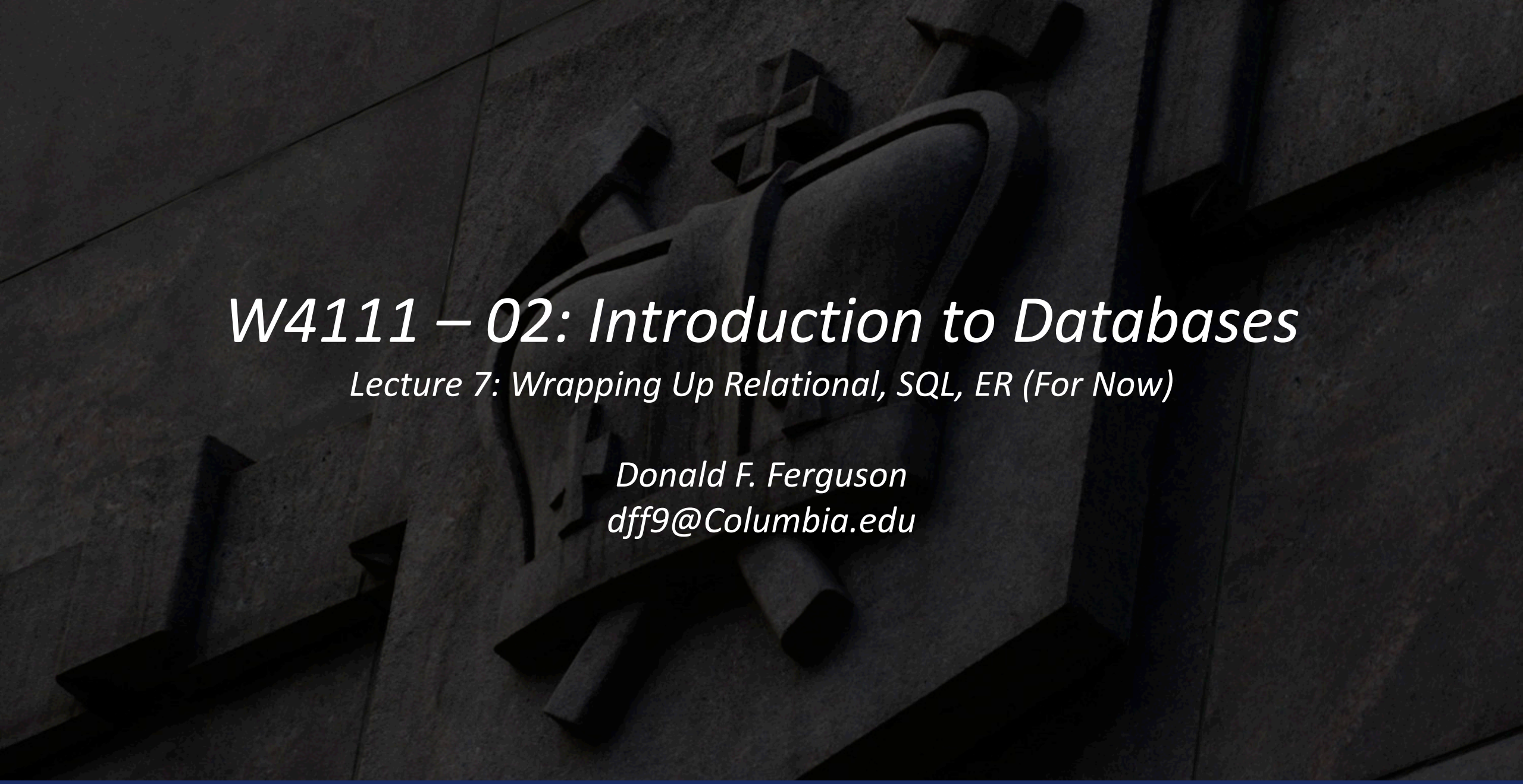


W4111 – 02: Introduction to Databases

Lecture 7: Wrapping Up Relational, SQL, ER (For Now)

Donald F. Ferguson
dff9@Columbia.edu

Contents

Contents

- ER Patterns and Realization
  - Complex/Compound Attributes; Derived Attributes
  - Weak Entities
  - Ternary (and more Complex) Relationships
  - Aggregation
  - Miscellaneous Design Issues.
- Advanced SQL
  - Recursion
  - Views and Security

ER Modeling and Realization

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

Attributes (https://www.tutorialspoint.com/dbms/er_model_basic_concepts.htm)

Attributes

Entities are represented by means of their properties, called **attributes**. All attributes have values. For example, a student entity may have name, class, and age as attributes.

There exists a domain or range of values that can be assigned to attributes. For example, a student's name cannot be a numeric value. It has to be alphabetic. A student's age cannot be negative, etc.

Types of Attributes

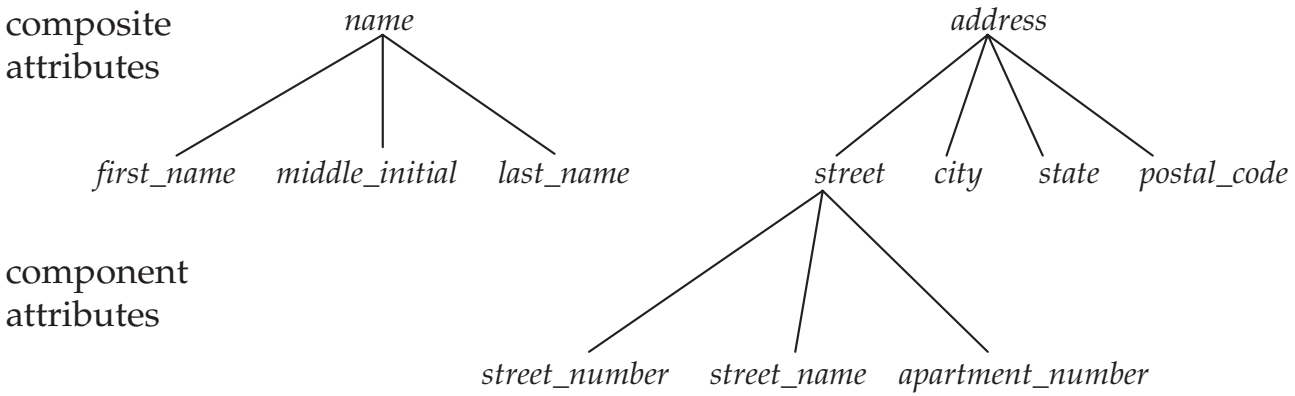
- **Simple attribute** − Simple attributes are atomic values, which cannot be divided further. For example, a student's phone number is an atomic value of 10 digits.
- **Composite attribute** − Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first_name and last_name.
- **Derived attribute** − Derived attributes are the attributes that do not exist in the physical database, but their values are derived from other attributes present in the database. For example, average_salary in a department should not be saved directly in the database, instead it can be derived. For another example, age can be derived from data_of_birth.
- **Single-value attribute** − Single-value attributes contain single value. For example − Social_Security_Number.
- **Multi-value attribute** − Multi-value attributes may contain more than one values. For example, a person can have more than one phone number, email_address, etc.

These attribute types can come together in a way like −

- simple single-valued attributes
- simple multi-valued attributes
- composite single-valued attributes
- composite multi-valued attributes

Composite Attributes

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



Representing Complex Attributes in ER Diagram

instructor
ID
name
first_name
middle_initial
last_name
address
street
street_number
street_name
apt_number
city
state
zip
{ phone_number }
date_of_birth
age ( )

```
{
    "ID": "dff9",
    "name": {
        "first_name": "Donald",
        "middle_initial": "F",
        "last_name": "Ferguson"
    },
    "addresses": {
        "office": {
            "street": {
                "number": "520",
                "street": "W 120th Street",
                "appt_room": "1234"
            },
            "city": "New York",
            "state": "NY",
            "country": "USA"
        },
        "home": {
            "street": {
                "number": "1234",
                "street": "Main St."
            },
            "city": "Smallville",
            "state": "KS",
            "country": "USA"
        }
    },
    "emails": [
        {"work": "dff9@columbia.edu", "note": "primary"},
        {"work": "dff@cs.columbia.edu"},
        {"work": "donald.ferguson@ansys.com", "note": "secondary"},
        {"personal": "vader@sith.org", "note": "personal"}
    ],
    "phone_numbers": [
        "212-555-1212",
        "516-555-1212"
    ],
    "DOB": "January 1, 1066"
}
```

Composite (Map)

Nested Composite (Map)

Multivalued of Composites (Map)

Multi-valued

Bad Format

First Normal Form

- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in **first normal form** if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - Example: Set of accounts stored with each customer, and set of owners stored with each account
  - We assume all relations are in first normal form (and revisit this in Chapter 22: Object Based Databases)

- (DFF) Normalization
  - **"[Relational] Database normalization** is the process of structuring a relational database in accordance with a series of so-called normal forms in order to reduce data redundancy and improve data integrity." (https://en.wikipedia.org/wiki/Database_normalization)
  - Normalization is a formal, methodology for think about what makes a "good relational schema."

First Normal Form (Cont.)

- Atomicity is actually a property of how the elements of the domain are used.
  - Example: Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*
  - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
  - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

First Normal Form (Cont.)

- Atomicity is actually a property of how the elements of the domain are used.

- There are several issues and problems that not applying 1NF can cause.
- Two of the most significant are:
  - Difficulty implementing integrity constraints.
  - Poor performance due to inability to use indexes.

1NF and Integrity

```
CREATE TABLE `course_non_nf` (
  `course_no` varchar(8) NOT NULL,
  `course_title` varchar(64) NOT NULL,
  `credits` int NOT NULL,
  PRIMARY KEY (`course_no`),
  CONSTRAINT `checkcredit` CHECK (((`credits` > 0) and (`credits` <= 4)))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

```
CREATE TABLE `course_nf` (
  `dept_code` char(4) NOT NULL,
  `level` enum('1','2','3','4','6','9') NOT NULL,
  `number` int NOT NULL,
  `title` varchar(32) NOT NULL,
  `faculty_code` enum('W','B','E','G') NOT NULL,
  `credits` int NOT NULL,
  `course_no` varchar(16) GENERATED ALWAYS AS (concat(`dept_code`,`faculty_code`,`level`,`number`)) STORED,
  PRIMARY KEY (`dept_code`,`faculty_code`,`level`,`number`),
  KEY `course_no1` (`level`,`number`),
  KEY `full_no` (`course_no`),
  CONSTRAINT `c_to_d` FOREIGN KEY (`dept_code`) REFERENCES `department_nf` (`dept_code`),
  CONSTRAINT `course_nf_chk_1` CHECK ((((`level` = _utf8mb4'4') and (`credits` = 4)) or
    ((`credits` <> _utf8mb4'4') and (`credits` > 0) and (`credits` <= 3))))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

- Consider a course number like “COMSW4111”
  - Strings/VARCHAR is usually an atomic domain.
  - But is not in this case.
- Representing the course number as a string
  - Precludes integrity constrains on the department code, faculty code and level.
  - Precludes adding an index on elements of the number like ‘4111.’
  - We will see why in the next module.
- If we want to treat the composite attribute as a string, we can
  - Generate the column.
  - If it is a stored, generate column, we can also index it.

How Not to Model in Relational

ID	Name	Address1	Address2	Emails	PhoneWork	PhoneHome
dff9	Donald F. Ferguson	1234 Happy St, Smallville, KS, USA	4567 W 166th St, NY, NY 10026	dff@cs.columbia.edu, dff9@columbia.edu	+1 212 555-1212	+44 000 21 01456
xy1	John Smith	111 Some St, Boomtown, MA USA		foo@bar.edu, cool@cat.pet, x@y.com		+1 516-555-1212


```
{
    "ID": "dff9",
    "name": {
        "first_name": "Donald",
        "middle_initial": "F",
        "last_name": "Ferguson"
    },
    "addresses": {
        "office": {
            "street": {
                "number": "520",
                "street": "W 120th Street",
                "appt_room": "1234"
            },
            "city": "New York",
            "state": "NY",
            "country": "USA"
        },
        "home": {
            "street": {
                "number": "1234",
                "street": "Main St."
            },
            "city": "Smallville",
            "state": "KS",
            "country": "USA"
        }
    },
    "emails": [
        {"work": "dff9@columbia.edu", "note": "primary"},
        {"work": "dff@cs.columbia.edu"},
        {"work": "donald.ferguson@ansys.com", "note": "secondary"},
        {"personal": "vader@sith.org", "note": "personal"}
    ],
    "phone_numbers": [
        "212-555-1212",
        "516-555-1212"
    ],
    "DOB": "January 1, 1066"
}
```


```
CREATE TABLE `W4111Examples`.`bad_table` (
             `id` VARCHAR(32) NULL,
            `name` VARCHAR(256) NULL,
            `address1` VARCHAR(256) NULL,
             `address2` VARCHAR(256) NULL,
            `emails` VARCHAR(256) NULL,
            `phone_work` VARCHAR(45) NULL,
            `phone_other` VARCHAR(45) NULL);
```

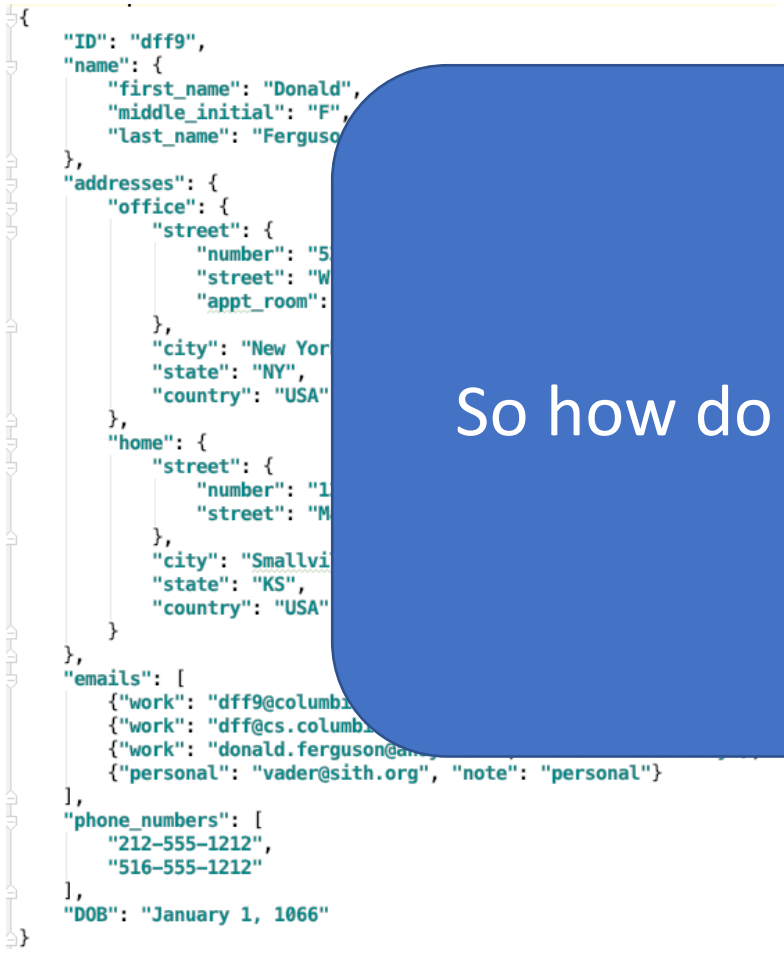
Approaches like this create several problems:

- Query on things like last_name or country.
- Integrity: emails are unique
- Insight:
  - Finding people in the same zip code.
  - Which people have business emails?
- Difficult to index for efficiency.

We will study database models that support and recommend “document” or “object” like approaches.

How **Not** to Model in Relational

ID	Name	Address1	Address2	Emails	PhoneWork	PhoneHome
dff9	Donald F. Ferguson	1234 Happy St, Smallville, KS, USA	4567 W 166th St, NY, NY 10026	dff@cs.columbia.edu, dff9@columbia.edu	+1 212 555-1212	+44 000 21 01456
xy1	John Smith	111 Some St, Boomtown, MA USA		foo@bar.edu, cool@cat.pet, x@y.com		+1 516-555-1212



CREATE TABLE `W4111Examples`.`bad_table` (
`id` VARCHAR(32) NULL

So how do you do this in the relational model?

- Difficult to index for efficiency.

We will study database models that support and recommend “document” or “object” like approaches.

Generated Columns, Column Types and Functions

Relational Databases have built in functions for many types.

- From https://www.mysqltutorial.org/mysql-functions.aspx for MySQL. List is complete.
- Other RDBs have similar sets of functions.

MySQL aggregate functions

- MySQL aggregate functions – provide a brief overview of the most commonly used MySQL aggregate functions
- AVG – calculate the average value of a set of values or an expression.
- COUNT – count the number of rows in a table.
- INSTR – return the position of the first occurrence of a substring in a string.
- SUM – calculate the sum of a set of values or an expression.
- MIN – find the minimum value in a set of values
- MAX – find the maximum value in a set of values
- GROUP_CONCAT – concatenate strings from a group into a string with various options such as `DISTINCT`, `ORDER BY`, and `SEPARATOR`.
- MySQL standard deviation functions – shows you how to compute population standard deviation and sample standard deviation.

Generated Columns, Column Types and Functions

Relational Databases have built in functions for many types.

- From https://www.mysqltutorial.org/mysql-functions.aspx for MySQL. List is complete.
- Other RDBs have similar sets of functions.

MySQL control flow functions

- CASE – return the corresponding result in `THEN` branch if the condition in the `WHEN` branch is satisfied, otherwise return the result in the `ELSE` branch.
- IF – return a value based on a given condition.
- IFNULL– return the first argument if it is not NULL, otherwise returns the second argument.
- NULLIF– return NULL if the first argument is equal to the second argument, otherwise, returns the first argument.

MySQL string functions

- CONCAT – combine two or more strings into one string.
- LENGTH & CHAR_LENGTH – get the length of strings in bytes and in characters.
- LEFT – get the left part of a string with a specified length.
- REPLACE – search and replaces a substring in a string.
- SUBSTRING – extract a substring starting from a position with a specific length.
- TRIM – remove unwanted characters from a string.
- FIND_IN_SET – find a string within a comma-separated list of strings.
- FORMAT – format a number with a specific locale, rounded to the number of decimals

Generated Columns, Column Types and Functions

MySQL date and time functions

- CURDATE – return the current date.
- DATEDIFF – calculate the number of days between two `DATE` values.
- DAY – get the day of the month of a specified date.
- DATE_ADD – add a time value to date value.
- DATE_SUB – subtract a time value from a date value.
- DATE_FORMAT – format a date value based on a specified date format.
- DAYNAME – get the name of a weekday for a specified date.
- DAYOFWEEK – return the weekday index for a date.
- EXTRACT – extract part of a date.
- NOW – return the current date and time at which the statement executed.
- MONTH– return an integer that represents a month of a specified date.
- STR_TO_DATE – convert a string into a date and time value based on a specified format.
- SYSDATE – return the current date.
- TIMEDIFF – calculate the difference between two `TIME` or `DATETIME` values.
- TIMESTAMPDIFF – calculate the difference between two `DATE` or `DATETIME` values.
- WEEK – return a week number of a date.
- WEEKDAY – return a weekday index for a date.
- YEAR – return the year part of a date value.

MySQL comparison functions

- COALESCE – return the first non-null arguments, which is very handy for substitution of null.
- GREATEST & LEAST – take n arguments and return the greatest and least values of the n arguments respectively.
- ISNULL – return 1 if the argument is null, otherwise return zero.

MySQL Math functions

- ABS – return the absolute value of a number.
- CEIL – return the smallest integer value greater than or equal to the input number.
- FLOOR – return the largest integer value not greater than the argument.
- MOD – return the remainder of a number divided by another.
- ROUND – rounds a number to a specified number of decimal places.
- TRUNCATE – truncate a number to a specified number of decimal places.

Other MySQL functions

- LAST_INSERT_ID – obtain the last generated sequence number of the last inserted record.
- CAST – convert a value of any type into a value with a specified type.

Some Examples

Switch to Notebook

Date Formatting and Parsing

Format	Description
%a	Abbreviated weekday name (Sun to Sat)
%b	Abbreviated month name (Jan to Dec)
%c	Numeric month name (0 to 12)
%D	Day of the month as a numeric value, followed by suffix (1st, 2nd, 3rd, ...)
%d	Day of the month as a numeric value (01 to 31)
%e	Day of the month as a numeric value (0 to 31)
%f	Microseconds (000000 to 999999)
%H	Hour (00 to 23)
%h	Hour (00 to 12)
%I	Hour (00 to 12)
%i	Minutes (00 to 59)
%j	Day of the year (001 to 366)
%k	Hour (0 to 23)
%l	Hour (1 to 12)
%M	Month name in full (January to December)
%m	Month name as a numeric value (00 to 12)
%p	AM or PM
%r	Time in 12 hour AM or PM format (hh:mm:ss AM/PM)
%S	Seconds (00 to 59)
%s	Seconds (00 to 59)
%T	Time in 24 hour format (hh:mm:ss)
%U	Week where Sunday is the first day of the week (00 to 53)
%u	Week where Monday is the first day of the week (00 to 53)
%V	Week where Sunday is the first day of the week (01 to 53). Used with %X
%v	Week where Monday is the first day of the week (01 to 53). Used with %X
%W	Weekday name in full (Sunday to Saturday)
%w	Day of the week where Sunday=0 and Saturday=6
%X	Year for the week where Sunday is the first day of the week. Used with %V
%x	Year for the week where Monday is the first day of the week. Used with %V
%Y	Year as a numeric, 4-digit value
%y	Year as a numeric, 2-digit value

Handling Compound/Composite Attributes

Let's use a simpler test case.

ID	Name	Address	Emails	Phone
dff9	Donald F. Ferguson	1234 Happy St, Smallville, KS, USA	dff@cs.columbia.edu	+1 212 555-1212
xy1	John Smith	111 Some St, Boomtown, MA USA	foo@bar.org	+44 038 931 7168

Analysis:

- Name is clear, and we should decompose into three columns.
- Some aspects of Address are clear:
  - We need to decompose into individual columns, e.g. street, town, state, country.
  - The level of decomposition is a judgement call, e.g. do we separate street into (no, name)?
  - There is a fundamental design question/judgment decision:
    - Is the Address a property of a Person?
    - Or, is a Person *related* to an instance of an *Address* entity?
  - We need to determine how to handle country and state, which have values that come from a fixed domain.
- Email seems relatively simple, if there is only one.
- Phone seems relatively simple if there is one. We need to determine if compound:
  - Country code
  - Telephone number

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);

<dff>

- Not sure how common using UDT is. I almost never see it.
- Support across various RDBMS products is inconsistent.

UDT Example

Here's an example of a UDT definition:

```
CREATE TYPE book_udt AS                 -- the UDT name will be book_udt
title CHAR(40),                         -- title is the first attribute
buying_price DECIMAL(9,2),              -- buying_price is the second attribute
selling_price DECIMAL(9,2)              -- selling_price is the third attribute
NOT FINAL                               -- this is a mandatory Finality Clause
METHOD profit( ) RETURNS DECIMAL(9,2); -- profit is a method, defined later
```

Columns Based on UDTs

Let us begin by making a Table, one of whose Columns is a UDT:

```
CREATE TABLE T (
   book_column book_udt,
   serial_number INTEGER);
```

https://crate.io/docs/sql-99/en/latest/chapters/27.html#udt-example

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'));

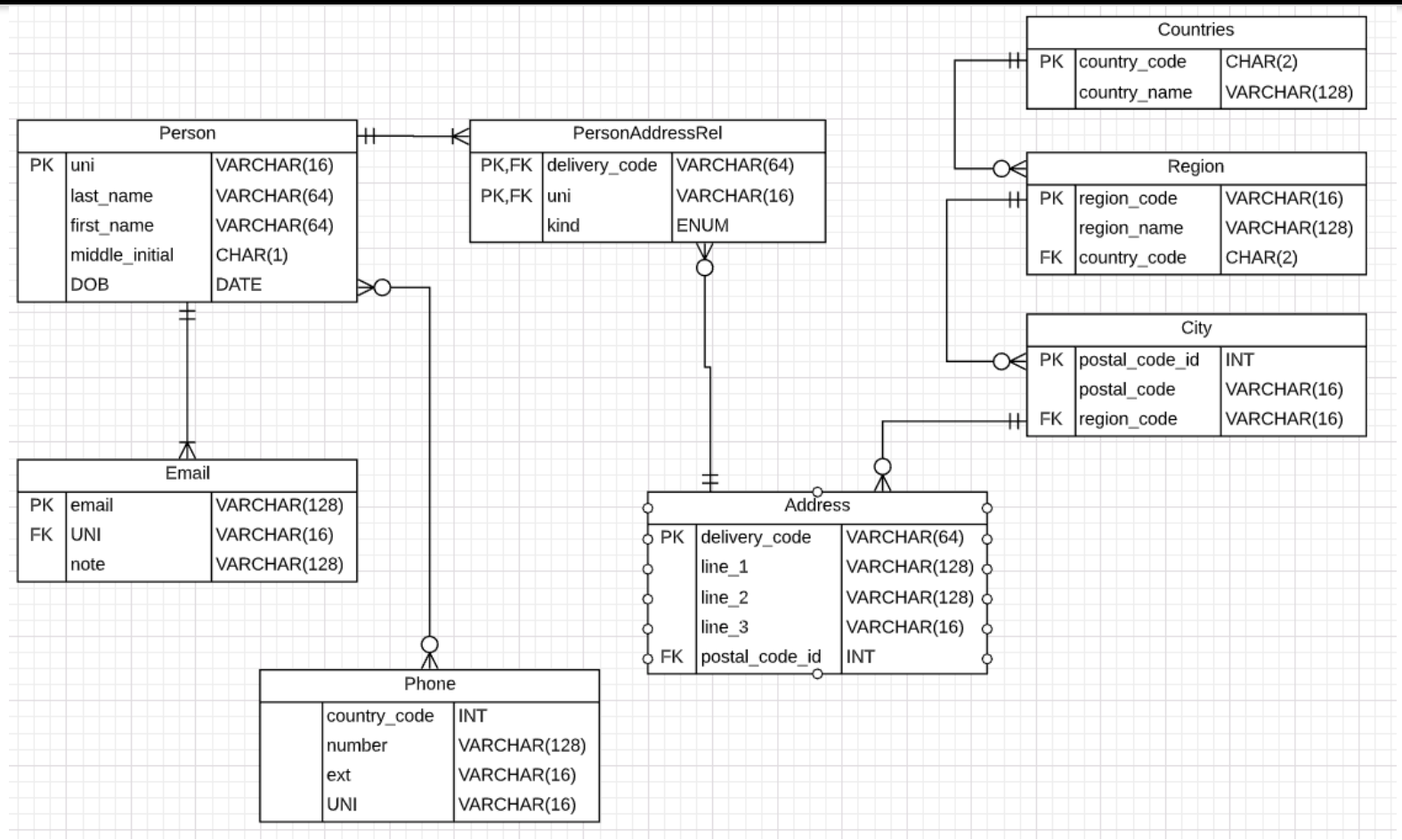
```
1  -- DOMAIN: public.email
2
3  -- DROP DOMAIN public.email;
4
5  CREATE DOMAIN public.email
6      AS character varying(['128'])
7      COLLATE pg_catalog."en_US.UTF-8"
8      DEFAULT NULL::character varying;
9
10 ALTER DOMAIN public.email OWNER TO postgres;
11
12 ALTER DOMAIN public.email
13     ADD CONSTRAINT has_at CHECK (VALUE::text ~~ '%@%'::text
```

This is the PostgreSQL of saying the value of an an element in a email domain must be LIKE “%@%”

Again, I am not sure how common this is or how consistent the support is.

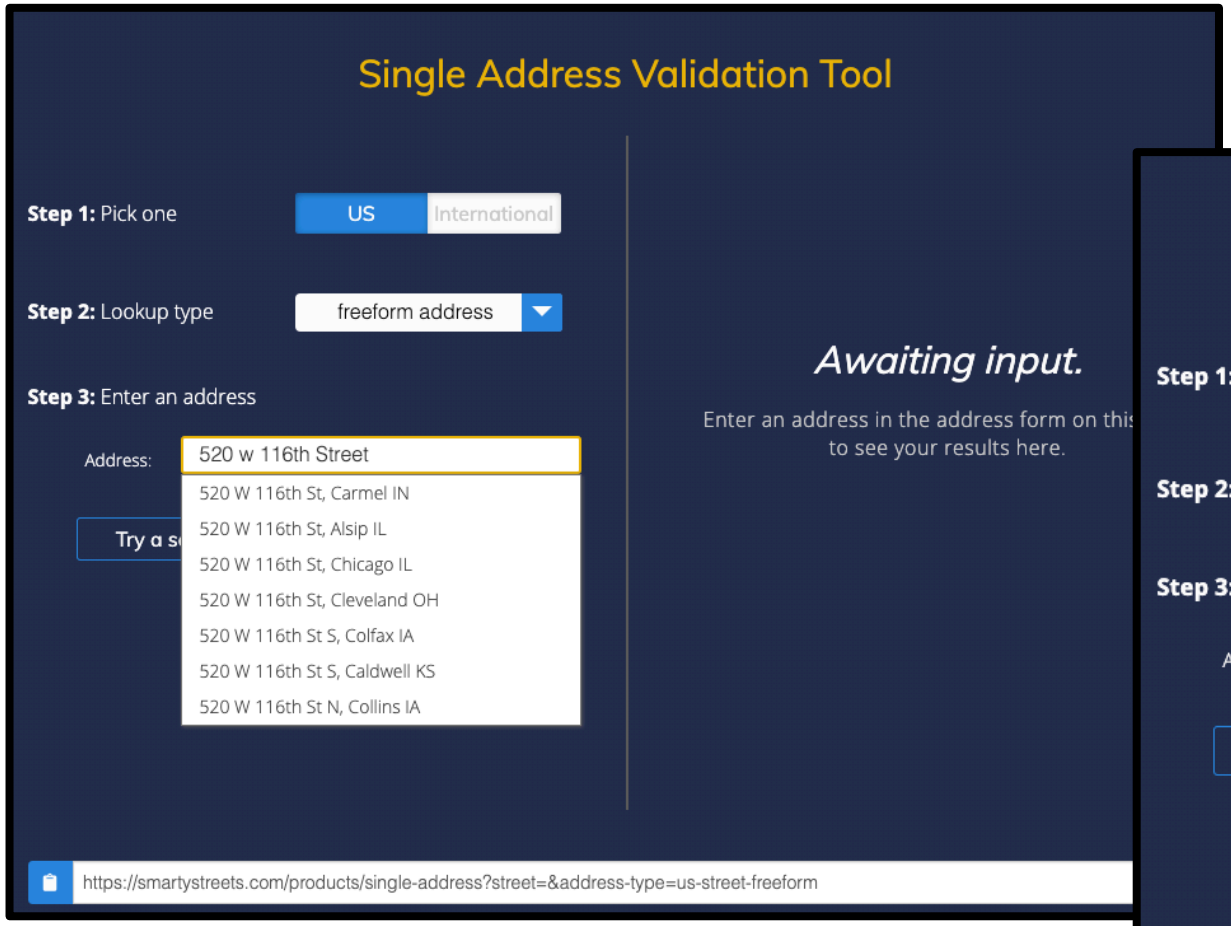
Back to Our Example – An Initial Pass

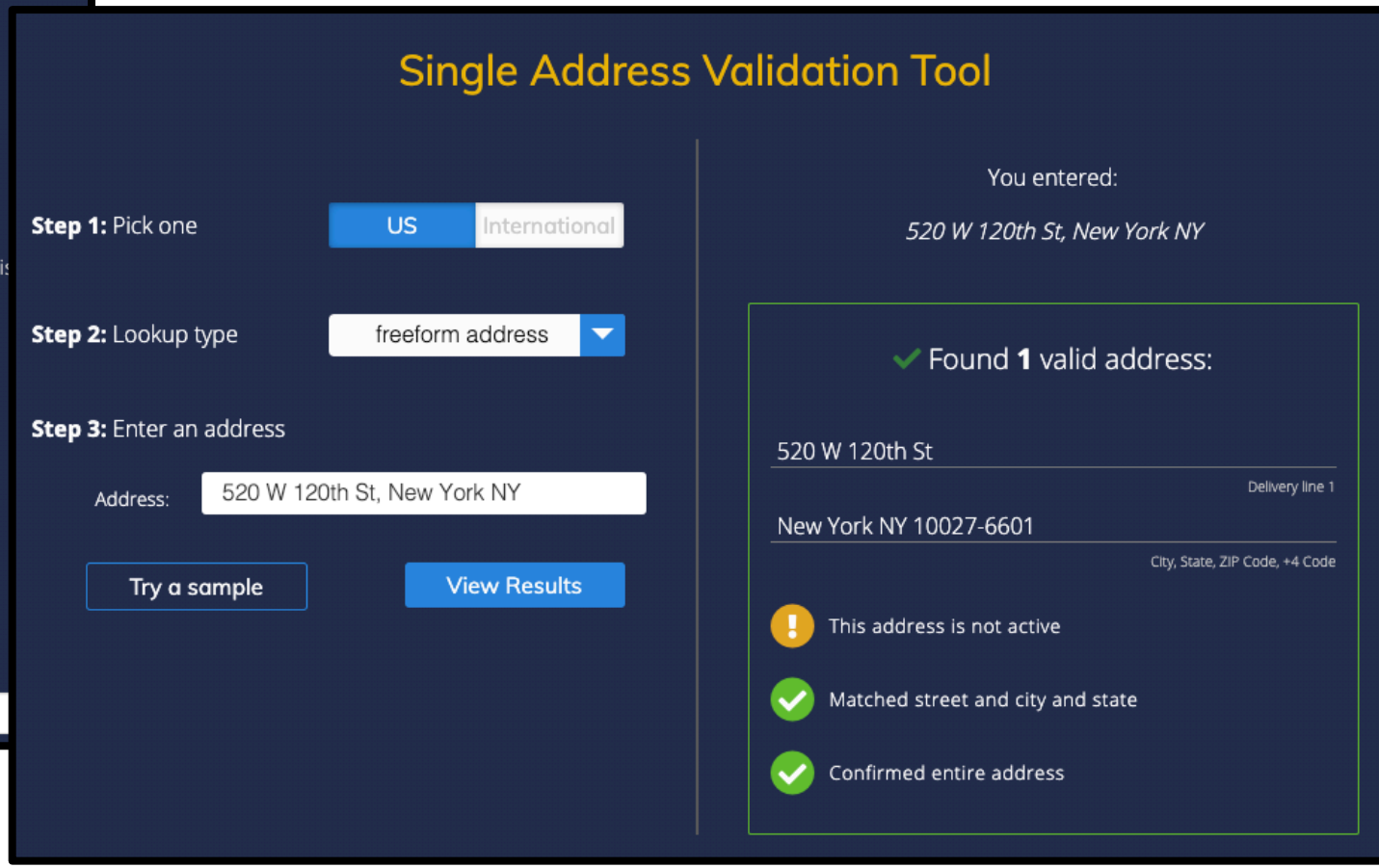
```
{
    "ID": "dff9",
    "name": {
        "first_name": "Donald",
        "middle_initial": "F",
        "last_name": "Ferguson"
    },
    "addresses": {
        "office": {
            "street": {
                "number": "520",
                "street": "W 120th Street",
                "appt_room": "1234"
            },
            "city": "New York",
            "state": "NY",
            "country": "USA"
        },
        "home": {
            "street": {
                "number": "1234",
                "street": "Main St."
            },
            "city": "Smallville",
            "state": "KS",
            "country": "USA"
        }
    },
    "emails": [
        {"work": "dff9@columbia.edu", "note": "primary"},
        {"work": "dff@cs.columbia.edu"},
        {"work": "donald.ferguson@ansys.com", "note": "secondary"},
        {"personal": "vader@sith.org", "note": "personal"}
    ],
    "phone_numbers": [
        "212-555-1212",
        "516-555-1212"
    ],
    "DOB": "January 1, 1066"
}
```



- This is a first pass and has some errors.
- Would not be sure until encountered some data.
- There are a couple of issues that I know about.

Addresses can be Surprisingly Complicated





Addresses can be Surprisingly Complicated

GET https://us-street.api.smartystreets.com/street-address?auth-id=1b0c3925-45fd-97fe-2dc4-fc3f16f35129&auth-token=XjP9Khk3tOF3KAqiQttV&candidates=5&street=520 W 120th St St&city=NY... Send Save

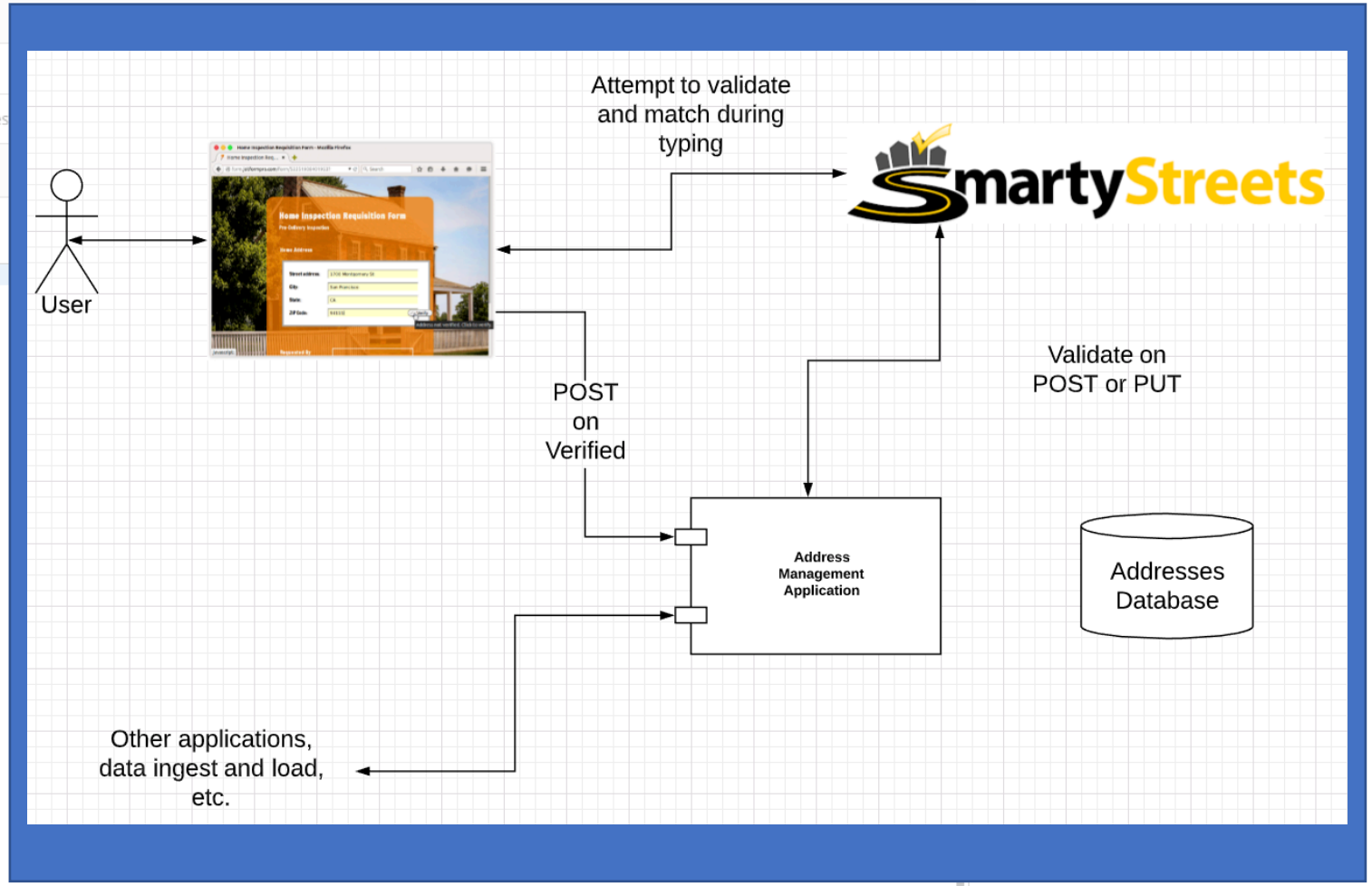
Params Authorization Headers Body Pre-request Script Tests

none form-data x-www-form-urlencoded raw binary

Body Cookies Headers (5) Test Results

Pretty Raw Preview JSON


```
[
    {
        "input_index": 0,
        "candidate_index": 0,
        "delivery_line_1": "520 W 120th St",
        "last_line": "New York NY 10027-6601",
        "delivery_point_barcode": "100276601205",
        "components": {
            "primary_number": "520",
            "street_predirection": "W",
            "street_name": "120th",
            "street_suffix": "St",
            "city_name": "New York",
            "default_city_name": "New York",
            "state_abbreviation": "NY",
            "zipcode": "10027",
            "plus4_code": "6601",
            "delivery_point": "20",
            "delivery_point_check_digit": "5"
        },
        "metadata": {
            "record_type": "S",
            "zip_type": "Standard",
            "county_fips": "36061",
            "county_name": "New York",
            "carrier_route": "C091",
            "congressional_district": "10",
            "rdi": "Commercial",
            "elot_sequence": "0169",
            "elot_sort": "A",
            "latitude": 40.80954,
            "longitude": -73.95993,
            "precision": "Zip9",
            "time_zone": "Eastern",
```

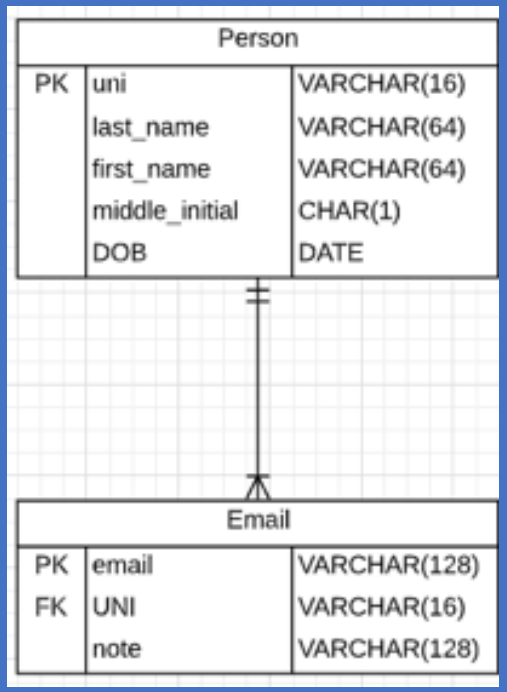



Screenshot

Back to Our Example – An Initial Pass

We have taken a quick, simple look at:

- Compound/composite attributes and mapping to the relational model.
- Compound attributes as:
  - Multiple columns in an entity.
  - A separate table/entity with multiple columns.
- Multi-valued attributes as related entities:
  - Many-to-Many
  - One-to-Many
  - Properties of the relationship.
- We are not going to implement the tables/schema right now.
  You should be able to implement most of the diagram, with some exceptions.
- There were some interesting constraints that we glossed over, e.g.
  - You cannot make a person without at least one Email.
  - Every email has exactly one Person.
  - What do you make 1st, the chicken or the egg?
  - Also, we have not considered *at least one.*



- There are a couple of issues that I know about.

Weak Entity

Weak Entity Sets

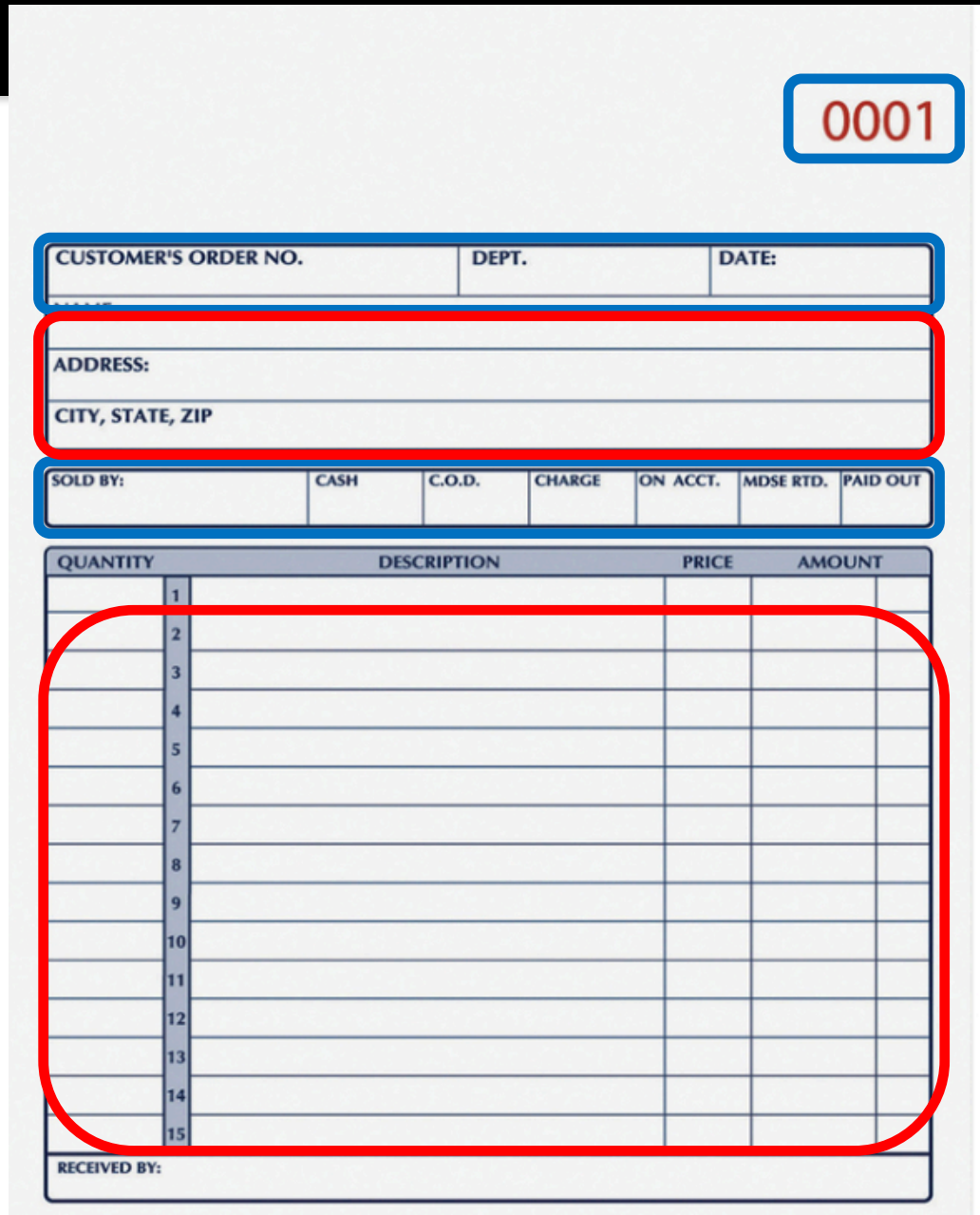
- Consider a *section* entity, which is uniquely identified by a *course_id*, *semester, year*, and *sec_id*.
- Clearly, section entities are related to course entities. Suppose we create a relationship set *sec_course* between entity sets *section* and *course*.
- Note that the information in *sec_course* 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.

"In a relational database, a **weak entity** is an entity that cannot be uniquely identified by its attributes alone; therefore, it must use a foreign key in conjunction with its attributes to create a primary key. The foreign key is typically a primary key of an entity it is related to." (https://en.wikipedia.org/wiki/Weak_entity)

One way to think about this is that a section does not "require" a course or "depend" on a course. The section is somehow inside of or part of a course.

Weak Entity – Consider and Order Form

- The order has attributes that are part of the order:
  - Date
  - Amount
  - Priority
  - etc.
- Dependency (Foreign Key Constraint) on other entities:
  - Customer
  - Product
  - Address
- But has these other weird things that
  - Repeat
  - Have attributes
  - Are sort of “inside” the order
  - These are weak entities.



Weak Entity Sets (Cont.)

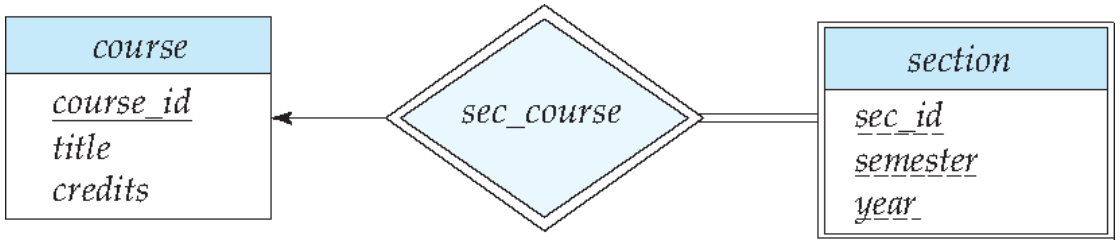
- An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *section_id*, *year*, and *semester.*
  - However, the entity set *section* then does not have enough attributes to identify a particular *section*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

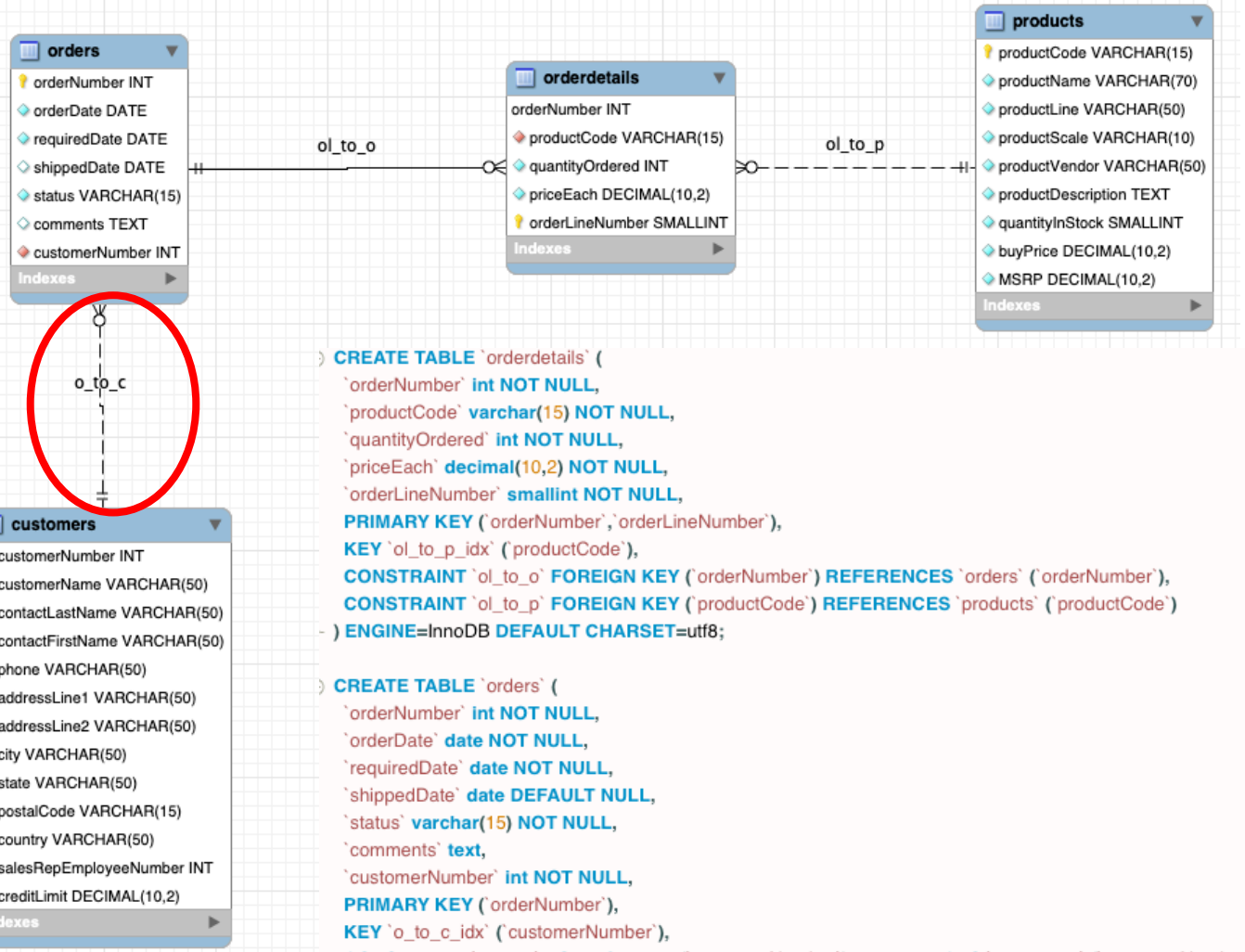
Expressing Weak Entity Sets

- 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*)



Consider Classicmodels Schema (https://www.mysqltutorial.org/mysql-sample-database.aspx)

A subset of the schema



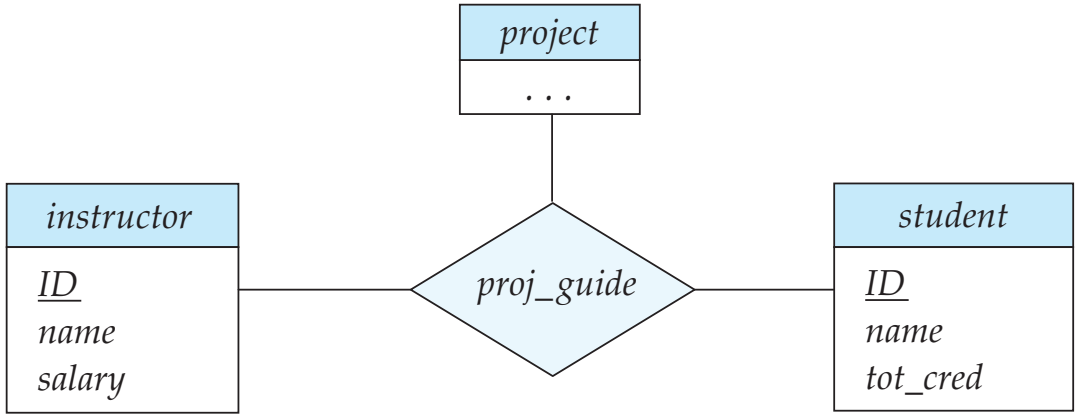

```
CREATE TABLE `orderdetails` (
  `orderNumber` int NOT NULL,
  `productCode` varchar(15) NOT NULL,
  `quantityOrdered` int NOT NULL,
  `priceEach` decimal(10,2) NOT NULL,
  `orderLineNumber` smallint NOT NULL,
  PRIMARY KEY (`orderNumber`,`orderLineNumber`),
  KEY `ol_to_p_idx` (`productCode`),
  CONSTRAINT `ol_to_o` FOREIGN KEY (`orderNumber`) REFERENCES `orders` (`orderNumber`),
  CONSTRAINT `ol_to_p` FOREIGN KEY (`productCode`) REFERENCES `products` (`productCode`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `orders` (
  `orderNumber` int NOT NULL,
  `orderDate` date NOT NULL,
  `requiredDate` date NOT NULL,
  `shippedDate` date DEFAULT NULL,
  `status` varchar(15) NOT NULL,
  `comments` text,
  `customerNumber` int NOT NULL,
  PRIMARY KEY (`orderNumber`),
  KEY `o_to_c_idx` (`customerNumber`),
  CONSTRAINT `o_to_c` FOREIGN KEY (`customerNumber`) REFERENCES `customers` (`customerNumber`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

- I put some improvement into the default schema you download.
- There are the obvious foreign key constraints:
  - orderdetails → orders
  - orderdetails → products
  - order → customers
- Why is one of the lines “dashed?”
  - MySQL refers to this as an “identifying relationship.”
  - orderdetails cannot form a primary key without a field from orders.
  - It is a Weak Entity
  - orderdetails primary key contains a column of orders primary key.
- An order ”owns” or “contains” its orderdetails.

Advanced Relationships, Aggregation

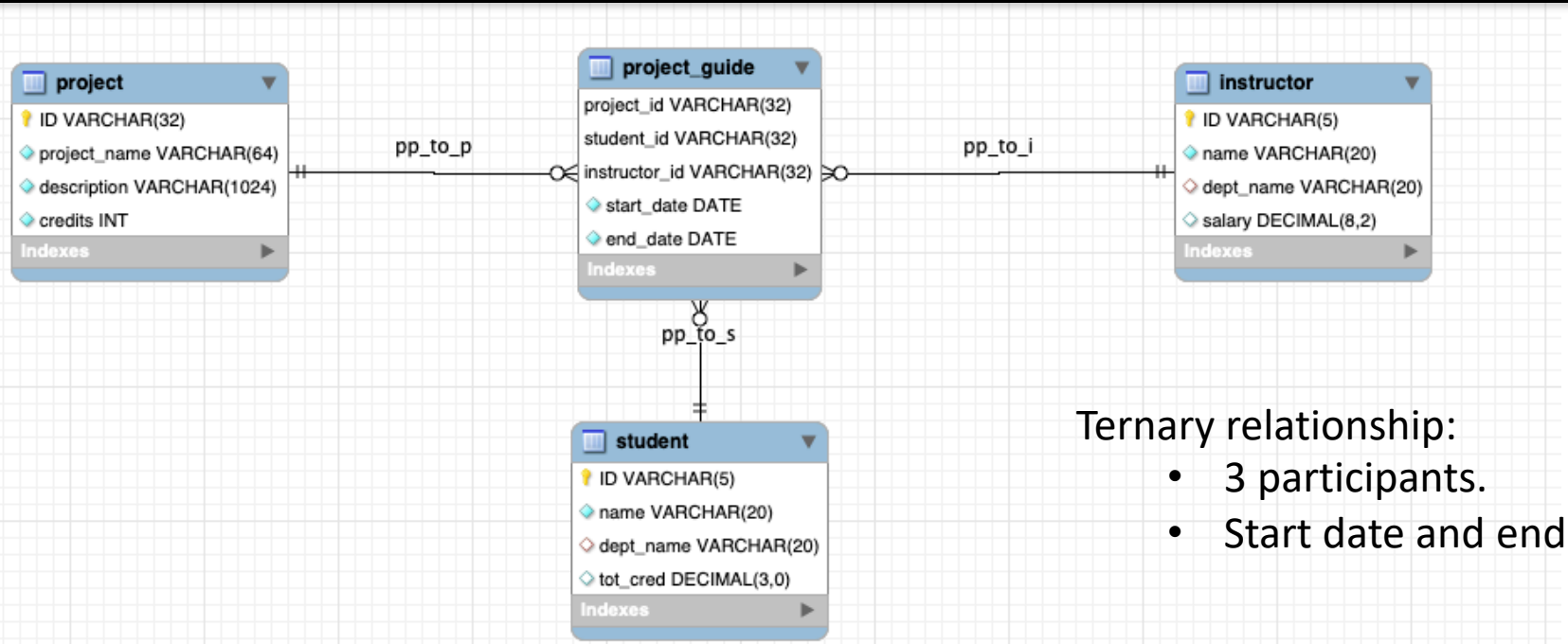
Non-binary Relationship Sets

- 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



- Note: In SQL/RDB, this requires four tables (relations)
  1. Instructor
  2. Student
  3. Project
  4. “Project Participants”

Consider Classicmodels Schema (https://www.mysqltutorial.org/mysql-sample-database.aspx)



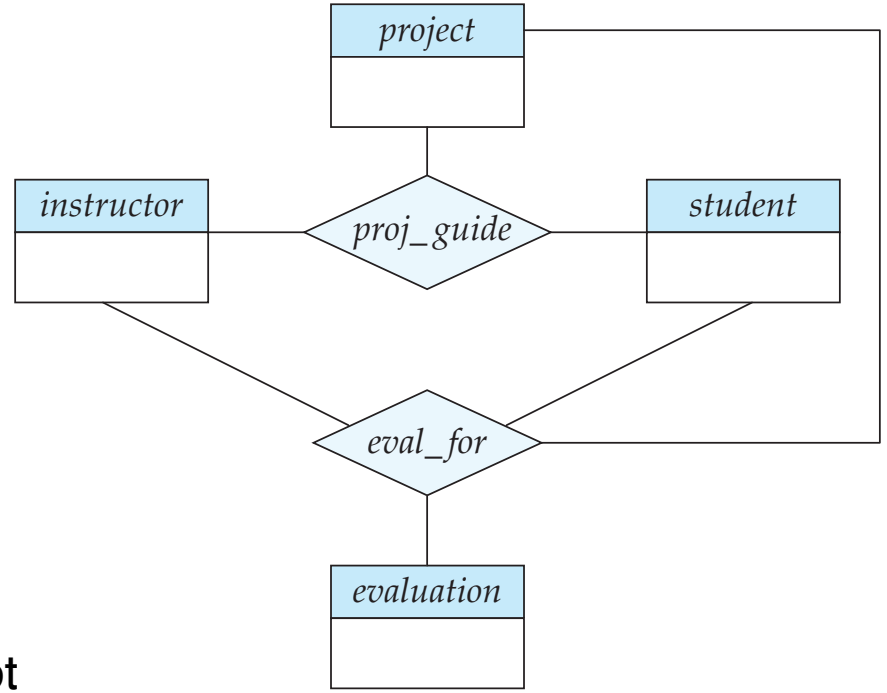
Ternary relationship:

- 3 participants.
- Start date and end date.

- The semantics of the model may not immediately be clear.
  - Clearly, the student exists independent of any specific project execution.
  - The same applies to the instructor.
  - Why doesn't the project just reference the instructor and student?
- The project is a "definition" of a project.
- The project "instance" comes into existence only when an instructor and student decide to work on a "predefined" project for some period of time.

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



OK, I admit it.
This looks weird.

This is also wrong. *eval_for* cannot reference any *project, student, instructor.*

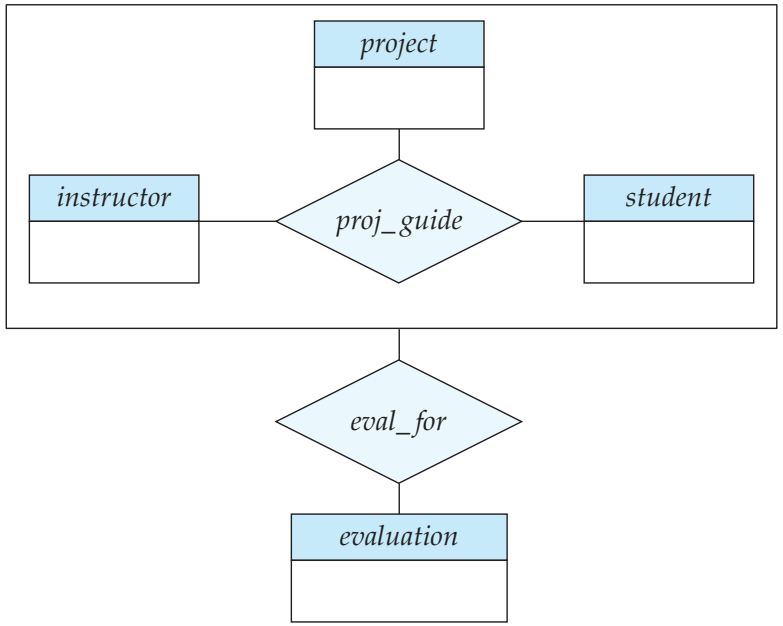
It must reference 3 reference by a *proj_guide.*

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.

- DFF Note: As long as you are OK with *evaluation_id* being NULL until there is an evaluation.

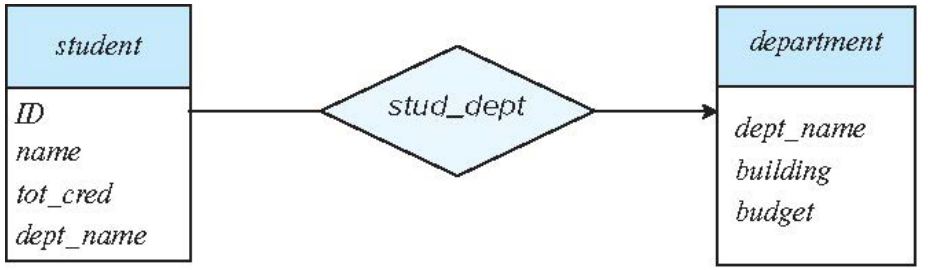
- *Enough slides. Let's build this.*

ER Design Issues

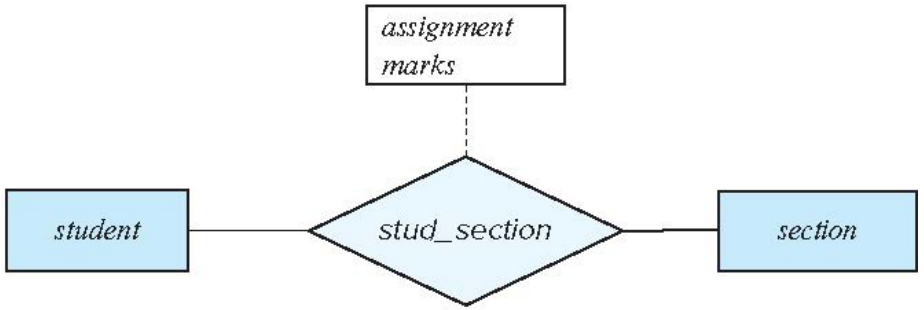
Design Issues

Common Mistakes in E-R Diagrams

- Example of erroneous E-R diagrams



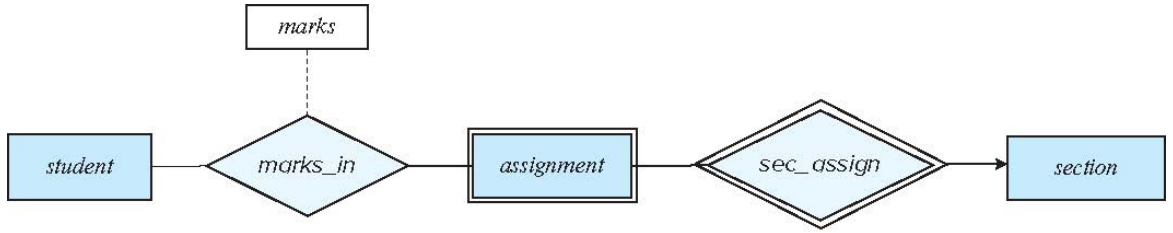
(a) Incorrect use of attribute



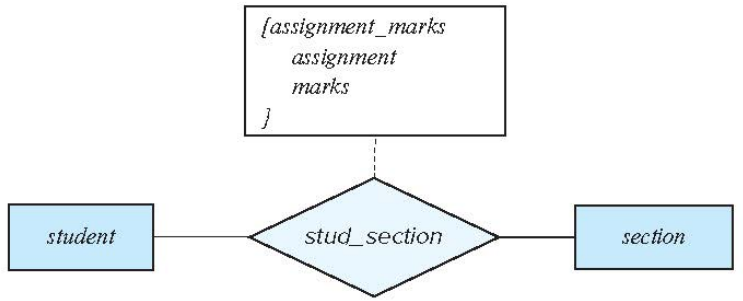
(b) Erroneous use of relationship attributes

Common Mistakes in E-R Diagrams (Cont.)

- Correct versions of the E-R diagram of previous slide



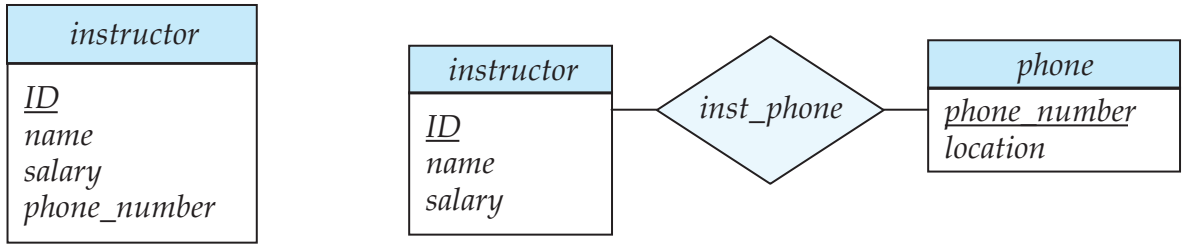
(c) Correct alternative to erroneous E-R diagram (b)



(d) Correct alternative to erroneous E-R diagram (b)

Entities vs. Attributes

- Use of entity sets vs. attributes

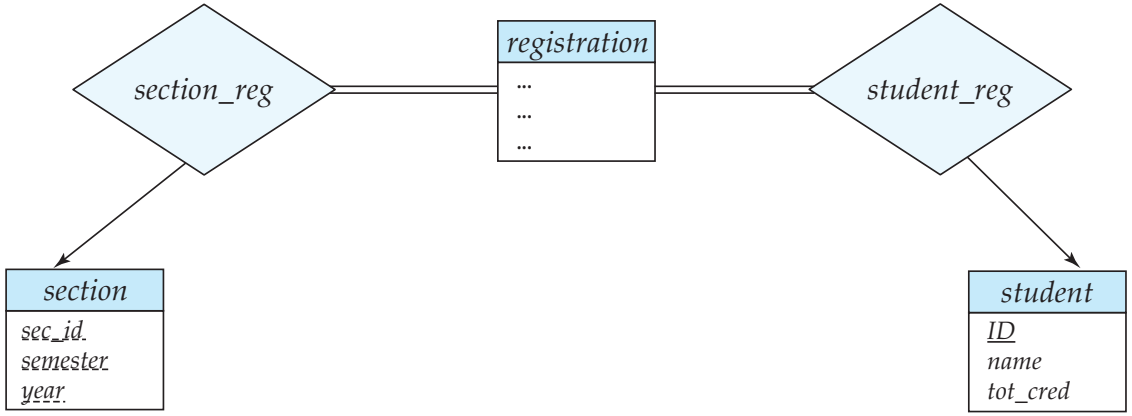


- Use of phone as an entity allows extra information about phone numbers (plus multiple phone numbers)

Entities vs. Relationship sets

- **Use of entity sets vs. relationship sets**

  Possible guideline is to designate a relationship set to describe an action that occurs between entities



- **Placement of relationship attributes**

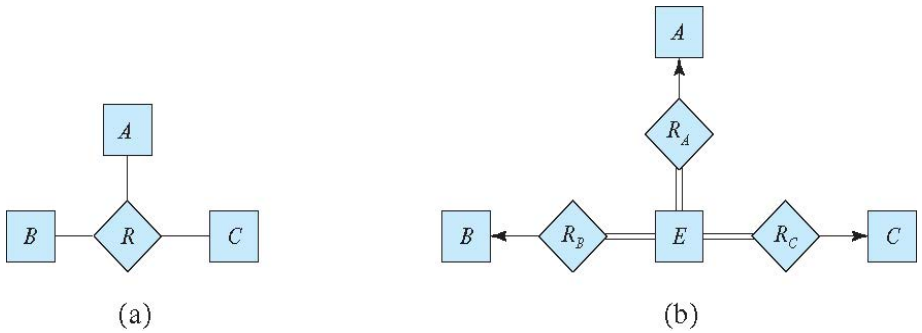
  For example, attribute date as attribute of advisor (sic?) or as attribute of student

Binary Vs. Non-Binary Relationships

- Although it is possible to replace any non-binary (n-ary, for $n > 2$) relationship set by a number of distinct binary relationship sets, a n-ary relationship set shows more clearly that several entities participate in a single relationship.
- Some relationships that appear to be non-binary may be better represented using binary relationships
  - For example, a ternary relationship *parents*, relating a child to his/her father and mother, is best replaced by two binary relationships, *father* and *mother*
    - Using two binary relationships allows partial information (e.g., only mother being known)
  - But there are some relationships that are naturally non-binary
    - Example: *proj_guide*

Converting Non-Binary Relationships to Binary Form

- In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
  - Replace R between entity sets A, B and C by an entity set E, and three relationship sets:
    1. R_A, relating E and A
    2. R_B, relating E and B
    3. R_C, relating E and C
  - Create an identifying attribute for E *and* add any attributes of R to E
  - For each relationship (a_i , b_i , c_i) in R, create
    1. a new entity e_i in the entity set E
    2. add (e_i , a_i) to R_A
    3. add (e_i , b_i) to R_B
    4. add (e_i , c_i) to R_C



Converting Non-Binary Relationships (Cont.)

- Also need to translate constraints
  - Translating all constraints may not be possible
  - There may be instances in the translated schema that cannot correspond to any instance of *R*
    - Exercise: *add constraints to the relationships* R_A, R_B *and* R_C to ensure that a newly created entity corresponds to exactly one entity in each of entity sets *A, B* and *C*
  - We can avoid creating an identifying attribute by making E a weak entity set (described shortly) identified by the three relationship sets

E-R Design Decisions

- The use of an attribute or entity set to represent an object.
- Whether a real-world concept is best expressed by an entity set or a relationship set.
- The use of a ternary relationship versus a pair of binary relationships.
- The use of a strong or weak entity set.
- The use of specialization/generalization – contributes to modularity in the design.
- The use of aggregation – can treat the aggregate entity set as a single unit without concern for the details of its internal structure.

Advanced SQL

Recursive Queries

Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation

DFF Note: This totally weirds me out. Please do not do this.

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book

The Power of Recursion

- Computing transitive closure using iteration, adding successive tuples to *rec_prereq*
  - The next slide shows a *prereq* relation
  - Each step of the iterative process constructs an extended version of *rec_prereq* from its recursive definition.
  - The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be **monotonic**. That is, if we add tuples to *prereq* the view *rec_prereq* contains all of the tuples it contained before, plus possibly more

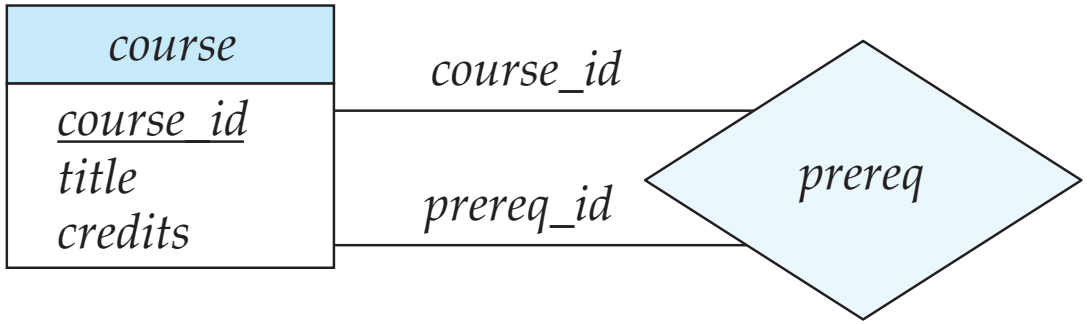
Example of Fixed-Point Computation

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-319	CS-101
CS-319	CS-315
CS-347	CS-319

Iteration Number	*Tuples in c1*
0	
1	(CS-319)
2	(CS-319), (CS-315), (CS-101)
3	(CS-319), (CS-315), (CS-101), (CS-190)
4	(CS-319), (CS-315), (CS-101), (CS-190)
5	done

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**.



- DFF Note: The concept of roles applies to all relationships/associations.

View and Security

Views and Security

- Views, which are a type of virtual tables allow users to do the following –
  - Structure data in a way that users or classes of users find natural or intuitive.
  - Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
  - Summarize data from various tables which can be used to generate reports.

  (https://www.tutorialspoint.com/sql/sql-using-views.htm)
- An Example – Consider this Section

Spring 2020			
Class ID	**Call#**	**Enrollment**	**Roster**
COMS 4111 W sec:002 **INTRODUCTION TO DATABASES**	13586	324/320	See Roster
COMS 4111 W sec:H02 **INTRODUCTION TO DATABASES**	25028	43/75	See Roster
COMS 4111 W sec:V02 **INTRODUCTION TO DATABASES**	25137	14/99	See Roster

This is the table definition for the CSV roster that I can download.

```
CREATE TABLE `class_roster` (
  `Uni` varchar(12) NOT NULL,
  `Student_ID` varchar(16) NOT NULL,
  `First_Name` varchar(64) NOT NULL,
  `Middle_Name` varchar(64) DEFAULT NULL,
  `Last_Name` varchar(64) NOT NULL,
  `Email` varchar(128) NOT NULL,
  `School` char(2) NOT NULL,
  `Level` char(3) NOT NULL,
  `Affiliation` varchar(8) NOT NULL,
  `Points` int NOT NULL,
  PRIMARY KEY (`Student_ID`),
  UNIQUE KEY `Uni_UNIQUE` (`Uni`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Loading Data (I)

```
use W4111Examples;

drop table if exists class_roster_0;

CREATE TABLE `class_roster_0` (
  `Uni` varchar(12) NOT NULL,
  `Student_ID` varchar(16) NOT NULL,
  `First_Name` varchar(64) NOT NULL,
  `Middle_Name` varchar(64) DEFAULT NULL,
  `Last_Name` varchar(64) NOT NULL,
  `Email` varchar(128) NOT NULL,
  `School` varchar(32) NOT NULL,
  `Level` varchar(32) NOT NULL,
  `Affiliation` varchar(32) NOT NULL,
  `Points` varchar(8) NOT NULL,
  PRIMARY KEY (`Uni`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

load data infile '/Users/donaldferguson/mysql/ClassRoster_20201_COMSW4111_002.csv'
   INTO TABLE class_roster_0 fields terminated by ',';

delete from class_roster_0 where uni='Uni';
```

- There are command to LOAD and export data to/from files.
- I am going to run the commands from an interactive shell.
- Normal use is inside an SQL script.
- There are also some security flags that I need to set.

I have to delete the header row.

Loading Data (II)

```
drop table if exists class_roster_h;
create table class_roster_h like class_roster_0;
load data infile '/Users/donaldferguson/mysql/ClassRoster_20201_COMSW4111_H02.csv'
   INTO TABLE class_roster_h fields terminated by ',';
delete from class_roster_h where uni='Uni';

drop table if exists class_roster_v;
create table class_roster_v like class_roster_0;
load data infile '/Users/donaldferguson/mysql/ClassRoster_20201_COMSW4111_V02.csv'
   INTO TABLE class_roster_v fields terminated by ',';
delete from class_roster_v where uni='Uni';
```

I thought I could start loading into a new table and setting the section to '002', 'H02', or 'V02' based on the table that I loaded.

Nope. One of you is in two sections, which messes up the *student_id* and 'uni.'

There is one in every crowd!

Loading Data (III) – How Did I Figure it Out?

After several failures, cursing, throwing my mouse. I ran the following query.

```
select a.uni, a.student_id, group_concat(a.uni) as unis,
    group_concat(a.student_id) as ids,
    group_concat(a.section) as sections,
    count(*) as count from
    (select *, '002' as section from class_roster_0
    union all
    select *, 'H02' as section from class_roster_h
    union all
    select *, 'V02' as section from class_roster_v) as a
    group by a.uni, a.student_id
    order by count desc;
```

There is only one. Now, I must decide what to do.
I decided to ignore the enrollment in the `H02` section and go with `002.`

Loading Data (IV) – Load the Data Into Table

```
drop table if exists class_roster;

create table class_roster as
      select class_roster_0.*, '002' as section from class_roster_0;

insert into class_roster
   select class_roster_h.*, 'H02' as section from class_roster_h
      where
         uni not in (select uni from class_roster);

insert into class_roster
   select class_roster_v.*, 'H02' as section from class_roster_v
      where
         uni not in (select uni from class_roster);
```

Loading Data (V) – Load the Data Into Table

```
update class_roster set Points=substr(Points,1,1);

ALTER TABLE `W4111Examples`.`class_roster`
CHANGE COLUMN `Points` `Points` INT NULL ,
CHANGE COLUMN `section` `section` ENUM('002', 'H02', 'V02') NOT NULL,
ADD PRIMARY KEY (`Student_ID`),
ADD UNIQUE INDEX `Uni_UNIQUE` (`Uni` ASC) VISIBLE;
```

Some schema/data cleanup!

View and Security

- Some of the fields are potentially sensitive, e.g.
  - UNI
  - student_id
- I can create a view that "hides" these columns from non-root users.

```
create view secure_roster as
SELECT
   concat(substr(uni,1,3),"******") as uni_prefix,
   First_name,
   Middle_name,
   Last_name,
   Email,
   School,
   Section
FROM W4111Examples.class_roster;
```

- Use GRANT/REVOKE to prevent access to underlying table.

Some Miscellaneous Stuff

Set Comparison

Set Comparison – "some" Clause

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

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

- Same query using > **some** clause

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

Definition of "some" Clause

- F <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**

Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor
    set salary = case
                    when salary <= 100000 then salary * 1.05
                    else salary * 1.03
                 end
```

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** $r \Leftrightarrow r = \emptyset$

Accessing SQL From Programming Languages (Hands-on-Walthrough)

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = '" + name + "'"
- Suppose the user, instead of entering a name, enters:
  - X' or 'Y' = 'Y
- then the resulting statement becomes:
  - "select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
  - which is:
    - select * from instructor where name = 'X' or 'Y' = 'Y'
  - User could have even used
    - X'; update instructor set salary = salary + 10000; --
- Prepared stament internally uses:
  "select * from instructor where name = 'X\' or \'Y\' = \'Y'
  - Always use prepared statements, with user inputs as parameters