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.

Structured data adheres to a specific format/schema and is easy to search and organize, where the raw data is mapped into predesigned fields that can be extracted and read through SQL easily. An example is a customer database where records are stored in rows and columns, with fields for name, address, phone number etc.

Semi-Structured data has some consistent and definite characteristics, does not follow a strict data model, and often contains tags or markers to enforce some structure. It contains some variability and inconsistency. An example of semi-structured data format is delimited files, as they contain elements that can break down the data into separate hierarchies.

Unstructured data is data present in absolute raw form, data that does not have a pre-defined data model or not organized in a pre-defined manner. It is difficult to process due to its complex arrangement and formatting. An example of unstructured data is social media posts, where the data is not in a specific format.

Reference: https://www.astera.com/type/blog/structured-semi-structured-and-unstructured-data/#:~:text=Organization%3A%20Structured%20data%20is%20well,is%20not%20organized%20at%20all.

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: system metadata must be stored in a relational representation, most often called the Data dictionary or system catalog. It must be accessible and manageable through the same relational operation. For example, information about relations, user and accounting information, statistical and descriptive data about the data, physical file organization information and information about indices are stored in relational representation. Relation_metadata could be one relation, index_metadata could be one relation, view_metadata could be on relation, attribute_metadata could be one relation and user_metadata could be one relation.
2. Security: security mechanisms should also be managed using relational capabilities in that the system should control access to data based on relational concepts. For example, security can be managed through the creation and manipulation of users, roles, and permissions stored in relational tables and controlled through relational operations. You can use a SQL command to grant a user read-only access to certain relations or revoke permissions from a role, e.g. "GRANT SELECT ON table_name TO user_name;".

Reference: https://chat.openai.com/

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
);
```

1. 2 Views that are not possible to update:

CREATE VIEW StudentSummary AS SELECT enrollment_year, COUNT(*) AS NumberOfStudents, AVG(total_credits) AS AverageCredits FROM student GROUP BY enrollment_year;

CREATE VIEW StudentDuplicates AS SELECT a.social_security_no, a.last_name AS left_student_last_name, b.last_name AS right_student_last_name, a.first_name, b.enrollment_year FROM student a, student b WHERE a.social_security_no = b.social_security_no AND a.enrollment_year <> b.enrollment_year;

2. One View that ispossible to update:

CREATE VIEW SimpleStudentView AS SELECT social_security_no, last_name, first_name FROM student;

Reference: https://chat.openai.com/

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. The key is nonatomic as it can be divided or decomposed into smaller parts with individual meaning: 2024 refers to the year, COMS4111 refers to the course and W002 refers to the section.
2. Non-atomic keys create problems for indexes because they contain multiple pieces of information combined into one field, which complicates search operations. When an index is built on a non-atomic key, it cannot efficiently support queries searching for individual components of the key since the index treats the key as a single unit. This reduces the effectiveness of the index in speeding up searches and can lead to slower query performance.

Reference: https://chat.openai.com/

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. A natural join is a JOIN operation that creates an implicit join clause for you based on the common columns in the two tables being joined. Common columns are columns that have the same name in both tables.
2. Equi-join performs a JOIN against equality or matching column(s) values of the associated tables. An equal sign (=) is used as comparison operator in the where clause to refer equality.
3. Theta join combines tuples from different relations provided they satisfy the theta condition.
4. Left join joins two tables and fetches all matching rows of two tables for which the SQL-expression is true, plus rows from the frist table that do not match any row in the second table.
5. Right join joins two tables and fetches rows based on a condition, which is matching in both the tables (before and after the JOIN clause mentioned in the syntax below) , and the unmatched rows will also be available from the table written after the JOIN clause
6. Outer join returns all rows from both the participating tables which satisfy the join condition along with rows which do not satisfy the join condition. The SQL OUTER JOIN operator (+) is used only on one side of the join condition.
7. Inner join selects all rows from both participating tables as long as there is a match between the columns. An SQL INNER JOIN is same as JOIN clause, combining rows from two or more tables.

https://docs.oracle.com/javadb/10.6.2.1/ref/rrefsqljnaturaljoin.html#:~:text=A%20NATURAL%20JOIN%20is%20a,The%20default%
https://www.w3resource.com/sql/tutorials.php https://www.tutorialspoint.com/dbms/database_joins.htm

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 automatic propagation of changes from one table to another due to the constraints of a foreign key relationship. The cascade rule of the foreign key states that when any entry is deleted or updated from the parent table then all the dependent rows in the child table should also get deleted or updated. This maintains data integrity and consistency between related tables.

2. For the above foreign key relationship: If an order is deleted from the orders table, cascading delete to orderdetails will automatically remove all order details related to that order. If the orderNumber in orders were to be updated, cascading update to orderdetails will also update all the corresponding orderNumber entries in orderdetails.

https://www.tutorialspoint.com/what-is-cascade-rule-for-the-foreign-key-on-one-table-referencing-to-another-table#:~:text=The%20CASCADE%20rule%20of%20the,table%20should%20also%20get%20deleted.

W7

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

1. We use associative entities to resolve many-to-many relationships, since foreign keys can only represent one-to-one or one-to-many relationships.
2. We use associative entities in cases we need to store properties about the relationships that are not attributes of the connected entities, and this cannot be implemented with foreign keys.

W8

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

SQL is closed under its operation since the result of a SQL query is another table which can be used as an input for another SQL query, allowing for the composition of SQL operations thorugh layering or nesting of queries, enabling more complex data retrieval and analysis.

Example:

SELECT EmployeeName FROM (SELECT EmployeeName, DepartmentID FROM Employees WHERE Year(HireDate) > 2010) AS RecentHires WHERE DepartmentID = 3;

Reference: https://chat.openai.com/

W9

Briefly explain the differences between:

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

Stored procedures are reusable sets of SQL statements that can accept parameters and return results; functions are named operations that return a single value or a table, and triggers are special stored procedures that are executed automatically in response to specific database events.

Functions cannot make changes to databases, but they can read them, and they must also return a value. Stored procedures can accomplish anything and anything with databases and can return 0 or many values.

Unlike stored procedures and functions, which are called explicitly, triggers work implicitly in the background and can't be called directly by users. They can be designed to execute before or after the data modification event they are associated with.

https://www.scholarhat.com/tutorial/sqlserver/difference-between-stored-procedure-and-function-in-sql-server#:~:text=Stored%20procedures%20are%20reusable%20sets,response%20to%20specific%20database%20events.

W10

List three benefits/use cases for defining views.

1. A view provides a mechanism to hide certain data from certain users, allowing them to see a relation that is not a conceptual model and not the entire logical model.
2. Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
3. Views can act as a security mechanism by allowing users to access data through the view, and not providing access to underlying relations.

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'

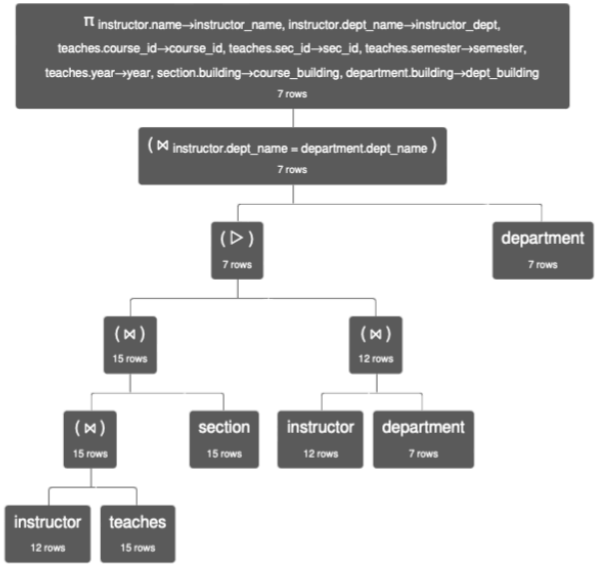
- Srinivasan taught CS-101, section 1 in Fall of 2009 in the Packard building. However, Srinivasan is in the CS department, whose building is Taylor.

Algebra statement:

```
 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

((((instructor ⋈ teaches) ⋈ section)
    ▷  (instructor⋈department))
    ⋈ 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) ▷ (instructor ⋈ department)) ⋈ instructor.dept_name = department.dept_name department)

Execution time: 4 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 **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:

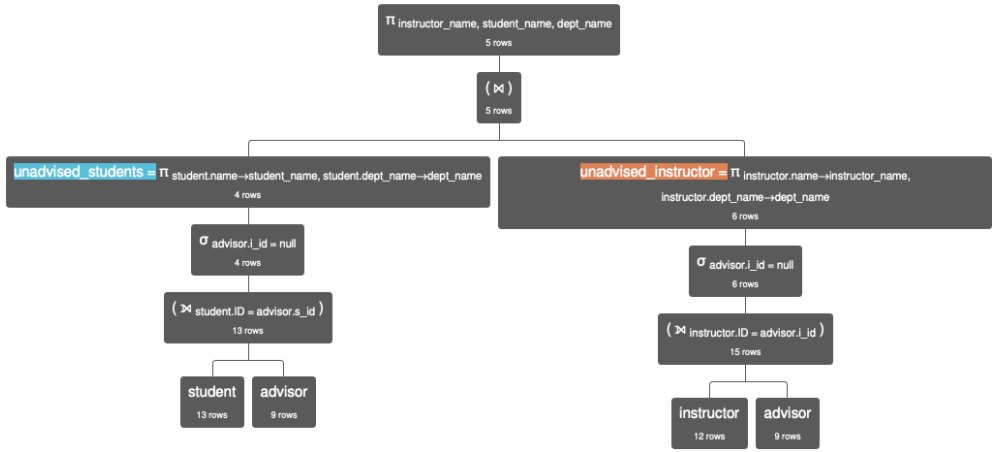
```
unadvised_students =
π student_name←student.name, dept_name←student.dept_name
σ advisor.i_id = NULL
(student
⟕ student.ID = advisor.s_id
advisor)

unadvised_instructor =
π instructor_name←instructor.name, dept_name←instructor.dept_name
```

```
(σ advisor.i_id = NULL
(instructor
⋈ instructor.ID = advisor.i_id
advisor))

π instructor_name, student_name, dept_name
(unadvised_students
⋈ unadvised_instructor)
```

Execution:



π instructor_name, student_name, dept_name (π student.name→student_name, student.dept_name→dept_name σ advisor.i_id = null (student ⋈ student.ID = advisor.s_id advisor) ⋈ π instructor.name→instructor_name, instructor.dept_name→dept_name (σ advisor.i_id = null (instructor ⋈ instructor.ID = advisor.i_id advisor)))

Execution time: 4 ms

instructor_name	student_name	dept_name
'El Said'	'Brandt'	'History'
'Califieri'	'Brandt'	'History'
'Brandt'	'Williams'	'Comp. Sci.'
'Mozart'	'Sanchez'	'Music'
'Gold'	'Snow'	'Physics'

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.
- 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:

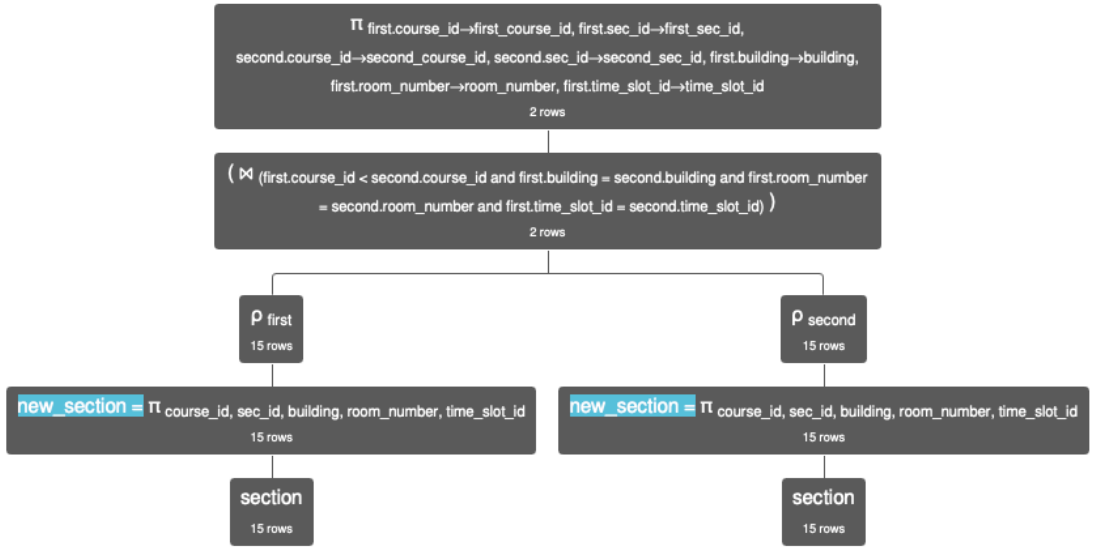
```
new_section = π course_id, sec_id, building, room_number, time_slot_id (section)
```

```
 π
    first_course_id←first.course_id,
    first_sec_id←first.sec_id,
    second_course_id←second.course_id,
    second_sec_id←second.sec_id,
    building←first.building,
    room_number←first.room_number,
    time_slot_id←first.time_slot_id
(
    (ρ first (new_section))
    ⋈


    (first.course_id < second.course_id
    ∧
    first.building = second.building
    ∧
    first.room_number = second.room_number
    ∧
    first.time_slot_id = second.time_slot_id)


    (ρ second (new_section))
)
```

Execution:



π first.course_id→first_course_id, first.sec_id→first_sec_id, second.course_id→second_course_id, second.sec_id→second_sec_id, first.building→building, first.room_number→room_number, first.time_slot_id→time_slot_id ((ρ first π course_id, sec_id, building, room_number, time_slot_id (section)) ⋈ (first.course_id < second.course_id and first.building = second.building and first.room_number = second.room_number and first.time_slot_id = second.time_slot_id) (ρ second π course_id, sec_id, building, room_number, time_slot_id (section)))

Execution time: 3 ms

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'

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.
    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.

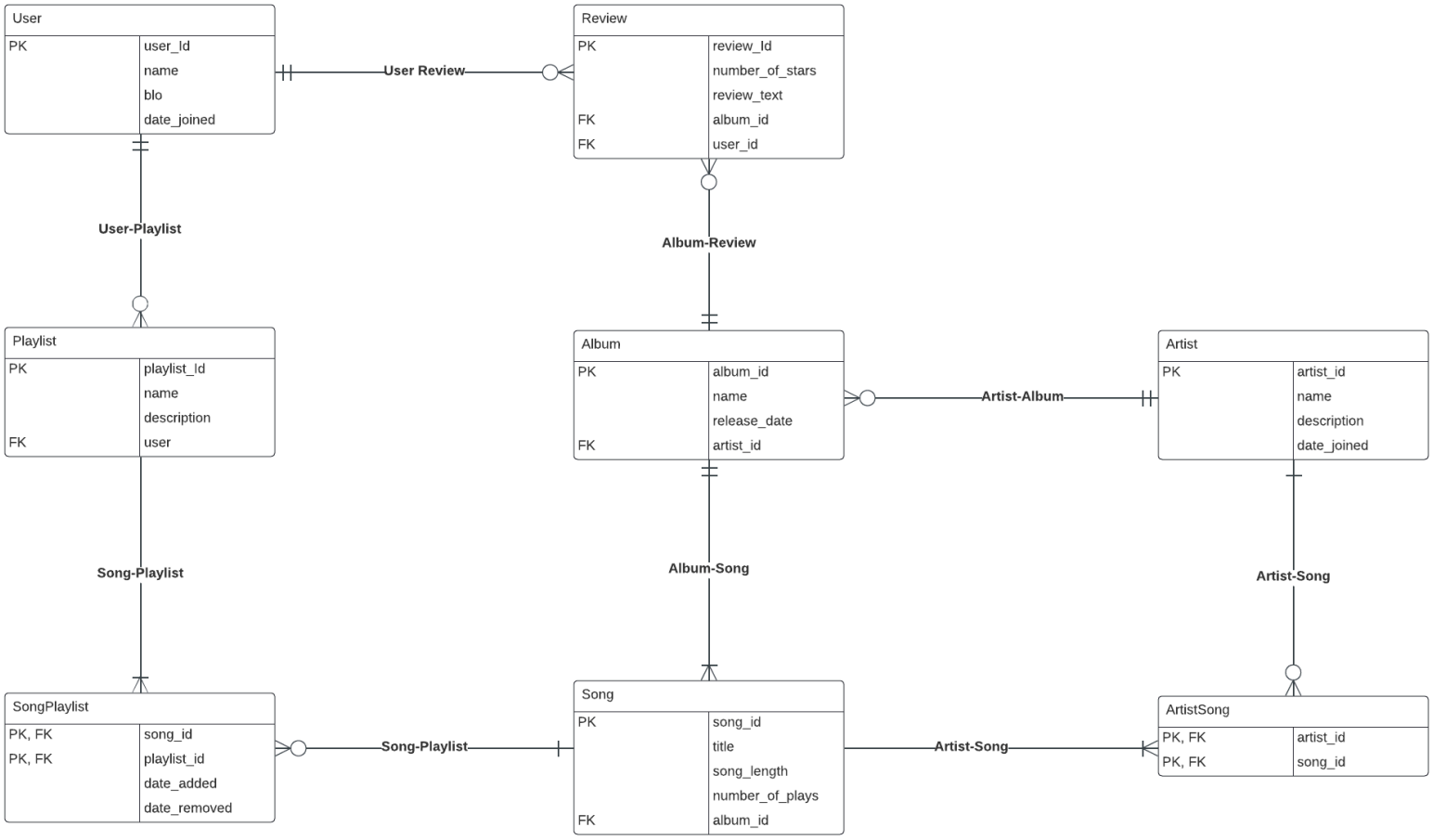
Assumptions and Documentation

1. the ArtistSong table represents the many-to-many relationship between artists and songs. No single column uniquely identifies a record since an artist can be associated with many songs and a song can have many artists. However, the combination of artist_id and song_id does uniquely identify each record, making it a composite primary key.
2. The SongPlaylist table helps track which songs are in which playlists and for how long. Assuming that a song can only be added once at any given time, the combination of song_id and playlist_id serves as a composite primary key.
3. A song can be added and removed from playlists multiple times. The date_added and date_removed fields in the SongPlaylist table are used to track these changes, with the understanding that a NULL in date_removed indicates the

song is currently in the playlist.

4. Requirement 2: to implement the tracking of current and previous reviews, as well as their dates, the Review table includes review_date. This allows tracking when the review was made. To handle multiple reviews by the same user for the same album, Use the review_date to identify the most recent review. Assume that only the most recent review per user per album is "active" or "current". Older reviews remain in the table for historical tracking but are not considered the active review. This historical tracking is done by maintaining all entries in the Review table but recognizing the one with the latest date for each user-album pair as the current review.
5. Requirement 3: The SongPlaylist associative entity tracks the songs in playlists over time with date_added and date_removed attributes. For the current songs in a playlist, date_removed would be NULL. This setup allows the system to see which songs were in a playlist at any historical point in time based on the date range they were added and removed. It supports querying for all songs in a playlist where the current date is between the date_added and date_removed (or date_removed is NULL) to find the current songs in a playlist. This also allows tracking the history of a playlist's composition.

Diagram:

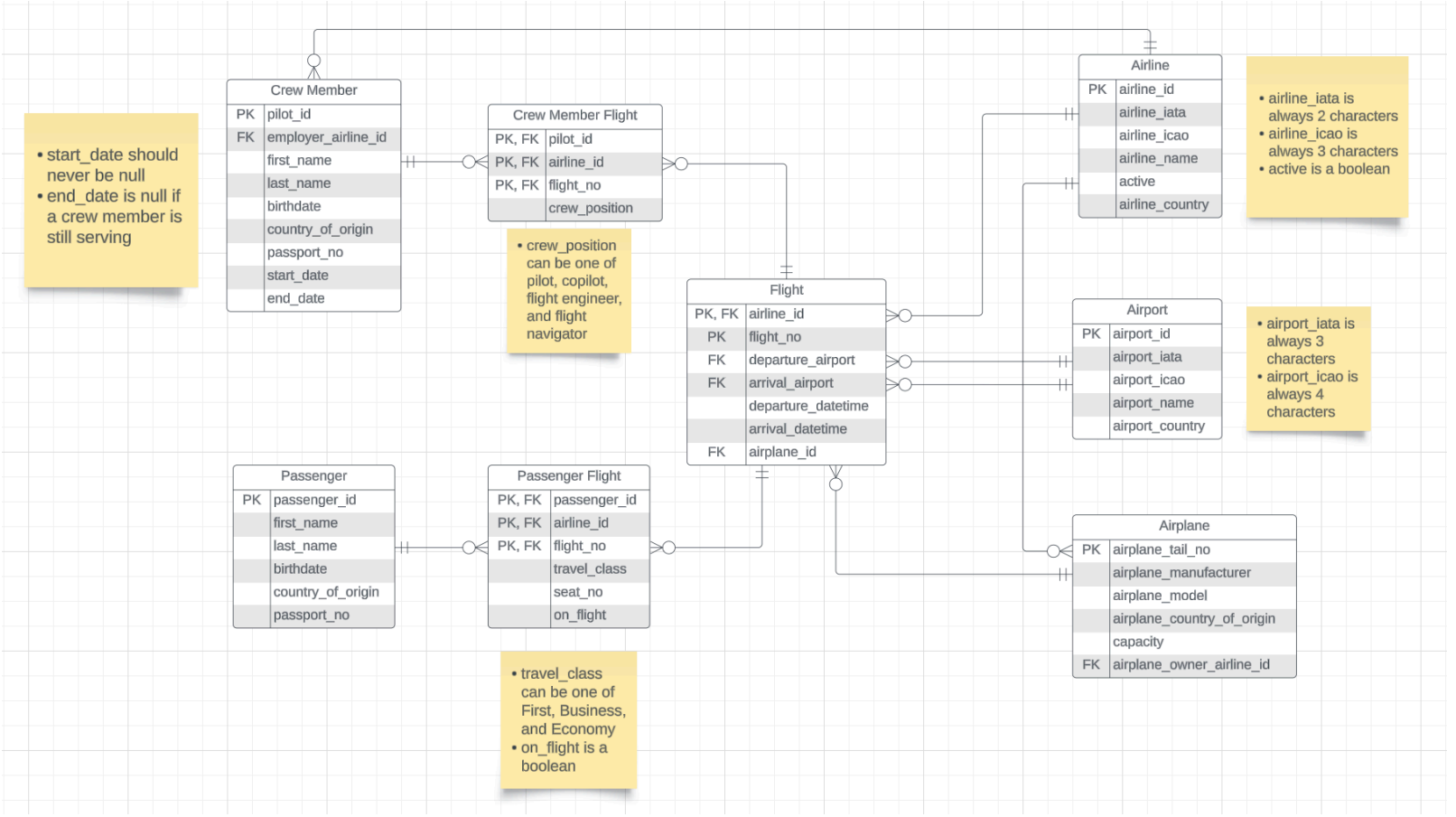


Definition to Model ER Diagram

Model to DDL

- This question tests your ability to convert an ER diagram to DDL.
- Given the ER diagram below, 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 the IDs for all tables can represented as unique integers.
- employer_airline_id in Crew Member table refers to the airline that the crew member is employed under, and references Airline.

- Crew_position values (pilot, copilot, flight engineer, and flight navigator), and travel_class values (first, business and economy) are not expected to change frequently. Hence ENUM is used instead of CHECK.
- passport_no has maximum 10 characters and comprises both numbers and letters.
- seat_no has maximum 5 characters and comprises both numbers and letters.
- flight_no has maximum 10 characters and comprises both numbers and letters.
- All names and countries are given more characters as they could be a very large length.

```
CREATE TABLE CrewMember (
    pilot_id INT PRIMARY KEY,
    employer_airline_id INT,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    birthdate DATE,
    country_of_origin VARCHAR(255),
    passport_no VARCHAR(10),
    start_date DATE NOT NULL,
    end_date DATE,
    FOREIGN KEY (employer_airline_id) REFERENCES Airline(airline_id)
);


CREATE TABLE CrewMemberFlight (
    pilot_id INT,
    airline_id INT,
    flight_no VARCHAR(10),
    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 Flight (
    airline_id INT,
    flight_no VARCHAR(10),
```

```
    departure_airport VARCHAR(255),
    arrival_airport VARCHAR(255),
    departure_datetime DATETIME,
    arrival_datetime DATETIME,
    airplane_id INT,
    PRIMARY KEY (airline_id, flight_no),
    FOREIGN KEY (airline_id) REFERENCES Airline(airline_id),
    FOREIGN KEY (departure_airport) REFERENCES Airport(airport_code), -- Assuming there's
an Airport table with 'airport_code' as PK
    FOREIGN KEY (arrival_airport) REFERENCES Airport(airport_code),
    FOREIGN KEY (airplane_id) REFERENCES Airplane(airplane_id) -- Assuming there's an
Airplane table with 'airplane_id' as PK
);

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(255)
);

CREATE TABLE Airport (
    airport_id INT PRIMARY KEY,
    airport_iata CHAR(3),
    airport_icao CHAR(4),
    airport_name VARCHAR(255),
    airport_country VARCHAR(255)
);

CREATE TABLE Airplane (
    airplane_tail_no VARCHAR(255) PRIMARY KEY,
    airplane_manufacturer VARCHAR(255),
    airplane_model VARCHAR(255),
    airplane_country_of_origin VARCHAR(255),
    capacity INT,
    airplane_owner_airline_id INT,
    FOREIGN KEY (airplane_owner_airline_id) REFERENCES Airline(airline_id)
```

```
);

CREATE TABLE Passenger (
    passenger_id INT PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    birthdate DATE,
    country_of_origin VARCHAR(255),
    passport_no VARCHAR(10)
);

CREATE TABLE PassengerFlight (
    passenger_id INT,
    airline_id INT,
    flight_no VARCHAR(10),
    travel_class ENUM('First', 'Business', 'Economy'),
    seat_no VARCHAR(5),
    on_flight BOOLEAN,
    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

In [145…
```
%%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[145]: []

In [146…
```
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[146]: 246

In [147…
```
%%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[147]:

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 [148…
```sql
%%sql

SELECT DISTINCT customers.country
FROM customers
LEFT JOIN countries ON customers.country = countries.short_name
WHERE countries.short_name IS NULL;
```

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

Out[148]:

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 [149…

```
%%sql

UPDATE customers
SET country = TRIM(country)
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';
```

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

Out[149]: []

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 [150…

```
%%sql

ALTER TABLE customers
ADD COLUMN iso_code CHAR(3);

ALTER TABLE customers
ADD CONSTRAINT fk_customers_iso_code
FOREIGN KEY (iso_code)
REFERENCES countries(alpha_3_code);

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

ALTER TABLE customers
DROP COLUMN country;

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

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

Out[150]: []

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 [151…

```
%%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[151]:

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 [152…
```
%%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[152]: []

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 [161…
```
%%sql
```

```
UPDATE products
SET product_code_letter = LEFT(productCode, 1), -- Correct, extracts 'S'
    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 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[161]: []

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 [163…

```
%%sql

SELECT
    product_code_letter,
    product_code_scale,
    product_code_number,
    productScale,
    productName
```

```
FROM
    products
WHERE
    product_code_scale != CAST(productScale AS UNSIGNED)
ORDER BY
    productName;
```

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

Out[163]:

product_code_letter	product_code_scale	product_code_number	productScale	productName
S	24	2011	1:24	18th century schooner
S	18	3136	1:18	18th Century Vintage Horse Carriage
S	24	2841	1:24	1900s Vintage Bi-Plane
S	24	4278	1:24	1900s Vintage Tri-Plane
S	18	3140	1:18	1903 Ford Model A
S	18	4522	1:18	1904 Buick Runabout
S	18	2248	1:18	1911 Ford Town Car
S	24	3151	1:24	1912 Ford Model T Delivery Wagon
S	18	2949	1:18	1913 Ford Model T Speedster
S	18	1749	1:18	1917 Grand Touring Sedan
S	18	3320	1:18	1917 Maxwell Touring Car
S	18	2432	1:18	1926 Ford Fire Engine
S	24	1785	1:24	1928 British Royal Navy Airplane
S	32	4289	1:32	1928 Ford Phaeton Deluxe
S	18	2795	1:18	1928 Mercedes-Benz SSK
S	50	1341	1:50	1930 Buick Marquette Phaeton
S	18	4409	1:18	1932 Alfa Romeo 8C2300 Spider Sport
S	18	2325	1:18	1932 Model A Ford J-Coupe
S	18	2957	1:18	1934 Ford V8 Coupe
S	24	4258	1:24	1936 Chrysler Airflow
S	18	2625	1:18	1936 Harley Davidson El Knucklehead
S	24	3969	1:24	1936 Mercedes Benz 500k Roadster
S	18	1367	1:18	1936 Mercedes-Benz 500K Special Roadster

product_code_letter	product_code_scale	product_code_number	productScale	productName
S	24	3420	1:24	1937 Horch 930V Limousine
S	18	1342	1:18	1937 Lincoln Berline
S	24	2022	1:24	1938 Cadillac V-16 Presidential Limousine
S	18	4668	1:18	1939 Cadillac Limousine
S	24	1937	1:24	1939 Chevrolet Deluxe Coupe
S	24	3816	1:24	1940 Ford Delivery Sedan
S	18	1097	1:18	1940 Ford Pickup Truck
S	18	4600	1:18	1940s Ford truck
S	18	3856	1:18	1941 Chevrolet Special Deluxe Cabriolet
S	18	1889	1:18	1948 Porsche 356-A Roadster
S	18	3685	1:18	1948 Porsche Type 356 Roadster
S	24	2766	1:24	1949 Jaguar XK 120
S	32	3207	1:32	1950's Chicago Surface Lines Streetcar
S	10	1949	1:10	1952 Alpine Renault 1300
S	24	2887	1:24	1952 Citroen-15CV
S	32	2509	1:32	1954 Greyhound Scenicruiser
S	24	3856	1:18	1956 Porsche 356A Coupe
S	12	4473	1:12	1957 Chevy Pickup
S	18	4721	1:18	1957 Corvette Convertible
S	18	4933	1:18	1957 Ford Thunderbird
S	18	3782	1:18	1957 Vespa GS150
S	24	2840	1:24	1958 Chevy Corvette Limited Edition
S	12	1666	1:12	1958 Setra Bus

product_code_letter	product_code_scale	product_code_number	productScale	productName
S	24	2000	1:24	1960 BSA Gold Star DBD34
S	24	4620	1:18	1961 Chevrolet Impala
S	50	1514	1:50	1962 City of Detroit Streetcar
S	10	4962	1:10	1962 LanciaA Delta 16V
S	24	2300	1:24	1962 Volkswagen Microbus
S	18	2319	1:18	1964 Mercedes Tour Bus
S	18	1589	1:18	1965 Aston Martin DB5
S	24	1628	1:24	1966 Shelby Cobra 427 S/C
S	12	3380	1:12	1968 Dodge Charger
S	12	1099	1:12	1968 Ford Mustang
S	24	3191	1:24	1969 Chevrolet Camaro Z28
S	12	3148	1:18	1969 Corvair Monza
S	12	4675	1:12	1969 Dodge Charger
S	18	3278	1:18	1969 Dodge Super Bee
S	12	3891	1:12	1969 Ford Falcon
S	10	1678	1:10	1969 Harley Davidson Ultimate Chopper
S	24	1046	1:24	1970 Chevy Chevelle SS 454
S	24	1444	1:24	1970 Dodge Coronet
S	12	3990	1:12	1970 Plymouth Hemi Cuda
S	18	4027	1:18	1970 Triumph Spitfire
S	24	3371	1:24	1971 Alpine Renault 1600s
S	10	4757	1:10	1972 Alfa Romeo GTA
S	32	4485	1:32	1974 Ducati 350 Mk3 Desmo

product_code_letter	product_code_scale	product_code_number	productScale	productName
S	18	3482	1:18	1976 Ford Gran Torino
S	32	1268	1:32	1980’s GM Manhattan Express
S	18	1662	1:18	1980s Black Hawk Helicopter
S	700	2824	1:18	1982 Camaro Z28
S	24	2360	1:24	1982 Ducati 900 Monster
S	32	2206	1:32	1982 Ducati 996 R
S	24	2972	1:24	1982 Lamborghini Diablo
S	18	3233	1:18	1985 Toyota Supra
S	18	3232	1:18	1992 Ferrari 360 Spider red
S	24	4048	1:24	1992 Porsche Cayenne Turbo Silver
S	18	1129	1:18	1993 Mazda RX-7
S	18	1984	1:18	1995 Honda Civic
S	10	2016	1:10	1996 Moto Guzzi 1100i
S	32	3522	1:32	1996 Peterbilt 379 Stake Bed with Outrigger
S	32	1374	1:32	1997 BMW F650 ST
S	24	1578	1:24	1997 BMW R 1100 S
S	18	2238	1:18	1998 Chrysler Plymouth Prowler
S	18	2870	1:18	1999 Indy 500 Monte Carlo SS
S	18	3029	1:18	1999 Yamaha Speed Boat
S	12	1108	1:12	2001 Ferrari Enzo
S	24	3432	1:24	2002 Chevy Corvette
S	12	2823	1:12	2002 Suzuki XREO
S	50	4713	1:50	2002 Yamaha YZR M1

product_code_letter	product_code_scale	product_code_number	productScale	productName
S	10	4698	1:10	2003 Harley-Davidson Eagle Drag Bike
S	700	2466	1:700	America West Airlines B757-200
S	700	1691	1:700	American Airlines: B767-300
S	700	4002	1:700	American Airlines: MD-11S
S	700	2834	1:700	ATA: B757-300
S	72	1253	1:72	Boeing X-32A JSF
S	18	3259	1:18	Collectable Wooden Train
S	24	3949	1:24	Corsair F4U (Bird Cage)
S	50	1392	1:50	Diamond T620 Semi-Skirted Tanker
S	700	3167	1:72	F/A 18 Hornet 1/72
S	700	2047	1:700	HMS Bounty
S	18	2581	1:72	P-51-D Mustang
S	72	3212	1:72	Pont Yacht
S	700	1938	1:700	The Mayflower
S	700	3962	1:700	The Queen Mary
S	700	1138	1:700	The Schooner Bluenose
S	700	3505	1:700	The Titanic
S	700	2610	1:700	The USS Constitution Ship

SQL

- Use the `classicmodels` database for these questions.
- The suggestions on which tables to use are hints, not requirements.

In [164…
```
%sql use classicmodels
```

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

Out[164]: []

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 [191…
```
%%sql

select
    p.productName,
    p.productLine,
    p.productVendor,
    SUM(od.quantityOrdered * od.priceEach) AS totalRevenue
from
    products p
join
    orderdetails od ON p.productCode = od.productCode
group by
    p.productName, p.productLine, p.productVendor
having
    SUM(od.quantityOrdered * od.priceEach) > 150000
```

```
order by
    totalRevenue DESC
```

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

Out[191]:

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 [228…

```
%%sql

select
    products.productCode,
    products.productName,
```

```
    products.productVendor,
    count(distinct(customerNumber)) as customerCount
from
    orders
join
    orderdetails on orders.orderNumber = orderdetails.orderNumber
join
    products on orderdetails.productCode = products.productCode
group by
    products.productCode, products.productVendor, products.productName
order by
    customerCount desc, productCode asc
limit 10
```

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

Out[228]:

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 [267…

```
%%sql


select
    c.customerName,
    MONTH(p.paymentDate) as month,
    YEAR(p.paymentDate) as year,
    IFNULL(SUM(p.amount), 0) as monthlyExpenditure,
    c.creditLimit
from
    customers c
left join
    payments p on c.customerNumber = p.customerNumber
group by
    c.customerName, month, year, c.creditLimit
having
    monthlyExpenditure > c.creditLimit or monthlyExpenditure = 0
order by
    monthlyExpenditure desc, c.customerName
limit 10
```

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

Out[267]:

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 [250…

```
%%sql
select distinct
    p.productCode,
    p.productName,
    p.productLine,
```

```
    p.productVendor,
    p.productDescription
from
    products p
where not exists (
    select 1
    from orders o
    join orderdetails od on o.orderNumber = od.orderNumber
    join customers c on o.customerNumber = c.customerNumber
    where c.country = 'France' and od.productCode = p.productCode
)
order by
    p.productCode
```

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

Out[250]:

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 [268…

```sql
%%sql
select
    e.firstName,
    e.lastName,
    count(distinct c.customerNumber) as totalCustomers,
    count(distinct o.orderNumber) as totalCustomerOrders,
    case
        when count(distinct c.customerNumber) >= 8 and count(distinct o.orderNumber) >= 30 then 'both'
        when count(distinct c.customerNumber) >= 8 then 'customers'
        when count(distinct o.orderNumber) >= 30 then 'orders'
    end as raiseBecause
from
    employees e
left join
    customers c on e.employeeNumber = c.salesRepEmployeeNumber
left join
    orders o on c.customerNumber = o.customerNumber
group by
    e.firstName, e.lastName
having
    raiseBecause is not null
order by
    e.firstName
```

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

Out[268]:

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