COMS W4111: Introduction to Databases
Spring 2024, Sections 002/V02

Homework 2: Common

Introduction

This notebook contains HW2 Common. **Students on both tracks should complete this part.** To ensure everything runs as expected, work on this notebook in Jupyter.

Submission instructions:

- You will submit **a PDF** for this assignment
  - 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.

Written Questions

W1

Explain Codd's 3rd Rule.

- What are some interpretations of a NULL value?
- An alternative to using NULL is some other value for indicating missing data, e.g., using -1 for the value of a weight

column. Explain the benefits of NULL relative to other approaches.

Codd's third rule is that all the NULL values in a database should be treated the same.

- In a database NULL can mean that data is missing, data is not known, or data is not applicable.
- NULL gets propogated in calculations, whereas some column-specific invalid value is not reflected in aggregations. For example, if you wanted the sum of a column of weights, and the "null" value was -1, the sum would incorrectly treat -1 as an actual negative weight, which is nonsensical. Using NULL, the sum would be NULL, too, which is appropriate.

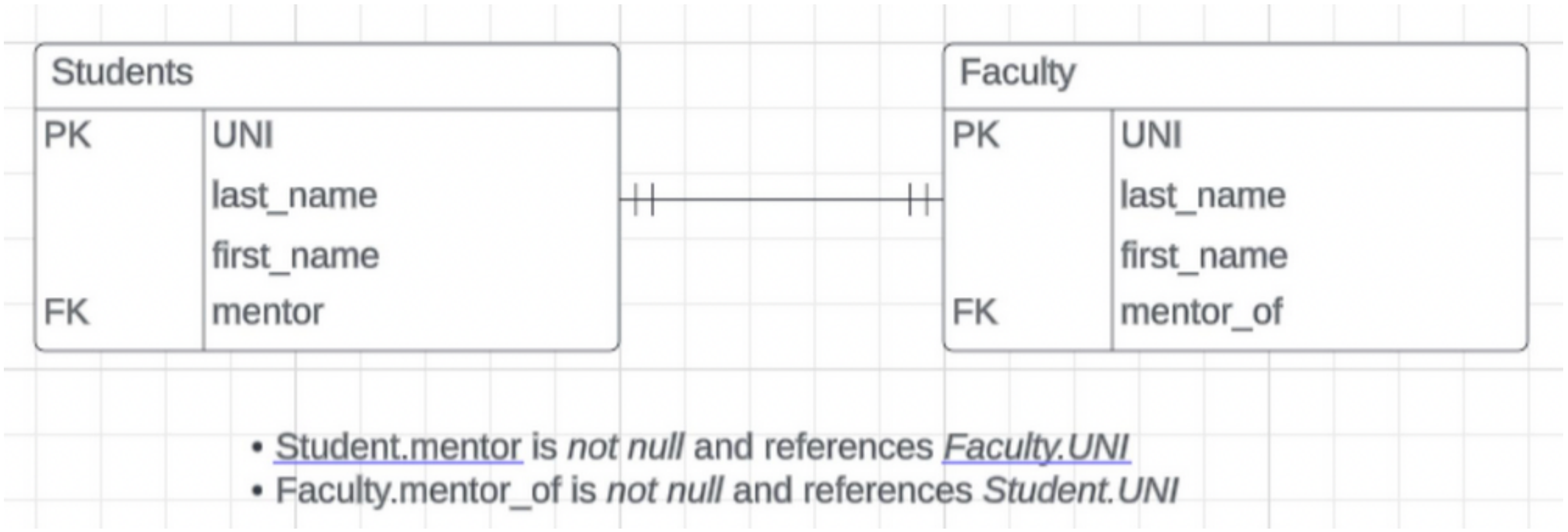
W2

Briefly explain the following concepts:

1. Primary key
2. Candidate key
3. Super key
4. Alternate key
5. Composite key
6. Unique key
7. Foreign key

1. Primary key: an attribute (or set of attributes) of an entity that is unique and can be used to identify that entity specifically. There can only be one primary key in a table, and the database designer chooses it.
2. Candidate key: a minimal set of attributes that uniquely identify a row/entity (subset of super keys). The values of a candidate key cannot be null.
3. Super key: a set of keys that together idenify an entity. Does not have to be minimal.
4. Alternate key: a candidate key that was not chosen to be the primary key.
5. Composite key: A type of key that is a combination of many columns/attributes. Can combine both candidate keys and the primary key
6. Unique key: Set of one or more values that can be used to identify a table row. Can be non-minimal.
7. Foreign key: a common key in two database tables.

W3


Students
PK
UNI
last_name
first_name
FK
mentor
Faculty
PK
UNI
last_name
first_name
FK
mentor_of
• Student.mentor is *not null* and references *Faculty.UNI*
• Faculty.mentor_of is *not null* and references *Student.UNI*

Consider the logical data model above. The one-to-one relationship is modeled using two foreign keys, one in each table.

- Why does this make it difficult to insert data into the tables?
- What is a (simple) fix for this, i.e., how would you model a one-to-one relationship?

Having two foreign keys to represent a relationship means that when inserting data, you have to address a "chicken-and-egg" problem: To insert a record in Students, you would need a record in Faculty, and vice-versa.

You can fix this by making the primary key of Student non-minimal (UNI, mentor) and using this composite primary key as a foreign key in Faculty. This way you would know to always insert the student before the mentor.

```
In [ ]:
```

W4

The relational model places restrictions on attributes. Many data scenarios have more complex types of attributes. Briefly explain the following types of attributes:

1. Simple attribute
2. Composite attribute
3. Derived attribute
4. Single-value attribute
5. Multi-value attribute

1. Simple attribute: an atomic, indivisible attribute representing a single value or entity.
2. Composite attribute: a collection of simple attributes that mean something all together.
3. Derived attribute: an attribute that depends on/is calculated from another attribute.
4. Single-value attribute: an attribute that holds a single value per instance of the entity.
5. Multi-value attribute: an attribute that can hold multiple values per each instance of an entity.

W5

The slides associated with the recommended textbook list six basic relational operators:

1. select: σ
2. project: π
3. union: ∪
4. set difference: -
5. Cartesian product: ×
6. rename: ρ

The list does not include join: ⋈. This is because it is possible to derive join using more basic operators. Explain how to derive join from the basic operators.

Take the cartesian product of the two tables, and then select the rows where the attributes match each other. Example from lecture:

σ instructor.ID=teaches.ID (instructor × teaches) = instructor ⋈ teaches

W6

Explain how using a natural join may produce an incorrect/unexpected answer.

If we have two attributes with the same name that are technically unrelated, natural join will combine the rows as if they are related. It only looks at the names of the attributes.

W7

The UNION and JOIN operations both combine two tables. Describe their differences.

UNION is used to combine rows from two or more result sets vertically, while JOIN is used to combine columns from two or more tables horizontally based on a specified condition. There are many types of JOINs and only one type of UNION.

W8

Briefly explain the importance of integrity constraints. Why do non-atomic attributes cause problems/difficulties for integrity constraints?

Integrity constraints make it so that nonsensical changes to the database can't be made. Having atomic attributes is itself a kind of inherent integrity constraint; at least, having only atomic attributes makes it so we can be more specific about the integrity constraints of each attribute. As an example, if we had "date" represented as 3 fields rather than one (month, day, year), we would be able to say that $1 \leq month \leq 12$, $1 \leq day \leq 31$, and $0 \leq year \leq 2023$.

W9

What is the primary reason for creating indexes? What are the negative effects of creating unnecessary indexes?

We create indexes when we reference a certain section of the data a lot, to save runtime on this common query by avoiding scanning through the whole database. Unecessary indexes use more disk space and create maintainance overhead, because adding to the database requires a check that the inserted record should be accessible via the index.

W10

Consider the table `time_slot` from the sample database associated with the recommended textbook.

- The data type for the column `day` is `char(1)`. Given the data types MySQL supports, what is a better data type for `day`?
- What is a scenario that would motivate creating an index on `day`?

A better type for `day` would be `ENUM`, which restricts the domain of `day` to the 5 characters representing weekdays. This change would maintain data integrity by not allowing nonsensical day values!

We might need an index on `day` if we wanted to view the timeslots grouped by the day.

Relational Algebra

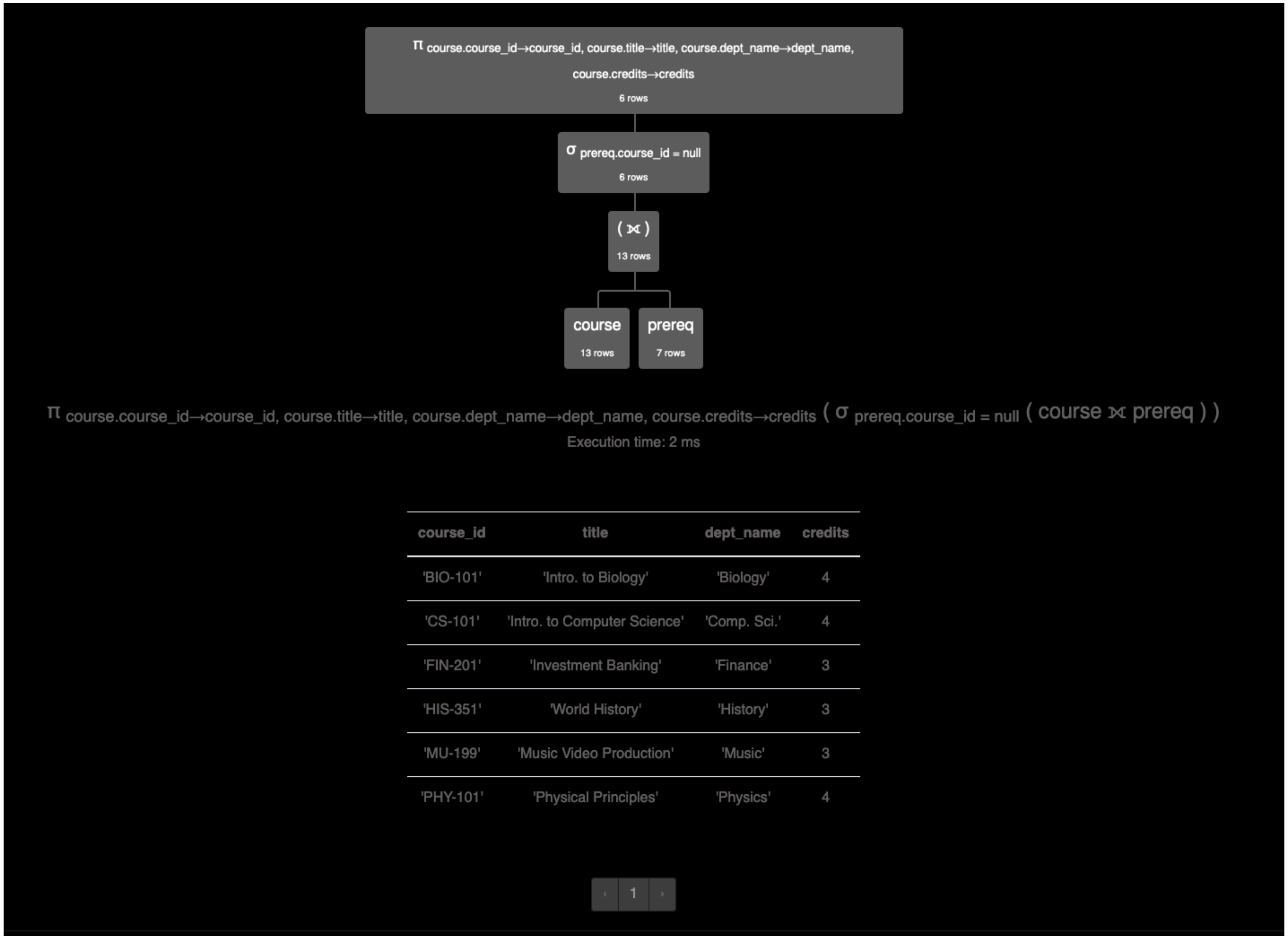
R1

- Write a relational algebra statement that produces a relation showing **courses that do not have a prereq**
- Your output should have the following columns (names should match exactly; there should be no prefixes):
  - `course_id`
  - `title`
  - `dept_name`
  - `credits`
- You may not use the anti-join: $\triangleright$ operator
- You should use the `course` and `prereq` tables

Algebra statement:

```
π course_id ← course.course_id, title ← course.title, dept_name ← course.dept_name,
credits←course.credits (
σ prereq.course_id = null(course ⟗ prereq)
)
```

Execution:


π course.course_id→course_id, course.title→title, course.dept_name→dept_name, course.credits→credits
6 rows
σ prereq.course_id = null
6 rows
(⋉)
13 rows
course
13 rows
prereq
7 rows

$\pi_{\text{course.course\_id}\to\text{course\_id, course.title}\to\text{title, course.dept\_name}\to\text{dept\_name, course.credits}\to\text{credits}} (\sigma_{\text{prereq.course\_id = null}} (\text{course} ⟕ \text{prereq}))$

Execution time: 2 ms

course_id	title	dept_name	credits
'BIO-101'	'Intro. to Biology'	'Biology'	4
'CS-101'	'Intro. to Computer Science'	'Comp. Sci.'	4
'FIN-201'	'Investment Banking'	'Finance'	3
'HIS-351'	'World History'	'History'	3
'MU-199'	'Music Video Production'	'Music'	3
'PHY-101'	'Physical Principles'	'Physics'	4

R1 Execution Result

R2

- Write a relational algebra query that produces a relation showing **students who have taken sections taught by their advisors**
  - 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):
  - `student_name`
  - `instructor_name`
  - `course_id`
  - `sec_id`
  - `semester`
  - `year`
  - `grade`
- You should use the `takes`, `teaches`, `advisor`, `student`, and `instructor` tables
- As an example, one row you should get is

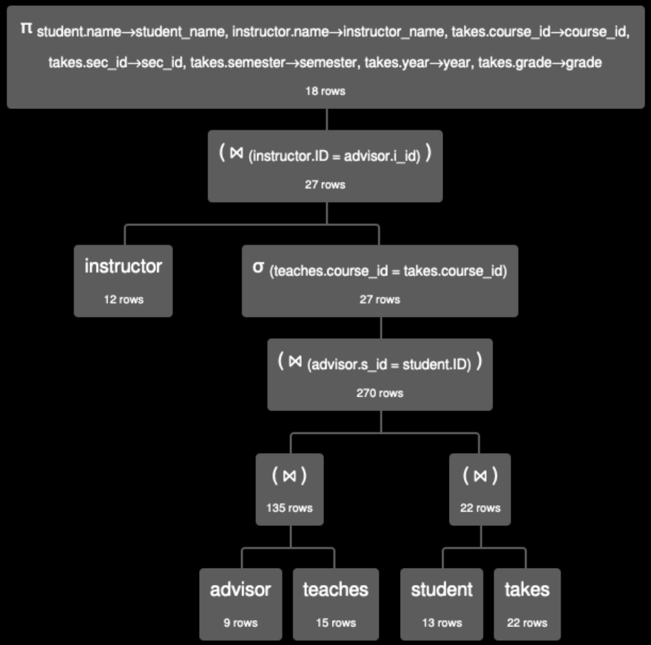
student_name	instructor_name	course_id	sec_id	semester	year	grade
'Shankar'	'Srinivasan'	'CS-101'	1	'Fall'	2009	'C'

- Shankar took CS-101, section 1 in Fall of 2009, which was taught by Srinivasan. Additionally, Srinivasan advises Shankar

Algebra statement:

```
π student_name ← student.name, instructor_name ← instructor.name, course_id ←
takes.course_id, sec_id ← takes.sec_id, semester ← takes.semester, year←takes.year, grade
← takes.grade (
    instructor join (instructor.ID = advisor.i_id) (σ  (teaches.course_id =
takes.course_id) ((advisor join teaches) join (advisor.s_id = student.ID) (student join
takes)))
)
```

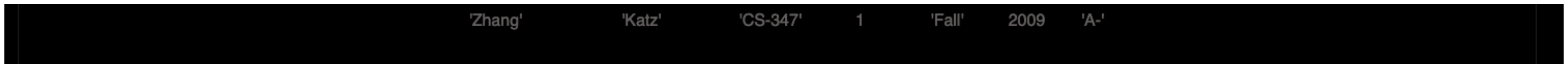
Execution:


π student.name→student_name, instructor.name→instructor_name, takes.course_id→course_id, takes.sec_id→sec_id, takes.semester→semester, takes.year→year, takes.grade→grade
18 rows
(⋈ (instructor.ID = advisor.i_id))
27 rows
instructor
12 rows
σ (teaches.course_id = takes.course_id)
27 rows
(⋈ (advisor.s_id = student.ID))
270 rows
(⋈)
135 rows
(⋈)
22 rows
advisor
9 rows
teaches
15 rows
student
13 rows
takes
22 rows

$\pi_{\text{student.name→student\_name, instructor.name→instructor\_name, takes.course\_id→course\_id, takes.sec\_id→sec\_id, takes.semester→semester, takes.year→year, takes.grade→grade}}$ (instructor $\bowtie_{\text{(instructor.ID = advisor.i\_id)}}$ ($\sigma_{\text{(teaches.course\_id = takes.course\_id)}}$ ((advisor $\bowtie$ teaches) $\bowtie_{\text{(advisor.s\_id = student.ID)}}$ (student $\bowtie$ takes))))

Execution time: 2 ms

student_name	instructor_name	course_id	sec_id	semester	year	grade
'Shankar'	'Srinivasan'	'CS-101'	1	'Fall'	2009	'C'
'Shankar'	'Srinivasan'	'CS-315'	1	'Spring'	2010	'A'
'Shankar'	'Srinivasan'	'CS-347'	1	'Fall'	2009	'A'
'Shankar'	'Srinivasan'	'CS-190'	2	'Spring'	2009	'A'
'Peltier'	'Einstein'	'PHY-101'	1	'Fall'	2009	'B-'
'Levy'	'Einstein'	'CS-101'	1	'Fall'	2009	'F'
'Levy'	'Einstein'	'CS-101'	1	'Spring'	2010	'B+'
'Levy'	'Einstein'	'CS-319'	1	'Spring'	2010	'B'
'Zhang'	'Katz'	'CS-101'	1	'Fall'	2009	'A'



```
'Zhang'     'Katz'     'CS-347'     1     'Fall'     2009     'A-'
```

R2 Execution Result (there are more pages of results, too)

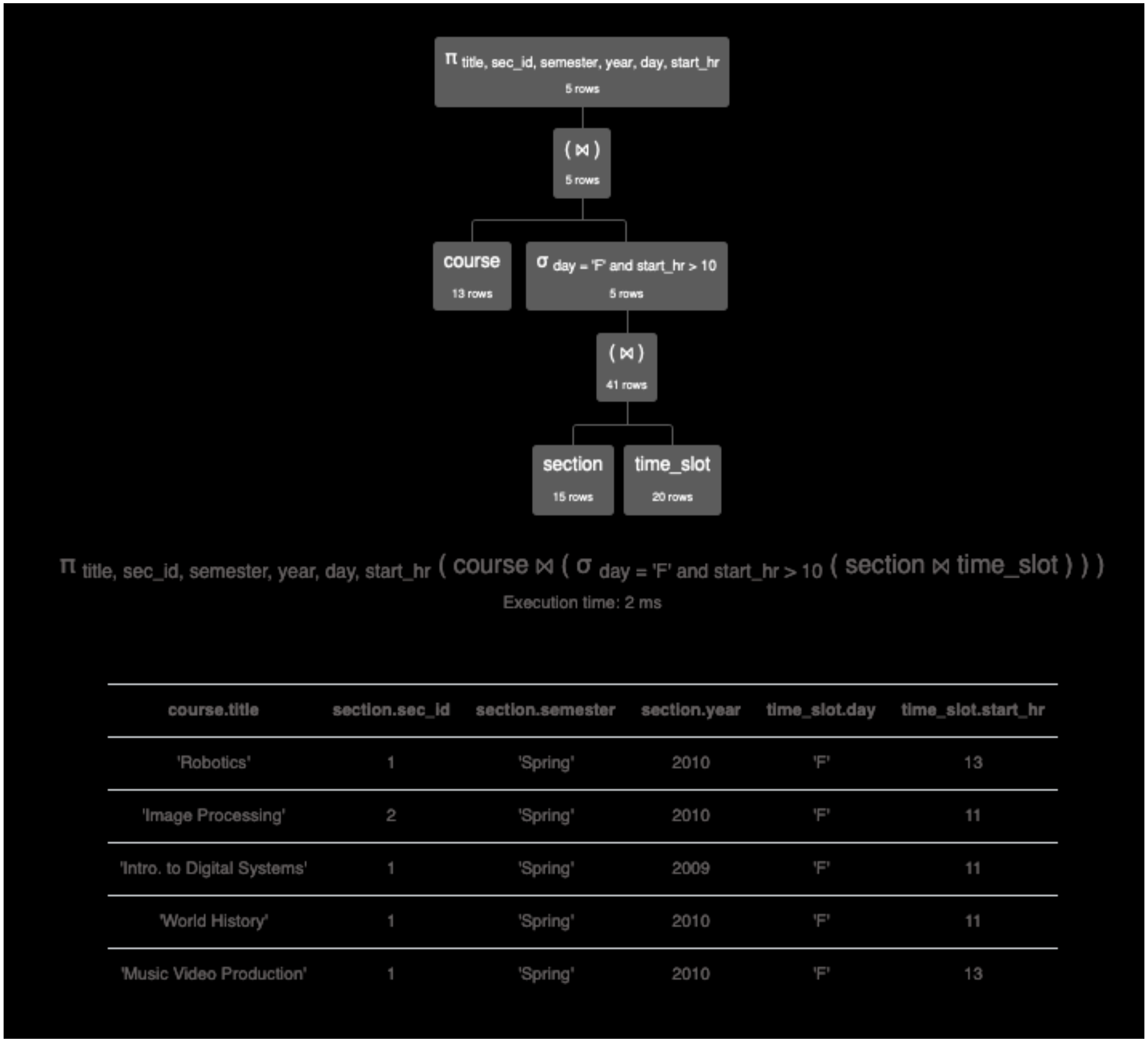
R3

- Write a relational algebra query that produces a relation showing **sections that occur on Friday and start after 10 AM**
- Your output should have the following columns (names should match exactly; there should be no prefixes):
  - `course_title`
  - `sec_id`
  - `semester`
  - `year`
  - `day`
  - `start_hr`
- You should use the `course` , `section` , and `time_slot` tables

Algebra statement:

```
π title, sec_id, semester, year, day, start_hr ( course join (sigma day='F' and
start_hr>10 (section join time_slot)) )
```

Execution:



$\pi_{\text{title, sec\_id, semester, year, day, start\_hr}}$ (course ⋈ ($\sigma_{\text{day = 'F' and start\_hr > 10}}$ (section ⋈ time_slot)))

Execution time: 2 ms

course.title	section.sec_id	section.semester	section.year	time_slot.day	time_slot.start_hr
'Robotics'	1	'Spring'	2010	'F'	13
'Image Processing'	2	'Spring'	2010	'F'	11
'Intro. to Digital Systems'	1	'Spring'	2009	'F'	11
'World History'	1	'Spring'	2010	'F'	11
'Music Video Production'	1	'Spring'	2010	'F'	13

R3 Execution Result

```
In [ ]:
```