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

```
The sql extension is already loaded. To reload it, use:
  %reload_ext sql
```

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

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.

- **Structured Data**: Structured data is laid out in a way that is efficiently addressable/accessible. It is based on a relational database table.
  - Example: spreadsheet
- **semi-structured data**: semi-structured data is data which is not fully structured but still has elements (hierarchy, keys, or something like that) that make it easier to access and analyze than unstructured data.
  - Example: JSON, XML
- **unstructured data**: unstructured data has no predefined organization. It does not fit into the relational database model easily.
  - Example: a bunch of PDFs

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

metadata

Codd's 0th rule means that information about tables, columns, relationships, etc should be stored/queried using relational principles.

For example, in a relational database, metadata about tables and their attributes (columns) can be stored in system tables. These system tables contain information about all the tables in the database, such as the names of the tables themselves, column names, data types, etc.

security

Codd's 0th rule means that RDBMSs must have built-in support for authorizing users and protecting the data.

For example, SQL provides commands to manage security in a relational database by defining access control and permissions. There are types of authorizations called *priveleges* (read, update, insert, delete) which define how users can make changes to a table's rows, and further, authorizations which define how users can interact with the schema as a whole. SQL helps database adminstrators manage this with the `grant` and `revoke` keyword.

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,
    first_name varchar(64) null,
    enrollment_year year null,
    total_credits int null
);
```

```
# View 1: Not Possible to Update (Due to Aggregation)
CREATE VIEW view_not_updateable_1 AS
SELECT
    enrollment_year,
    COUNT(*) AS student_count
FROM student
GROUP BY enrollment_year;

# View 2: Not Possible to Update (Due to Join and Aggregation)
CREATE VIEW view_not_updateable_2 AS
SELECT
    s.social_security_no,
    s.last_name,
```

```
    s.first_name,
    s.enrollment_year,
    s.total_credits,
    c.course_name
FROM student s
JOIN course c ON s.social_security_no = c.student_social_security_no
WHERE s.enrollment_year = 2022;

# View 3: Possible to Update
CREATE VIEW view_updateable AS
SELECT
    social_security_no,
    last_name,
    first_name,
    enrollment_year,
    total_credits
FROM student
WHERE enrollment_year = 2023;
```

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 within the key, there are fields like year, course_department, course_number, and section.
2. Indexing on non-atomic keys creates more overhead than with atomic keys because you might need to look at multiple fields to do things like compare or update entries.

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
    - It is a type of theta join that automatically identifies common attributes based on their names without requiring explicit specification.
2. Equi-join
    - An equi-join is a specific type of theta join where the condition for combining rows from two relations is based on the equality of values in specified attributes. It is a more general term that includes natural joins but allows for explicit definition of the equality condition.
3. Theta join
    - A theta join is a join operation where rows from two relations are combined based on a specified condition (called a theta condition), which can include comparisons using operators such as "=", "<", or ">", allowing for a broader range of join conditions than equi-joins.
4. Left join
    - retrieves all rows from the left (or first) relation and the matching rows from the right (or second) relation. If no match is found in the right relation, the result contains null values for columns from the right relation.
5. Right join
    - Like a left join, but with left and right swapped.
6. Outer join
    - Includes all tuples from at least one of the relations involved (both if it's a full outer join). This means it includes rows with null values in the matching attribute column.
7. Inner join
    - a type of join that does not preserve nonmatched tuples.

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. A cascading action on a foriegn key means that when the record in the parent table is affected, so is the corresponding record in the child table.
2. Cascading actions with foreign keys would be helpful in the Classic Models because it would allow for sychronization of changes. For instance, deleting an `order` record would necessarily mean the corresponding `orderdetails` record would also be deleted.

W7

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

1. An associative entity allows for the relationship itself to have attributes (rather than the relationsip just being its endpoints)
2. Associative entitites are a simpler way of representing many-to-many relationships.

W8

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

SQL is closed under its operations, which means that operations on tables produce tables. A simple example of this is with `select` followed by a `join`. Since SQL is closed, we can chain operations together, i.e.:

```
SELECT Employees.EmployeeID, Employees.EmployeeName, Departments.DepartmentName
FROM Employees
```

```
JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

W9

Briefly explain the differences between:

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

1. Stored procedures

   The most general of the three, stored procedures are precompiled sets of one or more SQL statements that we invoke to save runtime. They can make changes to data, and can return zero or more values.

2. Functions

   Are similar to stored procedures, but they can only return one value. They also cannot change data (they can't have side-effects).

3. Triggers

   Statements that are run automatically on an event (before or after INSERT, UPDATE, or DELETE). They never have a return value and can only have side-effects.

W10

List three benefits/use cases for defining views.

1. There might be multiple applications using the same database each in their own way. By using a view DB admins can give each application their own version of the database without actually changing the underlying representation of the data.
2. Views can simplify complex queries by providing a shorthand for a lengthy expression

3. We can use views to expose only specific rows/columns to certain users, which has a security benefit of protecting sensitive data.

Relational Algebra

- Use the Relax calculator 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

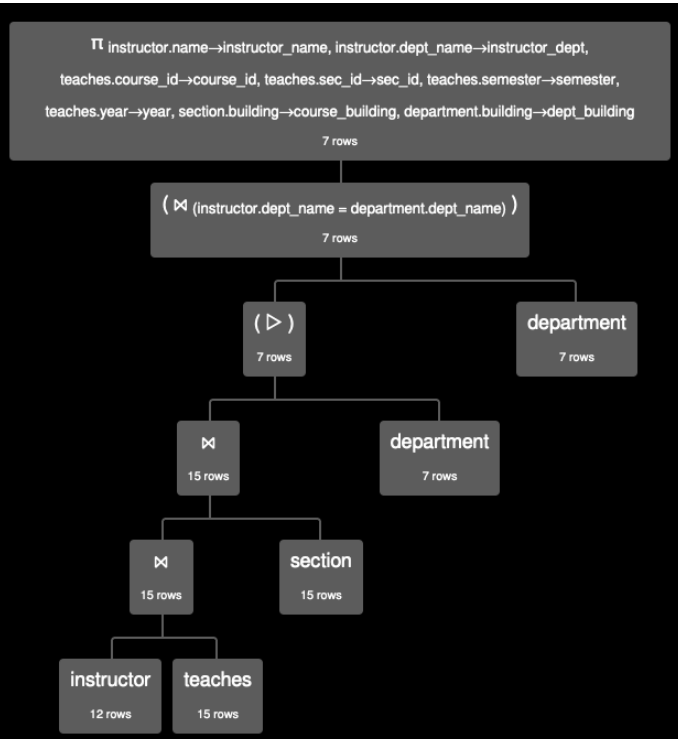
instructor_name	instructor_dept	course_id	sec_id	semester	year	course_building	dept_building
'Srinivasan'	'Comp. Sci.'	'CS-101'	1	'Fall'	2009	'Packard'	'Taylor'

- 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:

```
pi 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 (
    (instructor join teaches join section anti join department) join
(instructor.dept_name = department.dept_name) department
)
```

Execution:



π instructor.name→instructor_name, instructor.dept_name→instructor_dept, teaches.course_id→course_id, teaches.sec_id→sec_id, teaches.semester→semester, teaches.year→year, section.building→course_building, department.building→dept_building ((((instructor ⋈ teaches) ⋈ section) ▷ department) ⋈ (instructor.dept_name = department.dept_name) department)

Execution time: 0 ms

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'	'Comp. Sci.'	'CS-315'	1	'Spring'	2010	'Watson'	'Taylor'
'Wu'	'Finance'	'FIN-201'	1	'Spring'	2010	'Packard'	'Painter'
'Katz'	'Comp. Sci.'	'CS-101'	1	'Spring'	2010	'Packard'	'Taylor'
'Katz'	'Comp. Sci.'	'CS-319'	1	'Spring'	2010	'Watson'	'Taylor'
'Crick'	'Biology'	'BIO-101'	1	'Summer'	2009	'Painter'	'Watson'
'Crick'	'Biology'	'BIO-301'	1	'Summer'	2010	'Painter'	'Watson'

R1 Execution Result

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, =, ≠, ∧ (and), ∨ (or), ρ, ←, ⋈, ⋉, ⋊, ⋋
  - 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'

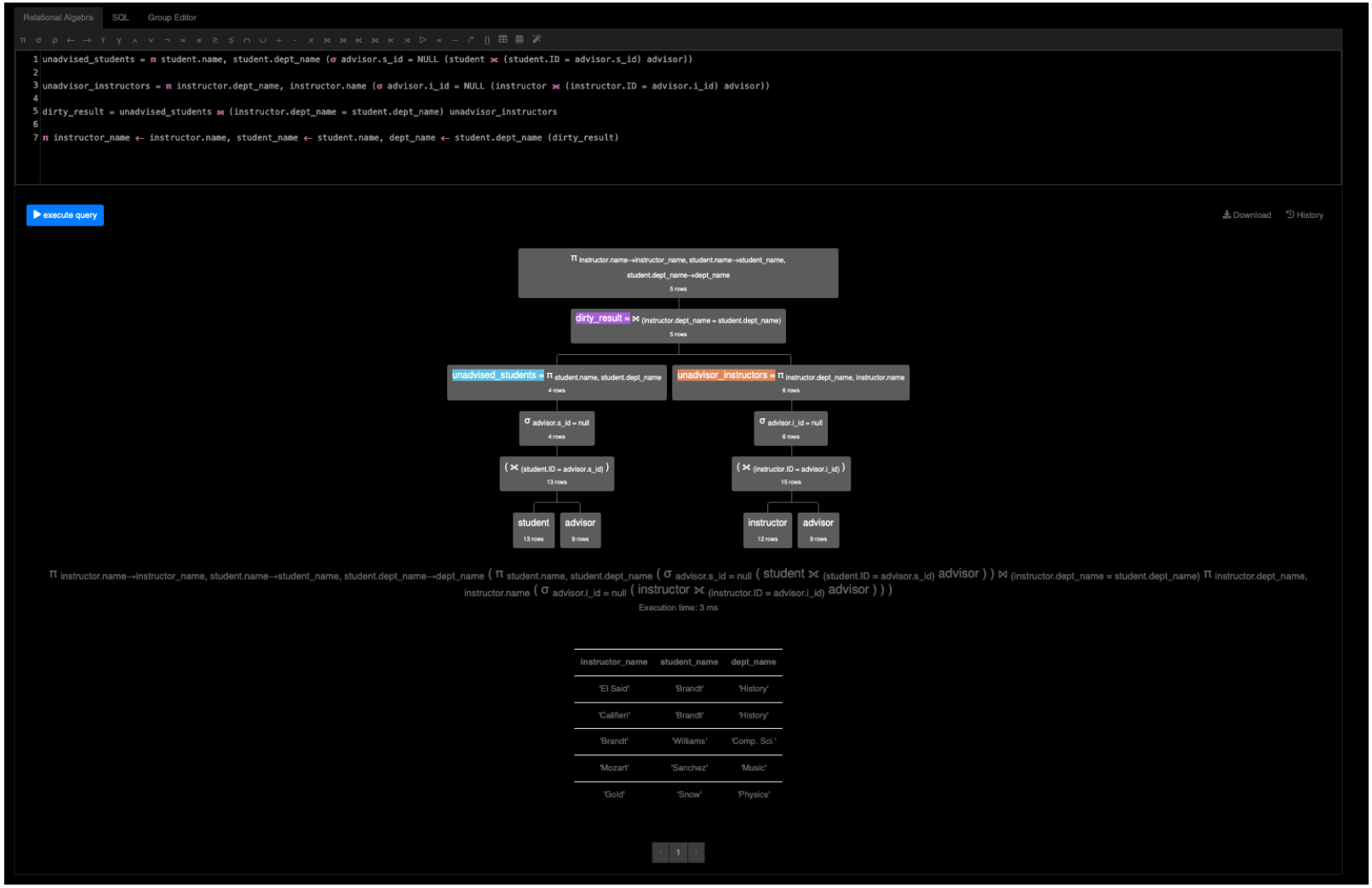
- El Said has no advisees, and Brandt has no advisor. They are both in the history department.
- The same instructor may show up multiple times, but the student should be different each time. Similarly, the same student may show up multiple times, but the instructor should be different each time.

Algebra statement:

```
unadvised_students = π student.name, student.dept_name (σ advisor.s_id = NULL (student ⟕
(student.ID = advisor.s_id) advisor))

unadvisor_instructors = π instructor.dept_name, instructor.name (σ advisor.i_id = NULL
(instructor ⟕ (instructor.ID = advisor.i_id) advisor))

dirty_result = unadvised_students ⋈ (instructor.dept_name = student.dept_name)
unadvisor_instructors

π instructor_name ← instructor.name, student_name ← student.name, dept_name ←
student.dept_name (dirty_result)
```

In [ ]:

Execution:



R2 Execution Result

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!

first_course_id	first_sec_id	second_course_id	second_sec_id	building	room_number	time_slot_id
'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:

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


dirty_out = σ section.course_id ≠ second_course_id and sec_id < second_sec_id
(new_section ⋈ (b=building and r=room_number and t=time_slot_id) (π
second_course_id←course_id, second_sec_id←sec_id, b←building, r←room_number,
t←time_slot_id (new_section)))

π first_course_id ← course_id,
first_sec_id←sec_id,second_course_id,second_sec_id,building
←building,room_number←room_number,time_slot_id←time_slot_id (dirty_out)
```

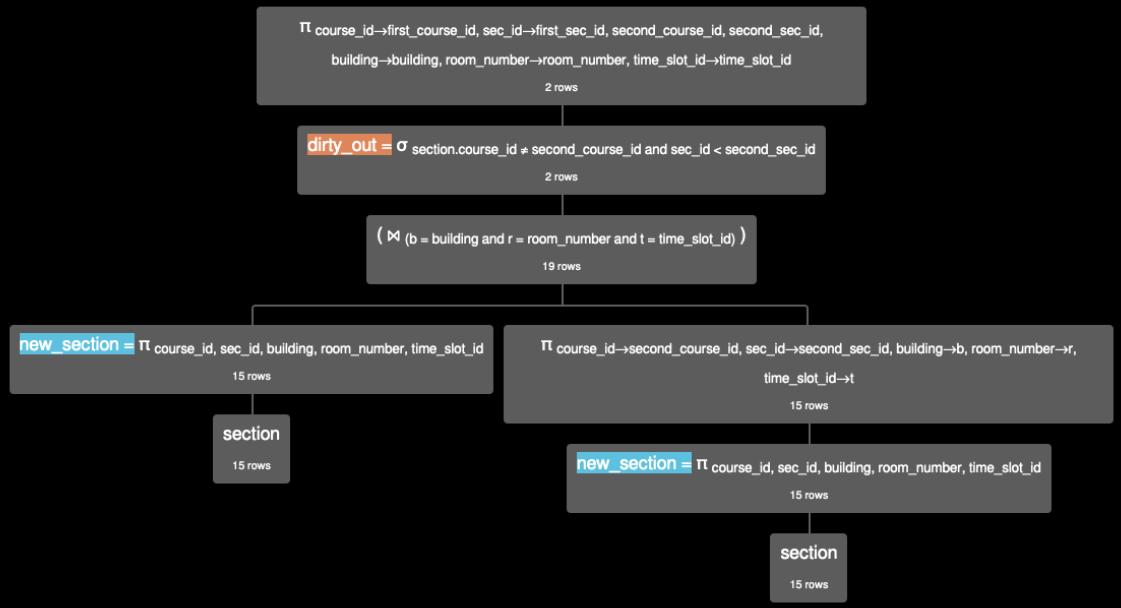
Execution:

Relational Algebra | SQL | Group Editor

```
1 new_section = π  course_id, sec_id, building, room_number, time_slot_id (section)
2
3
4 dirty_out = σ section.course_id ≠ second_course_id and sec_id < second_sec_id (new_section ⋈ (b=building and r=room_number and t=time_slot_id) (π
  second_course_id←course_id, second_sec_id←sec_id, b←building, r←room_number, t←time_slot_id (new_section)))
5
6 π first_course_id ← course_id, first_sec_id←sec_id,second_course_id,second_sec_id,building
  ←building,room_number←room_number,time_slot_id←time_slot_id (dirty_out)
```

execute query

Download | History



π course_id→first_course_id, sec_id→first_sec_id, second_course_id, second_sec_id, building→building, room_number→room_number, time_slot_id→time_slot_id σ section.course_id ≠ second_course_id and sec_id < second_sec_id (π course_id, sec_id, building, room_number, time_slot_id (section) ⋈ (b = building and r = room_number and t = time_slot_id) (π course_id→second_course_id, sec_id→second_sec_id, building→b, room_number→r, time_slot_id→t π course_id, sec_id, building, room_number, time_slot_id (section)))

Execution time: 4 ms

first_course_id	first_sec_id	second_course_id	second_sec_id	building	room_number	time_slot_id
'CS-347'	1	'CS-190'	2	'Taylor'	3128	'A'

R3 Execution Result

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 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**

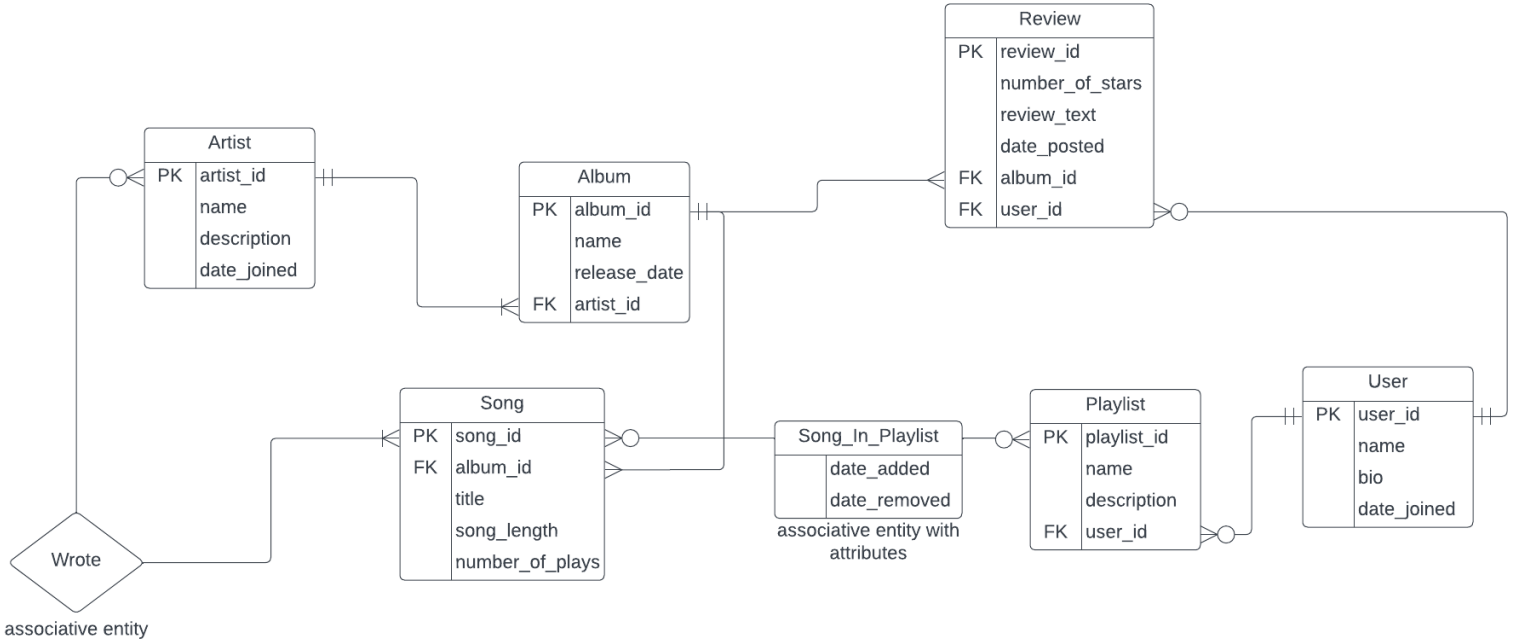
about foreign keys.

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

- Requirement 2
    - I assumed that the date posted is sufficient enough to tell us if the review is current (i.e., a users' most recent review is their one review for the album)
    - Users can make many reviews, even for a single album. They are dated so we know which is current.
- Requirement 3
    - I used an associative entity to track the many-to-many relationship between songs in playlists, and this associative entity has dates for being added and removed.
    - If the date removed is null, the song is still on the playlist.

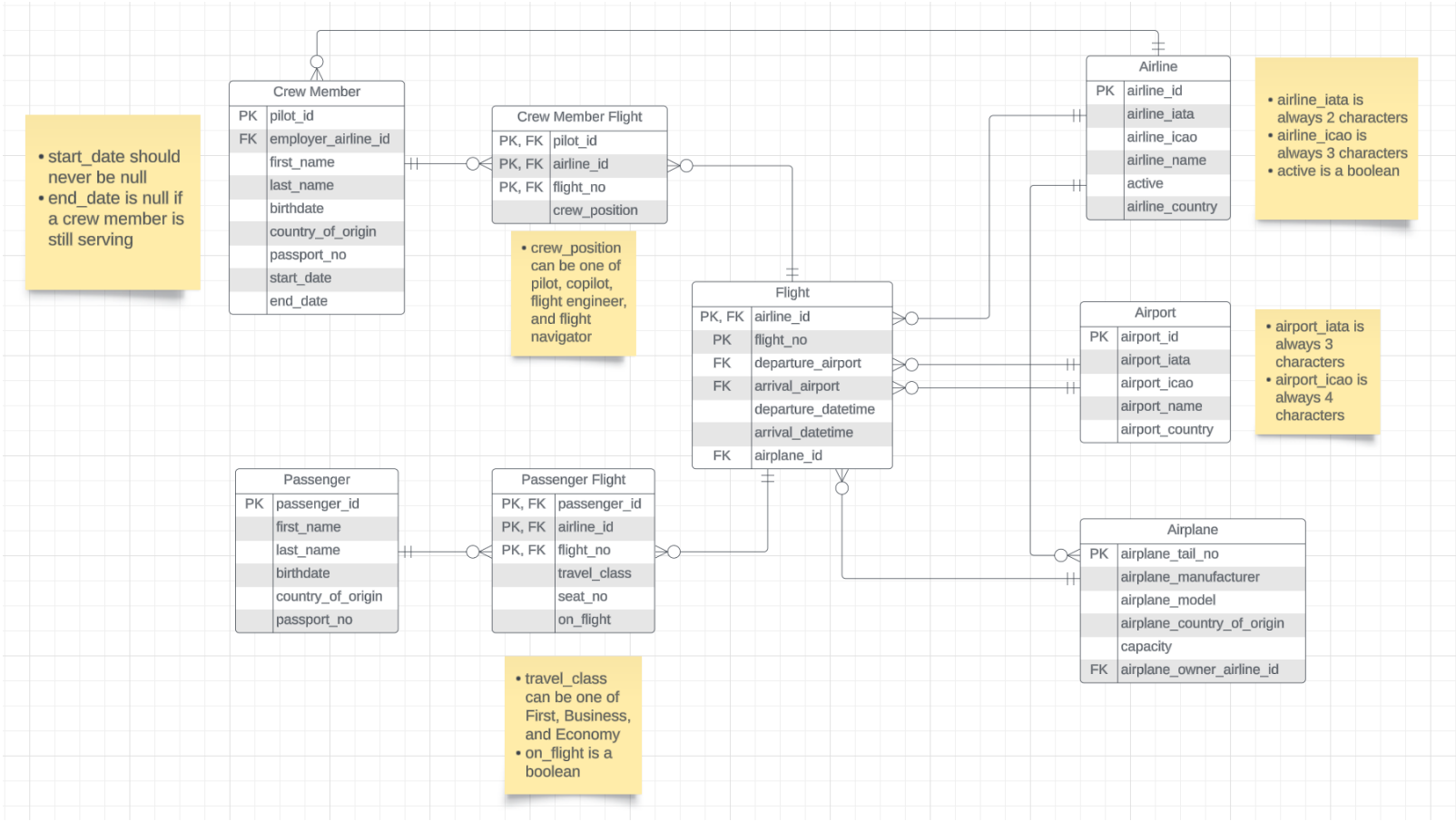
Diagram:



Definition to Model ER Diagram

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.



Model to DDL ER Diagram

Answer:

Assumptions and Documentation

- flight_No, seat_no, and passport_no are not strictly numeric
- technically country of origin should be enum, but there are so many possibilities and I don't know them all. For this example I made it a varchar.

```
CREATE TABLE Airport (
```

```
    airport_id INT PRIMARY KEY,
    airport_iata CHAR(3),
    airport_icao CHAR(4),
    airport_name VARCHAR(255),
    airport_country VARCHAR(100)
);
CREATE TABLE Airline (
    airline_id INT PRIMARY KEY,
    airline_iata CHAR(2),
    airline_icao CHAR(3),
    airline_name VARCHAR(255),
    active BOOLEAN,
    airline_country VARCHAR(100)
);

CREATE TABLE Passenger (
    passenger_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    birthdate DATE,
    country_of_origin VARCHAR(100),
    passport_no VARCHAR(20) UNIQUE
);

CREATE TABLE Airplane (
    airplane_tail_no INT PRIMARY KEY,
    airplane_manufacturer VARCHAR(50),
    airplane_model VARCHAR(50),
    airplane_country_of_origin VARCHAR(100),
    capacity INT,
    airplane_owner_airline_id INT,
    FOREIGN KEY (airplane_owner_airline_id) REFERENCES Airline(airline_id)
);

CREATE TABLE `CrewMember` (
    pilot_id INT PRIMARY KEY,
    employer_airline_id INT,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
```

```
    birthdate DATE,
    country_of_origin VARCHAR(100),
    passport_no VARCHAR(20),
    start_date DATE NOT NULL,
    end_date DATE,
    FOREIGN KEY (employer_airline_id) REFERENCES Airline(airline_id)
);

CREATE TABLE Flight (
    airline_id INT,
    flight_no VARCHAR(20),
    departure_airport CHAR(3),
    arrival_airport CHAR(4),
    departure_datetime DATETIME,
    airplane_id VARCHAR(10),
    PRIMARY KEY (airline_id, flight_no),

    FOREIGN KEY (airline_id) REFERENCES Airline(airline_id),
    FOREIGN KEY (departure_airport) REFERENCES Airport(airport_iata),
    FOREIGN KEY (arrival_airport) REFERENCES Airport(airport_icao),
    FOREIGN KEY (airplane_id) REFERENCES Airplane(airplane_tail_no)
);

CREATE TABLE CrewMemberFlight (
    pilot_id INT,
    airline_id INT,
    flight_no VARCHAR(20),
    crew_position ENUM('pilot', 'copilot', 'flight_engineer', 'flight_navigator'),
    PRIMARY KEY (pilot_id, airline_id, flight_no),
    FOREIGN KEY (pilot_id) REFERENCES CrewMember(pilot_id),
    FOREIGN KEY (airline_id, flight_no) REFERENCES Flight(airline_id, flight_no)
);

CREATE TABLE PassengerFlight (
    passenger_id INT,
    airline_id INT,
    flight_no VARCHAR(20),
    travel_class ENUM('First', 'Business', 'Economy'),
    seat_no VARCHAR(10),
```

```
    on_flight BOOLEAN,
    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)
);
```

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

In [ ]:
```
%%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[ ]: []

In [ ]:
```
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[ ]: 246

In [ ]:
```
%%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[ ]:

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 [ ]:

```
%%sql
SELECT DISTINCT customers.country
FROM customers
WHERE customers.country NOT IN (SELECT short_name FROM countries);
```

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

Out[ ]:

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 [ ]:

```
%%sql

UPDATE customers
SET country = 'Norway'
WHERE country = 'Norway__';

UPDATE customers
SET country = 'United States'
WHERE country = 'USA';
```

```
UPDATE customers
SET country = 'United Kingdom'
WHERE country = 'UK';

UPDATE customers
SET country = 'Russian Federation'
WHERE country = 'Russia';

# SELECT DISTINCT customers.country from CUSTOMERS;
```

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

Out[ ]: []

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`?

Answer

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

ALTER TABLE customers
```

```
ADD CONSTRAINT fk_iso_code
FOREIGN KEY (iso_code)
REFERENCES countries(alpha_3_code);

UPDATE customers
SET iso_code = (
    SELECT alpha_3_code
    FROM countries
    WHERE countries.short_name = customers.country
);

ALTER TABLE customers
DROP COLUMN country;

CREATE VIEW customers_country AS
SELECT c.customerNumber, c.customerName, co.short_name AS country, c.iso_code
FROM customers c
JOIN countries co ON c.iso_code = co.alpha_3_code;
```

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

Out[ ]: []

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 [ ]:
```
%%sql
```

```
SELECT iso_code AS country_iso, COUNT(*) AS number_of_customers
FROM customers_country
GROUP BY iso_code
ORDER BY number_of_customers DESC
LIMIT 10;
```

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

Out[ ]:

country_iso	number_of_customers
USA	36
DEU	13
FRA	12
ESP	7
AUS	5
GBR	5
NZL	4
ITA	4
FIN	3
CHE	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 [ ]:

```
%%sql
ALTER TABLE products
```

```
ADD COLUMN product_code_letter VARCHAR(1);

ALTER TABLE products
ADD COLUMN product_code_scale INT;

ALTER TABLE products
ADD COLUMN product_code_number INT;
```

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

Out[ ]: []

In [ ]:
```
# %%sql
# SELECT * FROM products
```

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 [ ]:
```
%%sql
UPDATE products
SET
    product_code_letter = LEFT(products.productCode, 1),
    product_code_scale = CAST(SUBSTRING(productCode, 2, LOCATE('_', productCode) - 2) AS SIGNED),
    product_code_number = CAST(SUBSTRING_INDEX(productCode, '_', -1) AS SIGNED)
;

ALTER TABLE products
ADD CONSTRAINT PRIMARY KEY (product_code_letter, product_code_scale, product_code_number);
```

```
ALTER TABLE products
DROP COLUMN productCode;
```

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

Out[ ]: []

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 [ ]:
```
%%sql
SELECT
    product_code_letter,
    product_code_scale,
    product_code_number,
    productScale,
    productName
FROM
    products
WHERE
        CAST(SUBSTRING_INDEX(productScale, ':', -1) AS SIGNED) != product_code_scale
ORDER BY
    productName;
```

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

Out[ ]:

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 [ ]:
```
%sql use classicmodels
```

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

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

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 [ ]:
```
%%sql
SELECT
    products.productName,
    products.productLine,
    products.productVendor,
    SUM(orderdetails.quantityOrdered * orderdetails.priceEach) AS totalRevenue
FROM
    products
JOIN
    orderdetails ON products.productCode = orderdetails.productCode
GROUP BY
    products.productCode, products.productName, products.productLine, products.productVendor
HAVING
    totalRevenue > 150000
ORDER BY
    totalRevenue DESC;
```

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

Out[ ]:

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 [ ]:
```
%%sql
SELECT
    orderdetails.productCode,
    products.productName,
    products.productVendor,
    COUNT(DISTINCT orders.customerNumber) AS customerCount
FROM
    orderdetails
JOIN
    products ON orderdetails.productCode = products.productCode
JOIN
    orders ON orderdetails.orderNumber = orders.orderNumber
GROUP BY
    orderdetails.productCode, products.productName, products.productVendor
ORDER BY
    customerCount DESC, orderdetails.productCode
LIMIT 10;
```

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

Out[ ]:

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 [ ]:
```sql
%%sql
SELECT
    customers.customerName,
    EXTRACT(MONTH FROM payments.paymentDate) AS month,
    EXTRACT(YEAR FROM payments.paymentDate) AS year,
    COALESCE(SUM(payments.amount), 0) AS monthlyExpenditure,
    customers.creditLimit
FROM
    customers
LEFT JOIN
    payments ON customers.customerNumber = payments.customerNumber
GROUP BY
    customers.customerNumber, customers.customerName, month, year, customers.creditLimit
HAVING
    monthlyExpenditure > customers.creditLimit OR monthlyExpenditure = 0
ORDER BY
    monthlyExpenditure DESC, customers.customerName
```

```
LIMIT 10;
```

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

Out[ ]:

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 [ ]:

```
%%sql
SELECT
```

```
    products.productCode,
    products.productName,
    products.productLine,
    products.productVendor,
    products.productDescription
FROM
    products
WHERE
    products.productCode NOT IN (
        SELECT DISTINCT
            orderdetails.productCode
        FROM
            orderdetails
            JOIN orders ON orderdetails.orderNumber = orders.orderNumber
            JOIN customers ON orders.customerNumber = customers.customerNumber
        WHERE
            customers.country = 'France'
    )
ORDER BY
    products.productCode;
```

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

Out[ ]:

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 [ ]:
```sql
%%sql

SELECT
    employees.firstName,
    employees.lastName,
    COUNT(DISTINCT customers.customerNumber) AS totalCustomers,
    COUNT(DISTINCT orders.orderNumber) AS totalCustomerOrders,
    CASE
        WHEN COUNT(DISTINCT customers.customerNumber) >= 8 AND COUNT(DISTINCT orders.orderNumber) >= 30 THE
        WHEN COUNT(DISTINCT customers.customerNumber) >= 8 THEN 'customers'
        WHEN COUNT(DISTINCT orders.orderNumber) >= 30 THEN 'orders'
    END AS raiseBecause
FROM
    employees
LEFT JOIN
    customers ON employees.employeeNumber = customers.salesRepEmployeeNumber
LEFT JOIN
    orders ON customers.customerNumber = orders.customerNumber
GROUP BY
```

```
    employees.employeeNumber, employees.firstName, employees.lastName
HAVING
    raiseBecause IS NOT NULL
ORDER BY
    employees.firstName;
```

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

Out[ ]:

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