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")
```

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.

1. Structured data follows a strict template. It has or fits into a defined data model and can easily be queried, extracted, and updated. Eg: University data from ReLax.
2. Semi-structured data does not have a rigid/strict schema but contains certain defining features such as semantic tags and/or metadata that gives it some structure/heirarchy. Eg: JSON data, csv file contents.
3. This has no strict schema or defining characteristics, essentially raw data. Eg: Audio/video files.

Source: [https://careerfoundry.com/en/blog/data-analytics/structured-vs-unstructured-data/#what-is-structured-data (https://careerfoundry.com/en/blog/data-analytics/structured-vs-unstructured-data/#what-is-structured-data)](https://careerfoundry.com/en/blog/data-analytics/structured-vs-unstructured-data/#what-is-structured-data)

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 capa
bilities.
```

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

1. Metadata
2. Security

1. The metadata is essentially a collection of rules, definitions, and procedures about data. As per Codd's rule, it must explain how the data is to be managed, stored, manipulated, etc. The metadata must be stored within the system and must be able to operate on it with the system's operations. Eg: All the descriptive data about data, its organisation, infor about users, accounts, and their relations must be stored as views or indices within a relational DB.
2. Security is a broad concept that includes authentication, data protection. Per the rule, the system should be capable of controlling access and ensuring security solely based on its relational capabilities. Eg: This can be done through manipulation of user-level accesses (granting restrictive access to one set of users, admin privileges to another set of users, etc), creating views that only show restrictive content, and even granting read-only privilegs.

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
):
```

Used the following: https://dev.mysql.com/doc/refman/8.0/en/view-updatability.html (https://dev.mysql.com/doc/refman/8.0/en/view-updatability.html)

1. (Not updatable)

   ```
   create view enrollAvgCred as
   select enrollment_year, avg(total_credits) as avg_credits
   from student
   group by enrollment_year;
   ```

2. (Not updatable)

   ```
   create view enrollMaxCred as
   select enrollment_year, max(total_credits) as max_credits
   from student
   group by enrollment_year;
   ```

1. (Updatable)

   ```
   create view studentInfo as
   select first_name, last_name, total_credits
   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 is not atomic since it can be divided into "year" (2024), "department code"(COMS), "course code" (4111W), and "section" (002).
2. The purpose of an index is to make it efficient to query certain types of queries. WIth non-atomic keys, the index is larger so not space efficient and it is unable ot support queries into the sub-parts of the non-atomic key since it treats it as a single unit.

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

[https://www.prepbytes.com/blog/dbms/joins-in-dbms/ (https://www.prepbytes.com/blog/dbms/joins-in-dbms/)](https://www.prepbytes.com/blog/dbms/joins-in-dbms/)

1. It joins the tables on all columns that have matching names and type. If none is found, then it performs an outer join.
2. It joins table based on equality (=) between columns.
3. It joins table based on comparisons between columns such as (<, >, etc).
4. It matches rows on join condition and for those rows of left table that do not have a match, it fills it with null values. So it keeps all rows of left table irrespective of match.
5. Similar to left join, it matches rows on join condition and for those rows of right table that do not have a match, it fills it with null values. So it keeps all rows of right table irrespective of match.
6. Returns all rows from both tables including those that do not match. When it doesn't match, it fills in corresponding values with null.
7. Only returns rows that match in both tables on join condition

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 automatically propagates changes made in primary key table to foreign key. Thus, if a value is updated/deleted then the foreign key table also udpates/deletes values.
2. If there are multiple foreign key tables that reference one primary key, then we only need to change the primary key table to affect all the other tables. Thus maintaining and updating tables is easier and simpler.

W7

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

1. Makes it easier to represent many-many relationships since FKs can't do that (one-to-one or many-to-one only)
2. Certain attributes only pertain to relationships between entities and not the entities themselves so having an associative entity that represents the relationship makes it easier to model it rather than use FKs.

W8

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

Closure property essentially means that an sql query can further be acted upon by sql operations. That is, we can nest queries and continue building a complex query using sub-queries.

In the following example, we use a subquery to first find average salary of instructors then we use it as a where condition to find all instructors that earn above average salary.

Eg:

```
select name, dept_name, salary
from instructor
where salary > (select avg(salary) from instructor)
```

W9

Briefly explain the differences between:

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

1. Stored Procedure: These are sets of SQL statements that have been pre-compiled and stored. These can be reused and shared across applications.
2. Functions: These are DB objects that take in some input parameters and output some value/table.
3. Triggers: These are also a set of statements but are automatically executed should certain conditions come to pass.

Stored procedures can return 0,1,or many values and can perform a variety of operations. Functions must return a value and they can't directly affect the DB (only have read-access). Triggers aren't directly called by users but they can affect a variety of things before or after user procedures.

https://www.c-sharpcorner.com/blogs/about-store-proc-function-trigger-in-brif (https://www.c-sharpcorner.com/blogs/about-store-proc-function-trigger-in-brif)

W10

List three benefits/use cases for defining views.

1. Easier to make changes to underlying table structure by using a view. It gives us design flexibility.
2. We can hide the WHERE clause or other columns to which you do not want the user to have access, thus adding a layer of security and privacy.
3. Can simplify complex queries and joins by creating views.

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.

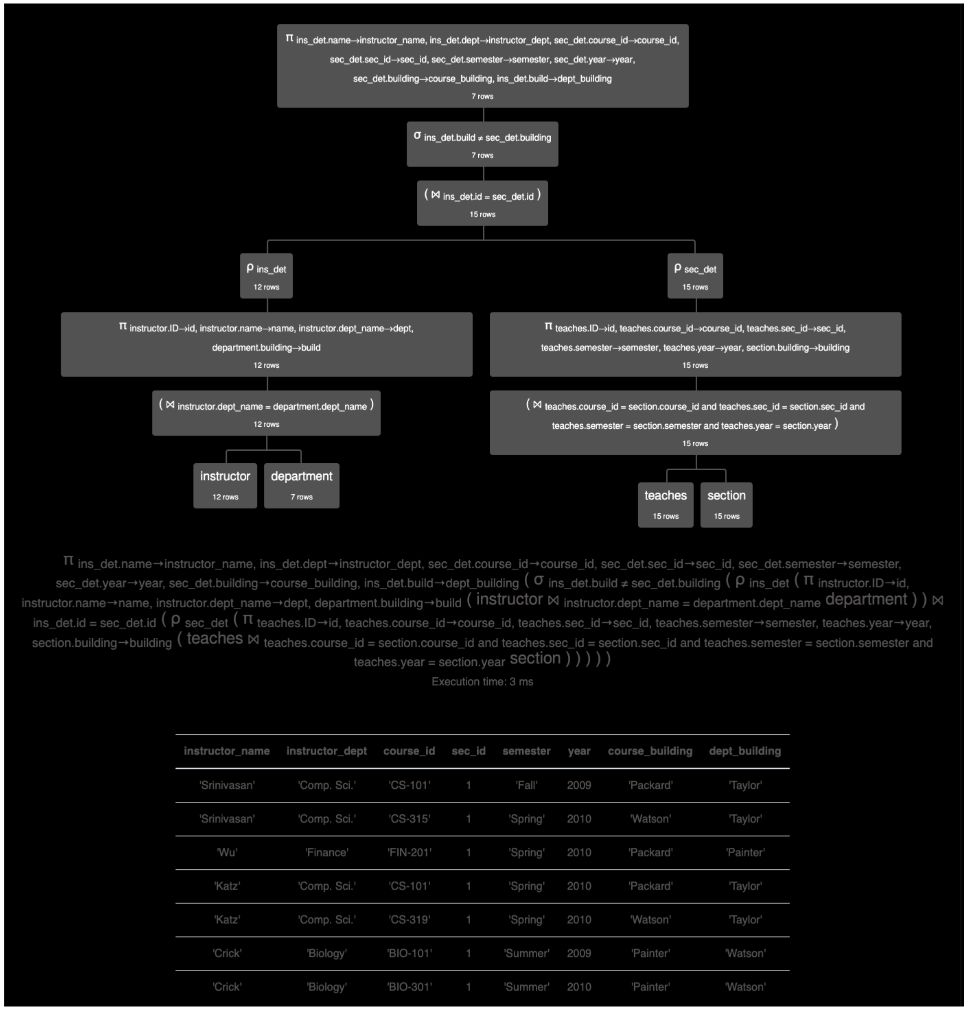
```
Algebra statement:

```
π   instructor_name <- ins_det.name,
    instructor_dept <- ins_det.dept,
    course_id ← sec_det.course_id,
    sec_id ← sec_det.sec_id,
    semester ← sec_det.semester,
    year ← sec_det.year,
    course_building <- sec_det.building,
    dept_building <- ins_det.build (σ ins_det.build != sec_det.building (
        ρ ins_det (
            π id <- instructor.ID,
            name ← instructor.name,
```
```

```
            dept ← instructor.dept_name,
            build ← department.building (
                instructor ⋈ instructor.dept_name = department.dept_name department))
⋈ ins_det.id = sec_det.id
    (ρ sec_det (π id <- teaches.ID,
            course_id ← teaches.course_id,
            sec_id ← teaches.sec_id,
            semester ← teaches.semester,
            year ← teaches.year,
            building ← section.building (
                teaches ⋈ teaches.course_id = section.course_id ∧
                            teaches.sec_id = section.sec_id ∧
                            teaches.semester = section.semester ∧
                            teaches.year = section.year section)))))
```

```
In [3]: from IPython.display import Image
        Image(filename='./R1.png', width=480, height=480)
```

Out[3]:



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'

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 **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.

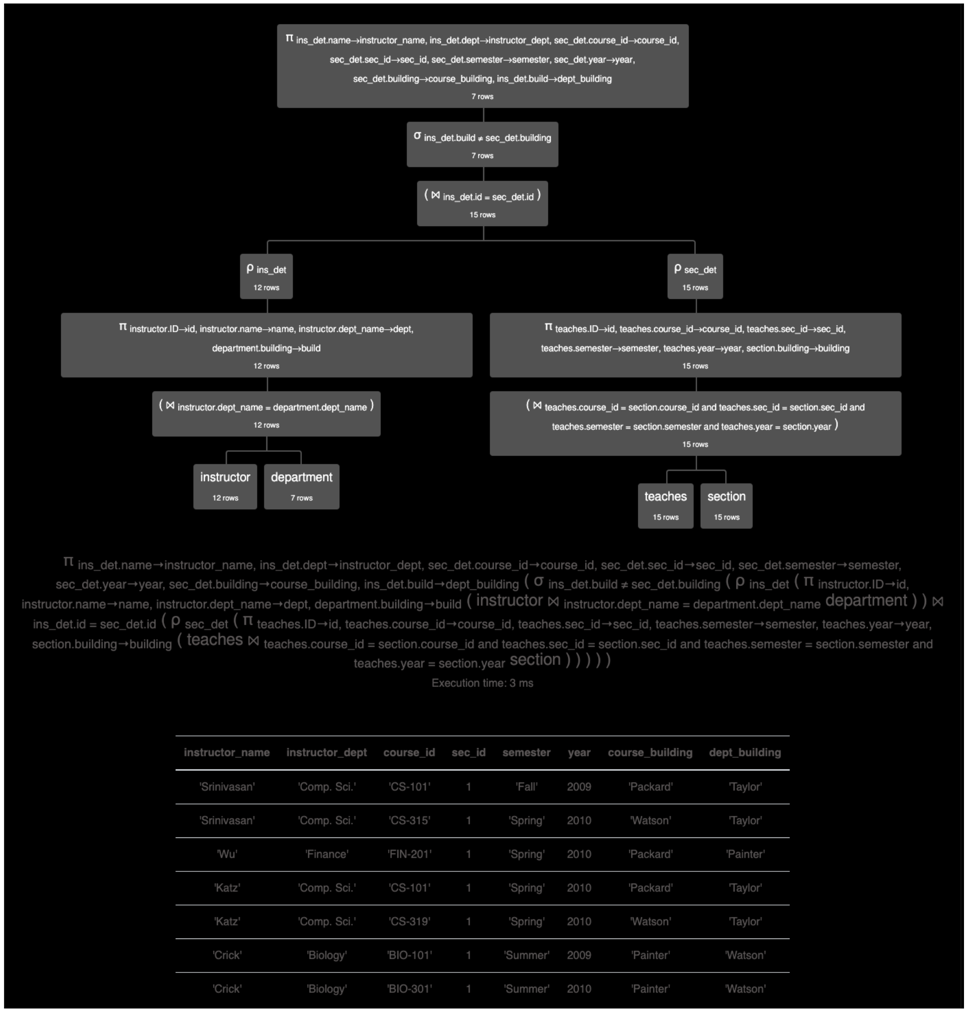
Algebra statement:

```
π
instructor_name <- ins.name,
student_name <- studs.name,
dept_name <- studs.dept (
(ρ studs (
π
id <- student.ID,
name <- student.name,
dept <- student.dept_name (
σ advisor.s_id = NULL (
student ⟖ student.ID = advisor.s_id advisor))
))
⋈ studs.dept = ins.dept
(ρ ins (
π
id <- instructor.ID,
name <- instructor.name,
dept <- instructor.dept_name (
σ advisor.i_id = NULL (
instructor ⟖ instructor.ID = advisor.i_id advisor))
```

```
))
)
```
```

In [4]:
```python
Image(filename='./R1.png', width=480, height=480)
```

Out[4]:



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'
```

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.
- 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`
- 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:

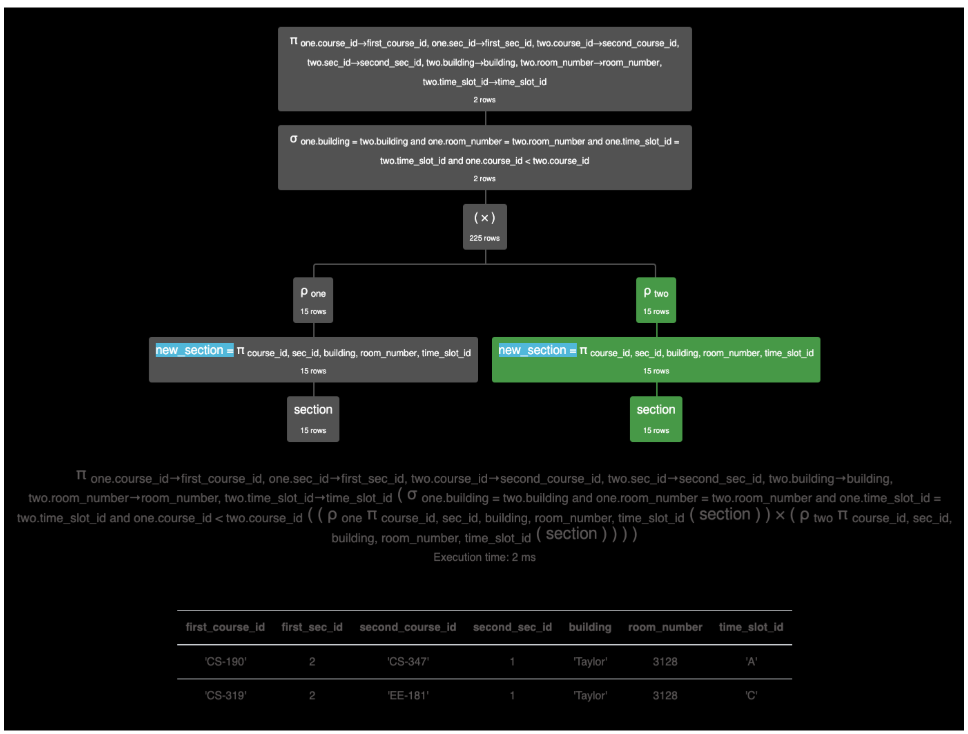
```
new_section = π course_id, sec_id, building, room_number, time_slot_id (section)
π
first_course_id <- one.course_id,
```
```

```
first_sec_id <- one.sec_id,
second_course_id <- two.course_id,
second_sec_id <- two.sec_id,
building <- two.building,
room_number <- two.room_number,
time_slot_id <- two.time_slot_id (
σ one.building = two.building ∧
one.room_number = two.room_number ∧
one.time_slot_id = two.time_slot_id ∧
one.course_id < two.course_id (
(ρ one new_section) × (ρ two new_section)))
```

Execution:

In [5]:
```
Image(filename='./R3.png', width=480, height=480)
```

Out[5]:



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'

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.
  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.

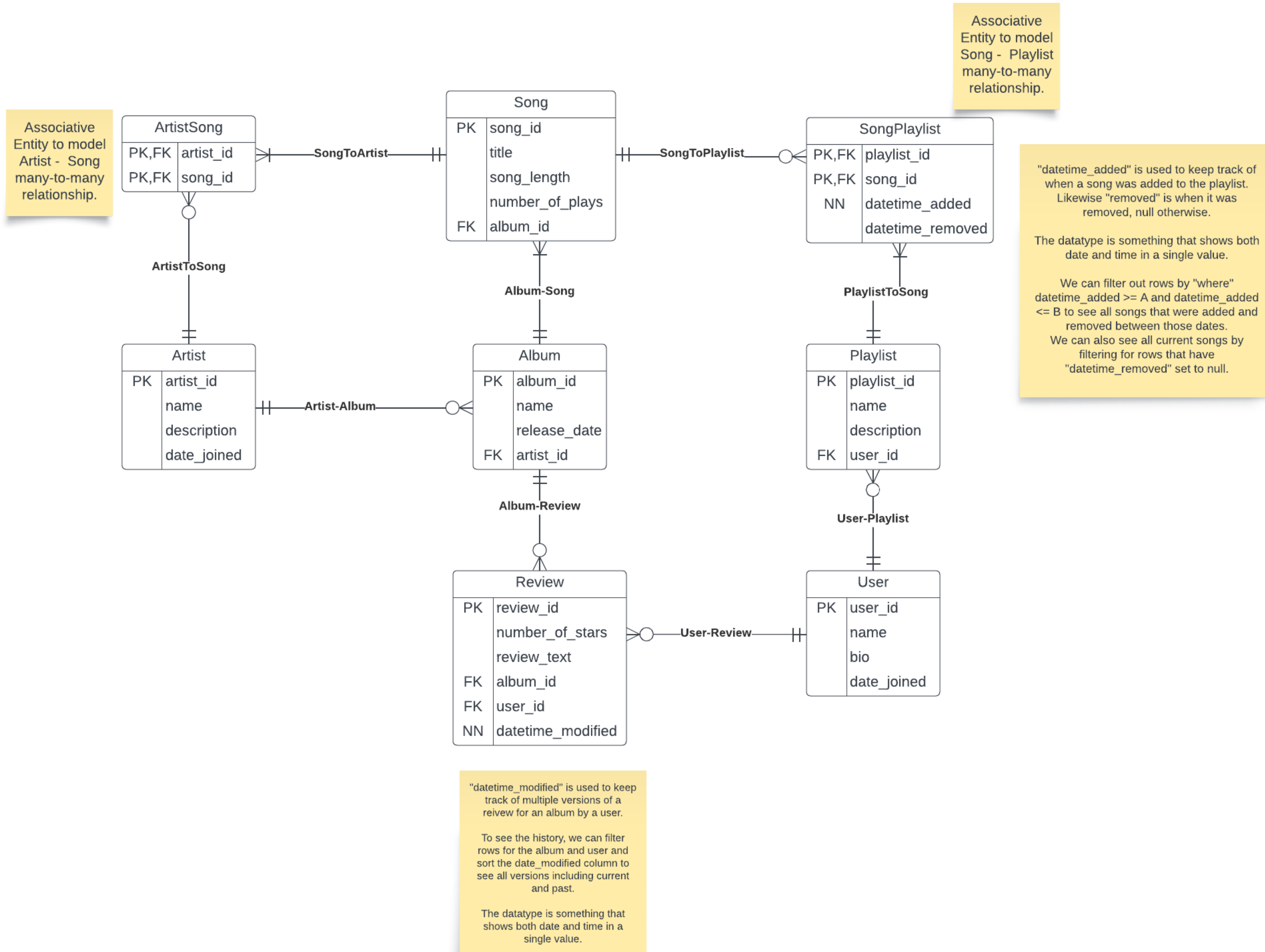
Assumptions and Documentation

In the diagram in yellow sticky notes.

In [6]:

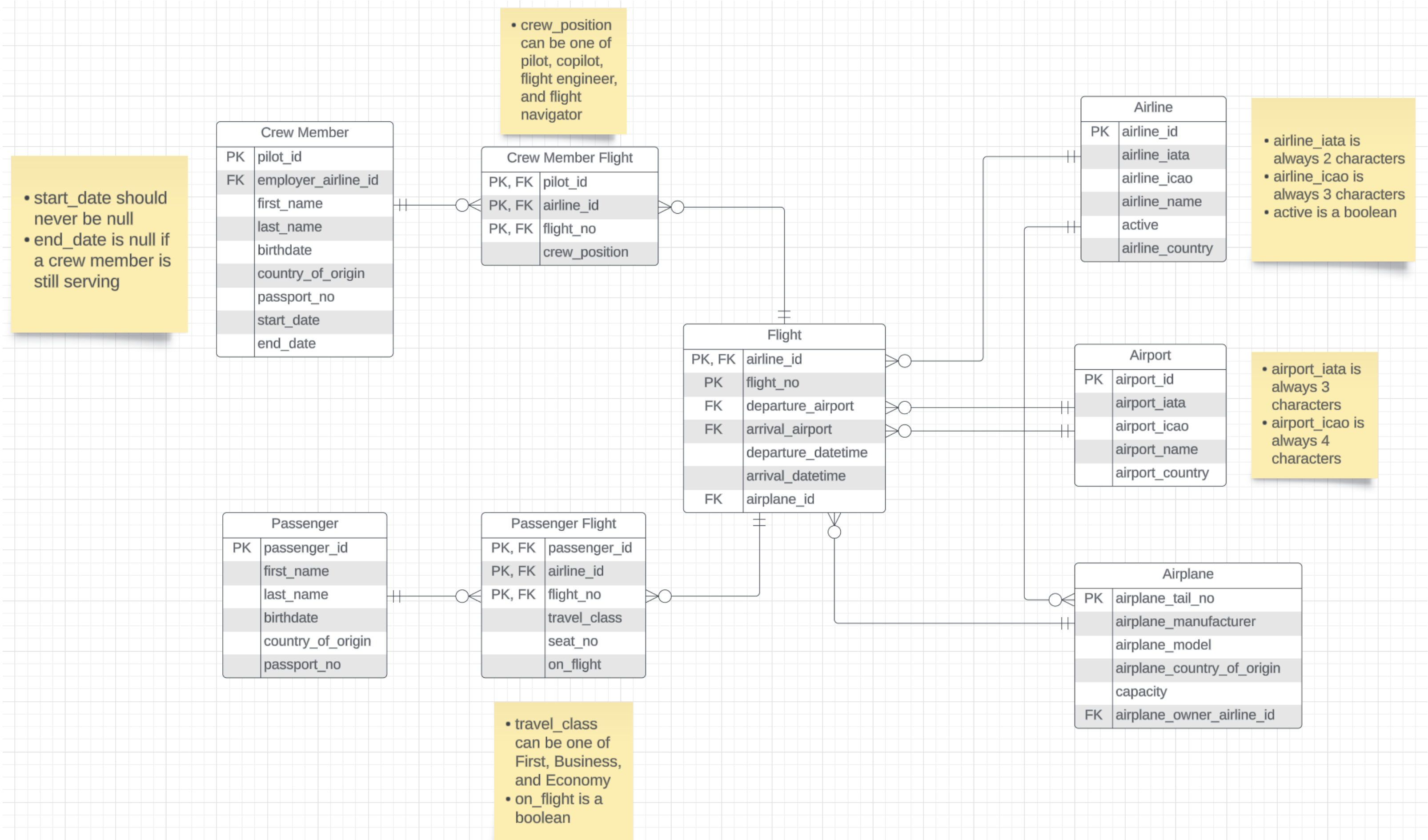
```python
# Diagram:
Image(filename='./ER-diagram.png')
```

Out[6]:



Model to DDL

- This question tests your ability to convert an ER diagram to DDL.
- Given the ER diagram below, 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.
  - 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_
- All \_id and flight_no fields are INTs
- Passport_no is varchar(30)
- Seat no is varchar(5)
- All names, country are varchar within 255 characters
```

create table Passenger
(
    passenger_id      int           PRIMARY KEY,
    first_name        varchar(255) not null,
    last_name         varchar(255) not null,
    birthdate         date         not null,
    country_of_origin varchar(255) not null,
    passport_no       varchar(30)  not null
);


create table Airline
(
    airline_id        int PRIMARY KEY,
    airline_iata      char(2) not null,
    airline_icao      char(3) not null,
    airline_name      varchar(255) not null,
    active            boolean not null,
    airline_country   varchar(255) not null
);


create table Airport
(
    airport_id        int PRIMARY KEY,
    airport_iata      char(3) not null,
    airport_icao      char(4) not null,
    airport_name      varchar(255) not null,
    airport_country   varchar(255) not null
);


create table Airplane
(
~~~

```
    airplane_tail_no             int PRIMARY KEY,
    airplane_manufacturer        varchar(255) not null,
    airplane_model               varchar(255) not null,
    airplane_country_of_origin   varchar(255) not null
    capactiy                     int not null,
    airplane_owner_airline_id    int not null,
    foreign key (airplane_owner_airline_id) references Airline(airline_id)
);


create table Flight
(
    airline_id              int,
    flight_no               int,
    departure_airport       int not null,
    arrival_airport         int not null,
    departure_datetime      datetime not null,
    arrival_datetime        datetime not null,
    airplane_id             int not null,
    primary key (airline_id, flight_no),
    foreign key (airline_id) references Airline(airline_id),
    foreign key (departure_airport) reference Airport(airport_id),
    foreign key (arrival_airport) reference Airport(airport_id)
);


create table CrewMember
(
    pilot_id              int PRIMARY KEY,
    employer_airline_id   int,
    first_name            varchar(255) not null,
    last_name             varchar(255) not null,
    birthdate             date no null,
    country_of_origin     varchar(255) not null,
    passport_no           varchar(30)  not null,
    start_date            date not null,
    end_date              date,
    foreign key (airline_id) references Airline(airline_id),
    foreign key (departure_airport) reference Airport(airport_id),
    foreign key (arrival_airport) reference Airport(airport_id)
);
```

````
CREATE TABLE CrewMemberFlight (
    pilot_id       int,
    airline_id     int,
    flight_no      int,
    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) references Airline(airline_id),
    foreign key (flight_no) references Flight(flight_no)
);


CREATE TABLE PassengerFlight (
    passenger_id   int,
    airline_id     int,
    flight_no      int,
    travel_class   ENUM('First', 'Business', 'Economy'),
    seat_no        varchar(5),
    on_flight      boolean not null,
    primary key (passenger_id, airline_id, flight_no),
    foreign key (passenger_id) references Passenger(passenger_id),
    foreign key (airline_id) references Airline(airline_id),
    foreign key (flight_no) references Flight(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 (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 country from
customers left join countries on customers.country = countries.short_name
where short_name is NULL
group by country
```

```
 * 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's a space 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 = "United States"
where country = "USA";

update customers
set country = "United Kingdom"
where country = "UK";

update customers
set country = "Russian Federation"
where country = "Russia";

update customers
set country = "Norway"
where country like "Norway% ";

# Used the following resource to learn about "LIKE" and "%"
# https://stackoverflow.com/questions/9946219/sql-like-wildcard-space-character
```

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

*Answer*

In [12]:
```
%%sql

alter table customers
add column iso_code char(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 customers.customerNumber as customerNumber,
        customers.customerName as customerName,
        countries.short_name as country,
        customers.iso_code as iso_code
from customers join countries on customers.iso_code = countries.alpha_3_code;
```

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

Out[12]: []

## 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 [13]:
```
%%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[13]:

| 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 [14]:
```
%%sql
alter table products
add column product_code_letter char(1),
add column product_code_scale int,
add column product_code_number int;
```

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

Out[14]: []

## 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. Change the PK of `products` from `productCode` to `(product_code_letter, product_code_scale, product_code_number)`.
3. Drop `productCode`.

In [15]:
```
%%sql
update products
set product_code_letter = substring(productCode, 1, 1),
    product_code_scale = cast(substring(substring_index(productCode, '_', 1), 2) as unsigned),
    product_code_number = cast(substring_index(productCode, '_', -1) as unsigned);

alter table products
add constraint triple_pk
primary key (product_code_letter, product_code_scale, product_code_number);

alter table products
drop productCode;


# Looked up CAST, SUBSTRING
# https://www.w3schools.com/sql/func_sqlserver_substring.asp
# https://www.w3schools.com/sql/func_sqlserver_cast.asp
```

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

Out[15]: []

## 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 [16]:
```
%%sql
select product_code_letter, product_code_scale, product_code_number, productScale, productName
from products
where product_code_scale != substring_index(productScale, ':', -1)
order by productName;
```

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

Out[16]:

| 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.

In [17]:
```
%sql use classicmodels
```

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

Out[17]: []

# 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 [18]:

```
%%sql
select productName, productLine, productVendor, sum(quantityOrdered*priceEach) as totalRevenue
from products join orderdetails on products.productCode = orderdetails.productCode
group by productName, productLine, productVendor
having totalRevenue > 150000
order by totalRevenue desc

# Debugged "where" error and turned it into "having"
#https://www.geeksforgeeks.org/having-vs-where-clause-in-sql/
```

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

Out[18]:

| 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 [19]:
```
%%sql
with one as (select products.productCode, productName, productVendor, orderNumber
    from products join orderdetails on products.productCode = orderdetails.productCode
    ),
    two as ( select productCode, productName, productVendor, customerNumber
    from one join orders on orders.orderNumber = one.orderNumber
    group by productCode, productName, productVendor, customerNumber
    )
select productCode, productName, productVendor, count(*) as customerCount
from two
group by productCode, productName, productVendor
order by customerCount desc, productCode
limit 10
```

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

Out[19]:

| 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. 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.
- 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 [20]:
```
%%sql
with one as (select
        customerNumber,
        substring_index(substring_index(paymentDate, '-', -2), '-', 1) as month,
        substring_index(paymentDate, '-', 1) as year,
        sum(amount) as monthlyExpenditure
    from payments
    group by customerNumber, month, year)
select customerName, month, year, IFNULL(monthlyExpenditure,0) as monthlyExpenditure, creditLimit
from one right join customers as c on one.customerNumber = c.customerNumber
where month is null or monthlyExpenditure > creditLimit
order by monthlyExpenditure desc, customerName
limit 10
```

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

Out[20]:

| 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 [21]:
```
%%sql
with one as (select productCode, productName, productLine, productVendor, productDescription
            from products),
    two as (select customerNumber, orders.orderNumber, productCode
           from orders left join orderdetails on orders.orderNumber = orderdetails.orderNumber),
    three as (select two.customerNumber, productCode, country
            from two join customers on two.customerNumber = customers.customerNumber)
select one.productCode as productCode, productName, productLine, productVendor, productDescriptio
from one left join three on one.productCode = three.productCode
group by productCode, productName, productLine, productVendor, productDescription
having max(case when country="France" then 1 else 0 end) = 0
order by productCode
```

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

Out[21]:

| 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 [22]:
```
%%sql
with one as (select customerNumber, employeeNumber, lastName, firstName
            from customers join employees on customers.salesRepEmployeeNumber = employees.employe
            ),
    two as (select  employeeNumber, lastName, firstName, count(*) as totalCustomers
            from one
            group by employeeNumber, lastName, firstName
            ),
    three as (select orders.orderNumber, orders.customerNumber, one.employeeNumber, one.lastName,
              from one join orders on one.customerNumber = orders.customerNumber),
    four as (select employeeNumber, lastName, firstName, count(*) as totalCustomerOrders
            from three
            group by employeeNumber, lastName, firstName
            ),
    five as (select two.firstName, two.lastName, two.totalCustomers, four.totalCustomerOrders
            from two left join four on two.employeeNumber = four.employeeNumber
            union
            select four.firstName, four.lastName, two.totalCustomers, four.totalCustomerOrders
            from four left join two on two.employeeNumber = four.employeeNumber)
select five.firstName as firstName, five.lastName as lastName, five.totalCustomers as totalCustom
       five.totalCustomerOrders as totalCustomerOrders,
       (case when totalCustomers >= 8 and totalCustomerOrders >= 30 then 'both'
             when totalCustomers >= 8 then 'customers'
             when totalCustomerOrders >= 30 then 'orders'
             else 'error' end) as raiseBecause
from five
where totalCustomers >= 8 or totalCustomerOrders >= 30
order by five.firstName
```

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

Out[22]:

| 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 |