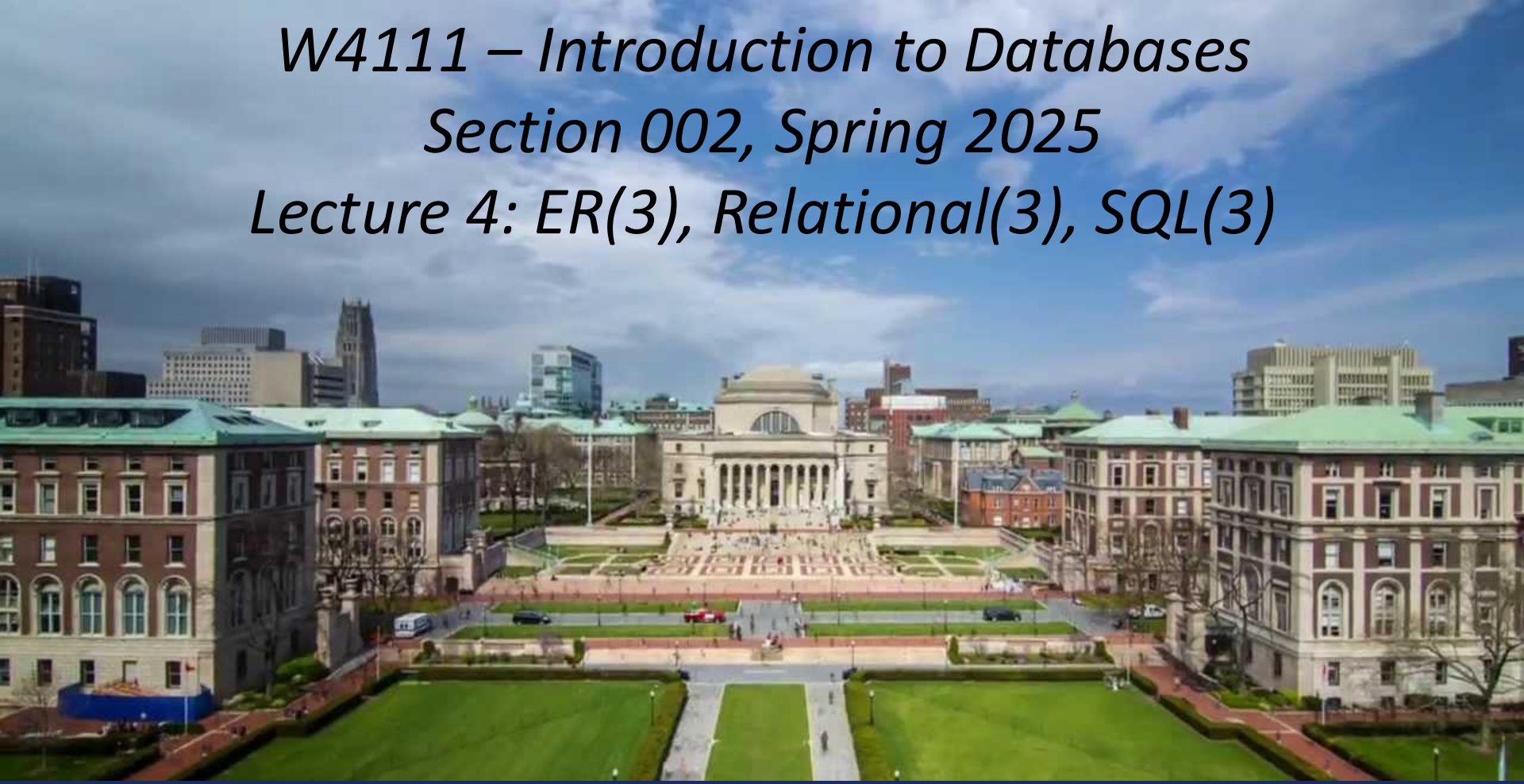
# W4111 – Introduction to Databases
# Section 002, Spring 2025
# Lecture 4: ER(3), Relational(3), SQL(3)



# *W4111 – Introduction to Databases*
# *Section 002, Spring 2025*
# *Lecture 4: ER(3), Relational(3), SQL(3)*

We will start in a few minutes.

# *Today's Contents*

# Contents

- Introduction and course updates.
- ER Modeling (3).
- SQL (3).
- Relational model and algebra (3).
- Homework and projects.

Swapping the order today because it is easier to understand some of the concepts in SQL.

# Introduction and Course Updates

# Course Updates

- Homework assignments:
  - I have been struggling with defining the homework assignments.
    - The initial cadence caused me to define homework assignments that contained questions on material from the next lecture. That is, the homework focused on content from two lectures, one of which we had not had.
    - This mismatch caused the whacky 1A and 1B cadence.
  - I am going to slightly modify the cadence and schedule.
    - Homework assignments will be every two weeks.
    - The assignment will contain questions only on material covered in lectures.
  - I will republish the schedule over the weekend.
- Final exam date:
  - The first day of class, I stated that the final exam would be the date of the last lecture (2025-MAY-02).
  - You lose the the “study days,” unfortunately.
  - I put a poll on Ed discussions to get a sense of student preference on a final the last day of class/last lecture versus holding the exam at the registrar assigned time of finals week.
  - Based on the results, I may suggest a change but accommodate students that have booked conflicts.

# ER Modeling

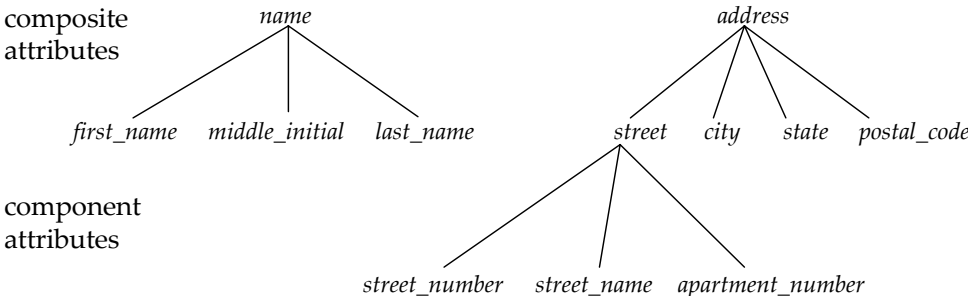
# Complex Attributes

# Complex Attributes

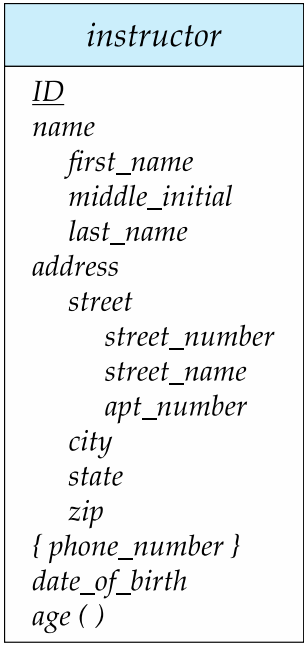
- Attribute types:
  - **Simple** and **composite** attributes.
  - **Single-valued** and **multivalued** attributes
    - Example: multivalued attribute: *phone_numbers*
  - **Derived** attributes
    - Can be computed from other attributes
    - Example:  age, given date_of_birth
- **Domain** – the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

# Composite Attributes

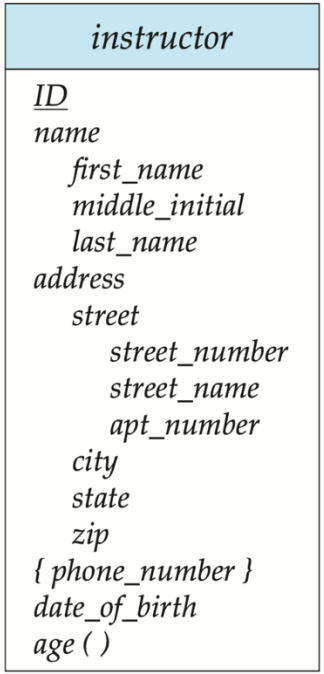
- Composite attributes allow us to divided attributes into subparts (other attributes).



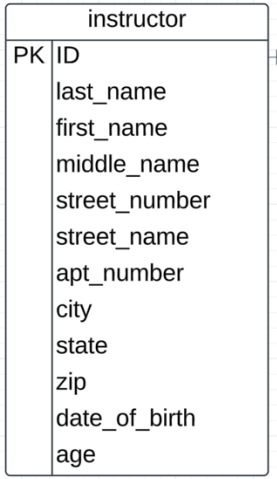
# Representing Complex Attributes in ER Diagram

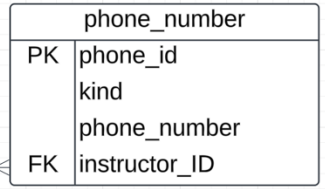


# Complex Attributes to Atomic




Simplest Mapping




age is generated

Notes:

- Composite and multi-valued are normally strings with delimiters when importing from files.
- Simple mapping is to:
  - Explode/flatten composites
  - Multi-valued become a separate, related table linked by a foreign key or associative entity.
- There are typically other things to consider, e.g.
  - Do multiple people have the same address?
  - Do multiple people have the same phone number?
  - Is the value of an attribute actually an enum or something that should be looked up? People are really bad a state names, for example.

# Data Cleanup/Validation

README | Code of conduct | ODbL-1.0 license


## Countries States Cities Database

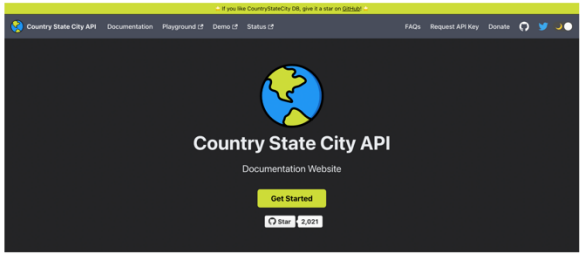
release v2.5 | size 1.7 GiB

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




simplemaps
Interactive Maps & Data

US Cities | Zips | Counties | Neighborhoods | World Cities | Pricing | All

## 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 | Most cities and towns | All populated places |

New 42 day free trial **Learn More** →

smarty | Products | Solutions | Resources | Company | Pricing | Cart | Log In | SIGN UP FREE

## 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)

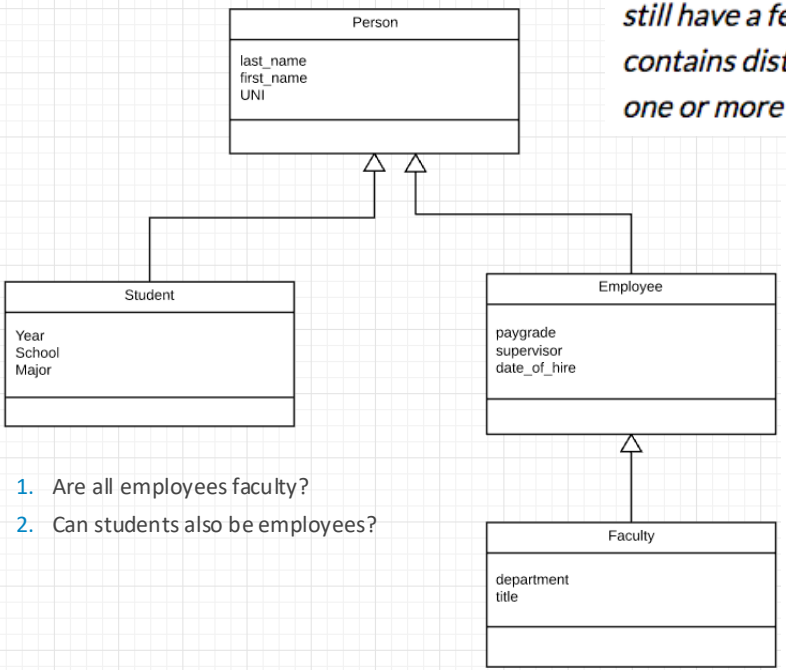
# Inheritance
# Specialization/Generalization

# Specialization

- Top-down design process; we designate sub-groupings within an entity set that are distinctive from other entities in the set.
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labeled ISA (e.g., *instructor* “is a” *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

# Inheritance/Specialization

*In the process of designing our entity relationship diagram for a database, we may find that attributes of two or more entities overlap, meaning that these entities seem very similar but still have a few differences. In this case, we may create a subtype of the parent entity that contains distinct attributes. A parent entity becomes a supertype that has a relationship with one or more subtypes.*




1. Are all employees faculty?
2. Can students also be employees?

The subclass association line is labeled with specialization constraints. Constraints are described along two dimensions:

**1 incomplete/complete**

- In an **incomplete** specialization only some instances of the parent class are specialized (have unique attributes). Other instances of the parent class have only the common attributes.
- In a **complete** specialization, every instance of the parent class has one or more unique attributes that are not common to the parent class.

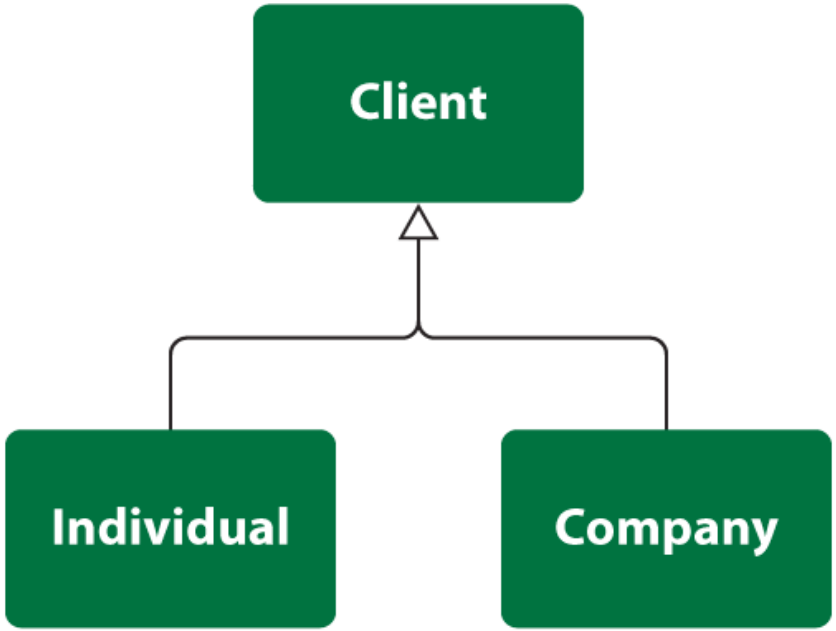
**2 disjoint/overlapping**

- In a **disjoint** specialization, an object could be a member of only one specialized subclass.
- In an **overlapping** specialization, an object could be a member of more than one specialized subclass.

http://www.vertabelo.com/blog/technical-articles/inheritance-in-a-relational-database

# Specialization

In class Client we distinguish two subtypes: Individual and Company. This specialization is disjoint (client can be an individual or a company) and complete (these are all possible subtypes for supertype).

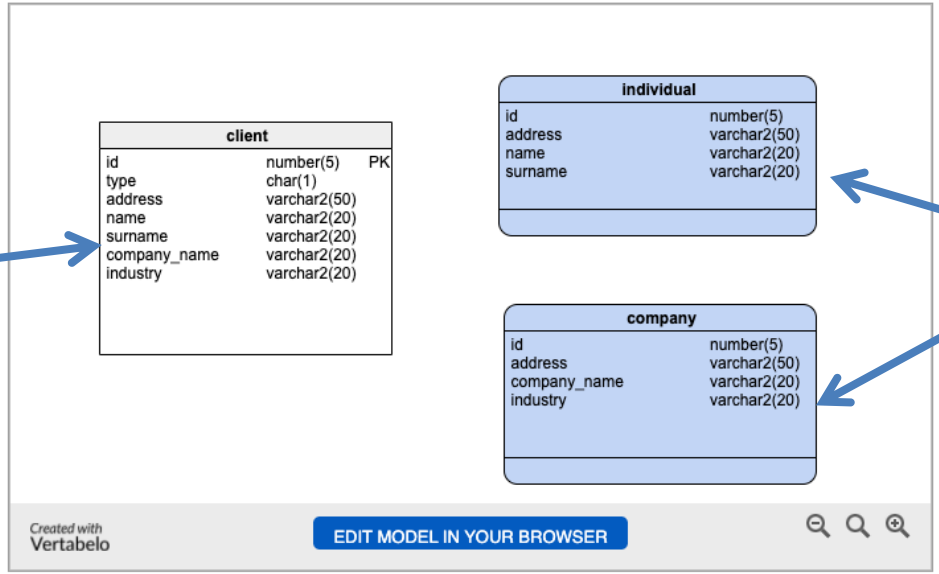


# One Table

## One table implementation

In a one table implementation, table `client` has attributes of both types.

The diagram below shows the table `client` and two views: `individual` and `company`:




Tables

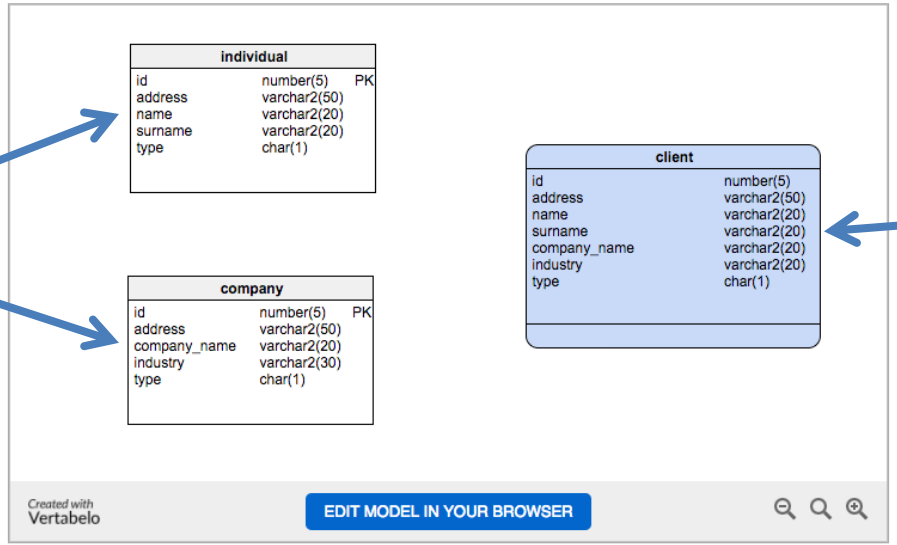
Views

# Two Table

## Two-table implementation

In a two-table implementation, we create a table for each of the subtypes. Each table gets a column for all attributes of the supertype and also a column for each attribute belonging to the subtype. Access to information in this situation is limited, that's why it is important to create a view that is the union of the tables. We can add an additional attribute called 'type' that describes the subtype.

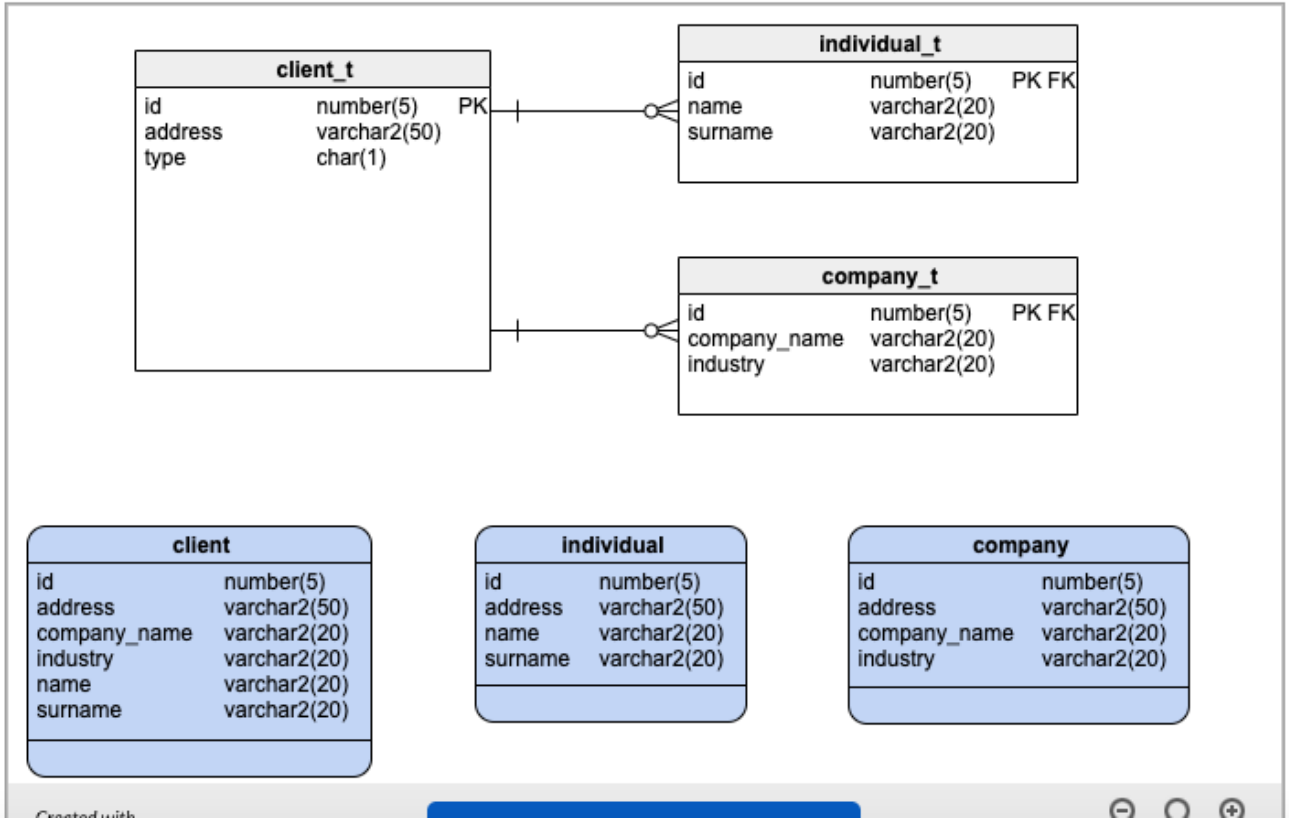
The diagram below presents two tables, `individual` and `company`, and a view (the blue one) called `client`.




Tables

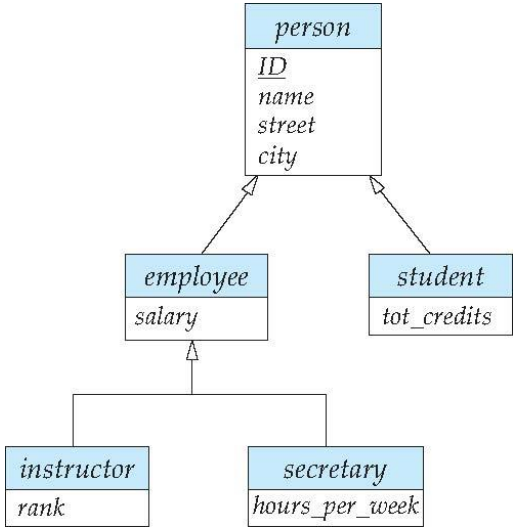
View

# Three Table



# Specialization Example

- **Overlapping** – *employee* and *student*
- **Disjoint** – *instructor* and *secretary*
- Total and partial



# Representing Specialization via Schemas

- Method 1:
  - Form a schema for the higher-level entity
  - Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

| schema | attributes |
|---|---|
| person | ID, name, street, city |
| student | ID, tot_cred |
| employee | ID, salary |

  - Drawback: getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema

# Representing Specialization as Schemas (Cont.)

- Method 2:
  - Form a schema for each entity set with all local and inherited attributes

| schema | attributes |
|---|---|
| person | ID, name, street, city |
| student | ID, name, street, city, tot_cred |
| employee | ID, name, street, city, salary |

  - Drawback: *name, street* and *city* may be stored redundantly for people who are both students and employees

# Generalization

- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably.

# Completeness constraint

- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
  - **total**: an entity must belong to one of the lower-level entity sets
  - **partial**: an entity need not belong to one of the lower-level entity sets

# Completeness constraint (Cont.)

- Partial generalization is the default.
- We can specify total generalization in an ER diagram by adding the keyword **total** in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrow-head to which it applies (for a total generalization), or to the set of hollow arrow-heads to which it applies (for an overlapping generalization).
- The *student* generalization is total: All student entities must be either graduate or undergraduate. Because the higher-level entity set arrived at through generalization is generally composed of only those entities in the lower-level entity sets, the completeness constraint for a generalized higher-level entity set is usually total

# Views

What are these views of which you speak? Prithee tell us more!

# SQL

# Introduction to Views

# Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  **select** *ID*, *name*, *dept_name*
  **from** *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.

# View Definition

- A view is defined using the **create view** statement which has the form

  **create view** *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

# View Definition and Use

- A view of instructors without their salary

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

- Find all instructors in the Biology department

```
select name
from faculty
where dept_name = 'Biology'
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

# Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation $v_1$ is said to ***depend directly*** on a view relation $v_2$ if $v_2$ is used in the expression defining $v_1$
- A view relation $v_1$ is said to ***depend on*** view relation $v_2$ if either $v_1$ depends directly to $v_2$ or there is a path of dependencies from $v_1$ to $v_2$
- A view relation $v$ is said to be ***recursive*** if it depends on itself.

# Views Defined Using Other Views

- **create view** ***physics_fall_2017*** **as**
  **select** *course.course_id*, *sec_id*, *building*, *room_number*
  **from** *course*, *section*
  **where** *course.course_id* = *section.course_id*
  **and** *course.dept_name* = 'Physics'
  **and** *section.semester* = 'Fall'
  **and** *section.year* = '2017';

- **create view** ***physics_fall_2017***_*watson* **as**
  **select** *course_id*, *room_number*
  **from** ***physics_fall_2017***
  **where** *building*= 'Watson';

# View Expansion

- Expand the view :

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from physics_fall_2017
    where building= 'Watson'
```

- To:

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2017')
    where building= 'Watson';
```

# View Expansion (Cont.)

- A way to define the meaning of views defined in terms of other views.
- Let view $v_1$ be defined by an expression $e_1$ that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

  **repeat**
  
  Find any view relation $v_i$ in $e_1$
  
  Replace the view relation $v_i$ by the expression defining $v_i$
  
  **until** no more view relations are present in $e_1$
- As long as the view definitions are not recursive, this loop will terminate

# Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called **Materialized view**:
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.

# Update of a View

- Add a new tuple to *faculty* view which we defined earlier

  **insert into** *faculty*

  **values** ('30765', 'Green', 'Music');

- This insertion must be represented by the insertion into the *instructor* relation
  - Must have a value for salary.
- Two approaches
  - Reject the insert
  - Inset the tuple

    ('30765', 'Green', 'Music', null)

    into the *instructor* relation

# Some Updates Cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  **select** *ID*, *name*, *building*
  **from** *instructor*, *department*
  **where** *instructor.dept_name= department.dept_name*;
- **insert into** *instructor_info*
  **values** ('69987', 'White', 'Taylor');
- Issues
  - Which department, if multiple departments in Taylor?
  - What if no department is in Taylor?

# And Some Not at All

- **create view** *history_instructors* **as**
  **select** *
  **from** *instructor*
  **where** *dept_name*= 'History';
- What happens if we insert

  ('25566', 'Brown', 'Biology', 100000)

  into *history_instructors?*

# View Updates in SQL

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause.

# 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*;

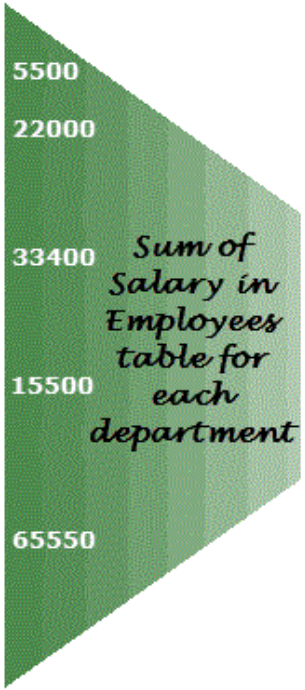
| *ID* | *name* | *dept_name* | *salary* |
|---|---|---|---|
| 76766 | Crick | Biology | 72000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| 98345 | Kim | Elec. Eng. | 80000 |
| 12121 | Wu | Finance | 90000 |
| 76543 | Singh | 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

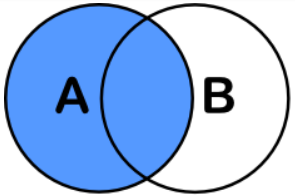
- Find the names and average salaries of all departments whose average salary is greater than 42000

  **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

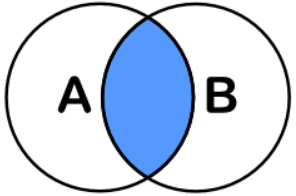
# Aggregate Functions

# One Way to Think About Joins

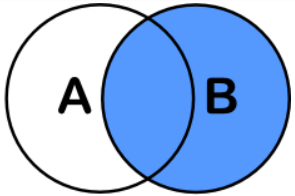



```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
```

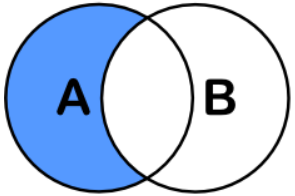
CHEATSHEET
SQL
JOINS



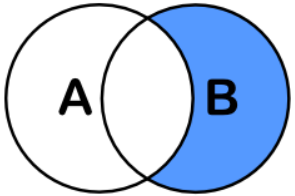

```
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key
```



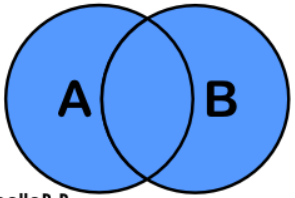

```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
```



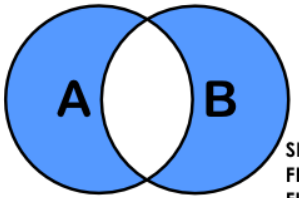

```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL
```




```
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
```

# Outer Join

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

# Outer Join Examples

- Relation *course*

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

- Relation *prereq*

| *course_id* | *prereq_id* |
|---|---|
| BIO-301 | BIO-101 |
| CS-190 | CS-101 |
| CS-347 | CS-101 |

- Observe that

  *course* information is missing CS-437

  *prereq* information is missing CS-315

- *X*

# Left Outer Join

- *course* **natural left outer join** *prereq*

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

- In relational algebra: *course* ⟕ *prereq*

# Right Outer Join

- *course* **natural right outer join** *prereq*

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

- In relational algebra: *course* ⟖ *prereq*

# Full Outer Join

- *course* **natural full outer join** *prereq*

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

- In relational algebra: *course* ⟗ *prereq*

# Joined Types and Conditions

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

| *Join types* |
| --- |
| **inner join** |
| **left outer join** |
| **right outer join** |
| **full outer join** |

| *Join conditions* |
| --- |
| **natural** |
| **on** <predicate> |
| **using** $(A_1, A_2, \ldots, A_n)$ |

# Joined Relations – Examples

- *course* **natural right outer join** *prereq*

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

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

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

# Joined Relations – Examples

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

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

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

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

# Joined Relations – Examples

- *course* **natural right outer join** *prereq*

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

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

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

# More Fun with Subqueris

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

(5 < **some** | 0 / 5 / 6 | ) = true (read: 5 < some tuple in the relation)

(5 < **some** | 0 / 5 | ) = false

(5 = **some** | 0 / 5 | ) = true

(5 ≠ **some** | 0 / 5 | ) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **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 <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <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 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≢ **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

# Relational Model

# 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?

- $\tau$ order by
- $\gamma$ group by
- $\neg$ negation
- $\div$ set division
- $\bowtie$ natural join, theta-join
- $⟕$ left outer join
- $⟖$ right outer join
- $⟗$ full outer join
- $\ltimes$ left semi join
- $\rtimes$ 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 operatorsbecause they may be common internships/job interview questions.

# What are all those other Symbols?

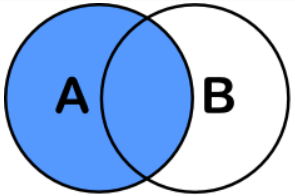
- τ order by
- γ group by
- ¬ negation
- ÷ set division
- ⋈ natural join, theta-join
- ⟕ left outer join
- ⟖ right outer join
- ⟗ full outer join
- ⋉ left semi join
- ⋊ right semi join
- ▷ anti-join

```
σ student_dept='Comp. Sci.' ∨ student_dept='Elec. Eng.'
(
  π student_id, student_name, student_dept,
    advisor_id, advisor_name, advisor_dept
  (
    (
      π student_id←ID, student_name←name,
        student_dept←dept_name, i_id
            (student ⋈ ID=s_id advisor)
    )
    ⋈ i_id=advisor_id
    (
      π advisor_id←ID, advisor_name←name, advisor_dept←dept_name
            (instructor ⋈ ID=i_id advisor)
    )
  )
)
```

Result

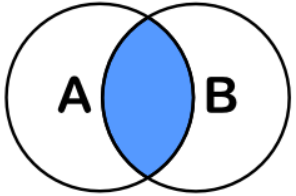
| student_id | student_name | student_dept | advisor_id | advisor_name | advisor_dept |
|---|---|---|---|---|---|
| 128 | Zhang' | Comp. Sci.' | 45565 | Katz' | Comp. Sci.' |
| 12345 | Shankar' | Comp. Sci.' | 10101 | Srinivasan' | Comp. Sci.' |
| 76543 | Brown' | Comp. Sci.' | 45565 | Katz' | Comp. Sci.' |
| 76653 | Aoi' | Elec. Eng.' | 98345 | Kim' | Elec. Eng.' |
| 98765 | Bourikas' | Elec. Eng.' | 98345 | Kim' | Elec. Eng.' |

# One Way to Think About Joins

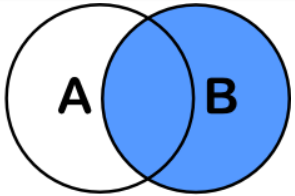



```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
```

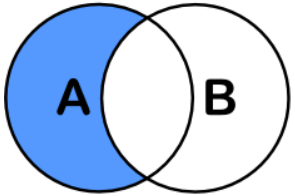
CHEATSHEET
SQL
JOINS



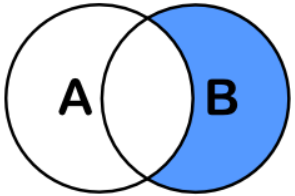

```
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key
```



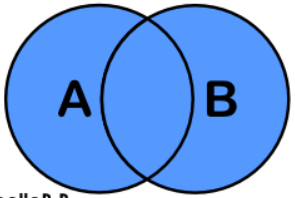

```
SELECT <auswahl>
FROM tabelleA A
RIGHT JOIN tabelleB B
ON A.key = B.key
```



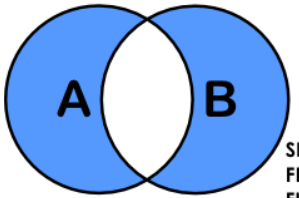

```
SELECT <auswahl>
FROM tabelleA A
LEFT JOIN tabelleB B
ON A.key = B.key
WHERE B.key IS NULL
```




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



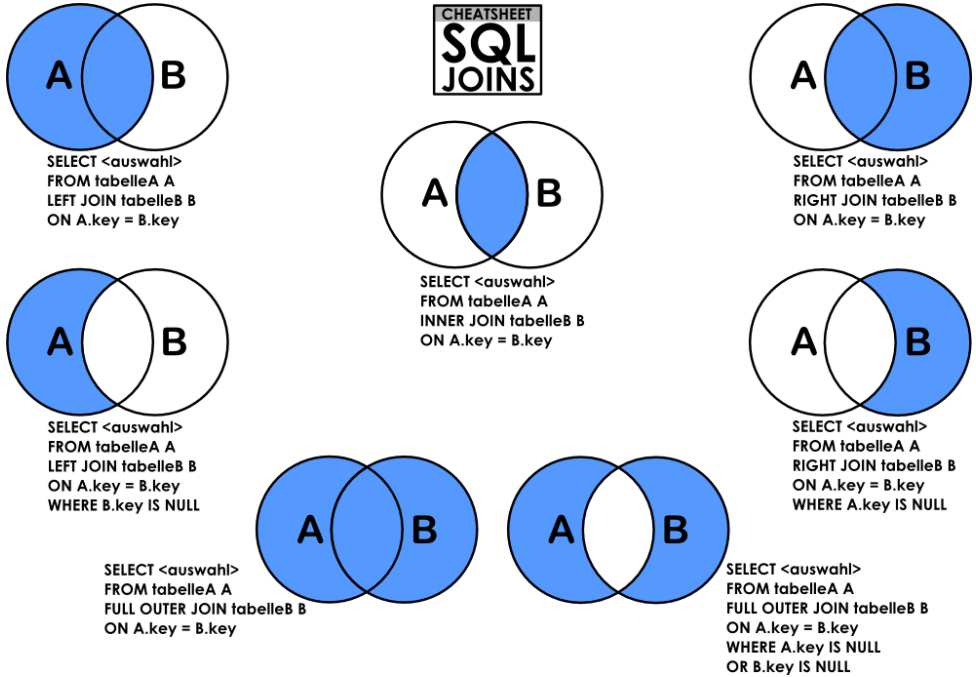

```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
```




```
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
```

# 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 I want is:
  - All the rows that matched.
  - And the rows from A that did not match?
  - OUTER JOIN (⟕, ⟖)



# 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 course and prereq to get
  - course info
  - prereq_id
- Join with
  - course using prereq_id
  - 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.

# 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))
)

# 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
                )
        )
```

# Relational Algebra

- We will do more examples in lectures and HW assignments.
- 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.
- You will be able to use a “cheat sheet” on exams. We will focus on “did you know what to do” and not “did you get the syntax completely correct.”

# Codd's 12 Rules

# 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.

# Random Thoughts on Projects

# Projects

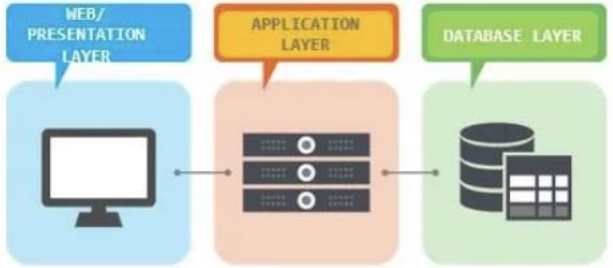
- The programming track will implement a simple, full stack web application.

## Full-stack Web Developer

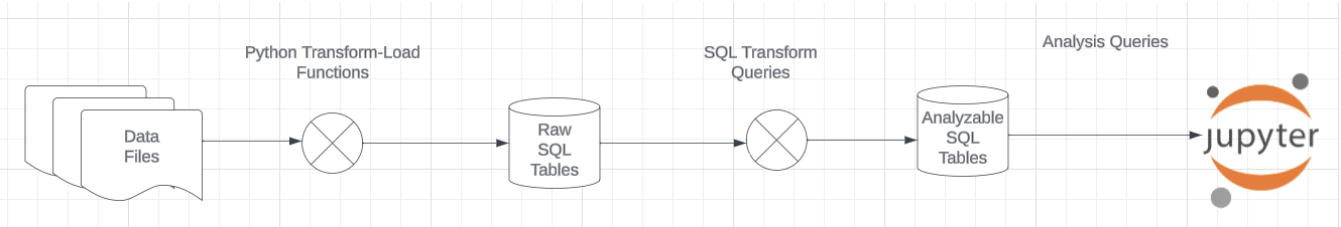
A full-stack web developer is a person who can develop both **client** and **server** software.

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.

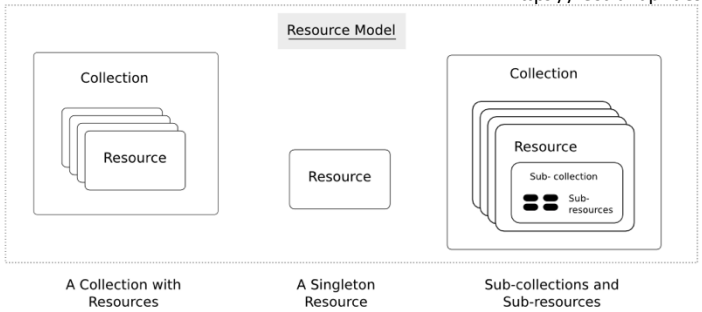



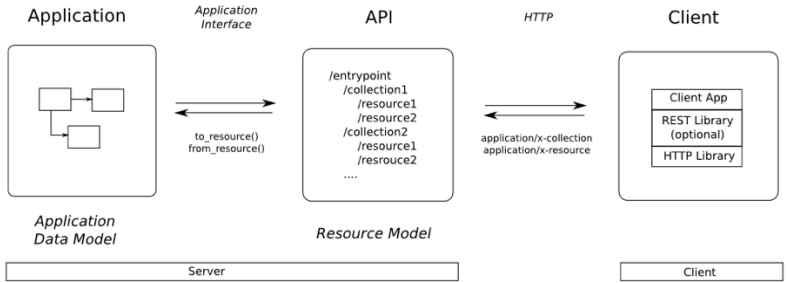
# REST and Web Applications

# Web Application Problem Statement

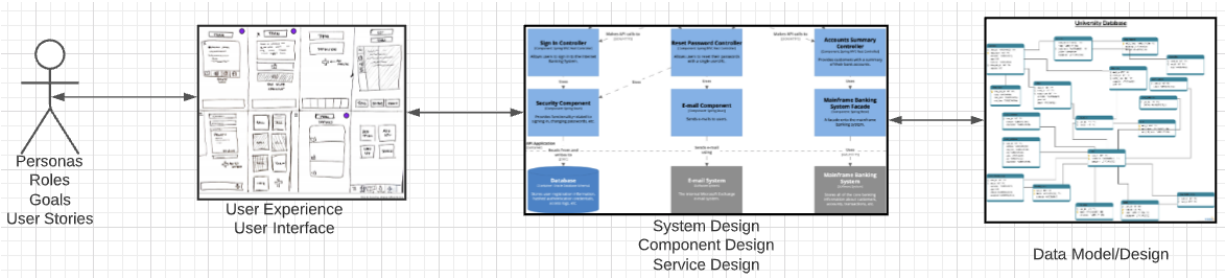
- We must build a system that supports create, retrieve, update and delete for IMDB and Game of Thrones Datasets.
- This requires implementing *create, retrieve, update and delete (CRUD)* for resources.

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.

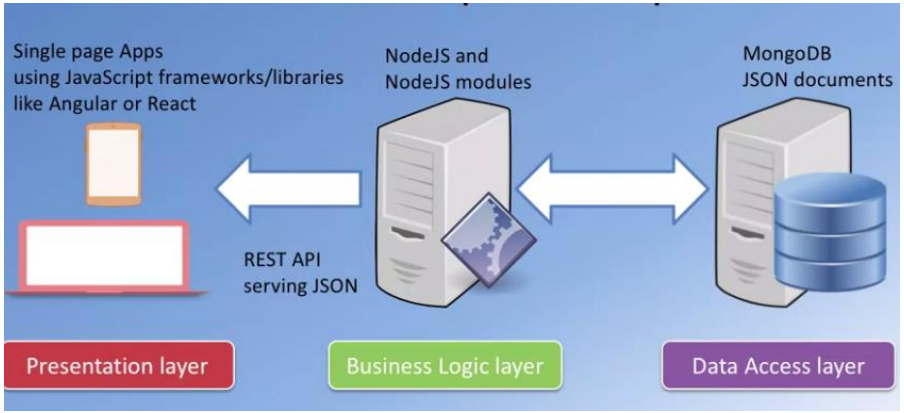



- In this course,
- We focus on the data dimension.
- We will get some insight into the other dimensions.

# 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.

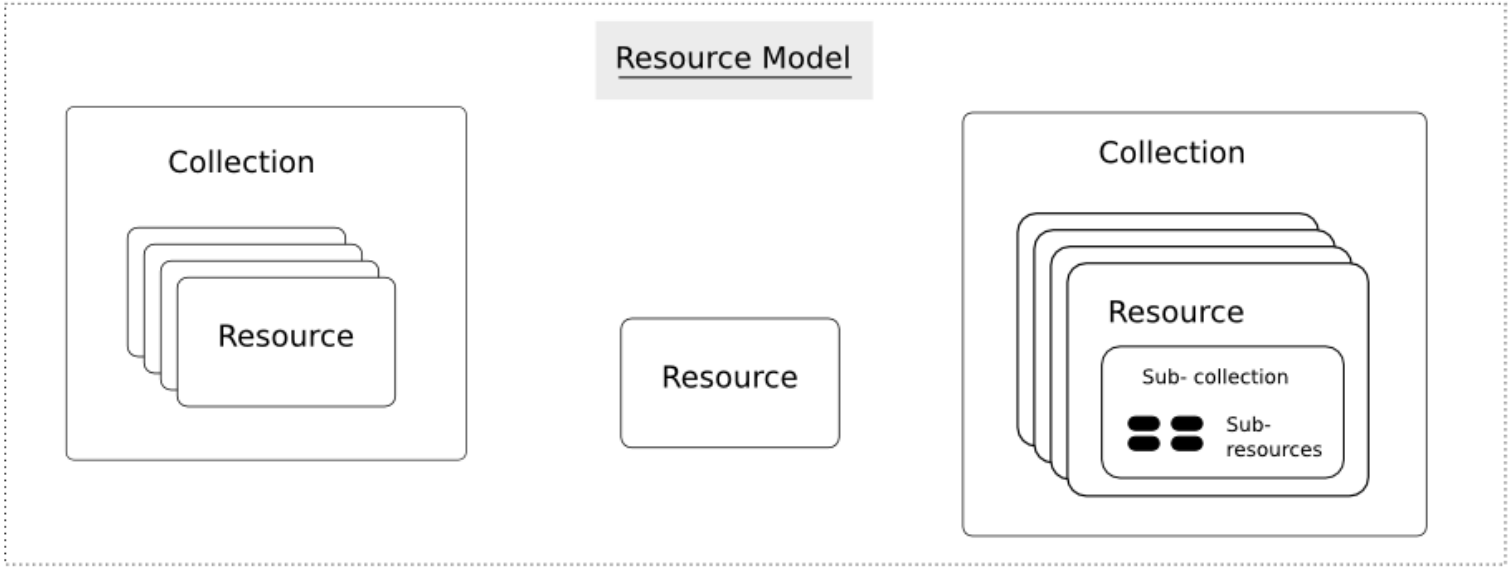
Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP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

## Creating RESTFul Webservice

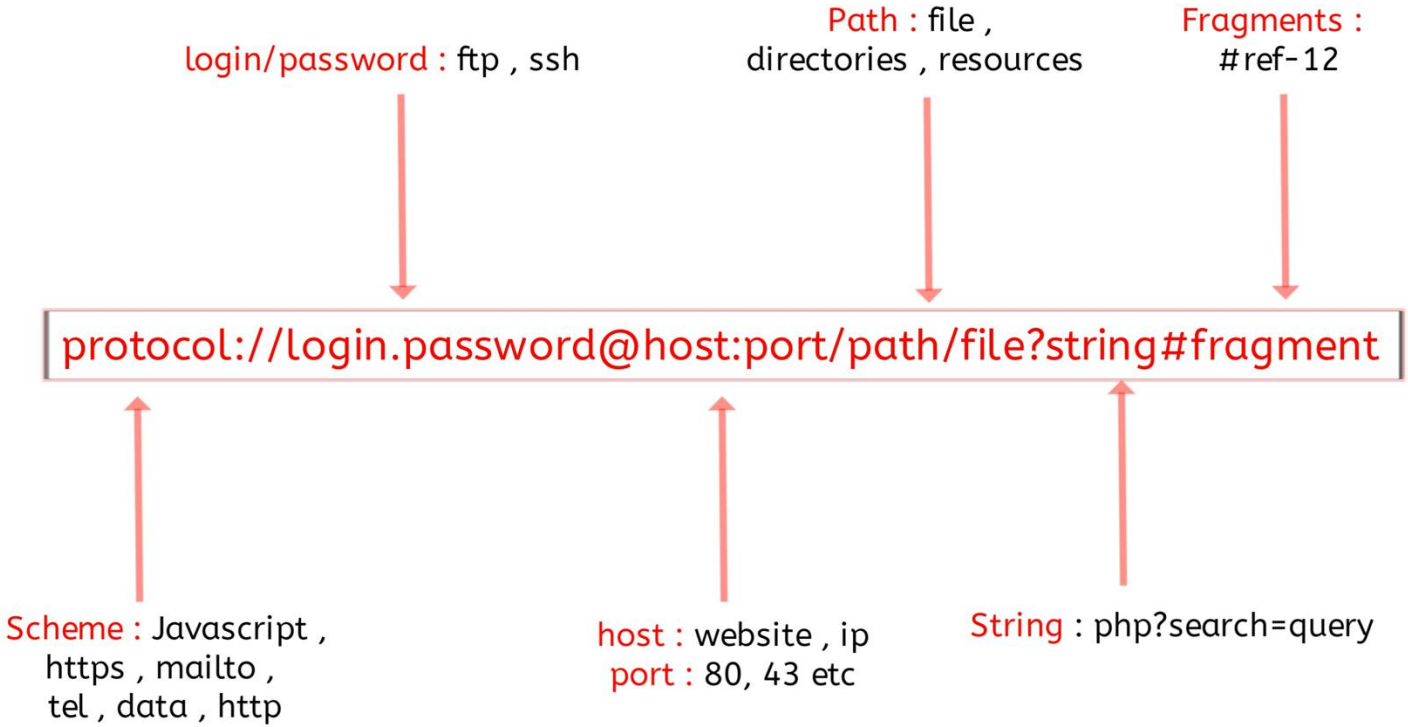
In next chapters, we'll create a webservice say user management with following functionalities –

| 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



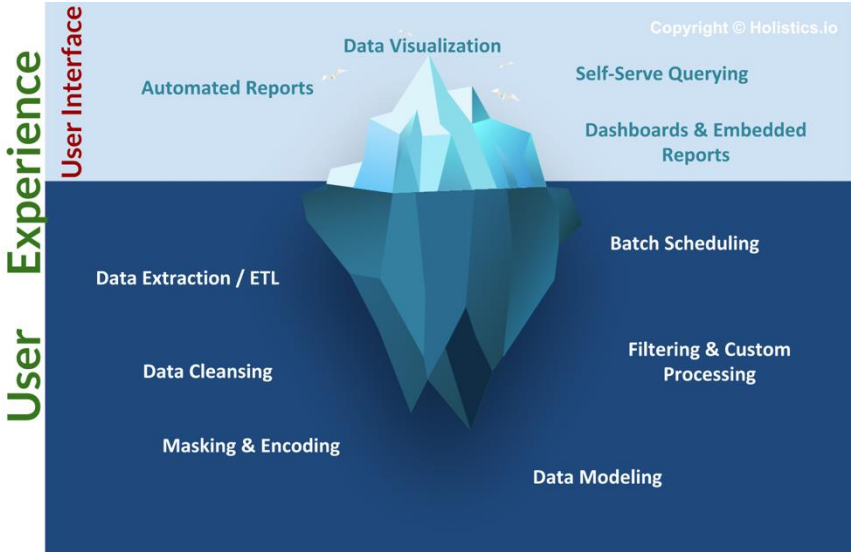
# URLs

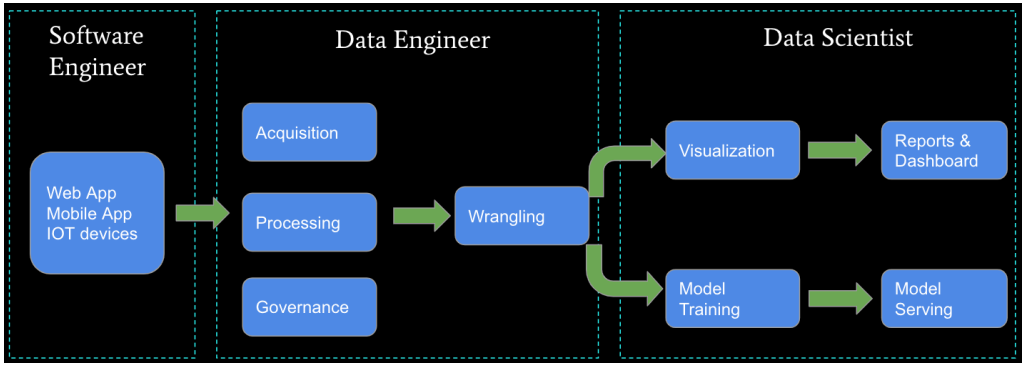


# Walk Through the Project Template

# Data Engineering

# Business Intelligence, Insight, Analysis, … …





- The “fun” stuff in data science and AI/ML is the “tip of the iceberg.”
- Data engineering is a necessary condition for producing analyzable data. This is often more than 80% of the hard work.
- We will do some small data engineering projects in this course.

# Walk Through the Sprint 1 Notebook