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 [222…
```
%load_ext sql
%sql mysql+pymysql://root:dbuserbdbuser@localhost
```

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

In [223…
```
import pandas
from sqlalchemy import create_engine
engine = create_engine("mysql+pymysql://root:dbuserbdbuser@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: it is highly organized and easily searchable by simple, straigtforward search engine algorithms or other search operations; it typically follows a tabular format with rows adn columns. A database table containing employee information is a good example, where each row represents an employee and columns represents attributes such as nmae, ID and salary.

Semi-structured data: it is not as rigid as structured data but still contains tags or other markerss to separate semantic elements and enforce hierarchies of records and fields within the data. This type of data doesn't fit neatly into a table but is not entirely without structure. JSON or XML is a good example, where data is organized into nested structures but doesn't strictluy adhere to a schema like structured data.

Unstructured data: it lacks a predefined data model or is not organized in a predefined manner, making it difficult to process and analyze using conventional databases and data models. It can be text-heavy, but it may also contain data such as dates, numbers, and facts. emails, video, audio files are good example, where there's a lot of content without a clear, predictable organization.

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: it is data that describles other data. For example, in a relational database, metadata would include information about table structures, column names, data types, and constraints. According to Codd's 0th rule, such metadata should itself be stored and managed in a relational format. This means you should be able to query metadata using the same relational operations (like select, join, etc) that you use for the actual data. Example: In a RDBMS like SQL Server, you can query system tables or use system views (like INFORMATION_SCHEMA.TABLES or sys.tables) to get information about the database structure. This makes metadata accessible and manageable using the same SQL language used for regular data, adhering to Codd's 0th rule.
2. Security: it refers to protecting data integrity, confidentiality, and accessiblity. According to Codd's 0th rule, security mechanisms, including access control, authentication, should be defined and managed using relational concepts. This means that permissions on tables, rows, and columns should be manageable using SQL commands or through relational tables that store security policies. Example: In Oracle, you can use SQL statements to manage security, such as GRANT and REVOKE, to give or take away access permissions to users and roles directly on database objects (like tables, views). Furthermore, Oracle uses data dictionary views (like DBA_TAB_PRIVS, DBA_COL_PRIVS) that store information about these permissions in a relational format, allowing for querying and managing security through its relational capabilities, thus complying with Codd's 0th rule.

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
);
```

Two views of the table that are not possible to update:

1. A view with aggregated data:

   ```
   CREATE VIEW student_summary AS
   SELECT enrollment_year, COUNT(*) AS student_count, AVG(total_credits) AS average_credits
   FROM student
   GROUP BY enrollment_year;
   ```
2. A view with joined data
   ```
   CREATE VIEW student_info AS
   SELECT CONCAT(first_name, ' ', last_name) AS full_name, total_credits
   FROM student;
   ```

A view that is possible to update:

```
CREATE VIEW student_view AS
SELECT social_security_no, last_name, first_name, enrollment_year, 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. It is not atomic because it is a composite key that contains multiple pieces of information (such as year, term, course code, and section number) and combines thme into a single string.

2. a. efficiency: Non-atomic keys can slow down index operations because the database system might need to parse each key to access individual components. b. complexity in querying: When keys are non-atomic, performing searches or queries that target only a portion of the key (e.g., all courses in a specific term) becomes more complex and potentially less efficient. c. range queries difficulty: Extracting and using parts of a non-atomic key for range queries (e.g., courses offered between certain years) is more complicated because the key must first be decomposed into its constituent parts. d. Maintenance and Scalability: Updating parts of a non-atomic key or reindexing based on new criteria can be more challenging and error-prone, as changes might affect the structure and interpretation of the key.

W5

Briefly explain the following concepts:

1. Natural join
2. Equi-join
3. Theta join
4. Left join
5. Right join
6. Outer join
7. Inner join

1. Natural join: A natural join is an operation in relational databases that combines two tables based on columns with the same name and data type. It returns all the rows from both tables where the values in the matched columns are equal.

2. Equi-Join: An equi-join is a type of join that combines two tables based on the equality of values in specified columns. It uses the equality operator (typically "=") to match rows in the tables where the values in the specified columns are equal.

3. Theta Join: A theta join is a join operation that combines tables using a condition other than equality. It employs a comparison operator (e.g., <, >, <=, >=) to define the relationship between columns in the joined tables.

4. Left join: This join returns all records from the left table and matched records from the right table. Where there is no match, the result set will have null values for columns from the right table.

5. Right join: Opposite to the left join, it returns all records from the right table and the matched records from the left table. Where there is no match, the result set will have null values for columns from the left table.

6. Outer join: It combines left, right and full outer joins. It returns all rows from both tables, with matched rows from both sides where available. Where there is no match, the result will have null values for the other side's columns.

7. Inner join: This join returns rows that have matching values in both tables involved in the join. If there is no match between the rows in the two tables, those rows are not returned in the result set.

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 in the context of foreign keys refer to automatic actions that the DBMS takes on related tables when a record in the primary key table is updated or deleted. These actions ensure referential integrity and consistency across related tables. The primary cascading actions are CASCADE, SET NULL, SET DEFAULT, and RESTRICT (or NO ACTION).

2. a. CASCADE on DELETE: If an order is deleted from the orders table, cascading the delete action would automatically remove all related orderdetails entries. This ensures that there are no orphaned order details that refer to a non-existent order. b. CASCADE on UPDATE: Although updating a primary key like orderNumber is less common, if it were to happen, cascading the update would ensure that all related entries in orderdetails are also updated, maintaining the link between orders and their details.

W7

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

1. Many-to-Many Relationships: When two entities have a many-to-many relationship, a direct foreign key implementation is not feasible. An associative entity is used to break down the many-to-many relationship into two one-to-many relationships, which can then be implemented using foreign keys.

2. Attributes on the Relationship: Sometimes, the relationship itself has attributes that need to be stored. For example, in a student-course relationship, you might need to store the grade that a student receives in a course. An associative entity is used to hold these relationship-specific attributes, as they do not belong solely to either of the entities but to the association between them.

W8

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

SQL is considered closed under its operations because the result of an SQL operation is itself a table, which can be used as the input for another SQL operation. Consider a scenario where we have a table 'employees' with columns 'id', 'name', and 'department_id', and a table departments with columns id and department_name. and we use the following operation to find the names of all employees who work in the "IT" department.

SELECT name FROM employees WHERE department_id IN (SELECT id FROM departments WHERE department_name = 'IT');

We can clearly see that the subquery (SELECT id FROM departments WHERE department_name = 'IT') produces a table of 'ids; that match the "IT" department. This result is then used as input for the query to filter 'employees' who belong to that department.

W9

Briefly explain the differences between:

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

1. Database stored procedures: Executable scripts that can perform complex operations, accept parameters, and return results. They can contain multiple SQL statements and control structures (like loops, conditions).
2. Database functions: Return a single value or a table. They're used within SQL statements and cannot perform actions that modify the database schema or data.
3. Database triggers: Automatically executed in response to specific events (e.g., insert, update, delete) on a table. Used to enforce rules or automate record updates.

W10

List three benefits/use cases for defining views.

1. Security: Views can restrict access to specific data, allowing users to see only what they need. This is useful for hiding sensitive information or complex data structures.
2. Simplification: Views can abstract and simplify complex queries, making it easier for users to query data without understanding the underlying complexities of the database schema.
3. Consistency: Views provide a consistent, unchanging interface to data, even if the underlying database schema changes, thereby insulating applications from changes in the database structure.

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'

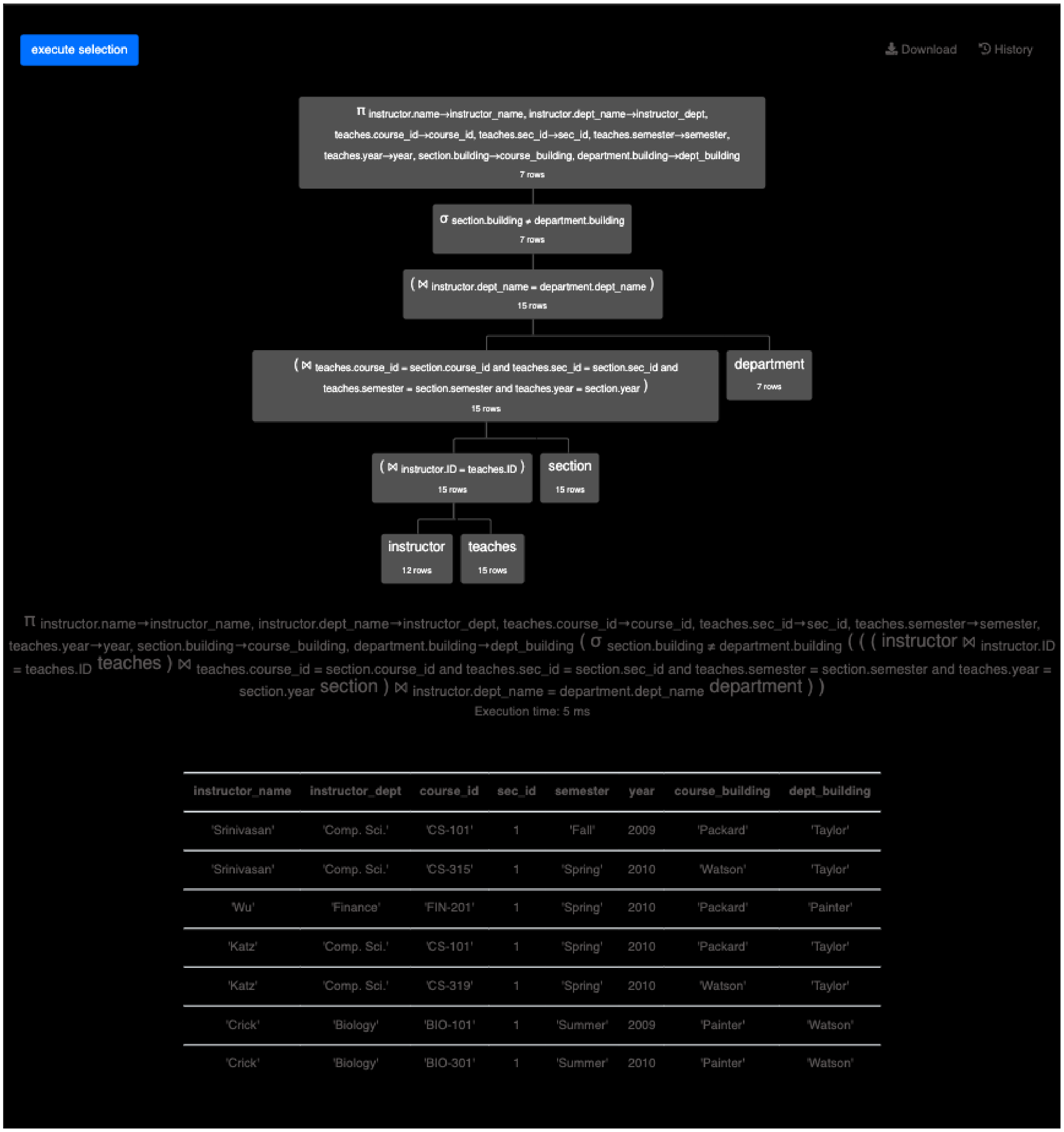
- Srinivasan taught CS-101, section 1 in Fall of 2009 in the Packard building. However, Srinivasan is in the CS department, whose building is Taylor.

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 (
σ section.building ≠ department.building (
(
    (instructor ⨝ instructor.ID = teaches.ID teaches)
    ⨝ teaches.course_id = section.course_id ∧ teaches.sec_id=section.sec_id ∧
teaches.semester=section.semester ∧ teaches.year=section.year section
) ⨝ instructor.dept_name = department.dept_name department

)
```

Execution:


execute selection
Download
History
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
7 rows
σ section.building ≠ department.building
7 rows
(⋈ instructor.dept_name = department.dept_name)
15 rows
(⋈ teaches.course_id = section.course_id and teaches.sec_id = section.sec_id and teaches.semester = section.semester and teaches.year = section.year)
15 rows
department
7 rows
(⋈ instructor.ID = teaches.ID)
15 rows
section
15 rows
instructor
12 rows
teaches
15 rows

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 (σ section.building ≠ department.building (((instructor ⋈ instructor.ID = teaches.ID teaches) ⋈ teaches.course_id = section.course_id and teaches.sec_id = section.sec_id and teaches.semester = section.semester and teaches.year = section.year section) ⋈ instructor.dept_name = department.dept_name department))

Execution time: 5 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

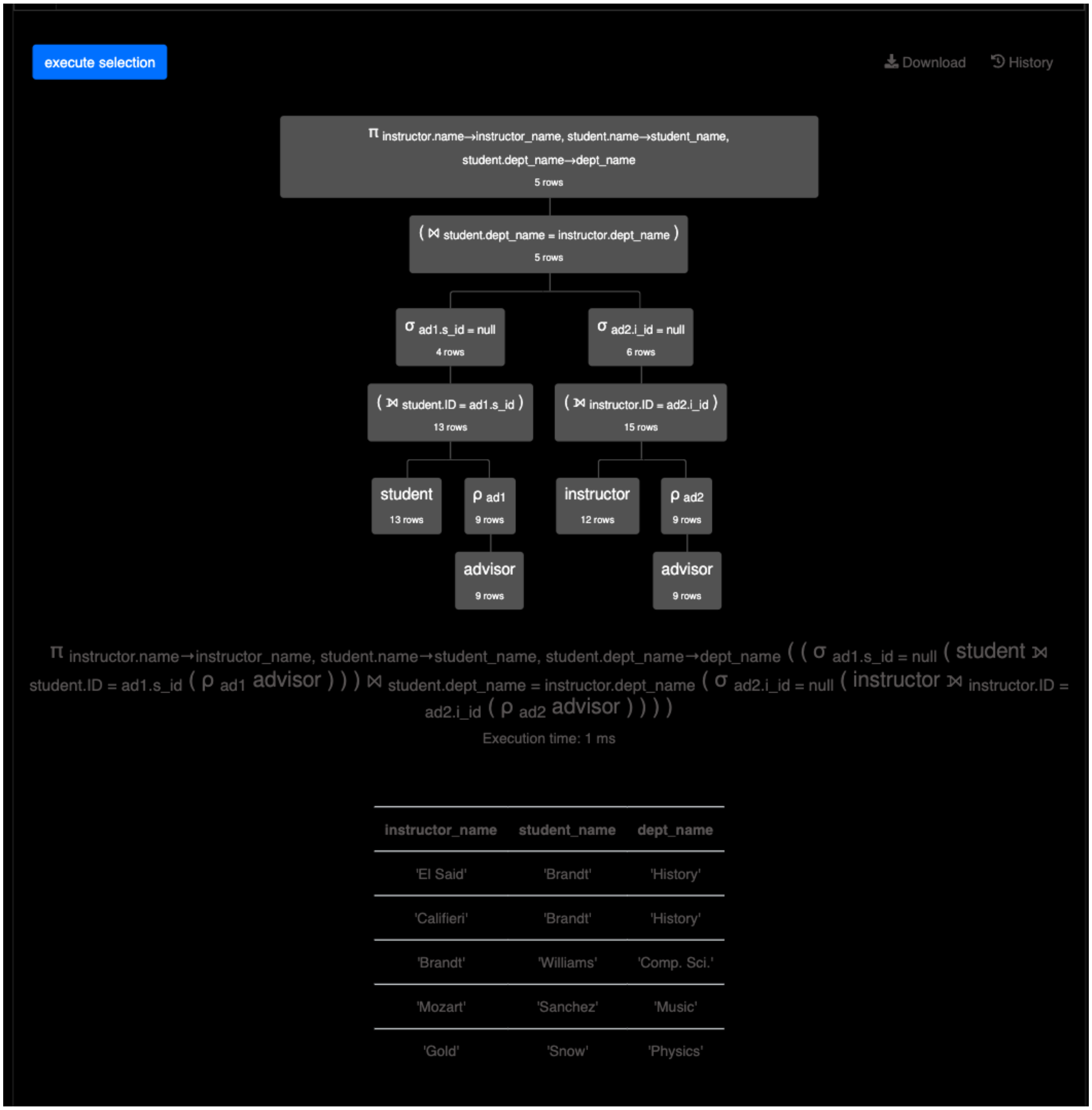
instructor_name	student_name	dept_name
'El Said'	'Brandt'	'History'

- 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:

```
π instructor_name ← instructor.name, student_name ← student.name,
dept_name←student.dept_name
(
    (σ ad1.s_id = null
        (student ⟕ student.ID = ad1.s_id (ρ ad1 advisor)
        )
    )
    ⋈ student.dept_name = instructor.dept_name
    (σ ad2.i_id = null
        (instructor ⟕ instructor.ID = ad2.i_id (ρ ad2 advisor)
        )
    )
)
```

Execution:


execute selection
Download
History
π instructor.name→instructor_name, student.name→student_name, student.dept_name→dept_name
5 rows
(⋈ student.dept_name = instructor.dept_name)
5 rows
σ ad1.s_id = null
4 rows
σ ad2.i_id = null
6 rows
(⟕ student.ID = ad1.s_id)
13 rows
(⟕ instructor.ID = ad2.i_id)
15 rows
student
13 rows
ρ ad1
9 rows
instructor
12 rows
ρ ad2
9 rows
advisor
9 rows
advisor
9 rows

π instructor.name→instructor_name, student.name→student_name, student.dept_name→dept_name ((σ ad1.s_id = null (student ⟕ student.ID = ad1.s_id (ρ ad1 advisor))) ⋈ student.dept_name = instructor.dept_name (σ ad2.i_id = null (instructor ⟕ instructor.ID = ad2.i_id (ρ ad2 advisor))))

Execution time: 1 ms

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

R1 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!

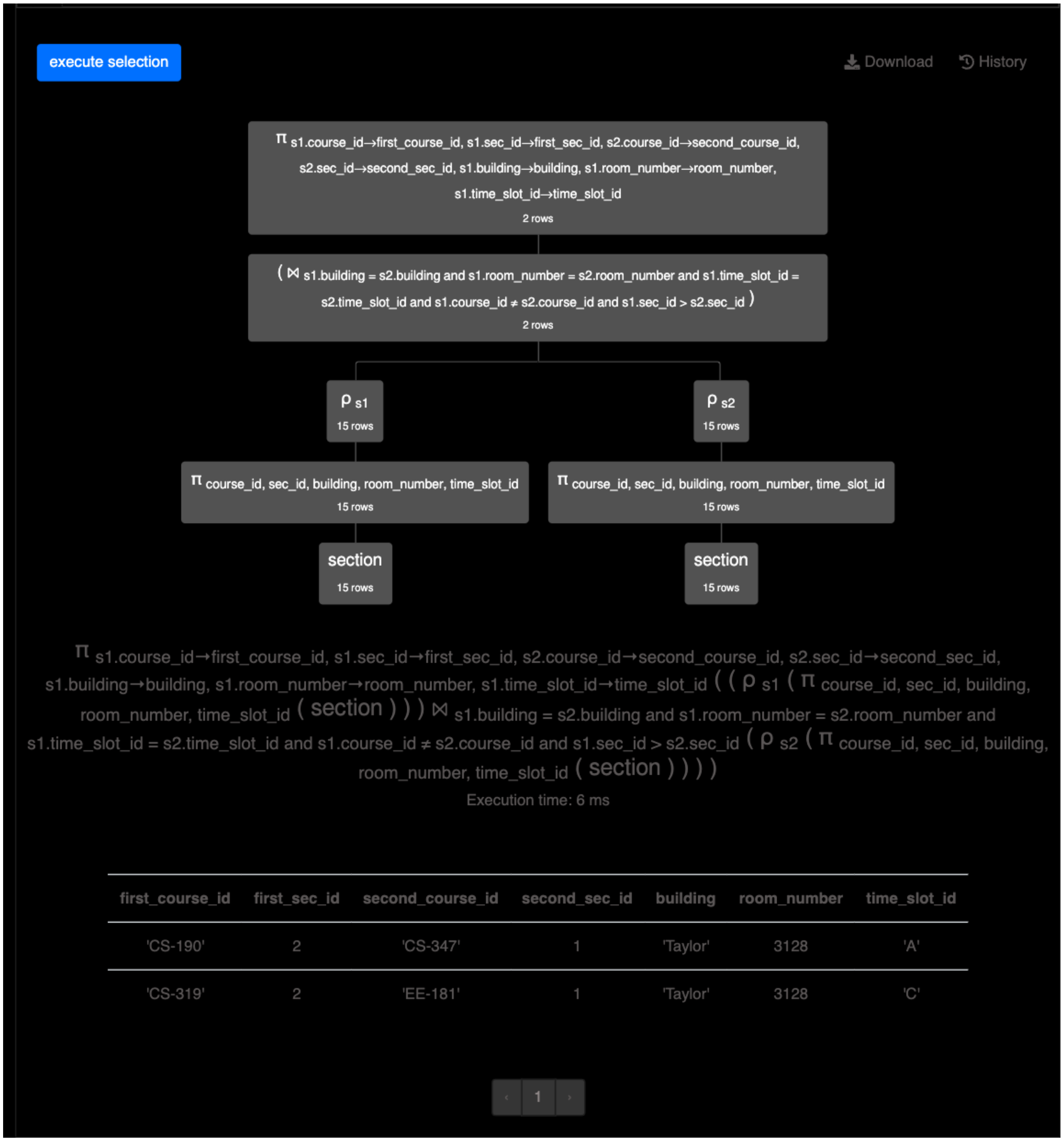
first_course_id	first_sec_id	second_course_id	second_sec_id	building	room_number	time_slot_id
'CS-190'	2	'CS-347'	1	'Taylor'	3128	'A'
'CS-319'	2	'EE-181'	1	'Taylor'	3128	'C'

- 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:

```
π first_course_id ← s1.course_id,
  first_sec_id ← s1.sec_id,
  second_course_id ← s2.course_id,
  second_sec_id ← s2.sec_id,
  building ← s1.building,
  room_number ← s1.room_number,
  time_slot_id ← s1.time_slot_id
((ρ s1 (π course_id, sec_id, building, room_number, time_slot_id
(section))) ⋈ s1.building = s2.building ∧ s1.room_number = s2.room_number ∧
s1.time_slot_id = s2.time_slot_id ∧ s1.course_id != s2.course_id ∧ s1.sec_id
> s2.sec_id (ρ s2 (π course_id, sec_id, building, room_number,
time_slot_id (section))))
```

Execution:


execute selection
Download
History
π s1.course_id→first_course_id, s1.sec_id→first_sec_id, s2.course_id→second_course_id, s2.sec_id→second_sec_id, s1.building→building, s1.room_number→room_number, s1.time_slot_id→time_slot_id
2 rows
(⋈ s1.building = s2.building and s1.room_number = s2.room_number and s1.time_slot_id = s2.time_slot_id and s1.course_id ≠ s2.course_id and s1.sec_id > s2.sec_id)
2 rows
ρ s1
15 rows
ρ s2
15 rows
π course_id, sec_id, building, room_number, time_slot_id
15 rows
π course_id, sec_id, building, room_number, time_slot_id
15 rows
section
15 rows
section
15 rows

π s1.course_id→first_course_id, s1.sec_id→first_sec_id, s2.course_id→second_course_id, s2.sec_id→second_sec_id, s1.building→building, s1.room_number→room_number, s1.time_slot_id→time_slot_id ((ρ s1 (π course_id, sec_id, building, room_number, time_slot_id (section))) ⋈ s1.building = s2.building and s1.room_number = s2.room_number and s1.time_slot_id = s2.time_slot_id and s1.course_id ≠ s2.course_id and s1.sec_id > s2.sec_id (ρ s2 (π course_id, sec_id, building, room_number, time_slot_id (section))))

Execution time: 6 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'

‹ 1 ›

R1 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.

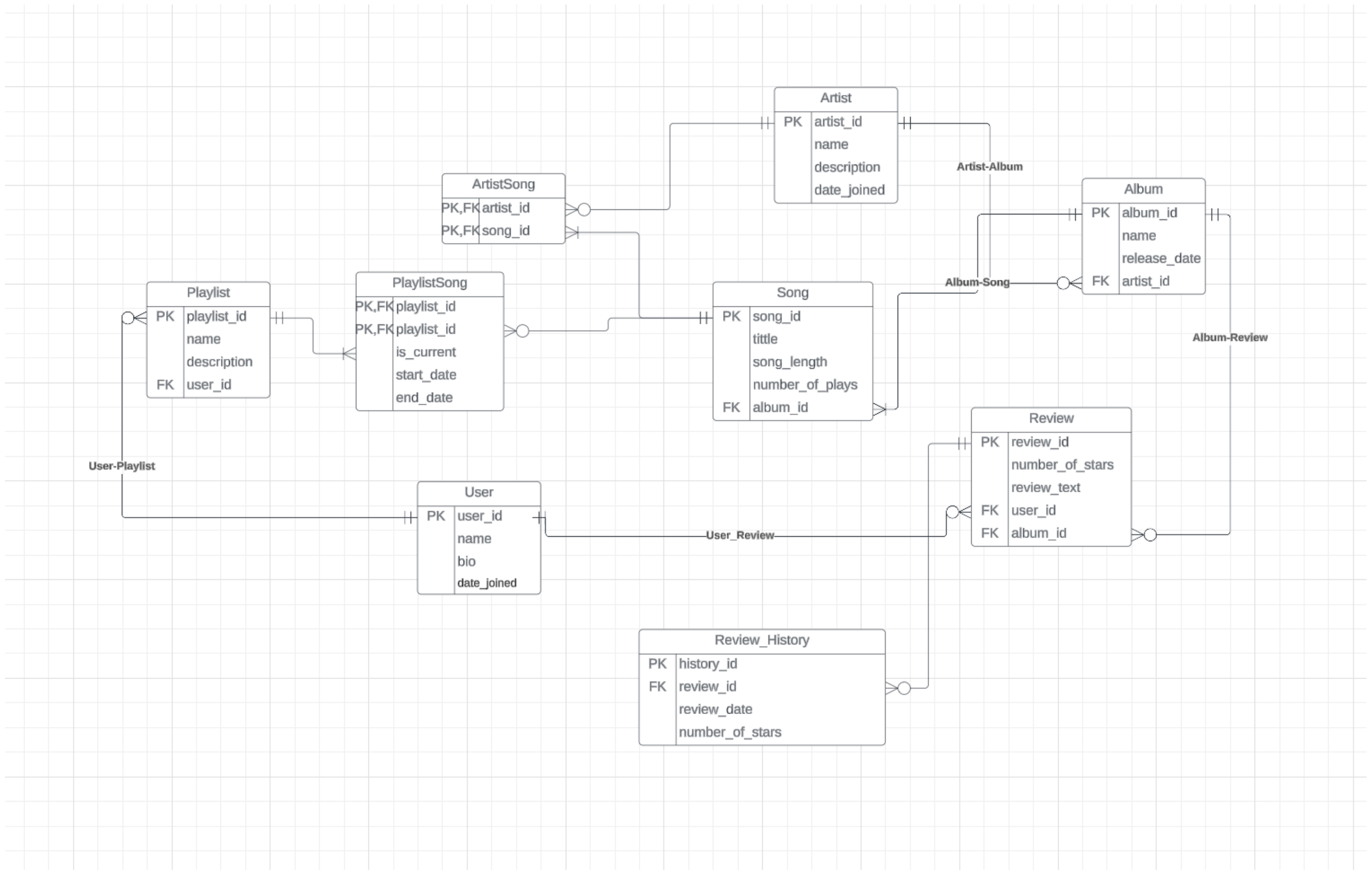
- Other requirements:
  1. You may **only** use the four Crow's Foot notations shown in class.
  2. A user can leave at most one review per album (you don't need to represent this in your diagram). However, reviews can change over time. Your model must support the ability to keep track of a user's current 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

- Add entity ArtistSong to handle many-to-many relationships between Artist and Song.
- Add entity PlaylistSong to handle many-to-many relationships between Playlist and Song. A PlaylistSong is associated with exactly one song. A song can be on any number of PlaylistSong. A PlaylistSong is associated with exactly one playlist. A playlist contains at least one Playlistsong.
- The Entity PlaylistSong includes start_date and end_date to manage the songs in a playlist over time, and is_current (boolean) to track of current songs in a playlist.
- Add entity UserReviewHistory to track of a user's current and previous reviews for an album as well as the dates for the reviews.

- The Entity UserReviewHistory includes history_id, review_date to history of a review, including changes in the review text and stars, along with the dates of those reviews. I also add number_of_stars to track how the user rate for the album.

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

- Integers are used for IDs, assuming they are unique across each entity.
- VARCHAR(255) is used for most string fields, which provides enough space for typical data.
- CHECK constraints for travel_class and crew_position to ensure only the valid values as described in the diagram are inserted.
- The start_date in CrewMember table cannot be NULL ensuring that every crew member has a start date. If a crew member is still serving, the end_date can be NULL.
- The active column in Airline is a BOOLEAN to represent if an airline is active or not.
- Data types for identifiers like IATA and ICAO codes are fixed-length characters (CHAR) as per the diagram's requirements.

```
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_id INT PRIMARY KEY,
    airplane_tail_no VARCHAR(255) NOT NULL,
    airplane_manufacturer VARCHAR(255) NOT NULL,
    airplane_model VARCHAR(255) NOT NULL,
    airplane_country_of_origin VARCHAR(255) NOT NULL,
    capacity INT NOT NULL,
    airplane_owner_airline_id INT,
    FOREIGN KEY (airplane_owner_airline_id) REFERENCES Airline(airline_id)
);

CREATE TABLE Flight (
    airline_id INT,
    flight_no INT PRIMARY KEY,
    departure_airport INT,
    arrival_airport INT,
    departure_datetime DATETIME NOT NULL,
    arrival_datetime DATETIME NOT NULL,
    airplane_id INT,
    FOREIGN KEY (airline_id) REFERENCES Airline(airline_id),
    FOREIGN KEY (departure_airport) REFERENCES Airport(airport_id),
    FOREIGN KEY (arrival_airport) REFERENCES Airport(airport_id),
    FOREIGN KEY (airplane_id) REFERENCES Airplane(airplane_id)
);

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(255) NOT NULL
);

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 NOT NULL,
    country_of_origin VARCHAR(255) NOT NULL,
    passport_no VARCHAR(255) NOT NULL,
    start_date DATE NOT NULL,
    end_date DATE,
    FOREIGN KEY (employer_airline_id) REFERENCES Airline(airline_id)
);

CREATE TABLE PassengerFlight (
    passenger_id INT,
    airline_id INT,
    flight_no INT,
    travel_class VARCHAR(50) NOT NULL CHECK (travel_class IN ('First', 'Business',
'Economy')),
    seat_no VARCHAR(10) NOT NULL,
    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)
);

CREATE TABLE CrewMemberFlight (
    pilot_id INT,
    airline_id INT,
    flight_no INT,
    crew_position VARCHAR(50) NOT NULL CHECK (crew_position IN ('pilot', 'copilot',
'flight engineer', '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)
);
```

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 [255…

```
%%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[255]:

```
 []
```

In [256…

```
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[256]:

```
 246
```

In [257…

```
%%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[257]:

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 [258…

```
%%sql
SELECT DISTINCT c.country
FROM customers c
LEFT JOIN countries co ON c.country = co.short_name
WHERE co.short_name IS NULL
```

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

Out[258]:

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 [259…

```
%%sql
-- Correcting 'USA' to 'United States'
UPDATE customers
SET country = 'United States'
WHERE country = 'USA';

-- Correcting 'UK' to 'United Kingdom'
UPDATE customers
SET country = 'United Kingdom'
WHERE country = 'UK';

-- Correcting 'Russia' to 'Russian Federation'
UPDATE customers
SET country = 'Russian Federation'
WHERE country = 'Russia';

-- Correcting 'Norway ' to 'Norway'
UPDATE customers
SET country = 'Norway'
WHERE country = 'Norway  ';
```

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

Out[259]:

```
 []
```

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`?

When we define a column as a foreign key in most RDBMS, the system automatically creates an index on that column if one does not already exist. In this case, when we added iso_code as a foreign key in the customers table referencing countries.alpha_3_code, the database likely created an index on customers.iso_code at that moment.

In [260…

```
%%sql

-- 1
ALTER TABLE customers
ADD COLUMN iso_code CHAR(3);

-- 2
ALTER TABLE customers
```

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

-- 3
UPDATE customers c
JOIN countries ct ON c.country = ct.short_name
SET c.iso_code = ct.alpha_3_code;

-- 4
ALTER TABLE customers
DROP COLUMN country;

-- 5
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[260]:

```
[]
```

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 [261…

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

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

Out[261]:

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 [262…

```
%%sql
ALTER TABLE products
ADD COLUMN product_code_letter CHAR(1);

ALTER TABLE products
ADD COLUMN product_code_scale INT;

ALTER TABLE products
ADD COLUMN product_code_number INT;
```

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

Out[262]: []

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 [263…

```
%%sql


UPDATE products

SET product_code_letter = LEFT(productCode, 1),
    product_code_scale = SUBSTRING(productCode, 2, LOCATE('_', productCode)-2),
    product_code_number = SUBSTRING(productCode, LOCATE('_', productCode) +1);

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[263]: []

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 [264…
```
%%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[264]:

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 [265…
```
%sql use classicmodels
```

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

Out[265]:
```
[]
```

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 [266…
```
%%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[266]:

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 [267…

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

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

Out[267]:

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 [268…

```
%%sql
SELECT
    c.customerName,
    IFNULL(MONTH(p.paymentDate), 'NULL') AS month,
    IFNULL(YEAR(p.paymentDate), 'NULL') AS year,
    COALESCE(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[268]:

customerName	month	year	monthlyExpenditure	creditLimit
Dragon Souveniers, Ltd.	12	2003	105743.00	103800.00
American Souvenirs Inc	NULL	NULL	0.00	0.00
ANG Resellers	NULL	NULL	0.00	0.00
Anton Designs, Ltd.	NULL	NULL	0.00	0.00
Asian Shopping Network, Co	NULL	NULL	0.00	0.00
Asian Treasures, Inc.	NULL	NULL	0.00	0.00
BG&E Collectables	NULL	NULL	0.00	0.00
Cramer Spezialitäten, Ltd	NULL	NULL	0.00	0.00
Der Hund Imports	NULL	NULL	0.00	0.00
Feuer Online Stores, Inc	NULL	NULL	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 [269…
```
%%sql


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

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

Out[269]:

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 [270…
```
%%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
JOIN
    customers c ON e.employeeNumber = c.salesRepEmployeeNumber
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[270]:

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