W4111 – Introduction to Databases
Section 002, Spring 2023

Lecture 7: ER, Relational, SQL
Advanced Topics and Scenarios

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

Contents

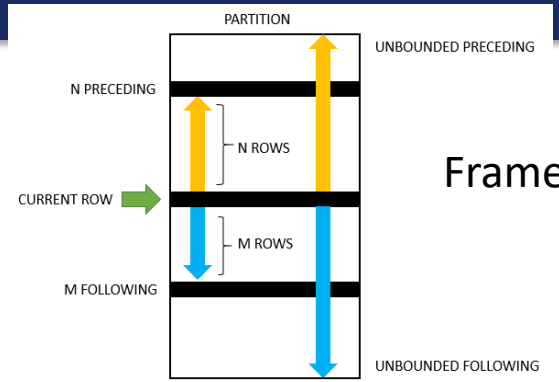
Contents

- REST, HATEOAS continued:
  - Review previously discussed concepts.
  - New concepts: Links, Location, … …
- Window Functions
- Advanced ER Modeling

Some Advanced and Miscellaneous SQL

Window Functions

MySQL Window Functions



Frames

- Syntax:

  window_function_name(expression) OVER (

  [partition_defintion]

  [order_definition]

  [frame_definition]

)

- PARTITION BY <expression>[{,<expression>...}]
  - The partition_clause breaks up the rows into chunks or partitions. Two partitions are separated by a partition boundary.
  - The window function is performed within partitions and re-initialized when crossing the partition boundary.
- ORDER BY <expression> [ASC|DESC], [{,<expression>...}]
  - The ORDER BY clause specifies how the rows are ordered within a partition.
- frame_unit {<frame_start>|<frame_between>}
  - A frame is a subset of the current partition.
  - A frame is defined with respect to the current row, which allows a frame to move within a partition depending on the position of the current row within its partition.

Comments

- Even by SQL standards and my explanations, this is baffling.
- The only way to learn is practice.
- Let’s switch to the notebook.

Interesting Things not Supported by MySQL

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- The following constraints, can be expressed using assertions:
- For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot
- An assertion in SQL takes the form:

  create assertion <assertion-name> **check** (<predicate>);

User-Defined Types

- **create type** construct in SQL creates user-defined type

  create type *Dollars* **as numeric (12,2) final**

- Example:

  create table *department*
  (*dept_name* **varchar** (20),
  building **varchar** (15),
  budget Dollars);

Domains

- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- Example:

```
create domain degree_level varchar(10)
    constraint degree_level_test
        check (value in ('Bachelors', 'Masters', 'Doctorate'));
```

Some Advanced ER Modeling (And Implementing It)

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 s*ec_course*; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.

Weak Entity Sets (Cont.)

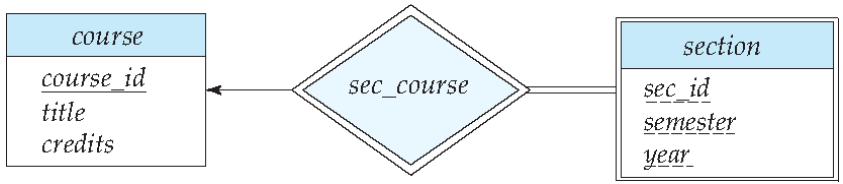
- An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *section_id*, *year*, and *semester.*
  - However, the entity set *section* then does not have enough attributes to identify a particular *section* 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.*

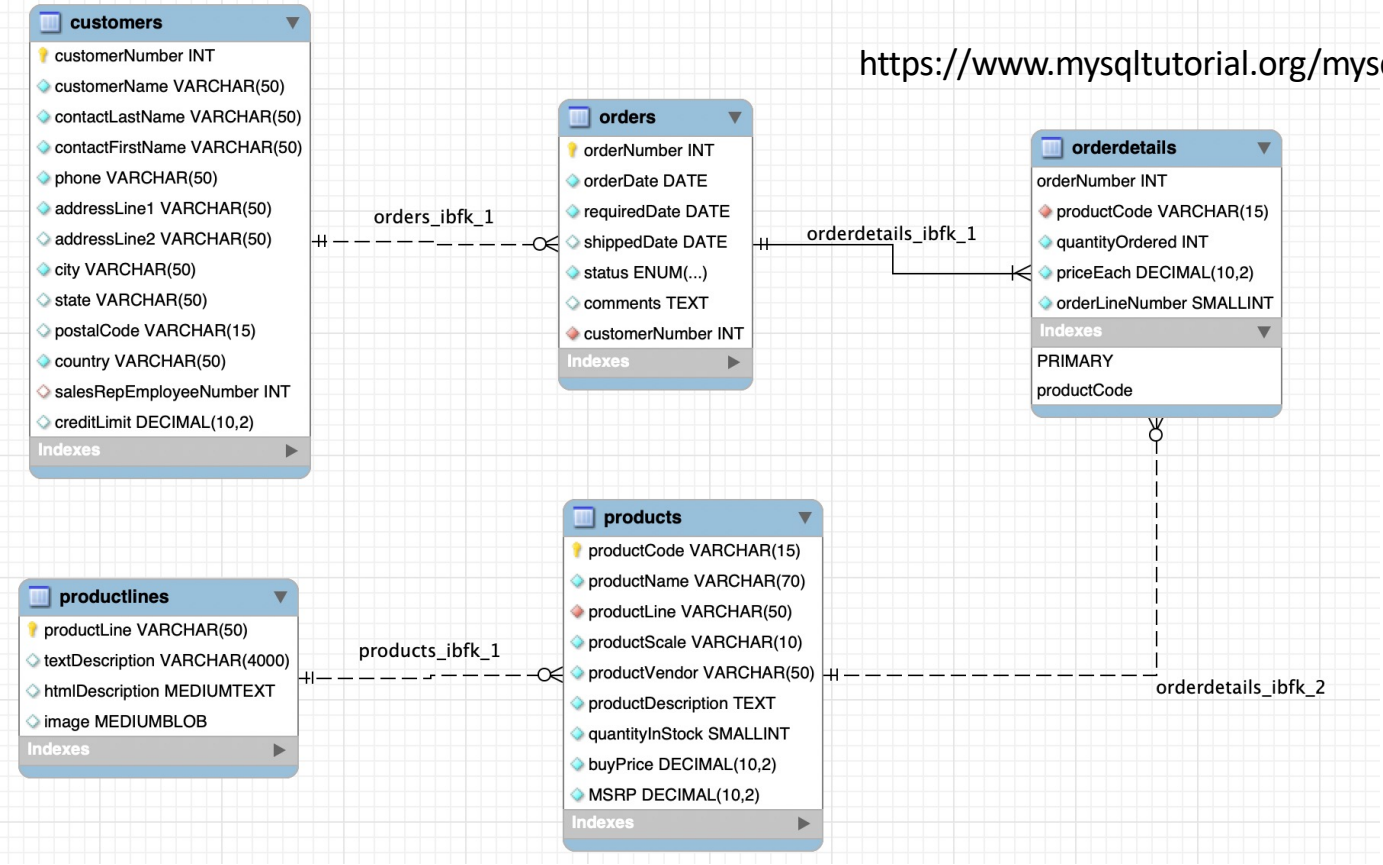
Expressing Weak Entity Sets

- 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*)



An Example – Classic Models

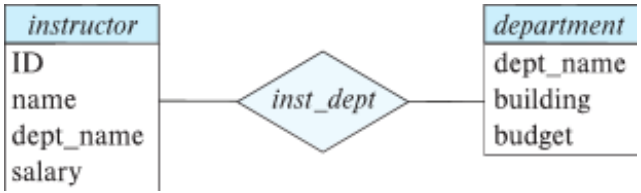
https://www.mysqltutorial.org/mysql-sample-database.aspx/



Redundant Attributes

Some of this only makes sense in this notation because it support relationships. Crow's foot does not have this problem.

- Suppose we have entity sets:
  - *instructor*, with attributes: *ID*, *name*, *dept_name, salary*
  - *department,* with attributes: *dept_name, building, budget*
- We model the fact that each instructor has an associated department using a relationship set *inst_dept*
- The attribute *dept_name* in *instructor* replicates information present in the relationship and is therefore redundant
  - and needs to be removed.
- BUT: when converting back to tables, in some cases the attribute gets reintroduced, as we will see later.



Design Alternatives

- In designing a database schema, we must ensure that we avoid two major pitfalls:
  - Redundancy: a bad design may result in repeat information.
    - **Redundant representation of information may lead to data inconsistency among the various copies of information**
  - Incompleteness: a bad design may make certain aspects of the enterprise difficult or impossible to model.
- Avoiding bad designs is not enough. There may be a large number of good designs from which we must choose.

Emphasis Added

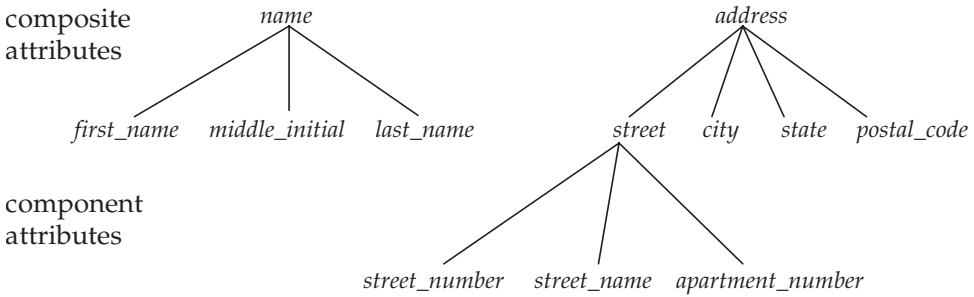
Complex Attributes

- 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

DFF: Jump into notebook and show examples from IMDB.

Composite Attributes

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



Example from IMDB

- Consider name_basics

10 rows | Tx: Auto | DDL | CSV

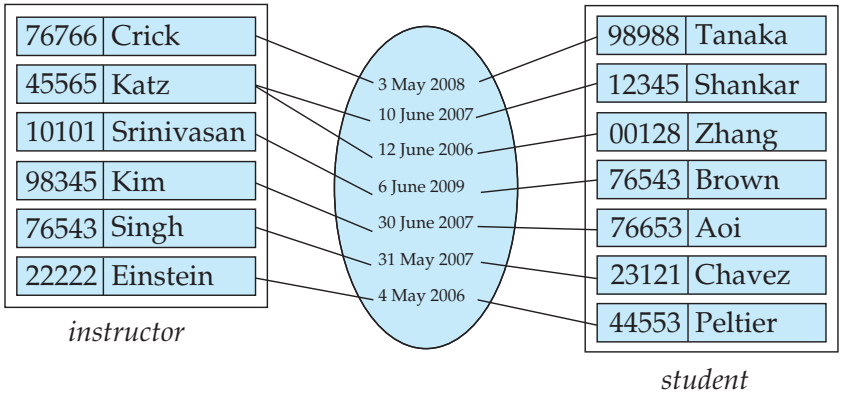
	nconst	primaryName	birthYear	deathYear	primaryProfession	knownForTitles
1	nm0000001	Fred Astaire	1899	1987	soundtrack,actor,miscellaneous	tt0050419,tt0031983,tt0072308,tt0053137
2	nm0000002	Lauren Bacall	1924	2014	actress,soundtrack	tt0071877,tt0117057,tt0037382,tt0038355
3	nm0000003	Brigitte Bardot	1934	<null>	actress,soundtrack,music_department	tt0049189,tt0056404,tt0057345,tt0054452
4	nm0000004	John Belushi	1949	1982	actor,soundtrack,writer	tt0077975,tt0072562,tt0080455,tt0078723
5	nm0000005	Ingmar Bergman	1918	2007	writer,director,actor	tt0050986,tt0060827,tt0069467,tt0050976
6	nm0000006	Ingrid Bergman	1915	1982	actress,soundtrack,producer	tt0077711,tt0038109,tt0034583,tt0036855
7	nm0000007	Humphrey Bogart	1899	1957	actor,soundtrack,producer	tt0043265,tt0034583,tt0042593,tt0037382
8	nm0000008	Marlon Brando	1924	2004	actor,soundtrack,director	tt0078788,tt0068646,tt0070849,tt0047296
9	nm0000009	Richard Burton	1925	1984	actor,soundtrack,producer	tt0061184,tt0087803,tt0057877,tt0059749
10	nm0000010	James Cagney	1899	1986	actor,soundtrack,director	tt0029870,tt0035575,tt0042041,tt0055256

- There
  - Is one composite attribute, primaryName.
  - Are two multivalued attributes: primaryProfession, knownForTitles
  - knownForTitles is also tricky, which we will see.
  - Names are also a little tricky

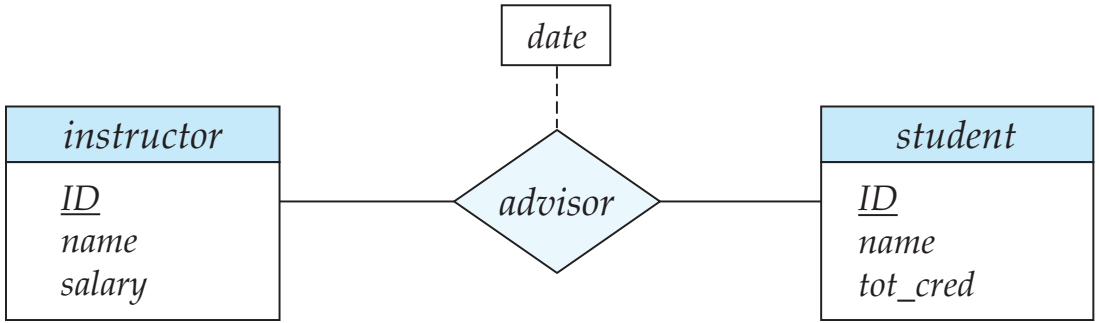
Switch to notebook

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

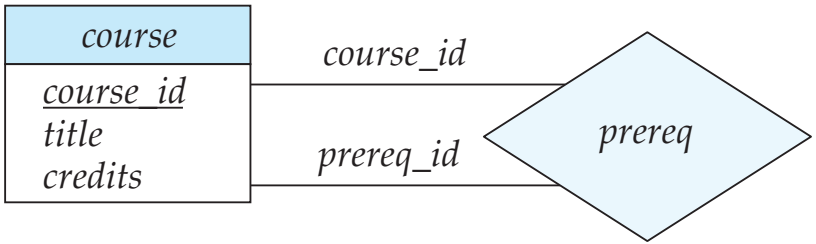


Relationship Sets with Attributes



Roles

- Entity sets of a relationship need not be distinct
  - Each occurrence of an entity set plays a “role” in the relationship
- The labels “*course_id*” and “*prereq_id*” are called **roles**.

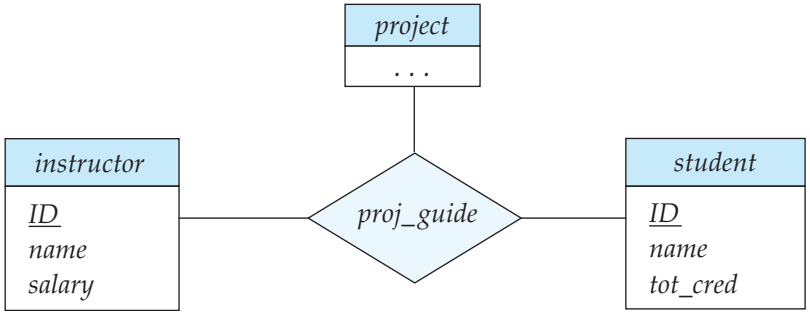


Degree of a Relationship Set

- Binary relationship
  - involve two entity sets (or degree two).
  - most relationship sets in a database system are binary.
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)
  - Example: *students* work on research *projects* under the guidance of an *instructor*.
  - relationship *proj_guide* is a ternary relationship between *instructor, student,* and *project*

Non-binary Relationship Sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary.
- E-R Diagram with a Ternary Relationship

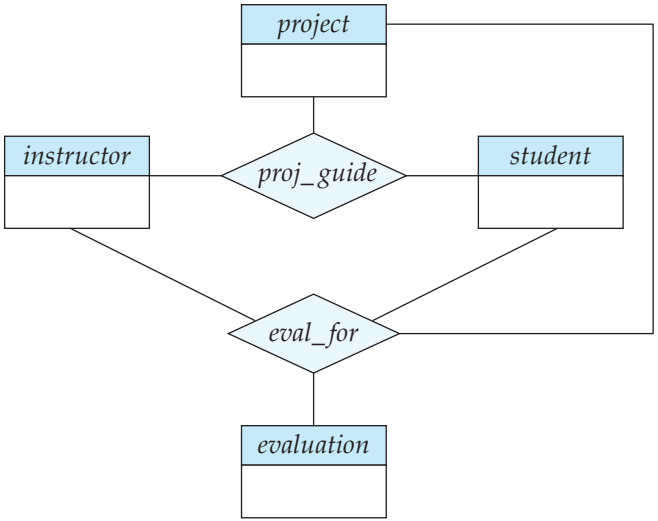


Mapping Cardinality Constraints

- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
  - One to one
  - One to many
  - Many to one
  - Many to many

Aggregation

- Consider the ternary relationship *proj_guide*, which we saw earlier
- Suppose we want to record evaluations of a student by a guide on a project

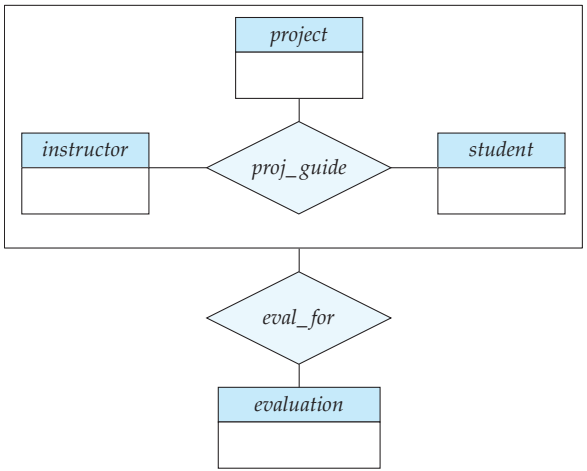


Aggregation (Cont.)

- Relationship sets *eval_for* and *proj_guide* represent overlapping information
  - Every *eval_for* relationship corresponds to a *proj_guide* relationship
  - However, some *proj_guide* relationships may not correspond to any *eval_for* relationships
    - So we can't discard the *proj_guide* relationship
- Eliminate this redundancy via *aggregation*
  - Treat relationship as an abstract entity
  - Allows relationships between relationships
  - Abstraction of relationship into new entity

Aggregation (Cont.)

- Eliminate this redundancy via *aggregation* without introducing redundancy, the following diagram represents:
  - A student is guided by a particular instructor on a particular project
  - A student, instructor, project combination may have an associated evaluation



Reduction to Relational Schemas

- To represent aggregation, create a schema containing
  - Primary key of the aggregated relationship,
  - The primary key of the associated entity set
  - Any descriptive attributes
- In our example:
  - The schema *eval_for* is:

    eval_for (*s_ID, project_id, i_ID, evaluation_id*)
  - The schema *proj_guide* is redundant.

Watch the Professor Do Stuff

REST (Again)

REST (https://restfulapi.net/)

What is REST

- REST is acronym for **RE**presentational **S**tate **T**ransfer. It is architectural style for **distributed hypermedia systems** and was first presented by Roy Fielding in 2000 in his famous dissertation.
- Like any other architectural style, REST also does have it's own 6 guiding constraints which must be satisfied if an interface needs to be referred as **RESTful**. These principles are listed below.

Guiding Principles of REST

- **Client–server** – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
- **Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.
- **Cacheable** – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
- **Uniform interface** – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.
- **Layered system** – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.
- **Code on demand (optional)** – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

Resources

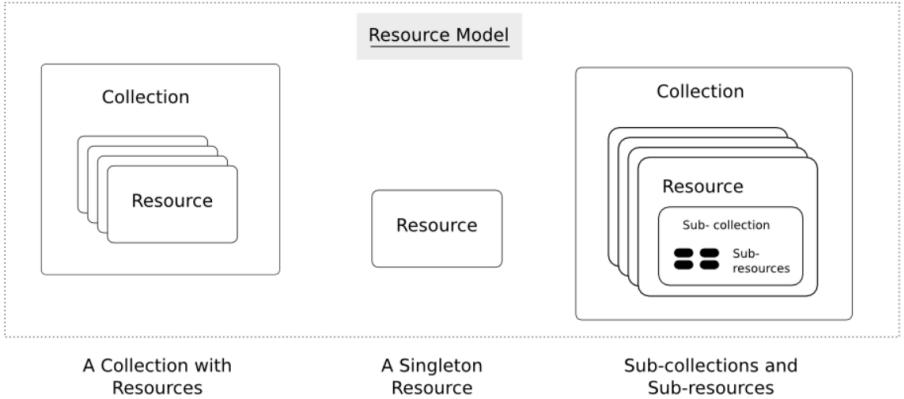
Resources are an abstraction. The application maps to create things and actions.

"A resource-oriented API is generally modeled as a resource hierarchy, where each node is either a *simple resource* or a *collection resource*. For convenience, they are often called a resource and a collection, respectively.

- A collection contains a list of resources of **the same type**. For example, a user has a collection of contacts.
- A resource has some state and zero or more sub-resources. Each sub-resource can be either a simple resource or a collection resource.

For example, Gmail API has a collection of users, each user has a collection of messages, a collection of threads, a collection of labels, a profile resource, and several setting resources.

While there is some conceptual alignment between storage systems and REST APIs, a service with a resource-oriented API is not necessarily a database, and has enormous flexibility in how it interprets resources and methods. For example, creating a calendar event (resource) may create additional events for attendees, send email invitations to attendees, reserve conference rooms, and update video conference schedules. (Emphasis added) (https://cloud.google.com/apis/design/resources#resources)



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

REST – Resource Oriented

- When writing applications, we are used to writing functions or methods:
  - openAccount(last_name, first_name, tax_payer_id)
  - account.deposit(deposit_amount)
  - account.close()

  We can create and implement whatever functions we need.
- REST only allows four methods:
  - POST: Create a resource
  - GET: Retrieve a resource
  - PUT: Update a resource
  - DELETE: Delete a resource

  That's it. That's all you get.

"The key characteristic of a resource-oriented API is that it emphasizes resources (data model) over the methods performed on the resources (functionality). A typical resource-oriented API exposes a large number of resources with a small number of methods." (https://cloud.google.com/apis/design/resources)

- A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

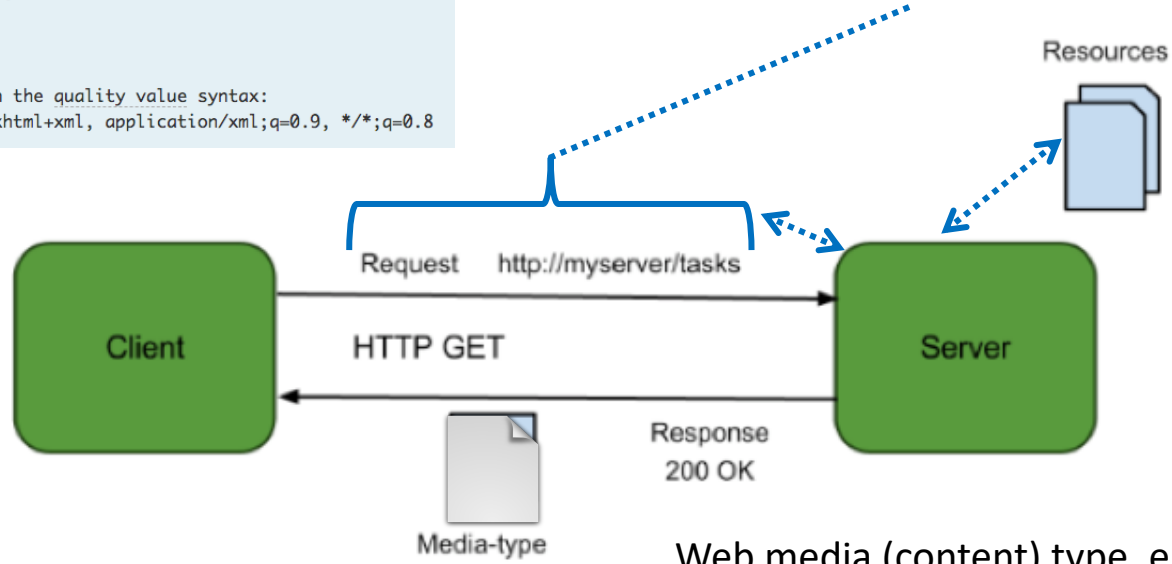
Resources, URLs, Content Types

- Relative URL identifies “resource” on the server.
- Server implementation maps abstract resource to tangible “thing,” file, DB row, … and any application logic.

Accept type in headers.

```
Accept: <MIME_type>/<MIME_subtype>
Accept: <MIME_type>/*
Accept: */*

// Multiple types, weighted with the quality value syntax:
Accept: text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8
```



Client may be

Browser

Mobile device

Other REST Service

… …

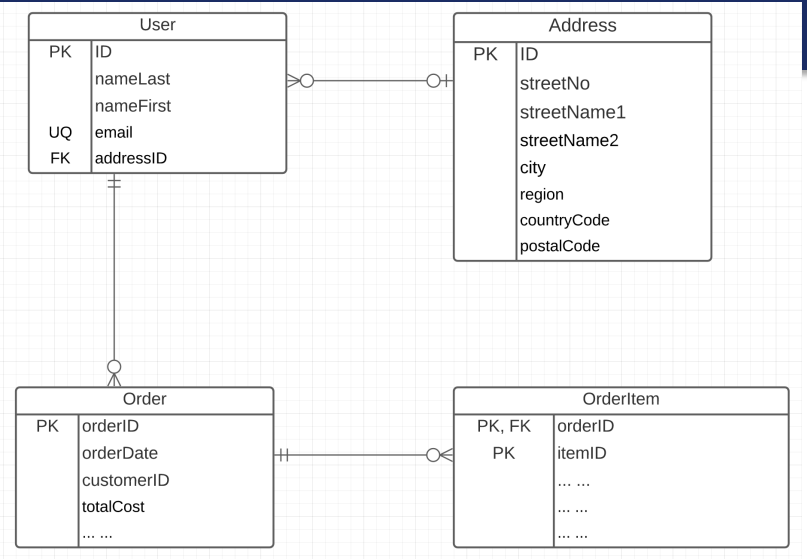
Web media (content) type, e.g.

- text/html
- application/json

REST Principles

- What about all of those principles?
  - Client/Server
  - Stateless
  - Cacheable
  - Uniform Interface
  - Layered System
  - Code on demand
- Some of these principles
  - Are simple and obvious.
  - Are subtle and have major implications.
  - Stateless can be baffling, but we will cover later.
- We will go into the principles in later lectures, ... but first linked resources ... ...

Linked Data



- Consider two microservices:
  - UserService handles two resources:
    - User
    - Address
  - OrderService handles two resources:
    - Order
    - OrderItem
- Associations/Link/Relationships can be surprisingly complicated. There are many considerations. (See https://www.uml-diagrams.org/association.html)
- In this example, we have
  - Association (User-Address, User-Order)
  - Composition (Order-OrderItem)

Resource Paths

The simple object model might/could/would/should have the following:

- /users
- /users/<userID>
- /users/<userID>/address
- /users/<userID>/orders
- /addresses
- /addresses/<addressID>/users
- /orders/<orderID>
- /orders/<orderID>/orderitems
- /orders<orderID>/orderitems/<itemID>
- … …

Composition

```
{
    "orderID": 2101,
    "orderDate": "2021-09-17",
    "totalCost": 10346.10,
    "orderItems": [
        {
            "itemID": 1,
            "someOtherFields": "... .."
        },
        {
            "itemID": 2,
            "someOtherFields": "... .."
        }
    ],
    "links": [
        {
            "href": "/users/gtg201",
            "rel": "user"
        }
    ]
}
```

Association

```
{
    "userID": "gtg101",
    "nameLast": "Ferguson",
    "nameFirst": "Donald",
    "email": "gtg@orthanc.gov",
    "links": [
        {
            "href": "/addresses/201",
            "rel": "address"
        },
        {
            "href": "/users/gtg101",
            "rel": "self"
        }
    ]
}
```

- How do you “point backwards” in a “foreign key relationship,” e.g. Address → User
  - “href” : “/users?addressID=201”
  - This means you store the addressID in the DB but return a link om REST.

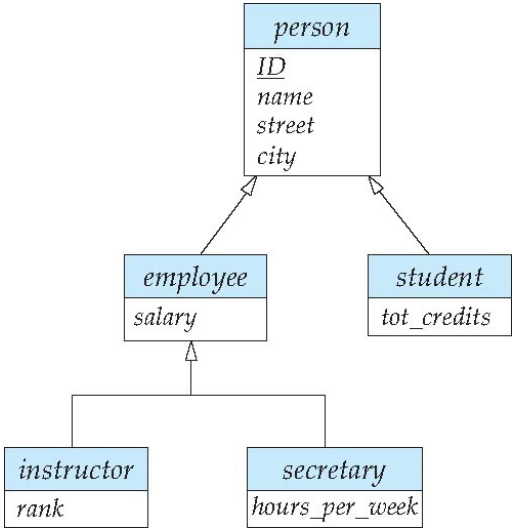
Backup

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.

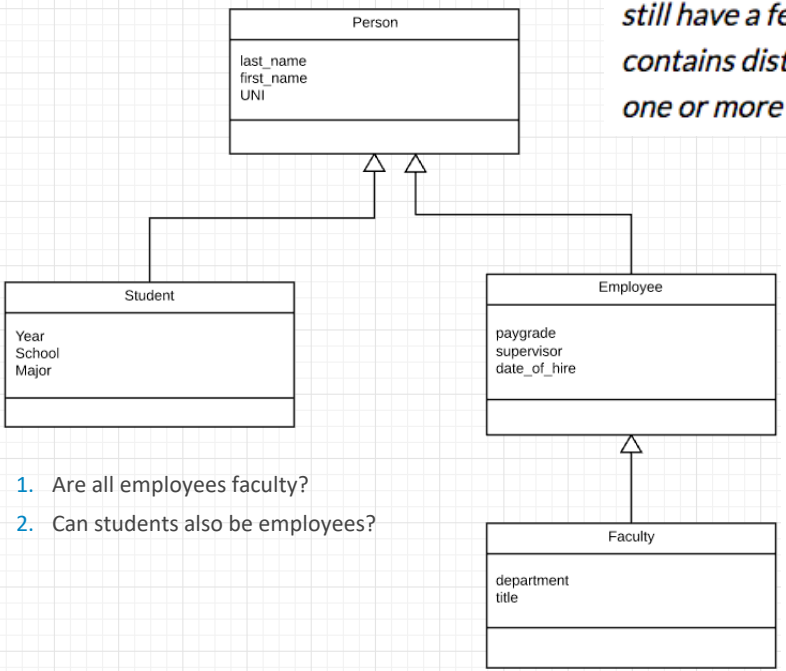
Specialization Example

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



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.

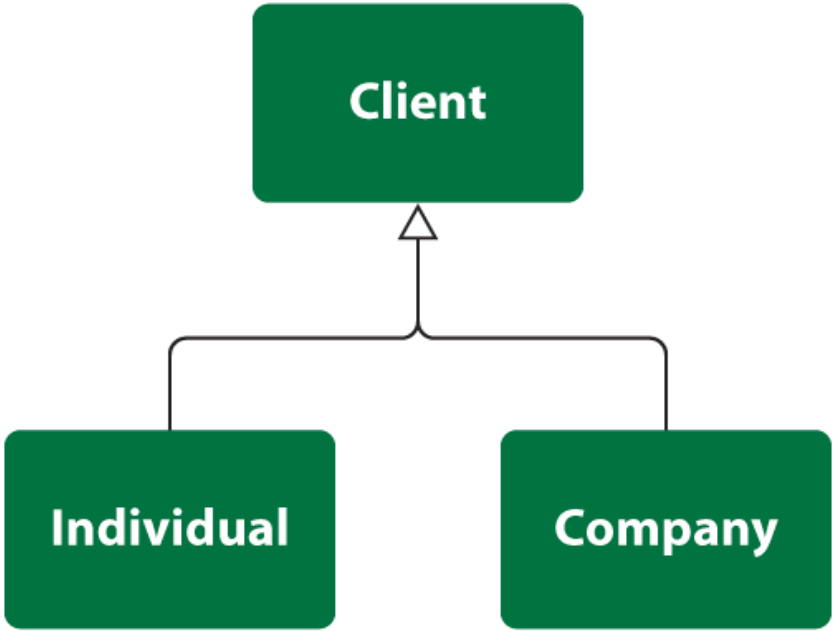
2 disjoint/overlapping

- 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).

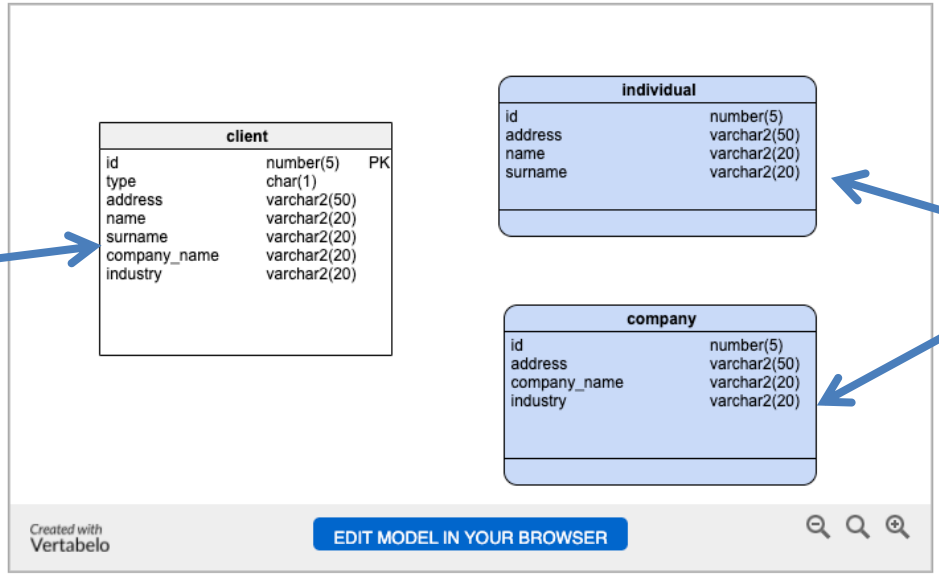


One Table

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

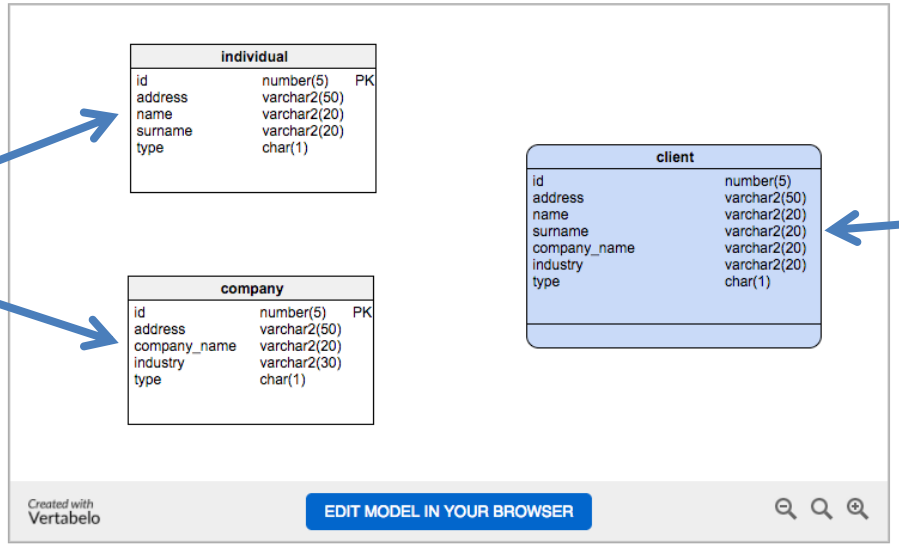
Views

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.

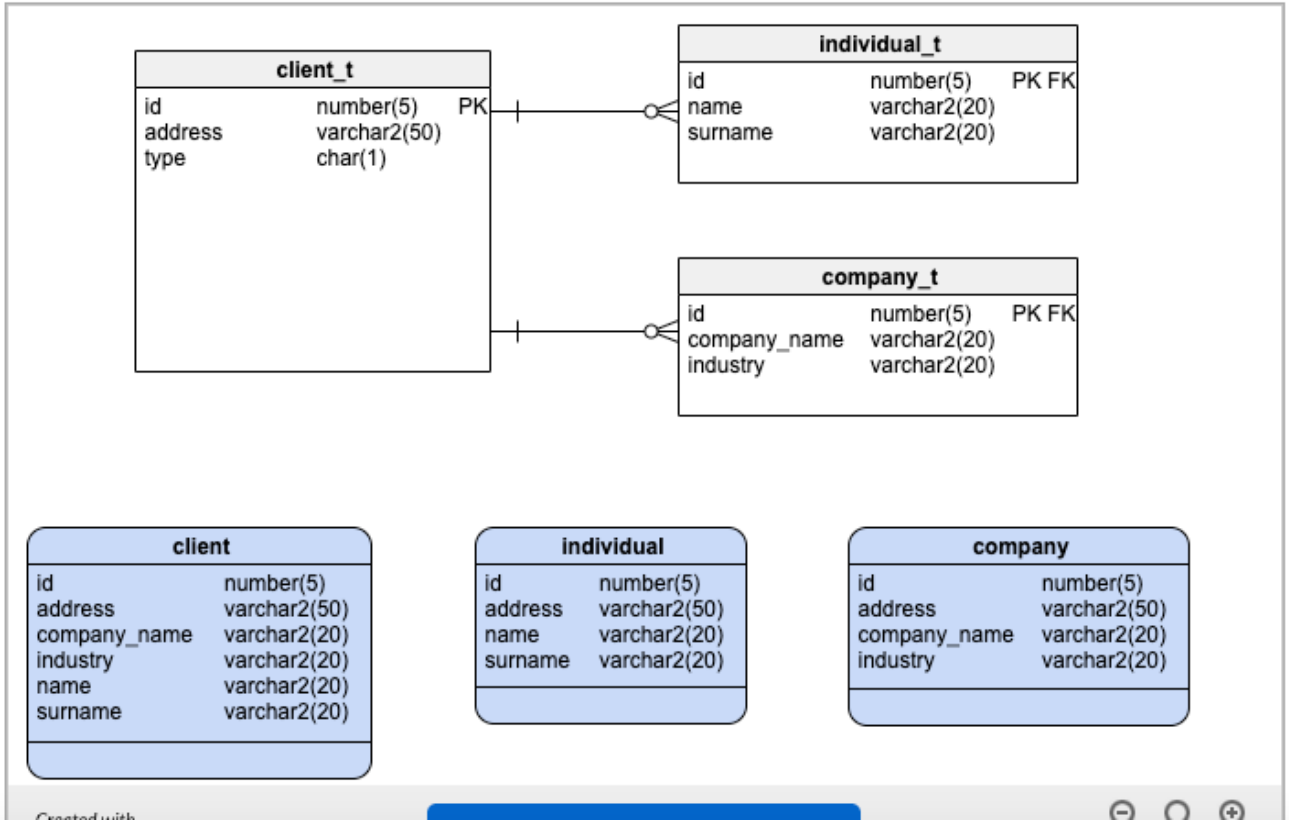
The diagram below presents two tables, `individual` and `company`, and a view (the blue one) called `client`.



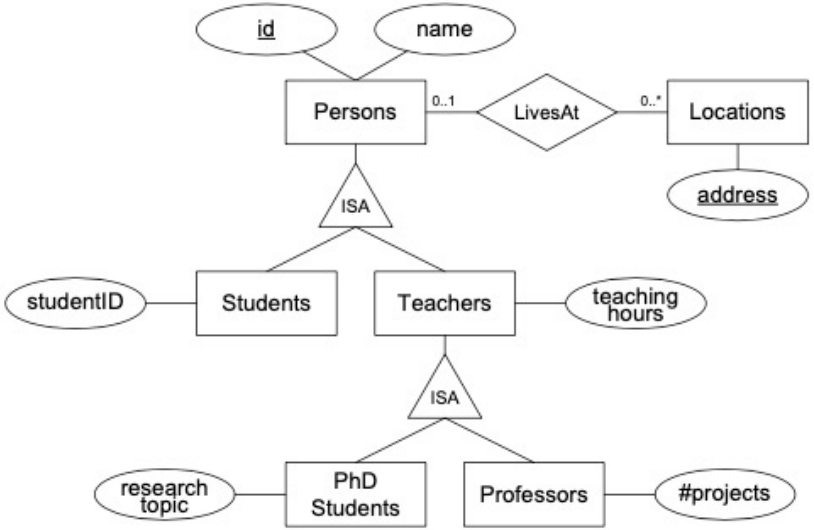
Tables

View

Three Table

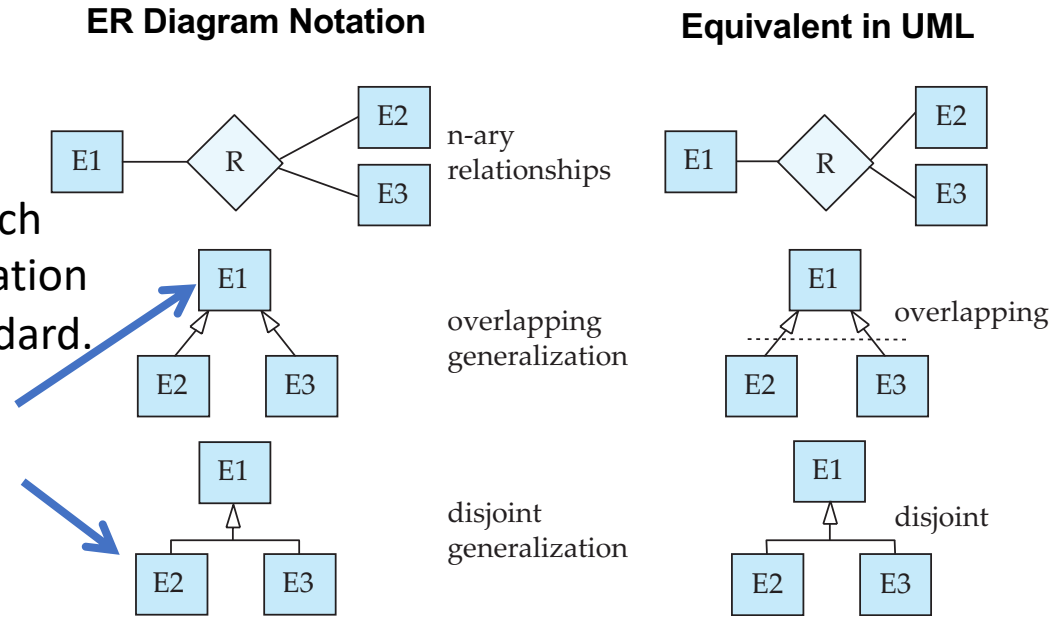


ISA Relationship



ER vs. UML Class Diagrams

I use this approach in Crow's Foot Notation but that is not standard.



* Generalization can use merged or separate arrows independent of disjoint/overlapping