COMS W4111-002/V02, Spring 22:
Take Home Midterm

Information and Instructions

- The midterm exam is due on 04-NOV at 11:59 PM. **You may not use late days.**
- See the Ed post #403 for submission instructions.
- Students should periodically check Ed post #404 for clarifications.
- You may use lecture notes, lecture slides, ... to help answer questions. You may also use online sources of information. If you use an online source,
  1. You must provide a link to the source.
  2. You are still responsible for ensuring the answer is correct. Not everything on the web is correct.
  3. You **MUST NOT** simply cut and paste, copy verbatim, ... You can use the information for guidance but must provide the answer in your own words and own code.
- You **MUST NOT** collaborate with other students or other people in any way. You may discuss the exam with TAs and instructors.

Environment Setup

Notes:

1. This section tests your environment.

2. You will need to change the MySQL userID and password in some of the cells below to match your configuration.

3. You may need to load data and copy databases. The relevant questions provide information.

4. You will need to:
   A. Install the Classic Models database if you have not already done so.
   B. Install the sample database that comes with the recommended textbook if you have not already done so.

In [1]:
```
%load_ext sql
```

In [2]:
```
%sql mysql+pymysql://root:dbuserbdbuser@localhost
```

In [3]:
```
%sql select * from classicmodels.customers where country='Spain'
```

```
 * mysql+pymysql://root:***@localhost
7 rows affected.
```

Out[3]:

customerNumber	customerName	contactLastName	contactFirstName	phone	addressLine1	addressLine2	city	state	postalCode	country	salesRepEmployeeNumber	creditLimit
141	Euro+ Shopping Channel	Freyre	Diego	(91) 555 94 44	C/ Moralzarzal, 86	None	Madrid	None	28034	Spain	1370	227600.00
216	Enaco Distributors	Saavedra	Eduardo	(93) 203 4555	Rambla de Cataluña, 23	None	Barcelona	None	08022	Spain	1702	60300.00
237	ANG Resellers	Camino	Alejandra	(91) 745 6555	Gran Vía, 1	None	Madrid	None	28001	Spain	None	0.00
344	CAF Imports	Fernandez	Jesus	+34 913 728 555	Merchants House	27-30 Merchant's Quay	Madrid	None	28023	Spain	1702	59600.00
458	Corrida Auto Replicas, Ltd	Sommer	Martín	(91) 555 22 82	C/ Araquil, 67	None	Madrid	None	28023	Spain	1702	104600.00
465	Anton Designs, Ltd.	Anton	Carmen	+34 913 728555	c/ Gobelas, 19-1 Urb. La Florida	None	Madrid	None	28023	Spain	None	0.00
484	Iberia Gift Imports, Corp.	Roel	José Pedro	(95) 555 82 82	C/ Romero, 33	None	Sevilla	None	41101	Spain	1702	65700.00

In [4]:
```
from sqlalchemy import create_engine
```

In [5]:
```
sql_engine = create_engine("mysql+pymysql://root:dbuserbdbuser@localhost")
```

In [6]:
```
import pandas as pd
```

In [7]:
```
sql = """
    select customerName, customerNumber, city, country from classicmodels.customers
        where country = 'Spain'
"""

res = pd.read_sql(sql, con=sql_engine)
```

In [8]:
```
res
```

Out[8]:

	customerName	customerNumber	city	country
0	Euro+ Shopping Channel	141	Madrid	Spain
1	Enaco Distributors	216	Barcelona	Spain
2	ANG Resellers	237	Madrid	Spain
3	CAF Imports	344	Madrid	Spain
4	Corrida Auto Replicas, Ltd	458	Madrid	Spain
5	Anton Designs, Ltd.	465	Madrid	Spain
6	Iberia Gift Imports, Corp.	484	Sevilla	Spain

In [9]:
```
import pymysql
```

In [10]:
```
sql_conn = pymysql.connect(
    user="root",
    password='dbuserbdbuser',
    host="localhost",
    port=3306,
    cursorclass=pymysql.cursors.DictCursor,
    autocommit=True)
```

In [11]:
```
try:
    cur = sql_conn.cursor()
    res = cur.execute(sql)
    res = cur.fetchall()
except Exception as e:
    print("Exception ", e, "is probably NOT good.")
```

In [12]:
```
res
```

Out[12]:
```
[{'customerName': 'Euro+ Shopping Channel',
  'customerNumber': 141,
  'city': 'Madrid',
  'country': 'Spain'},
 {'customerName': 'Enaco Distributors',
  'customerNumber': 216,
  'city': 'Barcelona',
  'country': 'Spain'},
 {'customerName': 'ANG Resellers',
  'customerNumber': 237,
  'city': 'Madrid',
  'country': 'Spain'},
 {'customerName': 'CAF Imports',
  'customerNumber': 344,
  'city': 'Madrid',
  'country': 'Spain'},
 {'customerName': 'Corrida Auto Replicas, Ltd',
  'customerNumber': 458,
  'city': 'Madrid',
  'country': 'Spain'},
 {'customerName': 'Anton Designs, Ltd.',
  'customerNumber': 465,
  'city': 'Madrid',
  'country': 'Spain'},
 {'customerName': 'Iberia Gift Imports, Corp.',
  'customerNumber': 484,
  'city': 'Sevilla',
  'country': 'Spain'}]
```

In [13]:
```
cur.close()
```

Written Questions

Note:

"If you can’t explain something in a few words, try fewer." – Robert Brault

"Professor Ferguson has the patience of a ferret that just drank a double espresso. If your answer is long, he gets bored and cranky, and deducts points." - Anonymous TA advising students in a previous semester.

- We expect brief, succinct answers.
- We deduct points for bloviating.

W1

Briefly explain the differences between:

1. *Candidate Key* and *Super Key*.
2. *Primary Key* and *Unique Key*.
3. *Natural Key* and *Surrogate Key*.

Answer

- A Super Key is a combination of one or more attributes that can uniquely identify tuples in a relation, whereas a Candidate Key is a type of superkey whose attributes can form into a superkey. It is often a subset of a superkey, where none of the proper subsets of a candidate key can be a superkey.
- There can only be one Primary Key, which is a selected Candidate Key, as opposed to multiple Unique Keys in a table. Moreover, Primary Keys cannot contain NULL values, while a Unique Key can accept NULL values.
- A Natural Key has meaning to a business outside of the database table, and already exists as a unique identifier. A Surrgoate Key is often only used within the context of the table, and is generated by a system or user to uniquely identify a tuple.

Ref: Textbook, https://www.mssqltips.com/sqlservertip/5431/surrogate-key-vs-natural-key-differences-and-when-to-use-in-sql-server/

W2

SQL supports the modifier *ON UPDATE* and *ON DELETE* for foreign key definitions. The database engines do not support *ON INSERT*. Why would implementing *ON INSERT* be impossible in most scenarios?

Answer

- *ON INSERT* wouldn't work for foreign key definitions since there are additional constraints and columns in the child class that are hard to account for when inserting a new tuple in the parent class, as opposed to *ON UPDATE* and *ON DELETE* cascades, which can simply apply the same update or deletion from the parent class to the child class. For example, if we insert a new row in the parent (referenced) class, we wouldn't necessarily know what values to insert in the child (referencing) class, which might have additional different columns.

W3

Codd's Third Rule states, "Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type."

Consider a table of the form:

```
create table if not exists orders
(
    uni            varchar(12) not null
        primary key,
    last_name      varchar(64) not null,
    first_name     varchar(64) not null,
    age            int         null
)
```

If we do not know the value of `age,` a poor design would use a convention like setting age to `-1` instead of using `NULL`. Give an example of a query for which not following Codd's Thurd Rule would result in an incorrect answer.

Answer

- If we wanted to query people whose age is over a nonnegative number, then people with an age of "-1" entered would be included in this query if NULL is not used, resulting in an incorrect answer. For example, for the query:

  ```
  SELECT
      *
  FROM orders
  WHERE
      age > 2;
  ```

W4

The relational model and SQL are *closed* under their operations. Briefly explain why this concept is critical joining three tables?

Answer

- The concept of closure implies that the output from one operation can be input into another operation, allowing for nested expressions-when joining three tables in succession, the output of joining two tables can be joined with the third table, allowing us to join all three tables:

  ```
  table1
  NATURAL JOIN
  table2
  NATURAL JOIN
  table3
  ```

W5

Codd's 6th rule states, "All views that are theoretically updatable are also updatable by the system."

Using the following table definition, use SQL (`CREATE VIEW`) to define:

1. Two views of the table for which it is impossible to update the base table through the view.
2. One view for which it is possible to update through the view.

- You do not need to execute the create statement. We are focusing on your understanding.

```
create table S22_W4111_Midterm.midterm_employees
(
    social_security_no char(9) not null
        primary key,
    last_name varchar(64) not null,
    first_name varchar(64) not null,
    dept_no char(4) not null,
    salary double not null
);
```

Answer

- It is impossible to update base table through the view if we use aggregate functions:

```
CREATE VIEW AS
    SELECT
        social_security_no,
        SUM(salary) AS total_salary
    FROM
        S22_W4111_Midterm.midterm_employees
    GROUP BY
        dept_no;
```

or

```
CREATE VIEW AS
    SELECT
        social_security_no,
        STD(salary) AS std_salary,
        VARIANCE(salary) AS var_salary
    FROM
        S22_W4111_Midterm.midterm_employees
    GROUP BY
        dept_no;
```

- A view with a simple query can be used to update the base view:

```
CREATE VIEW AS
    SELECT
        *
    FROM
        S22_W4111_Midterm.midterm_employees
    WHERE
        salary > 40000;
```

W6

Consider the following table:

```
create table S22_W4111_Midterm.midterm_employees
(
    phone_number varchar(64) not null primary key,
    last_name varchar(64) not null,
    first_name varchar(64) not null,
);
```

Telephone numbers are of the form `country code` followed by the phone number. Some examples are:

- `01 212-555-1212`
- `44 038 717 980 01`

Why is storing the number as a single `varchar` a poor design? What problems could that cause? How would you change the table definition.

Answer

- Storing as a varchar is unatomic and can cause data entry and processing errors, since the phone number can be broken down into smaller units. I would define an additional country_code column to make the definition more atomic. Moreover, to avoid discrepancies in phone numbers, I would add a constraint to prevent dashes in the phone number, so only numbers and spaces are included.

    ```
    country_code VARCHAR(4) NOT NULL,
    phone_number VARCHAR(64) NOT NULL,
    PRIMARY KEY (country_code, phone_number),
    CHECK (phone_number NOT LIKE '%-%')
    ```

W7

Briefly explain the differences between:

- Database stored procedure
- Database function
- Database trigger

Answer

- A stored procedure is a set of SQL statements that can be executed, accepts input and output parameters, and can modify data in a database. A database function, howeover, is a set of SQL statements that cannot modify data in a database, but can retrive data and return columns or tables—a function can be called from inside a stored procedure and trigger as well. Moreover, a function must always have a return statement, while procedures do not need to. A trigger automatically executes a set of SQL statements on update, delete, or insert-it can call stored procedures and functions, but cannot be called from a procedure or function. Moreover trigger does not return anything.

Ref: Textbook

W8

Briefly explain:

- Natural join
- Equi-join
- Theta join
- Self-join

Give a scenario in which a Natural Join would produce an incorrect answer.

Answer

- Natural join combines the left and right relations using the columns with the same attribute names and same corresponding values-it drops rows where the columns don't match.
- Equi-join is a type of join that only employs equality comparison operators, returning rows where specified columns are equal.
- Theta join combines the left and right relations using a general join condition with any type of comparison operator.
- Self-join joins a table with itself, using different comparison operators to compare data within a table, which is useful if there are references/inter-table relationships.

- Natural join wouldn't work correctly if unrelated columns in the left and right relations are mistakenly equated, and the correct columns are not specified in the join. For example, if we have two relations with columns —employee (employee_id, last_name, first_name, dept_name) and manager (employee_id, last_name, first_name, dept_name)—and did a natural join to match employees and managers with the same department name, without specifying a column, the output would be wrong, as the natural join would try to match employee_id, last_name, and first_name for both relations, along with dept_name. We could specify a column with _USING (dept*name)* when doing the natural join.

Ref: https://www.w3computing.com/sqlserver2012/theta-join-self-join-semi-join/

W9

We have seen examples in SQL of implementing relationships between two tables using an *associative entity* table instead of foreign keys. Give two reasons for using the associative entity design pattern.

Answer

- If we wanted to work with multi-valued attributes between two entities, we could use an associative entity to represent many-to-many connections/associations between them-they allow us to better represent non one-to-one relationships between entities in a more organized way.
- Instead of having to store repetitive data (same columns/attributes from multiple tables) for an entity in different tables, you can use an associative entity to represent that entity (for instance, the id of that entity), storing that association instead of having to store repetitive column data.

W10

Professor Ferguson often adds a `LIMIT` to his example queries in Jupyter Notebooks. Assume the table `customers` is very large. Why would the query

```
select * from customers
```

cause problems for the notebook? How does addining `limit 20` solve this problem for an example?

Answer

- Jupyter notebook has a limit/cap on the data output that can be displayed/processed, which can cause the notebook to hang and crash; limiting the output to 20 rows prevents this from happening and allows Jupyter to handle the output.

Relational Algebra

R1

- You can assume that the type for the columns in this question are `varchar(32).`
- Translate the following relational schema definition into an equivalent SQL `CREATE TABLE` statement.
- You do not need to execute the statement. We are focusing on understanding.

$$(\underline{policy\_type}, \underline{policy\_no}, policy\_date) \tag{1}$$

Answer

```
CREATE TABLE policies (
    policy_type varchar(32),
    policy_no varchar(32),
    policy_date varchar(32)

    PRIMARY KEY (policy_type, policy_no)
)
```

R2

Use the RelaX calculator with the textbook's sample data for this question.

Answer Format: Your answer to the relational algebra query should contain three secions:

1. A Markdown cell with the relational algebra statement.
2. An image capture of the query exection tree.
3. An image capture of the result table.

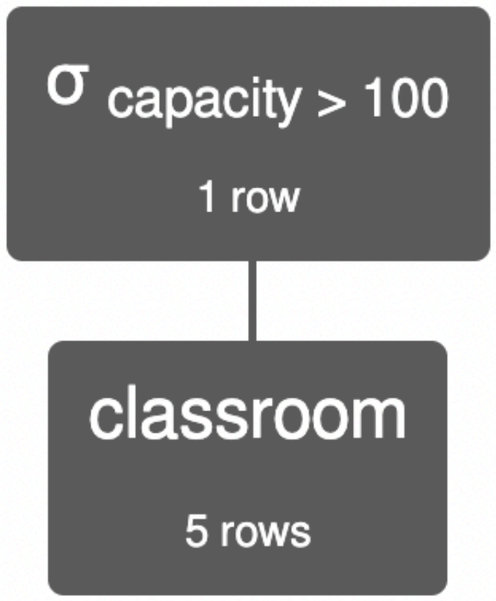
For example, a query returning all classrooms with `capacity > 100` would have the following cells:

Relational Algebra Statement

`σ capacity > 100 (classroom)`

Query Execution Tree

This must show the execution tree and relational algebra statement.



$$\sigma_{\text{capacity} > 100}(\text{classroom})$$

Execution time: 1 ms

Query Result

This must show the relational algebra statement and the result table.

$$\sigma_{\text{capacity} > 100}(\text{classroom})$$

Execution time: 1 ms

classroom.building	classroom.room_number	classroom.capacity
'Packard'	101	500

The Question

In the sample data,

- The relation `advisor` represents the advisor-student relationship between a `student` and an `instructor.`
- Write a relational algebra expression that produces the following information:
  - The `ID` and `name` of `student`, and the `ID` and `name` on `instructor`
  - For students and instructors in the CS department.
  - The information should be `null` if the instructor does not advise a student and vice-versa.
- To help, you are trying to produce the following information.
- **Note:**
  - You **may not** use full outer join.
  - You will have to use the column rename operation for project.

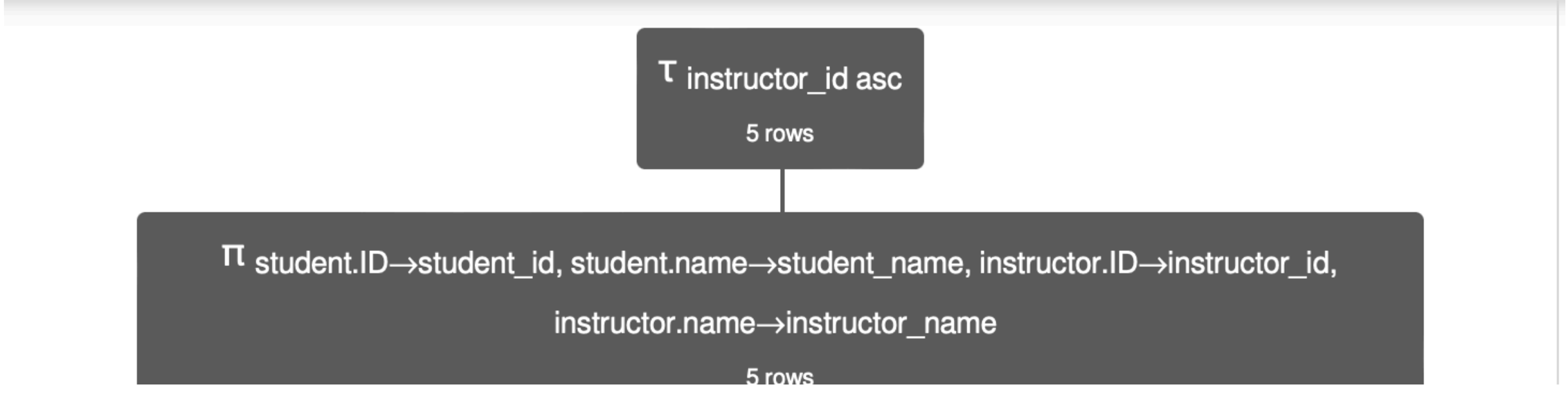
student_id	student_name	instructor_id	instructor_name
12345	'Shankar'	10101	'Srinivasan'
128	'Zhang'	45565	'Katz'
76543	'Brown'	45565	'Katz'
null	null	83821	'Brandt'
54321	'Williams'	null	null

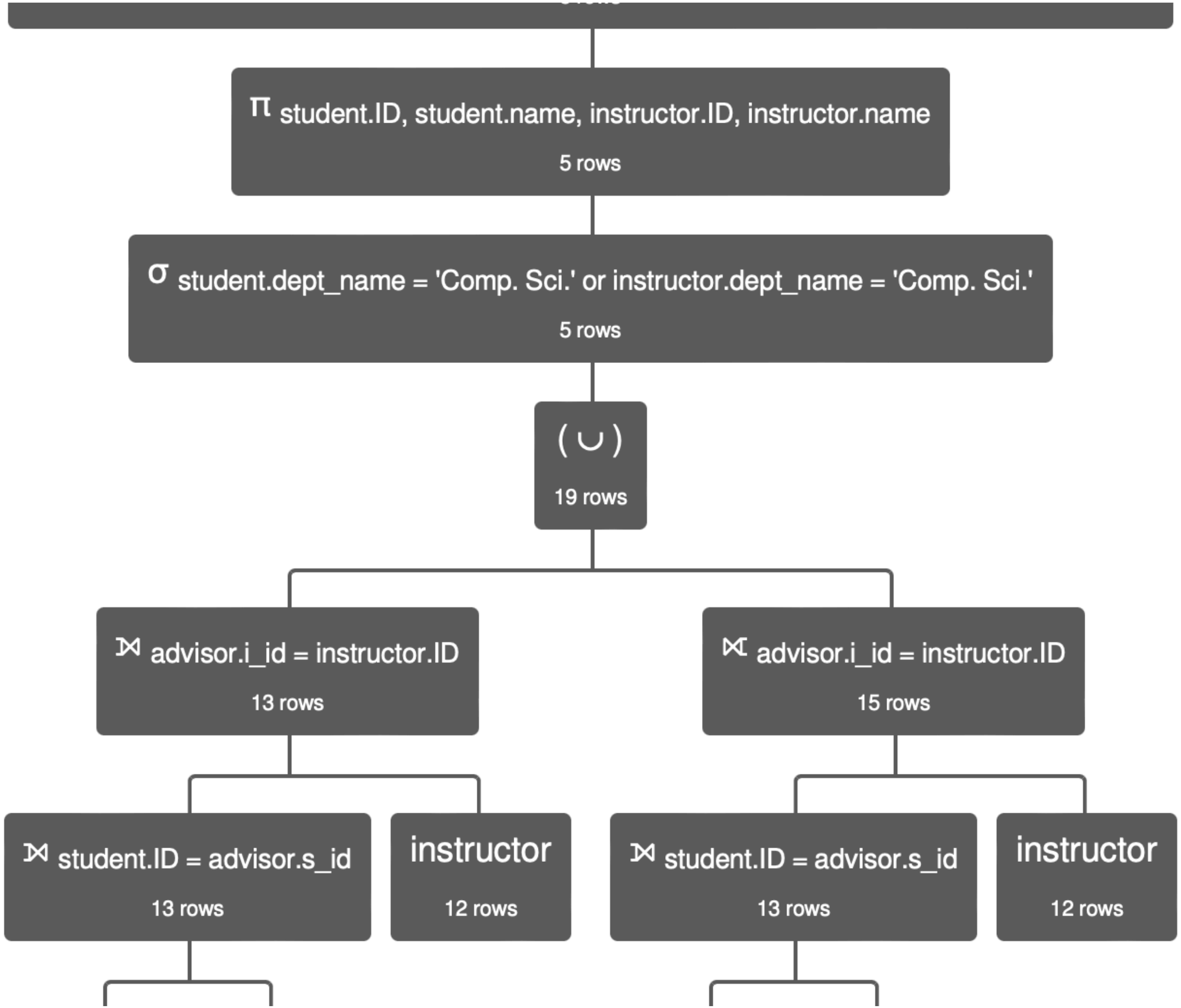
Answer

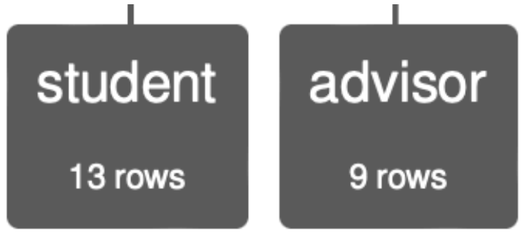
Query

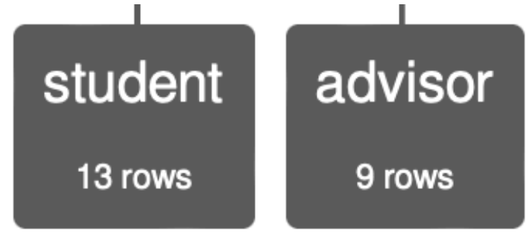
```
τ instructor_id asc (π student.ID→student_id, student.name→student_name, instructor.ID→instructor_id, instructor.name→instructor_name πstudent.ID, student.name,
instructor.ID, instructor.name (σ student.dept_name = 'Comp. Sci.' ∨ instructor.dept_name = 'Comp. Sci.' (student ⟖ student.ID = advisor.s_id advisor ⟖
advisor.i_id = instructor.ID instructor ∪ student ⟖ student.ID = advisor.s_id advisor ⟗ advisor.i_id = instructor.ID instructor)))
```

Query Tree




π student.ID, student.name, instructor.ID, instructor.name
5 rows
σ student.dept_name = 'Comp. Sci.' or instructor.dept_name = 'Comp. Sci.'
5 rows
(∪)
19 rows
⋈ advisor.i_id = instructor.ID
13 rows
⋈ advisor.i_id = instructor.ID
15 rows
⋈ student.ID = advisor.s_id
13 rows
instructor
12 rows
⋈ student.ID = advisor.s_id
13 rows
instructor
12 rows





$$\tau_{\text{instructor\_id asc}} (\pi_{\text{student.ID}\rightarrow\text{student\_id, student.name}\rightarrow\text{student\_name, instructor.ID}\rightarrow\text{instructor\_id, instructor.name}\rightarrow\text{instructor\_name}} \pi_{\text{student.ID, student.name, instructor.ID, instructor.name}} (\sigma_{\text{student.dept\_name = 'Comp. Sci.' or instructor.dept\_name = 'Comp. Sci.'}} (((\text{student} \bowtie_{\text{student.ID = advisor.s\_id}} \text{advisor}) \bowtie_{\text{advisor.i\_id = instructor.ID}} \text{instructor}) \cup ((\text{student} \bowtie_{\text{student.ID = advisor.s\_id}} \text{advisor}) \bowtie_{\text{advisor.i\_id = instructor.ID}} \text{instructor}))))$$

Execution time: 3 ms

Query Result

$$\tau_{\text{instructor\_id asc}} (\pi_{\text{student.ID}\rightarrow\text{student\_id, student.name}\rightarrow\text{student\_name, instructor.ID}\rightarrow\text{instructor\_id, instructor.name}\rightarrow\text{instructor\_name}} \pi_{\text{student.ID, student.name, instructor.ID, instructor.name}} (\sigma_{\text{student.dept\_name = 'Comp. Sci.' or instructor.dept\_name = 'Comp. Sci.'}} (((\text{student} \bowtie_{\text{student.ID = advisor.s\_id}} \text{advisor}) \bowtie_{\text{advisor.i\_id = instructor.ID}} \text{instructor}) \cup ((\text{student} \bowtie_{\text{student.ID}}$$

$$= \text{advisor.s\_id}\ \text{advisor}\ ) \bowtie \text{advisor.i\_id} = \text{instructor.ID}\ \text{instructor}\ ))))$$

Execution time: 3 ms

student_id	student_name	instructor_id	instructor_name
12345	'Shankar'	10101	'Srinivasan'
128	'Zhang'	45565	'Katz'
76543	'Brown'	45565	'Katz'
null	*null*	83821	'Brandt'
54321	'Williams'	*null*	*null*

Entity Relationship Model and Implementation

Explanation

For this problem,

- There is a written description of a data model.
- You must draw (using Lucidchart) a Crow's Foot notation ER diagram for the *logical model* implementing the written description. Note that not all concepts in the data model description can be modeled in the ER diagram.
- You must then write SQL DDL statements and execute the statements to create tables and constraints realizing the written data model description.

Written Description

There are the following entity types:

- `employee`:
  - `employee_id` is a 4 digit number that may begin with 0, e.g. `0201`. An employee must have a unique `employee_id`.
  - `last_name` is a string with maximum length 64. An employee must have a last name.
  - `first_name` is a string with maximum length 64. An employee must have a first name.
  - `employee_type` must be one of the following values, `regular`, `manager`, `executive.`
  - `employee_email` may be unknown, but if known it must be unique.

- `project`:
  - `project_code` is a two character code that must contain two uppercase English letters (A, B, ..., Z) and is unique.
  - `project_name` is a text string of maximum length 32.

- `project_team` is an associative entity of the form (none of the values may be NULL):
  - `project_code`
  - `sponsor_id` is `employee_id` of an employee who is an `executive.`
  - `manager_id` is the `employee_id` of an employee who is a `manager`.
  - `employee_id` is the `employee_id` of an employee working on the project.

- Constraints on `project_team`:
  - `project_code` is unique in the table.
  - An `employee_id` can appear at most three times.
  - The combination of `(sponsor_id, manager_id)` can appear at most once.

Note:

Being able to make sense out of a written description of a data model and producing a reasonably accurate diagram and DDL is an important skill. Most of the time, you will have to make assumptions or modify/extend constraints. The business statekholder/partner specifying the data model is not a database expert. There description may be incomplete or confused.

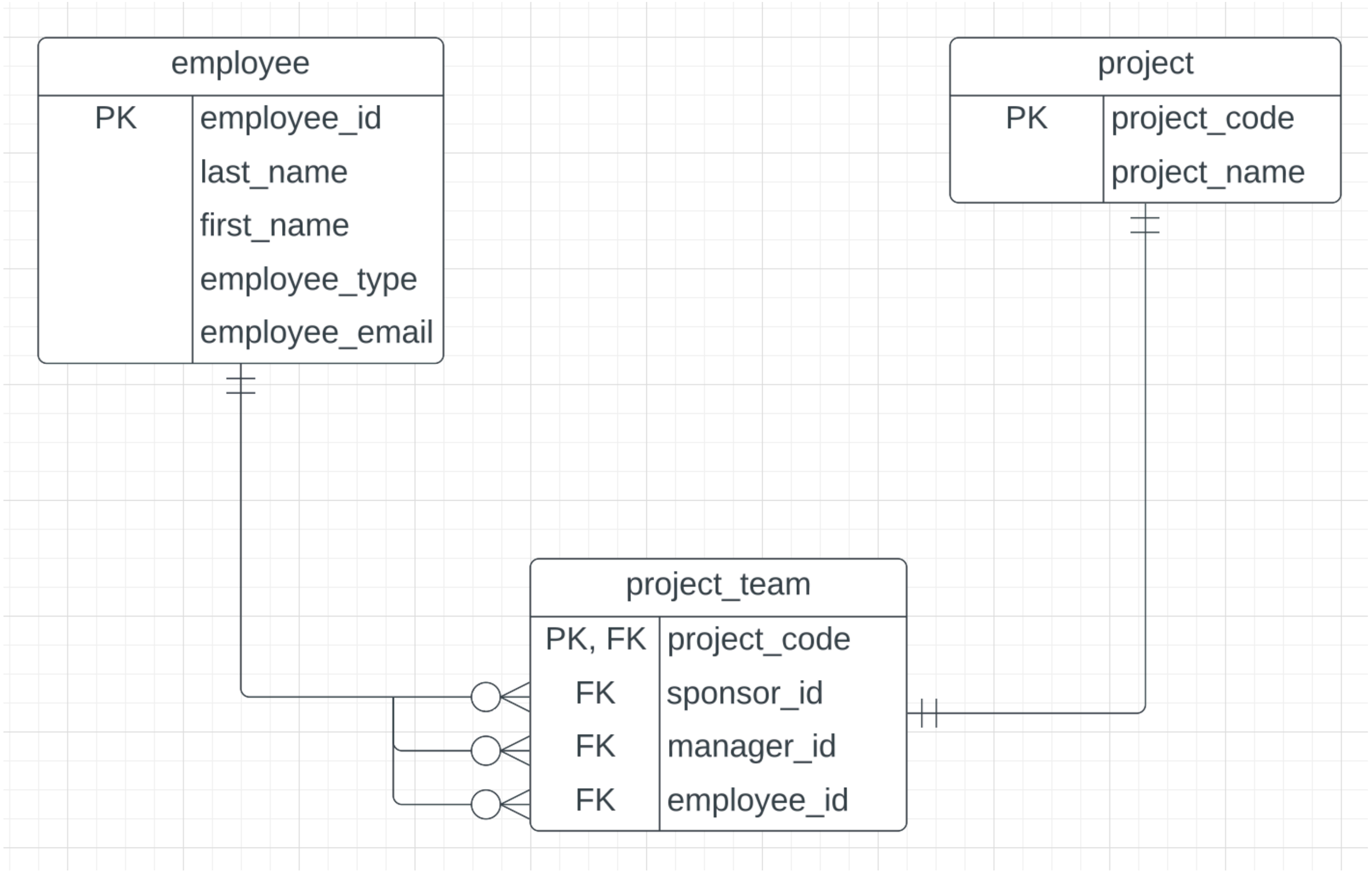
We are looking for your ability to apply what you have learned to a complex problem. If you have to make assumptions, note them. We will not deduct points for reasonable assumptions.

You may have to use check constraints, triggers, foreign keys, ... in your DDL.

Answer

Crow's Foot ER Diagram

- Assumptions:
  - Project and project team have one-to-one relationship (one project has one project team at any given moment)
  - Many to many relationship between employee and project
  - One to many relationship between employee and project team (employee has many project teams)



DDL Statements and Execution

In [14]:
```
%%sql

drop database if exists f22_midterm;

create database f22_midterm;
```

```
 * mysql+pymysql://root:***@localhost
3 rows affected.
1 rows affected.
```

Out[14]:
```
[]
```

In [15]:
```
%%sql
USE f22_midterm;

# Create tables
CREATE TABLE employee (
    employee_id VARCHAR(4) NOT NULL,
    last_name VARCHAR(64) NOT NULL,
    first_name VARCHAR(64) NOT NULL,
    employee_type ENUM ('regular', 'manager', 'executive') NOT NULL,
    employee_email VARCHAR(64),
    PRIMARY KEY (employee_id),
    CONSTRAINT employee_unique_email UNIQUE (employee_email),
    CONSTRAINT employee_check_id CHECK (employee_id REGEXP '[0-9][0-9][0-9][0-9]')
);

CREATE TABLE project (
    project_code VARCHAR(2) NOT NULL,
    project_name VARCHAR(32),
    PRIMARY KEY (project_code),
    CONSTRAINT project_check_code CHECK (project_code REGEXP '[A-Z][A-Z]')
);

CREATE TABLE project_team (
    project_code VARCHAR(2) NOT NULL,
    sponsor_id VARCHAR(4) NOT NULL,
    manager_id VARCHAR(4) NOT NULL,
    employee_id VARCHAR(4) NOT NULL,
    PRIMARY KEY (project_code),
    CONSTRAINT sponsor_manager_id_unique UNIQUE (sponsor_id, manager_id),
    FOREIGN KEY (project_code) REFERENCES project (project_code),
    FOREIGN KEY (sponsor_id) REFERENCES employee (employee_id),
    FOREIGN KEY (manager_id) REFERENCES employee (employee_id),
    FOREIGN KEY (employee_id) REFERENCES employee (employee_id)
);

# Functions: determine employee id count and (sponsor_id, manager_id) pair count in project_team
# Also check that sponsor_id corresponds to an executive, and manager_id correponds to a manager
CREATE FUNCTION employee_id_count (employee_id VARCHAR(4))
RETURNS integer DETERMINISTIC

BEGIN
    DECLARE e_count integer;
        SELECT
            COUNT(*) INTO e_count
        FROM project_team
        WHERE
            project_team.employee_id = employee_id;
    RETURN e_count;
END;
```

```
# Gets employee type with given employee_id
CREATE function get_employee_type (e_id VARCHAR(4))
returns VARCHAR(12) DETERMINISTIC

BEGIN
    DECLARE e_type VARCHAR(12);
        SELECT
            employee_type INTO e_type
        FROM employee
        WHERE
            employee_id = e_id;

    RETURN e_type;
END;

# Triggers: set values to null to trigger not null constraint before update/insert if conditions aren't satisfied
# so that values won't be inserted into table
CREATE TRIGGER insert_project_team_trigger BEFORE INSERT ON project_team
FOR EACH ROW
BEGIN
    IF employee_id_count(NEW.employee_id) >= 3 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot complete insert; employee_id occurs 3 times.';
    END IF;

    IF get_employee_type(NEW.sponsor_id) != 'executive' THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot complete insert; sponsor_id must correspond to employee type of executive';
    END IF;

    IF get_employee_type(NEW.manager_id) != 'manager' THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot complete insert; manager_id must correspond to employee type of manager';
    END IF;
END;

CREATE TRIGGER update_project_team_trigger BEFORE UPDATE ON project_team
FOR EACH ROW
BEGIN
    IF employee_id_count(NEW.employee_id) >= 3 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot complete update; employee_id occurs 3 times.';
    END IF;

    IF get_employee_type(NEW.sponsor_id) != 'executive' THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot complete update; sponsor_id must correspond to employee type of executive';
    END IF;

    IF get_employee_type(NEW.manager_id) != 'manager' THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot complete update; manager_id must correspond to employee type of manager';
    END IF;
END;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
0 rows affected.
```

Out[15]:
```
[]
```

SQL Queries

- You will use the Classic Models data for these questions.
- You loaded this database in a previous HW and tested that you have the database in the setup section.

S1

- Produce a table of the form: $(country, total_country_revenue)$.
- Each entry in `orderdetails` produces revenue `quantityOrdered*priceEach`.
- The revenue an `order` produces is the sum of the revenue from the `orderdetails` in the `order`, *but only if the order's status is* `shipped.`
- An `order` has a `customer` and the `customer` is in a country. The `total_country_revenue` is the sum over all shipped orders for customers in a country.
- The result table should have `total_country_revenue` nicely formated, sorted descending and have only countries with `total_country_revenue` >= 200,000.
- **NOTE:** You should be able to produce the answer without my providing the correct query output. I was giggling diabolically like the Riddler from Batman when writing the question. Then something like the following happened.

- So the output is below. You must match the output.

In [16]:
```sql
%%sql
USE classicmodels;
WITH
    total_revenue_grouped AS
        (
        SELECT
            country,
            SUM(quantityOrdered * priceEach) as total_country_revenue_numeric
        FROM orderdetails
        NATURAL JOIN
            orders
        NATURAL JOIN
            customers
        WHERE
            status='Shipped'
        GROUP BY
            country
        )
SELECT
    country,
    CONCAT('$', FORMAT(total_country_revenue_numeric, 2)) as total_country_revenue
FROM total_revenue_grouped
WHERE
    total_country_revenue_numeric >= 200000
ORDER BY
    total_country_revenue_numeric DESC;
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
11 rows affected.
```

Out[16]:

country	total_country_revenue
USA	$3,032,204.26
France	$965,750.58
Spain	$947,470.01
Australia	$509,385.82
New Zealand	$416,114.03
UK	$391,503.90
Italy	$360,616.81
Finland	$295,149.35
Norway	$270,846.30
Singapore	$263,997.78
Canada	$205,911.86

Out[17]:

country	total_country_revenue
USA	$3,032,204.26
France	$965,750.58
Spain	$947,470.01
Australia	$509,385.82
New Zealand	$416,114.03
UK	$391,503.90
Italy	$360,616.81
Finland	$295,149.35
Norway	$270,846.30
Singapore	$263,997.78
Canada	$205,911.86

S2

Return the product information for products not ordered by any French customer (Customer's `country` is France).

I did not want to get hit by Batman again. So, here is a sample answer.

```
 * mysql+pymysql://root:***@localhost
2 rows affected.
```

Out[18]:

productCode	productName	productLine	productScale	productVendor	productDescription	quantityInStock	buyPrice	MSRP
S18_3233	1985 Toyota Supra	Classic Cars	1:18	Highway 66 Mini Classics	This model features soft rubber tires, working steering, rubber mud guards, authentic Ford logos, detailed undercarriage, opening doors and hood, removable split rear gate, full size spare mounted in bed, detailed interior with opening glove box	7733	57.01	107.57
S18_4027	1970 Triumph Spitfire	Classic Cars	1:18	Min Lin Diecast	Features include opening and closing doors. Color: White.	5545	91.92	143.62

In [17]:
```
%%sql
USE classicmodels;
WITH
    products_from_france AS
        (
        SELECT
            productCode
        FROM orderdetails
        NATURAL JOIN
            orders
        NATURAL JOIN
            customers
        WHERE
            country = 'France'
        GROUP BY
            productCode
        )
SELECT
   *
FROM products
WHERE
    productCode NOT IN
        (
        SELECT
            productCode
        FROM products_from_france
        )
```

```
 * mysql+pymysql://root:***@localhost
0 rows affected.
2 rows affected.
```

Out[17]:

productCode	productName	productLine	productScale	productVendor	productDescription	quantityInStock	buyPrice	MSRP
S18_3233	1985 Toyota Supra	Classic Cars	1:18	Highway 66 Mini Classics	This model features soft rubber tires, working steering, rubber mud guards, authentic Ford logos, detailed undercarriage, opening doors and hood, removable split rear gate, full size spare mounted in bed, detailed interior with opening glove box	7733	57.01	107.57
S18_4027	1970 Triumph Spitfire	Classic Cars	1:18	Min Lin Diecast	Features include opening and closing doors. Color: White.	5545	91.92	143.62