGIS Geocoding
- Google Sheets
- Microsoft Excel

Multivalued Attributes

- The relational model and SQL **require** (strongly encourage) attribute domains to be *atomic.*
  - "Every domain must contain atomic values(smallest indivisible units) which means composite and multi-valued attributes are not allowed." (https://www.geeksforgeeks.org/constraints-on-relational-database-model/#)
  - This is sometimes known as "First Normal Form."
    We will cover normalization later in the semester.
- When loading data, e.g. the IMDB data, you often encounter multi-valued attributes.



```
tconst, titleType, primaryName, originalName, isAdult, startYear, endYear, runtimeMinutes, genres
tt0054518,tvSeries,The Avengers,The Avengers,0,1961,1969,50,"Action,Comedy,Crime"
tt0054571,tvSeries,Three Live Wires,Three Live Wires,0,1961,,30,Comedy
tt0055556,movie,"Two Living, One Dead","Two Living, One Dead",0,1961,,105,"Crime,Drama,Thriller"
```

- **How do we handle in relational/SQL? (Switch to notebook)**

Relational Algebra

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

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

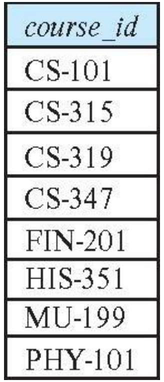
When the “arity” is not quite right, you can shape with project.

Union Operation (Cont.)

- Result of:

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

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



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

Set-Intersection Operation

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

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

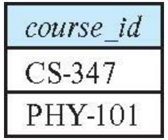
  - Result

course_id
CS-101

Set Difference Operation

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

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



course_id
CS-347
PHY-101

Simple Examples

- *σ dept_name='Comp. Sci.' (instructor ∪ student)* produces a weird answer because the union does not make any sense.
  - There is the correct number of columns and the types match.
  - The domains are different.
  - There are times when the tables have different numbers of columns, the column order is different, or the types are incompatible.
- You can fix the arity problem using a project.
- A somewhat artificial example is:

  σ dept_name='Comp. Sci.'

  ((π ID, name, dept_name, salary, tot_cred←0 (instructor))

  ∪

  (π ID, name, dept_name, salary←0, tot_cred (student)))
- Instead of using “0,” I should have set the not applicable values to NULL but the RelaX calculator did not like that.

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.

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

$$Music \leftarrow \sigma_{dept\_name=\text{“}Music\text{”}}(instructor)$$

$$Physics \cup Music$$

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

The Rename Operation

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

  $$\rho_x (E)$$

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

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

Putting Some Pieces Together

- Consider a course and its prereqs. (course ⋈ prereq) ⋈ course_id=prereq_id course
- There are a couple of problems with this query.
  - Does course.course_id refer to the first course or the prereq course?
  - We would wind up with a couple of columns with the same name.
- Something like this “fixes” the problem.
  (course ⋈ prereq) ⋈ the_prereq.course_id=prereq_id (ρ the_prereq(course))
- Renaming columns would also work and probably improve the answer.

```
π course_id ← course.course_id, course_title←course.title,
        prereq_id←the_prereq.course_id, prereq_title←the_prereq.title
                (
                        (course ⋈ prereq) ⋈ the_prereq.course_id=prereq_id (ρ the_prereq(course))
                )
```

SQL

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.

Create Table Construct

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

  create table *r*

  $(A_1\ D_1,\ A_2\ D_2,\ \ldots,\ A_n\ D_n,$
  (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);
```

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

- Integrity constraints as an integral part of DDL and implemented "in the database" is one of the great strengths of SQL.
- Dozens of applications and thousands of users may access and update data. Relying on correctly documenting and implementing in many places is error prone.
- Business changes that result in changed constraints creates a complex documentation and application modification problem.

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

Show example in notebook.

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')))
```

There is a better, or different, way to handle the *semester* "constraint."
Switch to notebook.

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**

- I tend to avoid cascading, but that is just my style. I think delete and update should be explicit.
- Cascade does make more sense for Weak Entities.

Complex Check Conditions

- The predicate in the check clause can be an arbitrary predicate that can include a subquery.

  check (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))

  The check condition states that the time_slot_id in each tuple in the *section* relation is actually the identifier of a time slot in the *time_slot* relation.

  - The condition has to be checked not only when a tuple is inserted or modified in *section* , but also when the relation *time_slot* changes

Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp:** date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - Example: interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Large-Object Types

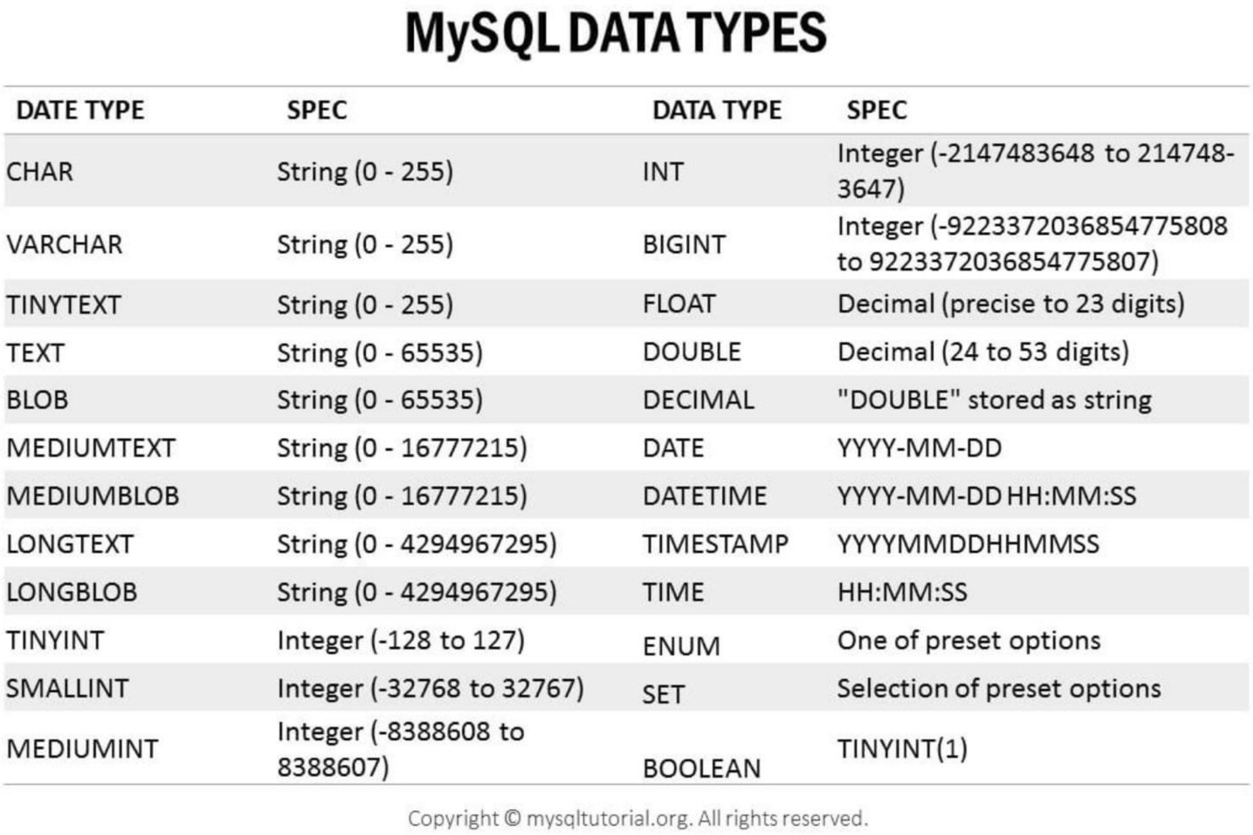
- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of character data
- When a query returns a large object, a pointer is returned rather than the large object itself.

Historically,

- BLOBs were for data like images, scans, … …
- CLOBs were for things like documents.

By and large with the emergence of the web and object/block storage, databases simply store URLs to the "large object."

MySQL Data Types (Partial)



MySQL DATA TYPES

DATE TYPE	SPEC	DATA TYPE	SPEC
CHAR	String (0 - 255)	INT	Integer (-2147483648 to 2147483647)
VARCHAR	String (0 - 255)	BIGINT	Integer (-9223372036854775808 to 9223372036854775807)
TINYTEXT	String (0 - 255)	FLOAT	Decimal (precise to 23 digits)
TEXT	String (0 - 65535)	DOUBLE	Decimal (24 to 53 digits)
BLOB	String (0 - 65535)	DECIMAL	"DOUBLE" stored as string
MEDIUMTEXT	String (0 - 16777215)	DATE	YYYY-MM-DD
MEDIUMBLOB	String (0 - 16777215)	DATETIME	YYYY-MM-DD HH:MM:SS
LONGTEXT	String (0 - 4294967295)	TIMESTAMP	YYYYMMDDHHMMSS
LONGBLOB	String (0 - 4294967295)	TIME	HH:MM:SS
TINYINT	Integer (-128 to 127)	ENUM	One of preset options
SMALLINT	Integer (-32768 to 32767)	SET	Selection of preset options
MEDIUMINT	Integer (-8388608 to 8388607)	BOOLEAN	TINYINT(1)

Summary

- All SQL implementations in a common core set of data types.
- All SQL implementations/products introduce their own additional and extended data types.
- The benefits of the types are:
  - Implementing constraints and integrity on setting data values.
  - There are often special functions that understand the domain.
- The easiest example to help understanding is time/date.
- Let’s see some examples in the notebook.
- When you have homework assignments, you will have to look at the documentation or Google or ChatGPT to know if there is an appropriate data type. Practice is the only way you learn. On exams, I will provide a cheat sheet.

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;
```

- These are also called common table expressions.
- These are very, very helpful when writing complex queries with many clauses.
- “Slow is smooth. Smooth is fast.”
- “Break it down Barney Style.”
- Let’s look at a complex example in the notebook.

Set Operations

- Find courses that ran in Fall 2017 or in Spring 2018

```
(select course_id  from section where sem = 'Fall' and year = 2017)
union
(select course_id  from section where sem = 'Spring' and year = 2018)
```

- Find courses that ran in Fall 2017 and in Spring 2018

```
(select course_id  from section where sem = 'Fall' and year = 2017)
intersect
(select course_id  from section where sem = 'Spring' and year = 2018)
```

- Find courses that ran in Fall 2017 but not in Spring 2018

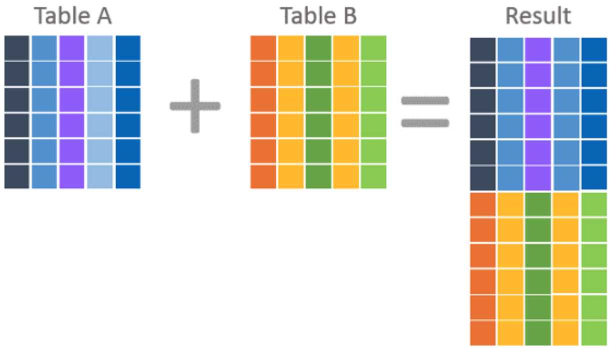
```
(select course_id  from section where sem = 'Fall' and year = 2017)
except
(select course_id  from section where sem = 'Spring' and year = 2018)
```

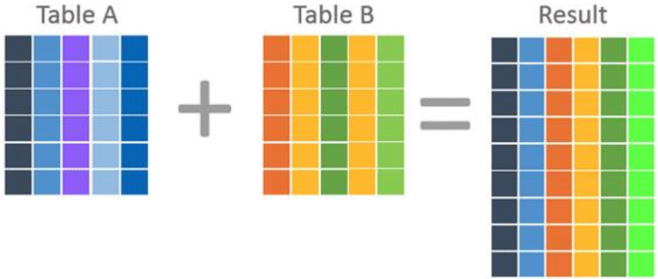
Set Operations (Cont.)

- Set operations **union**, **intersect**, and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the
  - **union all**,
  - **intersect all**
  - **except all**.

JOIN and UNION

- There are basically two approaches to putting tables together.
- UNION kind of “stacks” the tables.
- JOIN kind of ”puts them side by side.”
- We will see that there are more sophisticated patterns, including subqueries for combining tables.





Nested Subqueries

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

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  as follows:

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

    B <operation> (subquery)

    B is an attribute and <operation> to be defined later.
  - **Select clause:**

    A_i can be replaced be 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, ...

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;
```

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

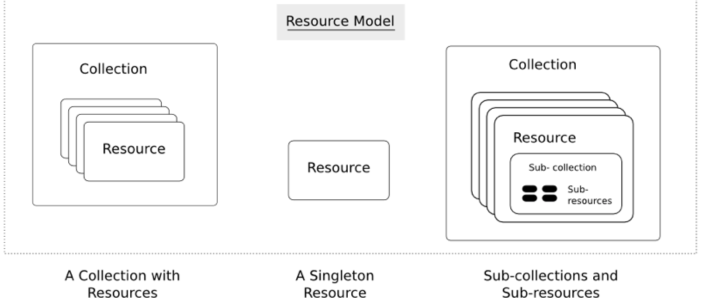
Applications, REST, Data Engineering Project

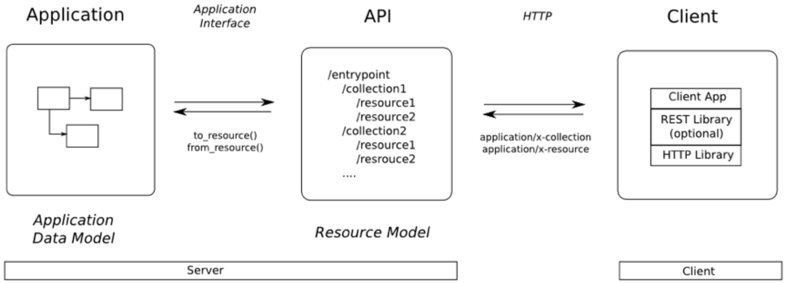
REST and Web Applications

Web Application Problem Statement

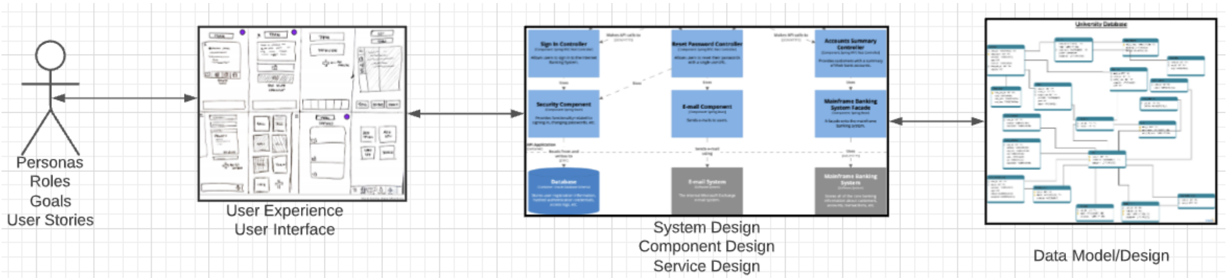
- We must build a system that supports create, retrieve, update and delete for IMDB and Game of Thrones Datasets.
- This requires implementing *create, retrieve, update and delete (CRUD)* for resources.

https://restful-api-design.readthedocs.io/en/latest/resources.html





- We will design, develop, test and deploy the system iteratively and continuously.
- There are four core domains.

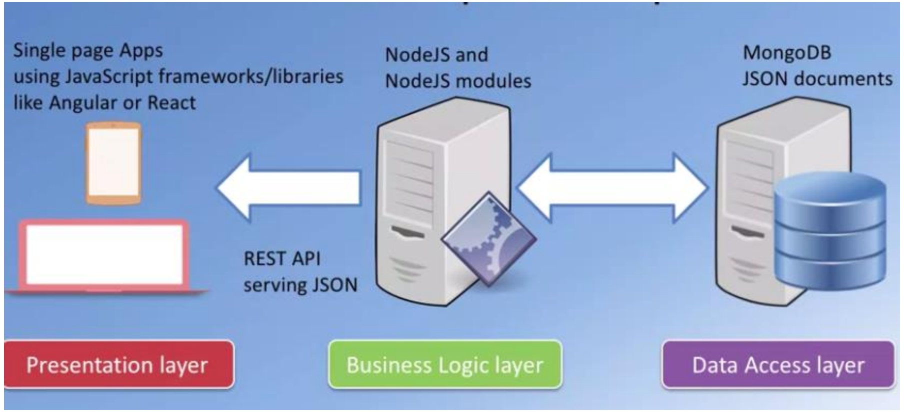


- In this course,
- We focus on the data dimension.
- We will get some insight into the other dimensions.

Interactive/Operational

- "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 (like using JavaScript, jQuery, Angular, or Vue)
  - Program a server (like using PHP, ASP, Python, or Node)
  - Program a database (like using SQL, SQLite, or MongoDB)"

  https://www.w3schools.com/whatis/whatis_fullstack.asp
- We will do a simple full stack app.
  - Three databases:
    - MySQL
    - MongoDB
    - Neo4j
  - The application tier will be Python and FastAPI.
  - The web UI will be Angular.
  - The primary focus is the data layer and application layer that access it.
  - I will provide a simple UI and template.

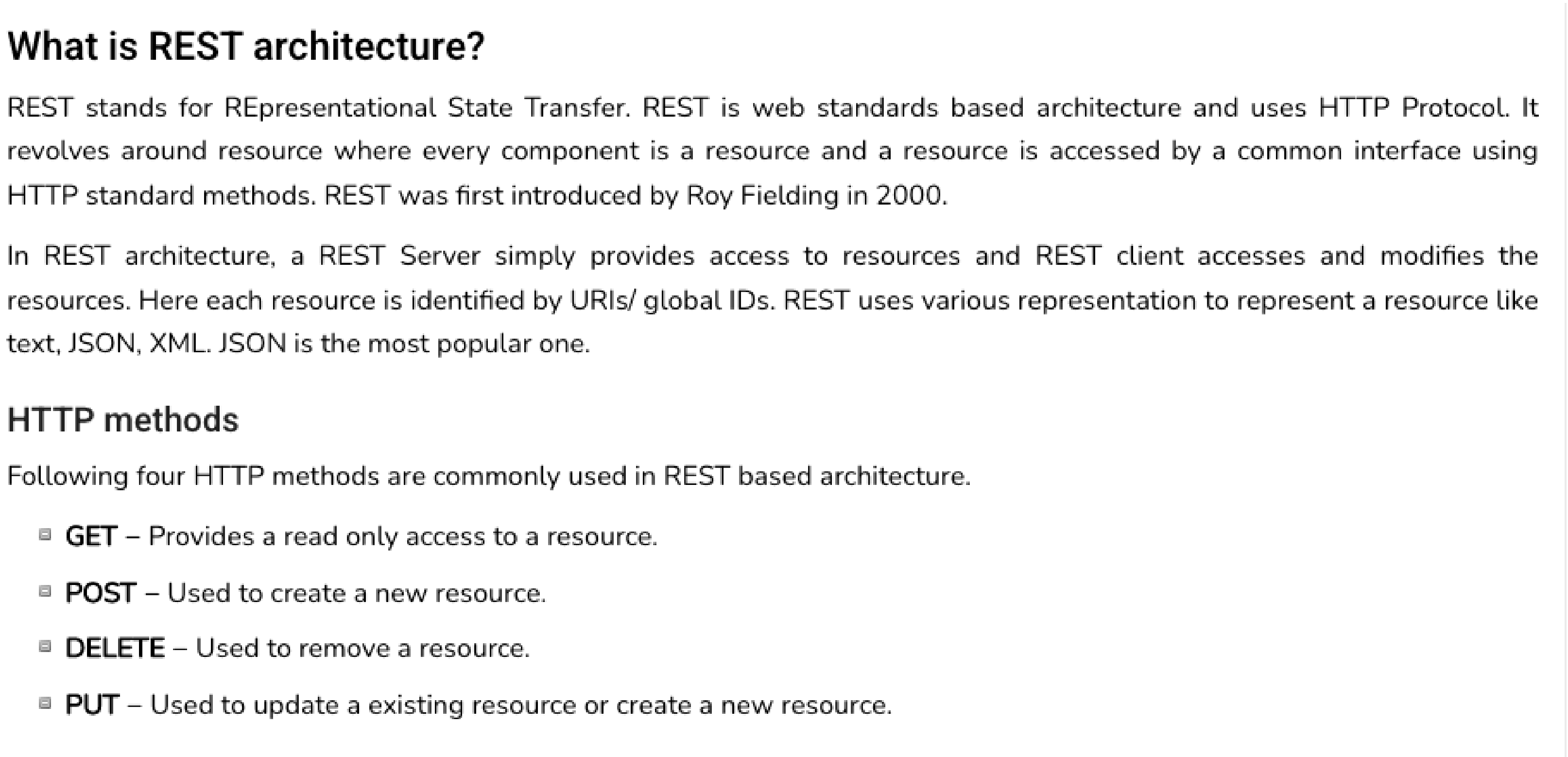


Data Modeling Concepts and REST

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/... ...
  - Host/database/table/pk

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)



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.

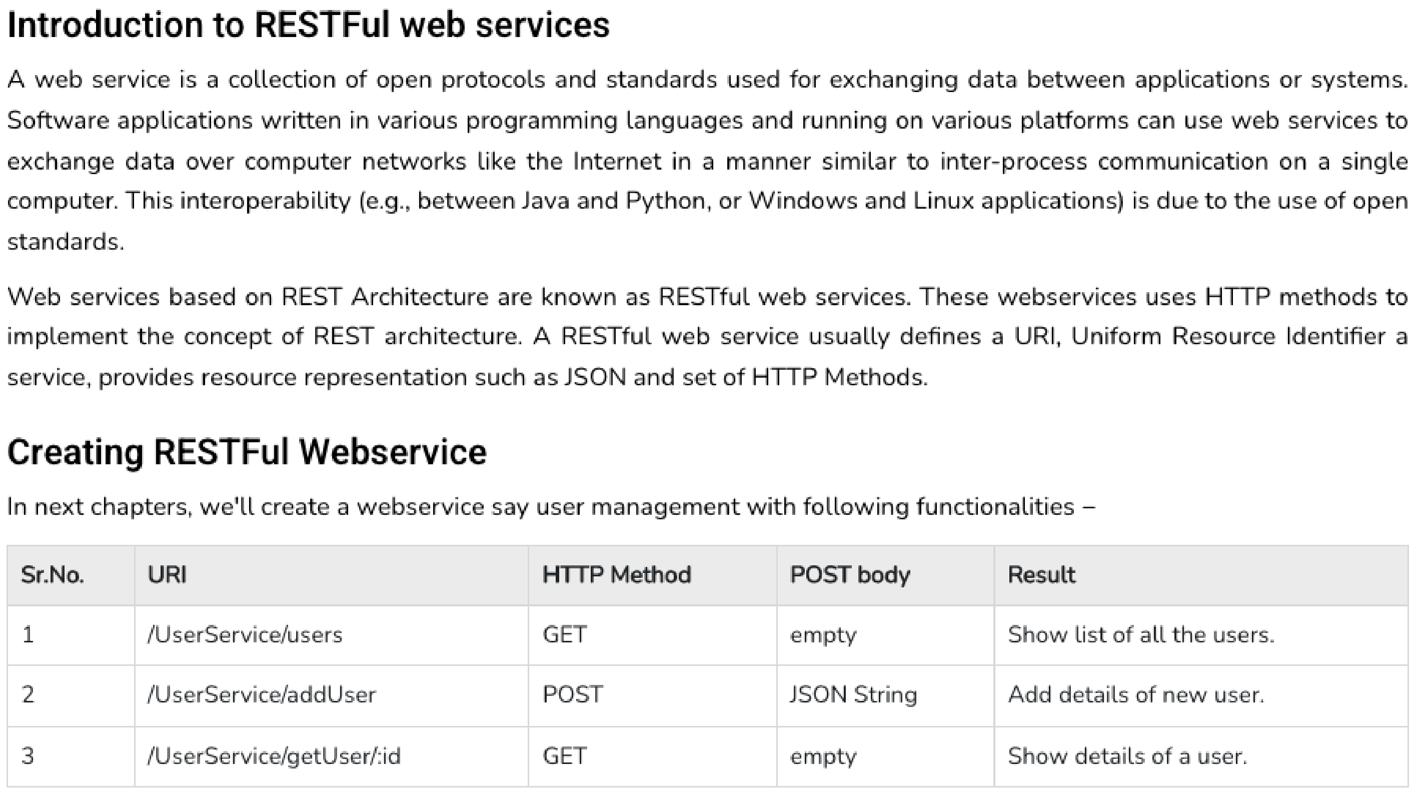
In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

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.

REST (https://www.tutorialspoint.com/restful/restful_introduction.htm)



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.

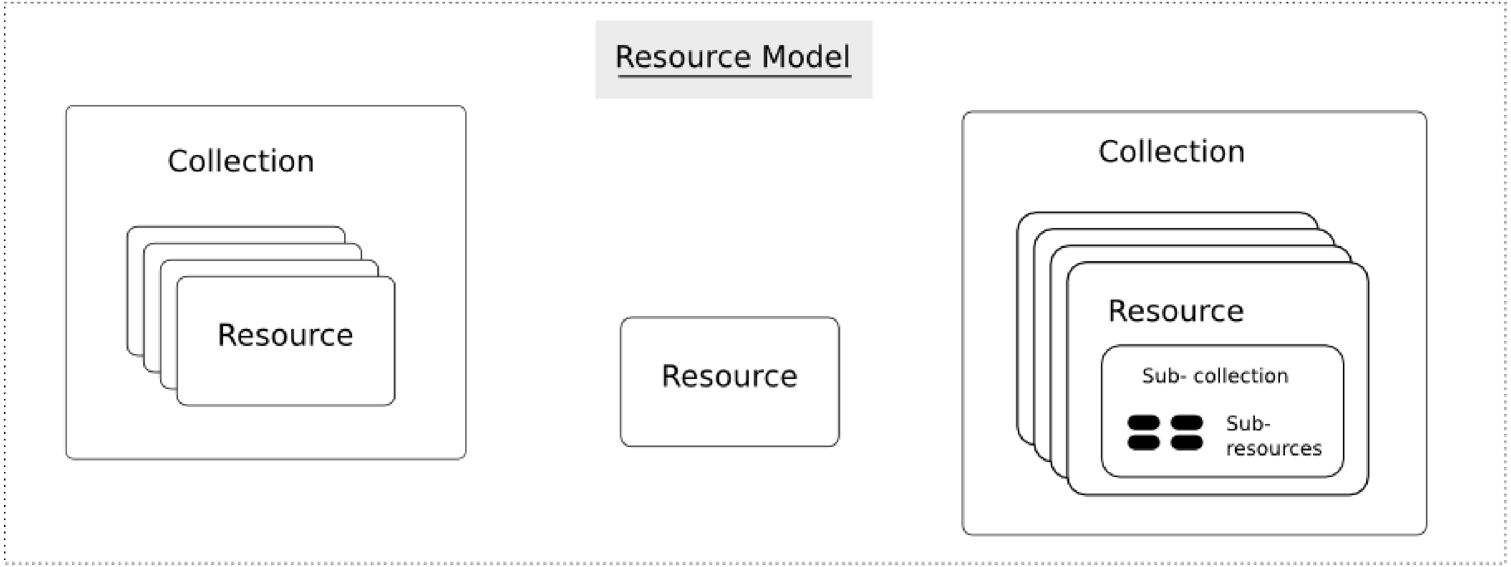
Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP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

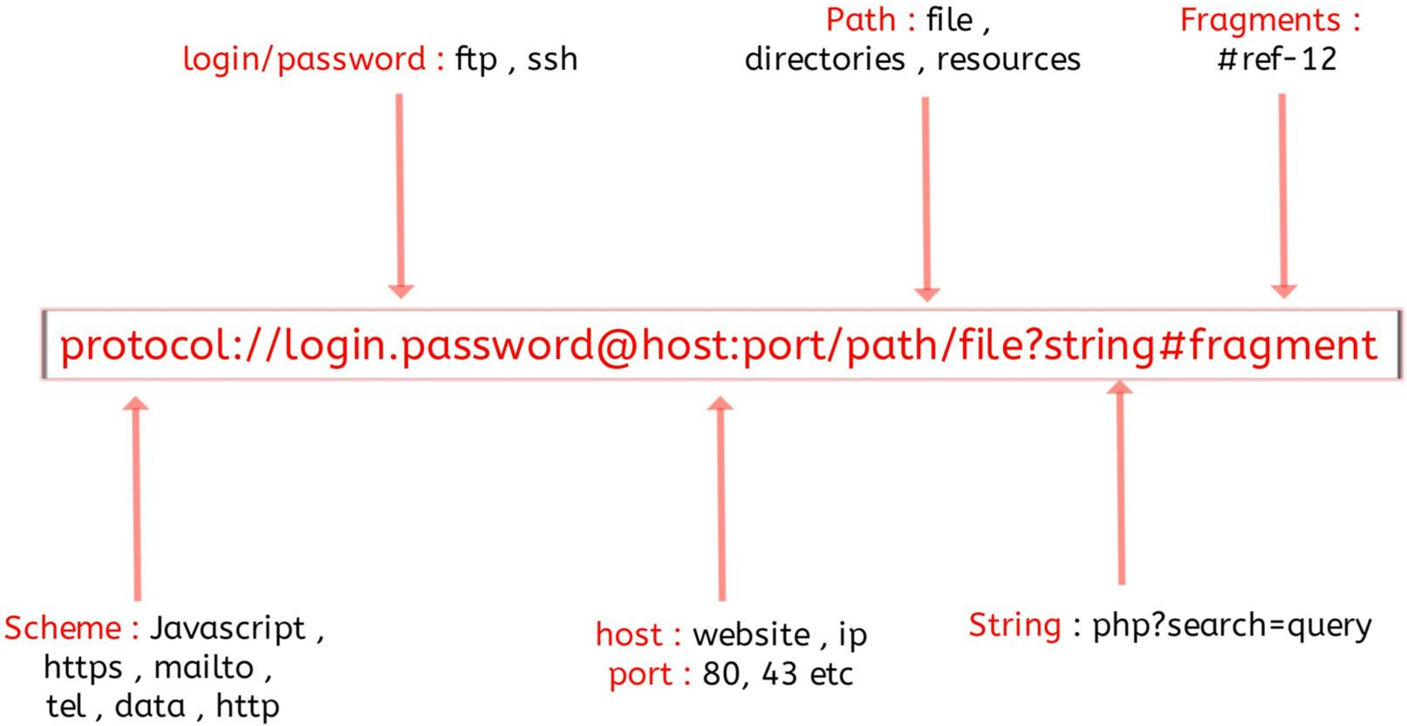


A Collection with
Resources

A Singleton
Resource

Sub-collections and
Sub-resources

URLs



login/password : ftp , ssh

Path : file , directories , resources

Fragments : #ref-12

protocol://login.password@host:port/path/file?string#fragment

Scheme : Javascript , https , mailto , tel , data , http

host : website , ip
port : 80, 43 etc

String : php?search=query