

COMS W4111: Introduction to Databases

Spring 2024, Sections 002/V02

Midterm

Introduction

This notebook contains the midterm. **Both Programming and Nonprogramming tracks should complete this.** To ensure everything runs as expected, work on this notebook in Jupyter.

- You may post **privately** on Edstem or attend OH for clarification
 - TAs will not be providing hints

Submission instructions:

- You will submit **PDF and ZIP files** for this assignment. Gradescope will have two separate assignments for these.
 - For the PDF:
 - The most reliable way to save as PDF is to go to your browser's menu bar and click `File -> Print`. Switch the orientation to landscape mode, and hit save.
 - **MAKE SURE ALL YOUR WORK (CODE AND SCREENSHOTS) IS VISIBLE ON THE PDF. YOU WILL NOT GET CREDIT IF ANYTHING IS CUT OFF.** Reach out for troubleshooting.
 - For the ZIP:
 - Zip a folder containing this notebook and any screenshots.
 - Further submission instructions may be posted on Edstem.
-

Setup

```
In [1]: %load_ext sql
        %sql mysql+pymysql://root:dbuserdbuser@localhost
```

```
In [2]: import pandas
        from sqlalchemy import create_engine
        engine = create_engine("mysql+pymysql://root:dbuserdbuser@localhost")
        from IPython.display import Image
```

Written

- You may use lecture notes, slides, and the textbook
- You may use external resources, but you must cite your sources
- As usual, keep things short

W1

Briefly explain structured data, semi-structured data, and unstructured data. Give an example of each type of data.

<https://www.geeksforgeeks.org/difference-between-structured-semi-structured-and-unstructured-data/>
(<https://www.geeksforgeeks.org/difference-between-structured-semi-structured-and-unstructured-data/>)

1. Structured data is highly organised that follow specific schemas is easy to carry out analysis. Example: relational database;
2. Semi-structured data does not have a rigid schema but has some level of data hierarchy. Example: JSON
3. Unstructured data does not have predefined data model at all with no structure. Example: Text

W2

Codd's 0th rule states:

For any system that is advertised as, or claimed to be, a relational database management system, that system must be able to manage databases entirely through its relational capabilities.

Briefly explain and give examples of how the rule applied to:

1. Metadata
2. Security

1. Metadata of the structure of the data is stored in database INFORMATION_SCHEMA that is queryable and can be manipulated. For example, TABLE is a table in INFORMATION_SCHEMA that contains details such as table name and table type
2. Similarly, there are tables that handle user privileges. For example, INFORMATION_SCHEMA has a USER_PRIVILEGES table that has information on what permission users has across the entire database system.

W3

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 that are not possible to update
2. One view that is possible to update

You do not need to execute the statements. We are focusing on your understanding.

```
create table student
(
    social_security_no char(9) not null primary key,
    last_name varchar(64) null,
```

Non-updateable:

1. CREATE VIEW nonupdateableView1 AS SELECT DISTINCT last_name, first_name FROM student
2. CREATE VIEW nonupdateableView2 AS SELECT AVG(total_credits), FROM student

Updateable:

1. CREATE VIEW updateableView AS SELECT social_security_no, last_name FROM student

W4

The Columbia University directory of courses uses 20241COMS4111W002 for this sections "key".

1. Is this key atomic? Explain.
 2. Explain why having non-atomic keys creates problems for indexes.
-
1. This key is not atomic because this key can be divided into year(2024), department(COMS), course number (4111), and section number (002).
 2. Non-atomic keys causes slower indexing because non-atomic keys require larger storage space and querying is not as flexible because you have to filter multiple attributes at once.

W5

Briefly explain the following concepts:

1. Natural join
2. Equi-join
3. Theta join
4. Left join

5. Right join
6. Outer join
7. Inner join

1. Natural join automatically finds a column with the same name in each of the table to be joined. Only rows with value of the matching column that appear in both tables are included in the joined table.
2. Equi-join is a conditional join where the join condition is values of a specified column in table1 is equal to values of a specified column in table2.
3. Theta join is a conditional join where the join condition is values of a specified column in table1 is equal, larger than, smaller than, or not equal to values of a specified column in table2.
4. Left join is similar to theta join but always includes all rows on the left table. If there are rows in the left table that do not match with the right table, the values of the right table columns of those rows will be NULL.
5. Right join is similar to theta join but always includes all rows on the right table. If there are rows in the right table that do not match with the left table, the values of the left table columns of those rows will be NULL.
6. Outer join is similar to theta join always includes all rows on both tables. If there are rows with no match, the values of the unmatched columns of those rows will be NULL.
7. Inner join is similar to theta join but only includes rows that meet the specified matching condition on both tables.

W6

The *Classic Models* database has several foreign key constraints. For instance, *orderdetails.orderNumber* references *orders.orderNumber*.

1. Briefly explain the concept of *cascading actions* relative to foreign keys.
2. How could cascading actions be helpful for the above foreign key relationship?

1. Cascading actions refers to when changes occur to a record in the parent table, children of the modified record will also be modified similarly.
2. Cascading actions maintains consistency across tables even with modification to table orders. For example, if *orderNumber* in *order* is updated or deleted, associated *orderNumber* in *orderdetails* will be updated or deleted automatically.

W7

Give two reasons for using an associative entity to implement a relationship instead of a foreign key.

1. Associative entity should be used if the relationship between two entities are many-to-many otherwise a record might have multiple foreign keys.
2. Associative entity can reduce ambiguity of the relationship between two entities by giving a definition to the relationship and allow additional attributes to be added to the relationship.

W8

Briefly explain how SQL is closed under its operations. Give a simple query that takes advantage of this.

SQL is closed because the result of any SQL operation on tables can be treated as another table which can be used as an input of other SQL operations;

```
SELECT sub.col1, sub.col2 FROM (SELECT table1.col1, table2.col2 FROM table1 INNER JOIN table2 ON table1.id=table2.id) sub
```

W9

Briefly explain the differences between:

1. Database stored procedures
2. Database functions
3. Database triggers

1. Database stored procedures are reusable sets of SQL statements that can be executed in a single unit
2. Database functions is similar to database stored procedures but the procedures are named that accepts a parameter and returns a single value or result set.
3. Database triggers are a set of database stored procedures that automatically executes in response to specific database events.

W10

List three benefits/use cases for defining views.

https://edtechbooks.org/learning_mysql/benefits_of_using_vi (https://edtechbooks.org/learning_mysql/benefits_of_using_vi)

1. Better security. Defining views can hide unselected information from users.
 2. Simplify queries against viewed data by encapsulating complex queries.
 3. Centralises logic so that you only need to change the logic encapsulated by the view if there is any change in underlying table structure. No changes are needed on the queries againsts viewed data
-

Relational Algebra

- Use the [Relax calculator](https://dbis-uibk.github.io/relax/calc/gist/4f7866c17624ca9dfa85ed2482078be8/relax-silberschatz-english.txt/0) (<https://dbis-uibk.github.io/relax/calc/gist/4f7866c17624ca9dfa85ed2482078be8/relax-silberschatz-english.txt/0>) for these questions.
- For each question, you need to show your algebra statement and a screenshot of your tree and output.
 - **For your screenshot, make sure the entire tree and output are shown.** You may need to zoom out.
- The suggestions on which relations to use are hints, not requirements.

R1

- Write a relational algebra statement that produces a relation showing **teachers that taught sections in buildings that didn't match their department's building**.
 - A section is identified by (course_id, sec_id, semester, year) .
- Your output should have the following columns (names should match exactly; there should be no prefixes):
 - instructor_name
 - instructor_dept
 - course_id

- sec_id
- semester
- year
- course_building
- dept_building
- You should use the teaches , section , instructor , and department relations.
- As an example, one row you should get is

instructor_name	instructor_dept	course_id	sec_id	semester	year	course_building	dept_building
'Srinivasan'	'Comp. Sci.'	'CS-101'	1	'Fall'	2009	'Packard'	'Taylor'

- Srinivasan taught CS-101, section 1 in Fall of 2009 in the Packard building. However, Srinivasan is in the CS department, whose building is Taylor.

Algebra statement:

```

'''
π instructor_name←instructor.name, instructor_dept←instructor.dept_name, course_id←teaches.course_id,
sec_id←teaches.sec_id, semester←teaches.semester, year←teaches.year, course_building←section.building,
dept_building←department.building (
    σ (section.building != department.building ∧ section.course_id = teaches.course_id ∧ section.sec_id =
teaches.sec_id ∧ section.semester = teaches.semester ∧ section.year = teaches.year) (
        section ⋈ (section.course_id = teaches.course_id) (
            department ⋈ (department.dept_name = instructor.dept_name) (
                teaches ⋈ (teaches.ID = instructor.ID) instructor
            )
        )
    )
)
'''

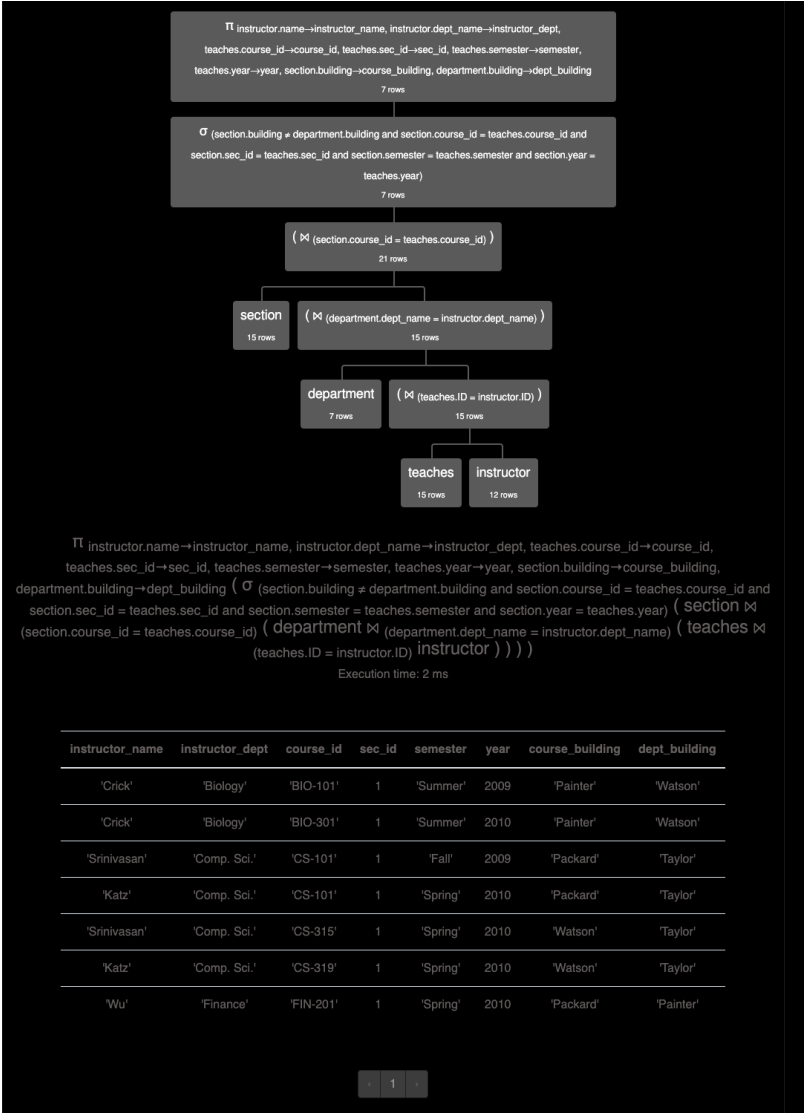
```

Execution:

| R1 Execution Result|


```
In [3]: Image("R1.png", width = 400)
```

Out [3]:



R2

- Some students don't have instructor advisors. Some instructors don't have student advisees.

- Write a relational algebra statement that produces a relation showing **all valid pairing between unadvised students and instructors with no advisees**.
 - A pairing is valid only if the student's department and instructor's department match.
- Your output should have the following columns (names should match exactly; there should be no prefixes):
 - instructor_name
 - student_name
 - dept_name
- You should use the `advisor`, `student`, and `instructor` relations.
- **You may only use the following operators:** π , σ , $=$, \neq , \wedge (and), \vee (or), ρ , \leftarrow , \bowtie , \Join , \ltimes , \Join
 - You may not need to use all of them.
 - Notably, you may **not** use anti-join or set difference.
- As an example, one row you should get is

instructor_name	student_name	dept_name
'El Said'	'Brandt'	'History'

- El Said has no advisees, and Brandt has no advisor. They are both in the history department.
- The same instructor may show up multiple times, but the student should be different each time. Similarly, the same student may show up multiple times, but the instructor should be different each time.

Algebra statement:

```

--
 $\pi$  instructor_name $\leftarrow$ instructor.name, student_name $\leftarrow$ student.name, student.dept_name (
   $\pi$  student.name, student.dept_name (
     $\sigma$  advisor.s_id = NULL (
      student  $\Join$  (student.ID = advisor.s_id)
      advisor
    )
  )  $\Join$  (student.dept_name=instructor.dept_name)
 $\pi$  instructor.name, instructor.dept_name (
   $\sigma$  advisor.i_id = NULL (
    instructor  $\Join$  (instructor.ID = advisor.i_id)
    advisor
  )
)
)

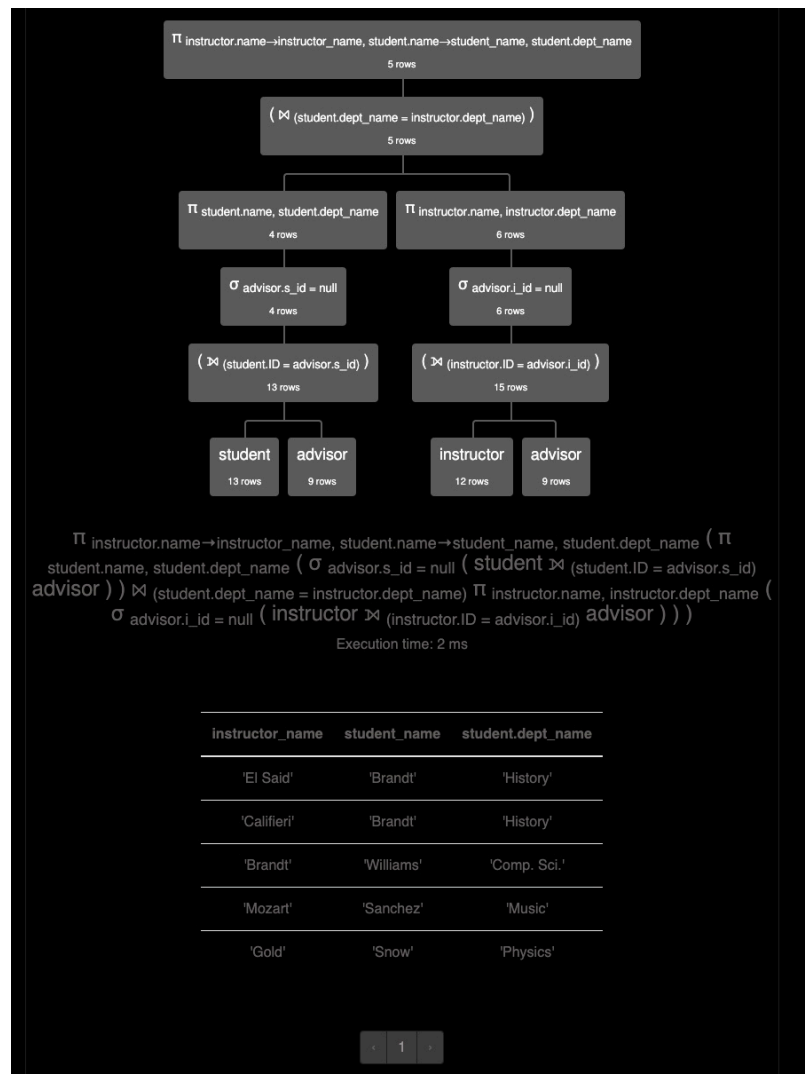
```

Execution:

| R2 Execution Result|

In [4]: Image("R2.jpeg", width = 400)

Out [4]:



R3

- Consider `new_section`, defined as:

`new_section = π course_id, sec_id, building, room_number, time_slot_id (section)`

- `new_section` contains sections, their time assignments, and room assignments independent of year and semester.
 - For this question, you can assume all the sections listed in `new_section` occur in the same year and semester.
 - You should copy the given definition of `new_section` to the top of your Relax calculator and treat it as a new relation.
- Write a relational algebra statement that produces a relation showing **conflicting sections**.
 - Two sections conflict if they have the same `(building, room_number, time_slot_id)`.
- Your output should have the following columns (names should match exactly; there should be no prefixes):
 - `first_course_id`
 - `first_sec_id`
 - `second_course_id`
 - `second_sec_id`
 - `building`
 - `room_number`
 - `time_slot_id`
- You should use the `new_section` relation.
- Your output cannot include courses and sections that conflict with themselves, or have two rows that show the same conflict.
- Good news: I'm going to give you the correct output!

<code>first_course_id</code>	<code>first_sec_id</code>	<code>second_course_id</code>	<code>second_sec_id</code>	<code>building</code>	<code>room_number</code>	<code>time_slot_id</code>
'CS-190'	2	'CS-347'	1	'Taylor'	3128	'A'
'CS-319'	2	'EE-181'	1	'Taylor'	3128	'C'

- Bad news: Your output must match mine **exactly**. The order of `first_course_id` and `second_course_id` cannot be switched.
 - Hint: You can do string comparisons in Relax using the inequality operators.

Algebra statement:

...

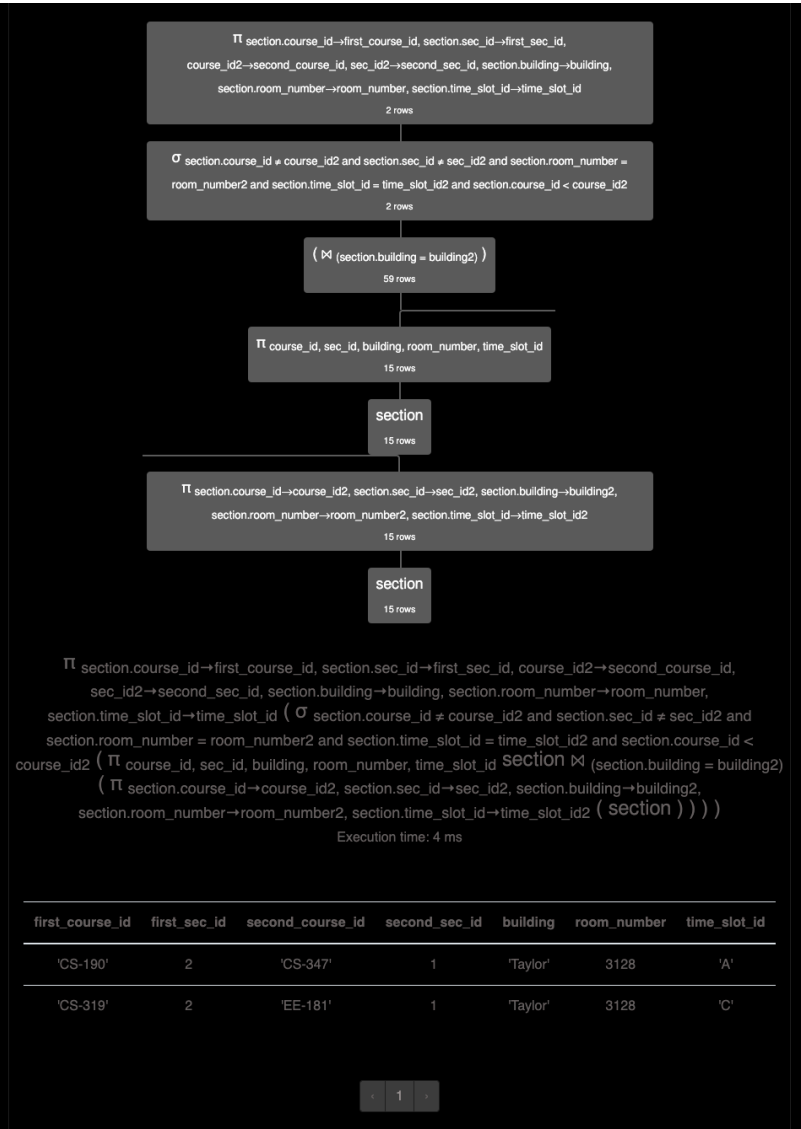
```
π first_course_id←section.course_id, first_sec_id←section.sec_id, second_course_id←course_id2,
second_sec_id←sec_id2, building←section.building, room_number←section.room_number,
time_slot_id←section.time_slot_id (
    σ section.course_id != course_id2 ∧ section.sec_id != sec_id2 ∧ section.room_number = room_number2 ∧
section.time_slot_id = time_slot_id2 ∧ section.course_id < course_id2 (
    π course_id, sec_id, building, room_number, time_slot_id
    section ⋈ (section.building = building2)
    (π course_id2←section.course_id, sec_id2←section.sec_id, building2←section.building,
room_number2←section.room_number, time_slot_id2←section.time_slot_id (section))
    )
)
...
```

Execution:

| R3 Execution Result|

In [5]: Image("R3.png", width = 400)

Out [5]:



ER Modeling

Definition to Model

- You're in charge of creating a model for a new music app, Dotify.
- The model has the following entities:
 1. `Artist` has the properties:
 - `artist_id` (primary key)
 - `name`
 - `description`
 - `date_joined`
 2. `Album` has the properties:
 - `album_id` (primary key)
 - `name`
 - `release_date`
 3. `Song` has the properties:
 - `song_id` (primary key)
 - `title`
 - `song_length`
 - `number_of_plays`
 4. `User` has the properties:
 - `user_id` (primary key)
 - `name`
 - `bio`
 - `date_joined`
 5. `Review` has the properties:
 - `review_id` (primary key)
 - `number_of_stars`
 - `review_text`
 6. `Playlist` has the properties:
 - `playlist_id` (primary key)
 - `name`

- description
- The model has the following relationships:
 1. Artist–Album : An artist can have any number of albums. An album belongs to one artist.
 2. Album–Song : An album can have at least one song. A song is on exactly one album.
 3. Artist–Song : An artist can have any number of songs. A song has at least one artist.
 4. Album–Review : An album can have any number of reviews. A review is associated with exactly one album.
 5. User–Review : A user can write any number of reviews. A review is associated with exactly one user.
 6. User–Playlist : A user can have any number of playlists. A playlist belongs to exactly one user.
 7. Song–Playlist : A song can be on any number of playlists. A playlist contains at least one song.
- Other requirements:
 1. You may **only** use the [four Crow's Foot](https://www.inf.usi.ch/faculty/soule/teaching/2014-spring/2014-03-07/images/crows-feet-notation.png) (<https://www.inf.usi.ch/faculty/soule/teaching/2014-spring/2014-03-07/images/crows-feet-notation.png>) notations shown in class.
 2. A user can leave at most one review per album (you don't need to represent this in your diagram). However, reviews can change over time. Your model must support the ability to keep track of a user's current and previous reviews for an album as well as the dates for the reviews.
 3. Playlists can change over time. Your model must support the ability to keep track of current songs in a playlist as well as which songs were on a playlist for what date ranges.
 - You don't need to keep track of a history of when a song was on a playlist (e.g., added Jan 1, then removed Jan 2, then re-added Jan 3, then re-removed Jan 4). You can just track the most recent date range (e.g., added Jan 3, then removed Jan 4).
 4. You may not directly link many-to-many relationships. You must use an associative entity.
 5. You may (and should) add attributes to the entities and create new entities to fulfill the requirements. **Do not forget about foreign keys.**
 6. You may add notes to explain any reasonable assumptions you make, either on the Lucidchart or below.
 - It would be beneficial, for instance, to document how you implemented requirements 2 and 3.

Assumptions and Documentation

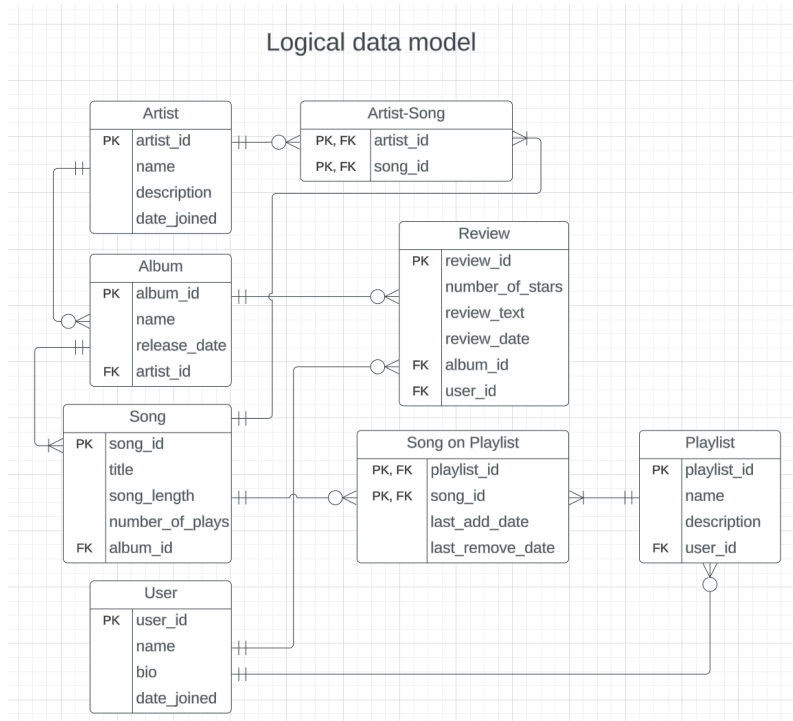
- For requirement 2, I added `review_date` to Review entity. I assume a user's review of a certain album with the newest date is the user's current review and all others are previous reviews of that certain album. To get a user's latest review on an album, you just need to filter review by user-id and album_id and choose the record with the latest date.
- For requirement 3, I added an associative entity `Song on Playlist` that associates song and playlist with two additional attributes `last_add_data` and `last_remove_data` to keep track of the most recent history of when a song was on a playlist.
- Added an associative entity `Artist–Song` that associates artist and song

Diagram:

| Definition to Model ER Diagram|

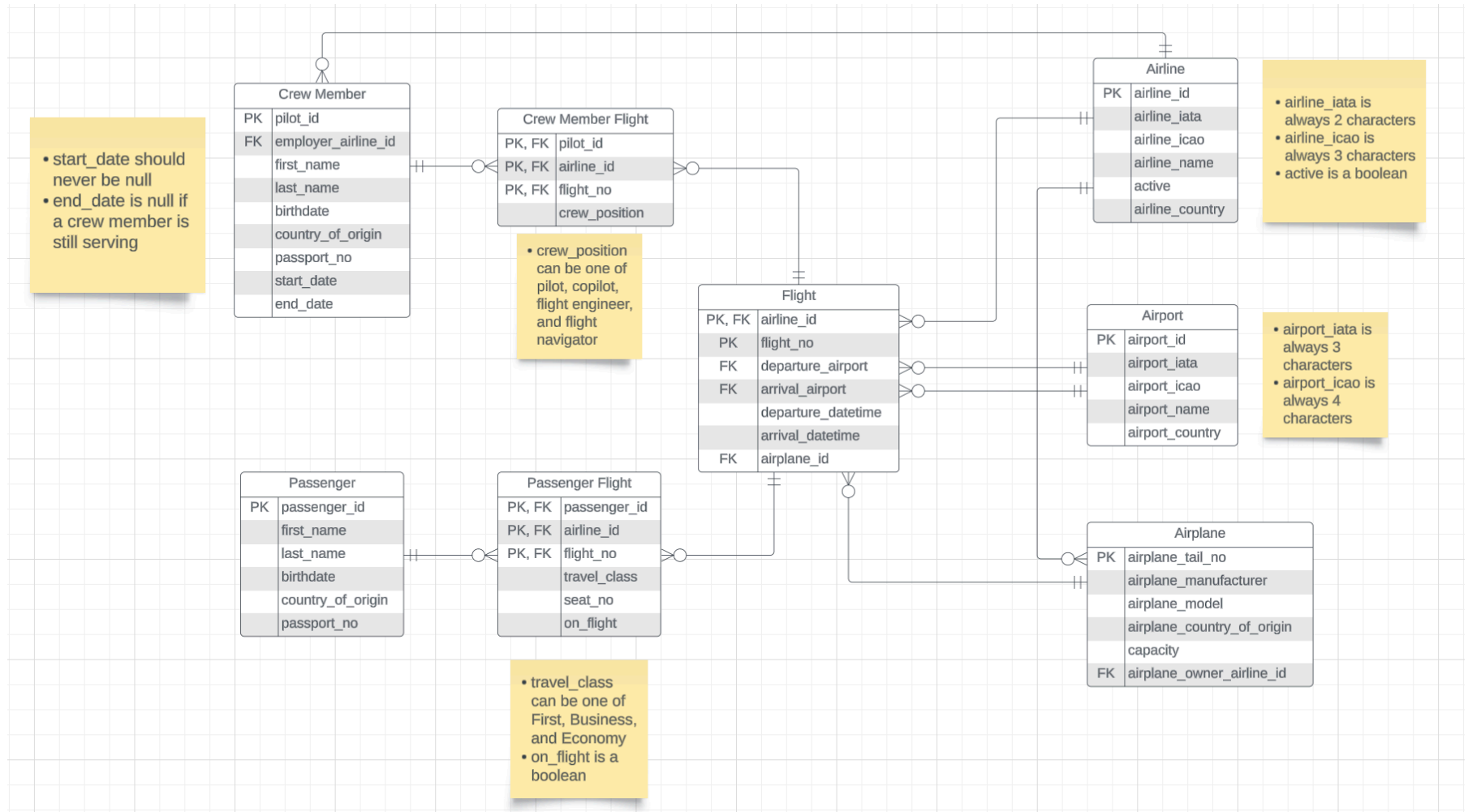
In [6]: `Image("ER.png", width = 400)`

Out [6]:



Model to DDL

- This question tests your ability to convert an ER diagram to DDL.
- Given the ER diagram below (**not your Dotify diagram**), write `create table` statements to implement the model.
 - You should choose appropriate data types, nullness, etc.
 - **You are required to implement the assumptions shown in the diagram.** You can document your other assumptions.
 - The required assumptions can be implemented through correct choices of data types and nullability. You aren't required to write checks or triggers for them.
 - You don't need to execute your statements. You also don't need to worry about details like creating/using a database.



Answer:

Assumptions and Documentation

- all ids cannot have more than 8 digits
- all names of any kind cannot have more than 20 characters
- all "no." attributes are unsigned bigint except for `seat_no`, which uses unsigned smallint
- `capacity` attribute uses unsigned smallint

...

```
create table "Crew Member"(  
    pilot_id varchar(8) not null,  
    employer_airline_id varchar(8) null,  
    first_name varchar(20) null,  
    last_name varchar(20) null,  
    birthdate date null,  
    country_of_origin varchar(20) null,  
    passport_no bigint unsigned null,  
    start_date date not null,  
    end_date date null,  
    primary key (pilot_id),  
    foreign key (employer_airline_id) references Airline(airline_id)  
);  
create table "Crew Member Flight"(  
    pilot_id varchar(8) not null,  
    airline_id varchar(8) not null,  
    flight_no bigint unsigned not null,  
    crew_position enum("pilot", "copilot", "flight engineer", "flight navigator") null,  
    primary key (pilot_id, airline_id, flight_no),  
    foreign key (pilot_id) references "Crew Member"(pilot_id),  
    foreign key (airline_id, flight_no) references Flight(airline_id, flight_no)  
);  
create table "Flight"(  
    airline_id varchar(8) not null,  
    flight_no bigint unsigned not null,  
    departure_airport varchar(20) null,  
    arrival_airport varchar(20) null,  
    departure_datetime datetime null,  
    arrival_datetime datetime null,  
    airplane_id varchar(8) null,  
    primary key (airline_id, flight_no),  
    foreign key (airline_id) references Airline(airline_id),  
    foreign key (departure_airport) references Airport(airport_id),  
    foreign key (arrival_airport) references Airport(airport_id),  
    foreign key (airplane_id) references Airplane(airplane_tail_no)  
);  
create table "Passenger Flight"(  
    passenger_id varchar(8) not null,  
    airline_id varchar(8) not null,  
    flight_no bigint unsigned not null,  
    travel_class enum("First", "Business", "Economy") null,
```

```

        seat_no smallint unsigned null,
        on_flight bool null,
        primary key (passenger_id, airline_id, flight_no),
        foreign key (passenger_id) references Passenger(passenger_id)
        foreign key (airline_id, flight_no) references Flight(airline_id, flight_no)
    );
create table "Passenger"(
    passenger_id varchar(8) not null,
    first_name varchar(20) null,
    last_name varchar(20) null,
    birthdate date null,
    country_of_origin varchar(20) null,
    passport_no bigint unsigned null,
    primary key (passenger_id)
);
create table "Airline"(
    airline_id varchar(8) not null,
    airline_iata char(2) null,
    airline_icao char(3) null,
    airline_name varchar(20) null,
    active bool null,
    airline_country varchar(20) null,
    primary key (airline_id)
);
create table "Airport" (
    airport_id varchar(8) not null,
    airport_iata char(3) null,
    airport_icao char(4) null,
    airport_name varchar(20) null,
    airport_country varchar(20) null,
    primary key (airport_id)
);
create table "Airplane" (
    airplane_tail_no bigint unsigned not null,
    airplane_manufacturer varchar(20) null,
    airplane_model varchar(20) null,
    airplane_country_of_origin varchar(20) null,
    capacity smallint unsigned null,
    airplane_owner_airline_id varchar(8) null,
    primary key (airplane_tail_no),
    foreign key (airplane_owner_airline_id) references Airline(airline_id)
)

```



Data and Schema Cleanup

Setup

- There are several issues with the `classicmodels` schema. Two issues are:
 - Having programs or users enter country names for `customers.country` is prone to error.
 - `products.productCode` is clearly not an atomic value.
- The following code does the following:
 1. Creates a schema for this question
 2. Creates copies of `classicmodels.customers` and `classicmodels.products`
 3. Loads a table of [ISO country codes \(https://en.wikipedia.org/wiki/List_of_ISO_3166_country_codes\)](https://en.wikipedia.org/wiki/List_of_ISO_3166_country_codes)

In [7]: %%sql

```
drop schema if exists classicmodels_midterm;
create schema classicmodels_midterm;
use classicmodels_midterm;

create table customers as select * from classicmodels.customers;
create table products as select * from classicmodels.products;

* mysql+pymysql://root:***@localhost
4 rows affected.
1 rows affected.
0 rows affected.
122 rows affected.
110 rows affected.
```

Out[7]: []

In [8]: iso_df = pandas.read_csv('./wikipedia-iso-country-codes.csv')
iso_df.to_sql('countries', schema='classicmodels_midterm',
 con=engine, index=False, if_exists="replace")

Out[8]: 246

In [9]: %%sql

```
alter table countries
  change `English short name lower case` short_name varchar(64) null;

alter table countries
  change `Alpha-2 code` alpha_2_code char(2) null;

alter table countries
  change `Alpha-3 code` alpha_3_code char(3) not null;

alter table countries
  change `Numeric code` numeric_code smallint unsigned null;

alter table countries
  change `ISO 3166-2` iso_text char(13) null;

alter table countries
  add primary key (alpha_3_code);

select * from countries limit 10;
```

* mysql+pymysql://root:***@localhost

246 rows affected.

246 rows affected.

246 rows affected.

246 rows affected.

246 rows affected.

0 rows affected.

10 rows affected.

Out [9]:

short_name	alpha_2_code	alpha_3_code	numeric_code	iso_text
Aruba	AW	ABW	533	ISO 3166-2:AW
Afghanistan	AF	AFG	4	ISO 3166-2:AF
Angola	AO	AGO	24	ISO 3166-2:AO
Anguilla	AI	AIA	660	ISO 3166-2:AI
Åland Islands	AX	ALA	248	ISO 3166-2:AX
Albania	AL	ALB	8	ISO 3166-2:AL
Andorra	AD	AND	20	ISO 3166-2:AD
Netherlands Antilles	AN	ANT	530	ISO 3166-2:AN
United Arab Emirates	AE	ARE	784	ISO 3166-2:AE
Argentina	AR	ARG	32	ISO 3166-2:AR

DE1

- There are four values in `customers.country` that do not appear in `countries.short_name`.
- Write a query that finds these four countries.
 - Hint: Norway should be one of these countries.


```
In [10]: %%sql
SELECT DISTINCT customers.country
FROM customers
WHERE customers.country NOT IN (SELECT countries.short_name FROM countries)

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

```
Out[10]: country
-----
USA
Norway
UK
Russia
```

DE2

- Norway actually does appear in `countries.short_name`. The reason it appeared in DE1 is because there are two spaces after the name (Norway__ instead of Norway).
- The mapping for the other countries is:

customers.country	countries.short_name
USA	United States
UK	United Kingdom
Russia	Russian Federation

- Write update table statements to correct the values in `customers.country` so that all the values in that attribute appear in `countries.short_name`.

```
In [11]: %%sql
UPDATE customers
SET country =
CASE
    WHEN country = 'Norway ' THEN 'Norway'
    WHEN country = 'USA' THEN 'United States'
    WHEN country = 'UK' THEN 'United Kingdom'
    WHEN country = 'Russia' THEN 'Russian Federation'
    ELSE country
END;
```

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

```
Out[11]: []
```

DE3

- The PK of `countries` is `alpha_3_code`. We want that as a FK in `customers`.

1. Create a column `customers.iso_code`
2. Set `customers.iso_code` as a FK that references `countries.alpha_3_code`
3. Fill `customers.iso_code` with the appropriate data based on `customers.country`
4. Drop `customers.country`
5. Create a view `customers_country` of form `(customerNumber, customerName, country, iso_code)`

Bonus point: I would ask you to create an index on `customers.iso_code`, but this is actually already done for us. When was an index created on `customers.iso_code`?

The index was automatically created at step 2 when I set `customers.iso_code` as a FK that references `countries.alpha_3_code`

```
In [12]: %%sql
ALTER TABLE customers
ADD COLUMN iso_code char(3)

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

Out[12]: []

```
In [13]: %%sql
ALTER TABLE customers
ADD FOREIGN KEY (iso_code) REFERENCES countries(alpha_3_code)

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

Out[13]: []

```
In [14]: %%sql
update customers
inner join countries on customers.country = countries.short_name
set customers.iso_code = countries.alpha_3_code

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

Out[14]: []

```
In [15]: %%sql
alter table customers
drop column country

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

Out[15]: []

```
In [16]: %%sql
create view customers_country as (
    select
        cu.customerNumber as customerNumber,
        cu.customerName as customerName,
        co.short_name as country,
        cu.iso_code as iso_code
    from
        customers cu
    join
        countries co on cu.iso_code = co.alpha_3_code
)

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

Out[16]: []

DE4

- To test your code, output a table that shows the number of customers from each country.
- You should use your `customers_country` view.
- Your table should have the following attributes:
 - `country_iso`
 - `number_of_customers`
- Order your table from greatest to least `number_of_customers`.
- Show only the first 10 rows.

```
In [17]: %%sql
select
    v.iso_code as country_iso,
    count(*) as number_of_customers
from
    customers_country v
group by
    v.iso_code
order by
    number_of_customers desc
limit 10

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

```
Out[17]:
```

country_iso	number_of_customers
USA	36
DEU	13
FRA	12
ESP	7
GBR	5
AUS	5
ITA	4
NZL	4
FIN	3
CAN	3

DE5

- `products.productCode` appears to be 3 separate values joined by an underscore.
 - I have no idea what the values mean, but let's pretend we do know for the sake of this question.
- Write `alter table` statements to create 3 new columns: `product_code_letter`, `product_code_scale`, and `product_code_number`.

- Choose appropriate data types. `product_code_letter` should always be a single letter.

```
In [18]: %%sql
alter table products
add column product_code_letter char(1) not null,
add column product_code_scale int unsigned not null,
add column product_code_number int unsigned not null

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

Out[18]: []

DE6

- As an example, for the product code `S18_3856`, the product code letter is `S`, the product code scale is `18`, and the product code number is `3856`.
 - I know the product code scale doesn't always match `products.productScale`. Let's ignore this for now.
- 1. Populate `product_code_letter`, `product_code_scale`, and `product_code_number` with the appropriate values based on `productCode`.
- 2. Set the PK of `products` to `(product_code_letter, product_code_scale, product_code_number)`.
- 3. Drop `productCode`.

```
In [19]: %%sql
update products
set
product_code_letter = substring(productCode,1,1),
product_code_scale =
cast(substring(productCode,2,position('_', ' ' in productCode)-2) as unsigned),
product_code_number =
cast(substring(productCode,position('_', ' ' in productCode)+1, length(productCode)) as unsigned);

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

Out[19]: []

```
In [20]: %%sql
alter table products
add primary key (product_code_letter, product_code_scale, product_code_number)

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

Out[20]: []

```
In [21]: %%sql
alter table products
drop column productCode

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

Out[21]: []

DE7

- To test your code, output a table that shows the products whose `product_code_scale` doesn't match `productScale` .
- Your table should have the following attributes:
 - `product_code_letter`
 - `product_code_scale`
 - `product_code_number`
 - `productScale`
 - `productName`
- Order your table on `productName` .

```
In [22]: %%sql
select
    product_code_letter,
    product_code_scale,
    product_code_number,
    productScale,
    productName
from
    products
where
    CAST(product_code_scale AS CHAR) != substring(productScale,3,length(productScale))
order by
    productName
```

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

```
Out [22]:
```

product_code_letter	product_code_scale	product_code_number	productScale	productName
S	24	3856	1:18	1956 Porsche 356A Coupe
S	24	4620	1:18	1961 Chevrolet Impala
S	12	3148	1:18	1969 Corvair Monza
S	700	2824	1:18	1982 Camaro Z28
S	700	3167	1:72	F/A 18 Hornet 1/72
S	18	2581	1:72	P-51-D Mustang

SQL

- Use the `classicmodels` database for these questions.
- The suggestions on which tables to use are hints, not requirements.
- All your answers should be a single select statement. **You may not create a new table.**
 - Subqueries (selects within a select) and the `with` keyword are fine. Just don't use the `create` keyword.

In [23]: %sql use classicmodels

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

Out[23]: []

SQL1

- Write a query that produces a table of form (productName, productLine, productVendor, totalRevenue) .
 - Attribute names should match exactly.
 - The totalRevenue for a product is the sum of quantityOrdered*priceEach across all the rows the product appears in in orderdetails .
 - You should consider all orders, regardless of orders.status .
- Only include products with totalRevenue greater than \$150,000.
- Order your output on totalRevenue descending.
- You should use the products and orderdetails tables.

```
In [24]: %%sql
select * from (
    select
        p.productName as productName,
        p.productLine as productLine,
        p.productVendor as productVendor,
        sum(o.quantityOrdered*o.priceEach) as totalRevenue
    from
        products p
    join
        orderdetails o on p.productCode = o.productCode
    group by
        o.productCode
) tmp
where
    totalRevenue > 150000
order by
    totalRevenue desc
```

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

Out [24]:

	productName	productLine	productVendor	totalRevenue
	1992 Ferrari 360 Spider red	Classic Cars	Unimax Art Galleries	276839.98
	2001 Ferrari Enzo	Classic Cars	Second Gear Diecast	190755.86
	1952 Alpine Renault 1300	Classic Cars	Classic Metal Creations	190017.96
	2003 Harley-Davidson Eagle Drag Bike	Motorcycles	Red Start Diecast	170686.00
	1968 Ford Mustang	Classic Cars	Autoart Studio Design	161531.48
	1969 Ford Falcon	Classic Cars	Second Gear Diecast	152543.02

SQL2

- Write a query that produces a table of form (productCode, productName, productVendor, customerCount) .
 - Attribute names should match exactly.
 - customerCount is the number of **distinct** customers that have bought the product.

- Note that the same customer may buy a product multiple times. This only counts as one customer in the product's `customerCount` .
- You should consider all orders, regardless of `status` .
- Order your table from largest to smallest `customerCount` , then on `productCode` alphabetically.
- Only show the first 10 rows.
- You should use the `orders` and `orderdetails` tables.

```
In [25]: %%sql
select
    od.productCode as productCode,
    p.productName as productName,
    p.productVendor as productVendor,
    count(distinct o.customerNumber) as customercount
from
    orderdetails od
join
    products p on od.productCode = p.productCode
join
    orders o on od.orderNumber = o.orderNumber
group by
    od.productCode
order by
    customerCount desc,
    od.productCode
limit 10
```

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

Out [25]:

productCode	productName	productVendor	customercount
S18_3232	1992 Ferrari 360 Spider red	Unimax Art Galleries	40
S10_1949	1952 Alpine Renault 1300	Classic Metal Creations	27
S10_4757	1972 Alfa Romeo GTA	Motor City Art Classics	27
S18_2957	1934 Ford V8 Coupe	Min Lin Diecast	27
S72_1253	Boeing X-32A JSF	Motor City Art Classics	27
S10_1678	1969 Harley Davidson Ultimate Chopper	Min Lin Diecast	26
S10_2016	1996 Moto Guzzi 1100i	Highway 66 Mini Classics	26
S18_1662	1980s Black Hawk Helicopter	Red Start Diecast	26
S18_1984	1995 Honda Civic	Min Lin Diecast	26
S18_2949	1913 Ford Model T Speedster	Carousel DieCast Legends	26

SQL3

- Write a query that produces a table of form (customerName, month, year, monthlyExpenditure, creditLimit) .
 - Attribute names should match exactly.
 - monthlyExpenditure is the total amount of payments made by a customer in a specific month and year based on the payments table.
 - Some customers have never made any payments ever. For these customers, monthlyExpenditure should be 0. month and year can be null.
- Only show rows where monthlyExpenditure exceeds creditLimit **or** the customer has never made any payments ever (so month and year should be null for these rows).
- Order your table on monthlyExpenditure descending, then on customerName alphabetically.
- Only show the first 10 rows.
- You should use the payments and customers tables.

```
In [26]: %%sql
select * from (
    select
        c.customerName as customerName,
        month(p.paymentDate) as month,
        year(p.paymentDate) as year,
        coalesce(sum(p.amount), 0) as monthlyExpenditure,
        c.creditlimit as creditLimit
    from
        customers c
    left join
        payments p on c.customerNumber = p.customerNumber
    group by
        c.customerNumber,
        month,
        year
) tmp
where
    monthlyExpenditure > creditLimit or monthlyExpenditure = 0
order by
    monthlyExpenditure desc,
    customerName
limit 10
```

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

Out [26]:

customerName	month	year	monthlyExpenditure	creditLimit
Dragon Souveniers, Ltd.	12	2003	105743.00	103800.00
American Souvenirs Inc	None	None	0.00	0.00
ANG Resellers	None	None	0.00	0.00
Anton Designs, Ltd.	None	None	0.00	0.00
Asian Shopping Network, Co	None	None	0.00	0.00
Asian Treasures, Inc.	None	None	0.00	0.00
BG&E Collectables	None	None	0.00	0.00
Cramer Spezialitäten, Ltd	None	None	0.00	0.00
Der Hund Imports	None	None	0.00	0.00
Feuer Online Stores, Inc	None	None	0.00	0.00

SQL4

- Write a query that produces a table of form (productCode, productName, productLine, productVendor, productDescription) .
 - Attribute names should match exactly.
- **You should only keep products that have never been ordered by a French customer.**
 - You should consider all orders, regardless of status .
- Order your table on productCode .
- You should use the customers , orders ,and orderdetails tables.

```
In [27]: %%sql
select
    p.productCode as productCode,
    p.productName as productName,
    p.productLine as productLine,
    p.productVendor as productVendor,
    p.productDescription as productDescription
from
    products p
where
    p.productCode not in (
        select
            od.ProductCode
        from
            orderdetails od
        join
            orders o on od.orderNumber = o.orderNumber
        join
            customers c on o.customerNumber = c.customerNumber
        where
            c.country = 'France'
    )
order by
    productCode
```

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

```
Out [27]:
```

productCode	productName	productLine	productVendor	productDescription
S18_3233	1985 Toyota Supra	Classic Cars	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
S18_4027	1970 Triumph Spitfire	Classic Cars	Min Lin Diecast	Features include opening and closing doors. Color: White.

SQL5

- A customer can have a sales rep employee.
- Corporate is deciding which employees to give raises to.

- A raise is given for the reason `customers` if an employee has 8 or more customers.
- A raise is given for the reason `orders` if the total number of orders made by customers associated with an employee is 30 or greater.
 - You should consider all orders, regardless of `status`.
- A raise is given for the reason `both` if both conditions above are true.
- Write a query that produces a table of form (`firstName`, `lastName`, `totalCustomers`, `totalCustomerOrders`, `raiseBecause`) .
 - Attribute names should match exactly.
 - `firstName` and `lastName` are for the employee.
 - `totalCustomers` is the total number of customers associated with an employee.
 - `totalCustomerOrders` is the total number of orders made by customers associated with an employee.
 - `raiseBecause` is one of `customers`, `orders`, and `both`.
- Your table should only show employees eligible for raises, i.e., `raiseBecause` should not be null.
- Order your table on `firstName`.
- You should use the `customers`, `orders`, and `employees` tables.

```

In [28]: %%sql
select * from (
    select
        e.firstName as firstName,
        e.lastName as lastName,
        count(distinct c.customerNumber) as totalCustomers,
        count(o.orderNumber) as totalCustomerOrders,
        case
            when count(distinct c.customerNumber) >= 8 and count(o.orderNumber) >= 30 then 'both'
            when count(distinct c.customerNumber) >= 8 then 'customers'
            when count(o.orderNumber) >= 30 then 'orders'
        end as raiseBecause
    from
        employees e
    join
        customers c on e.employeeNumber = c.salesRepEmployeeNumber
    join
        orders o on c.customerNumber = o.customerNumber
    group by
        e.employeeNumber) tmp
where
    raiseBecause is not null
order by
    firstname

```

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

Out [28]:

firstName	lastName	totalCustomers	totalCustomerOrders	raiseBecause
Barry	Jones	9	25	customers
George	Vanauf	8	22	customers
Gerard	Hernandez	7	43	orders
Larry	Bott	8	22	customers
Leslie	Jennings	6	34	orders
Pamela	Castillo	10	31	both

