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

Arman Ozcan, ao2794

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.

This rule states that a fully relational DBMS must allow each field to be null, which is a way for representing missing or inapplicable information in a systematic way, distinct from an empty string or a number like -1 or 0. There are different ways to interpret a NULL value, such as a value that is missing, unknown, not applicable, or just a placeholder to be filled later. Unlike other approaches, NULL values maintain standardization across different database systems. They are also automatically excluded from statistical calculations unlike other values such as -1, and this provides more accurate calculations. They also provide semantic clarity and do not require additional context or explanation as another value like -1 would.

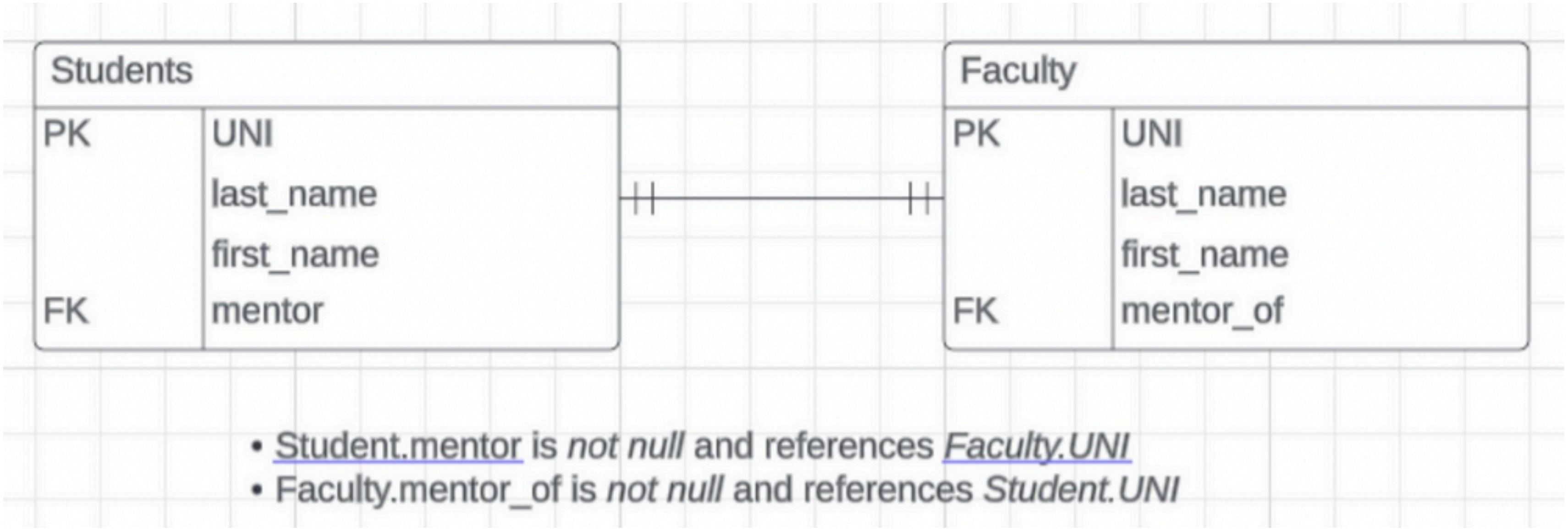
W2

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: A candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation.
2. Candidate key: Minimal super key. Technically speaking, superkey for which no proper subset is a superkey.
3. Super key: A set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation.
4. Alternate key: An alternate key is any candidate key that is not chosen to be the primary key.
5. Composite key: A key that consists of two or more columns in a table that are used together to uniquely identify a specific tuple, which are used when no single column can uniquely identify records within the table.
6. Unique key: A key that ensures all values in a column or a set of columns are unique across all records in a table. However, unlike primary keys, unique keys can accept null values, but only one null value if the unique constraint is applied to a single column.
7. Foreign key: A key that uniquely identifies a row of another table by referencing to the primary key of that another table. They are used to establish and enforce a link between the data in two 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?

Using two foreign keys, one in each table, leads to circular dependency; that is, one cannot insert an entry into Students without first inserting some entry into Faculty and vice versa due to foreign key constraint. A simple fix would be to only use one foreign key in one table that we designate as the dependent table, which uniquely references the primary table through its primary key. For example, in this case, we can model the one-to-one relationship by removing the Foreign Key on Faculty and just keep the Foreign Key on Student to link each student to a unique faculty member mentor.

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: Attributes that are not divided into subparts and instead is made as a single block. For example, "student_ID" could be a simple attribute that cannot be further divided.
2. Composite attribute: Attributes that can be divided into subparts (i.e., other attributes). To give an example, an attribute "name" could be structured as a composite attribute consisting of "first name", "middle initial", and "last name".
3. Derived attribute: Attributes whose value can be derived from the values of other related attributes or entities. For example, if there are two attributes "date of birth" and "age", the attribute "age" can be a derived attributed as it can be calculated using "date of birth" and the current date. In this case "date of birth" would be referred to as a base attribute, or a stored attribute, whereas the value of the derived attribute "age" would not be stored but be computed when required.
4. Single-value attribute: Attributes that only take a single value for each entity. For example, the "teacher_ID" attribute for a specific teacher entity refers to only one teacher ID.
5. Multi-value attribute: Attributes that take a set of values (zero, one, or more) for each entity. For example, the "email_address" attribute for a specific teacher entity may refer to multiple email addresses.

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.

The natural join operator joins two tables, producing a table that contains all Cartesian-product rows where the common column names have the same value. We can derive natural join by first using Cartesian product to take all combinations of pairs of rows from both tables. Then, we will use select to only take the rows that have matching values for the common (same-named) columns. (In practice, these same-named columns would be primary key of one table which another table references through foreign key.) Lastly, we will use project to only take one of the each same-named columns (as well as taking all the other different-named columns); in other words, we get rid of the repeated columns (same-named) by project operation.

W6

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

The natural join operator joins two tables, producing a table that contains all Cartesian-product rows where the common column names have the same value. So, the join condition is only based on the name of the attribute, which may be problematic as it does not take into account any foreign key relationship. If the two tables that are natural joined have columns with the same name but different meanings or data types, the result will be unexpected and incorrect. An example would be the attribute "phone number" for two different tables: customer and company. The attribute name for these columns are the same but they mean different things and have different functions, where the "phone number" for customer is a personal phone number and the "phone number" for company is the company phone number. Natural joining these two tables would be incorrect as it would only take the rows with same phone number, despite these phone numbers referring to different things, which would lead to data loss of different valued rows of phone numbers.

W7

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

JOIN operations are used to combine rows from two or more tables based on a matched join condition between them. UNION operation, on the other hand, combines two or more tables which have the same number and order of columns with the same data type in each column by stacking the rows together. A JOIN combines result sets horizontally, a UNION appends result set vertically, In other words, JOIN operations leads to additional number of columns by merging rows, whereas UNION operation leads to additional number of rows with same columns.

W8

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

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency. Therefore, integrity constraints guard against unintended damage to the database. Examples include constraints on single relations, such as not null or unique constraints, or referential integrity constraints for foreign keys. Non-atomic attributes in databases violate the principle of atomicity, leading to complexities in enforcing data integrity and uniqueness since they contain multiple values within a single field. This complicates query operations, making data manipulation, updates, and retrievals more difficult and less efficient.

W9

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

Many database queries target a limited subset of records within a table, rather than the entire dataset. Reading through every record to locate one with a specific value is a resource-intensive process. By implementing an index on a table's attribute, a database system can quickly identify tuples in the table that match a given attribute value, eliminating the need to examine each tuple in the table comprehensively. An index would be considered unnecessary if it is not often used by a query or it is redundant because some other compound index already covers it. Creating unnecessary indexes in a database leads to increased storage requirements, as each index occupies additional space. These surplus indexes slow down data modification operations (inserts, updates, deletes) due to the overhead of maintaining the indexes in sync with the table data.

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`?

Only using one character for a day, though efficient in space, is difficult to read, especially considering that both Tuesday and Thursday start with the same letter "T", an issue the sample database solved by denoting Thursday as "R". A better data type for day may be ENUM where we will define a list of acceptable string values for day, from "MONDAY" to "SUNDAY". This makes the data more readable and restricts the values to the specified set of days, providing better data integrity and making the data more understandable than single-character codes, which can be potentially any character.

Creating an index on the day column would be helpful in cases where queries frequently filter or group data based on the day of the week. For instance, if an application often needs to retrieve records for courses scheduled on specific days, such as all courses happening on Fridays, indexing the day column would significantly speed up these operations.

Relational Algebra

R1

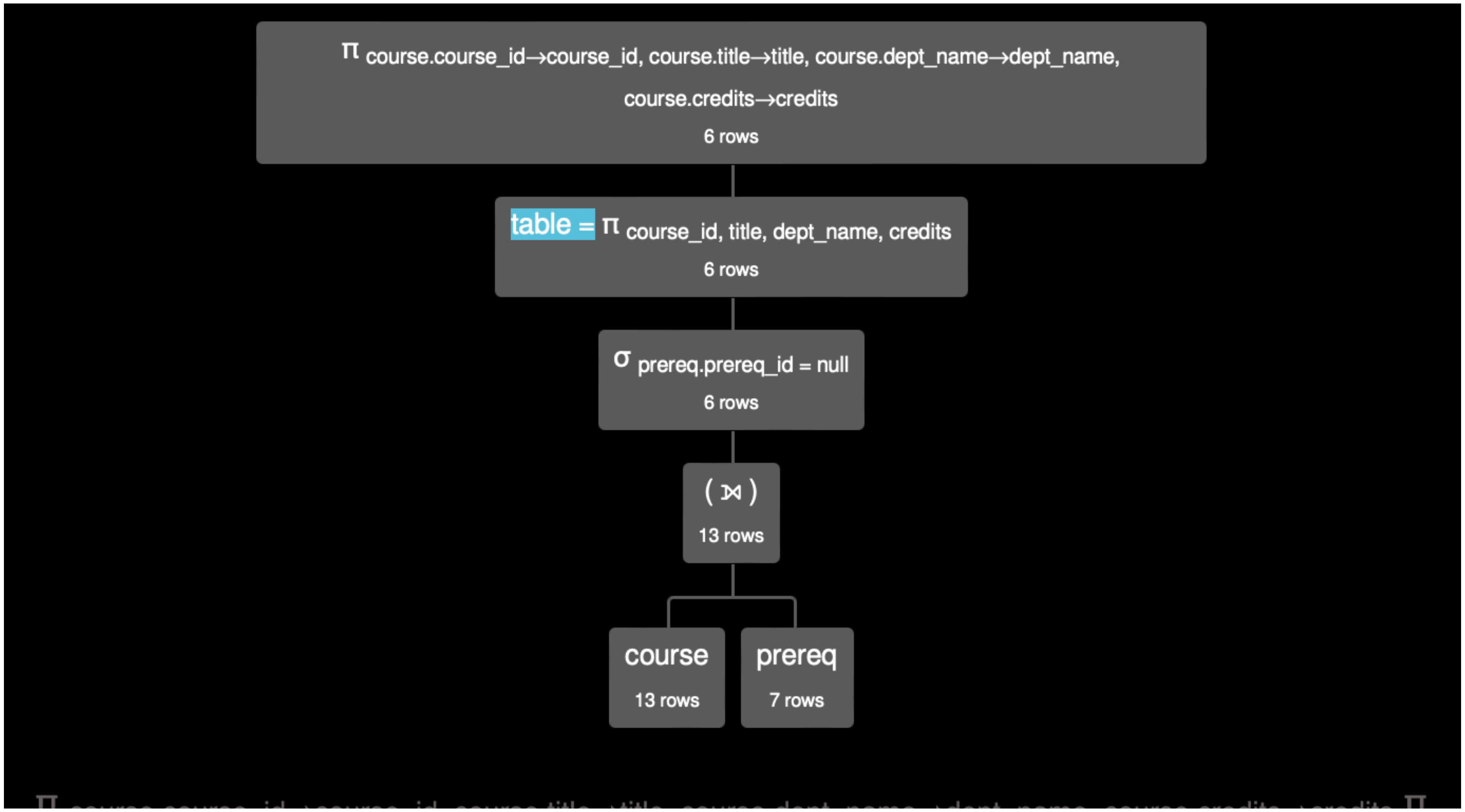
- Write a relational algebra statement that produces a relation showing **courses that do not have a prereq**
- Your output should have the following columns (names should match exactly; there should be no prefixes):
    - `course_id`
    - `title`
    - `dept_name`
    - `credits`
- You may not use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

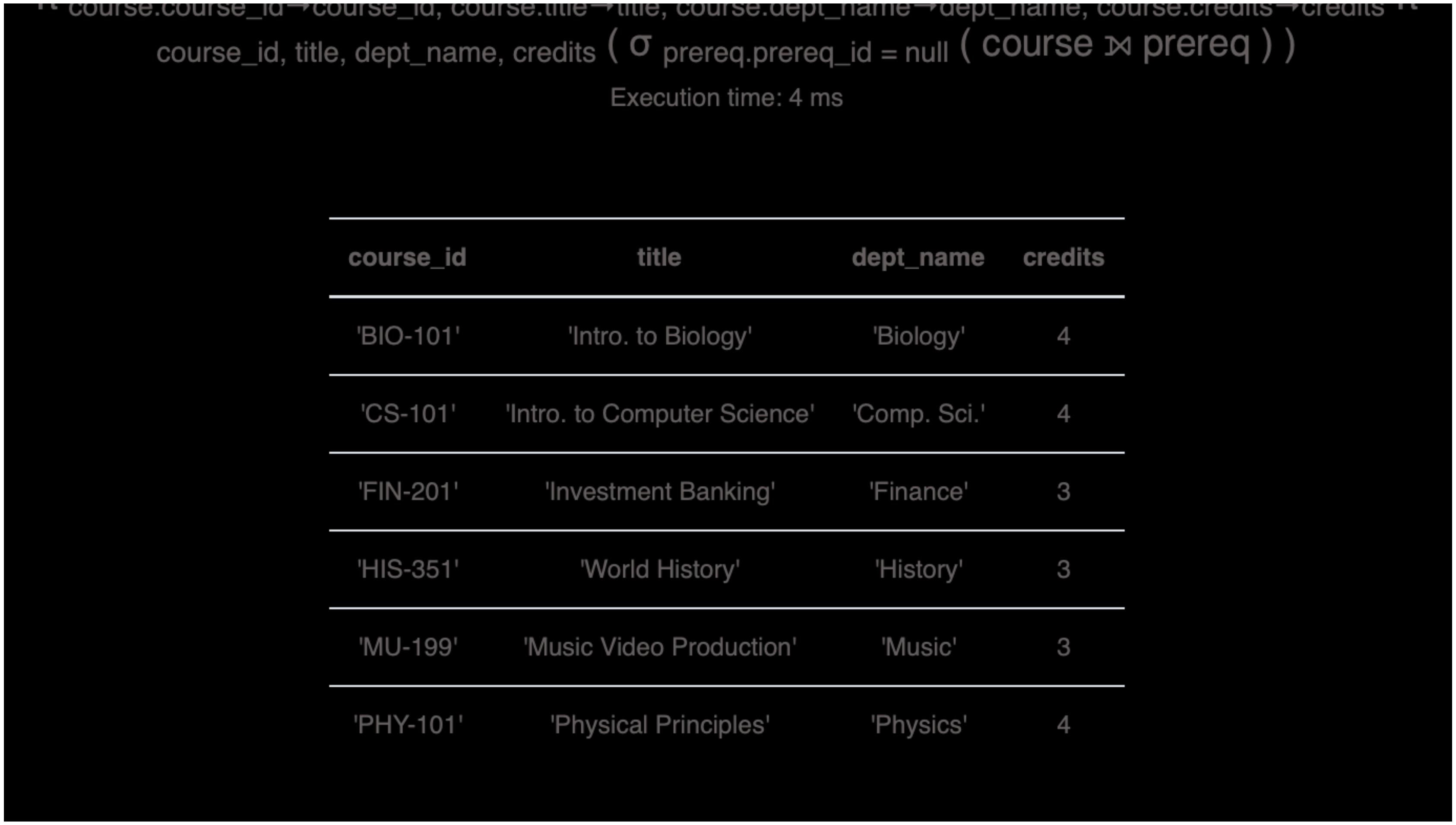
Algebra statement:

```
table = π course_id, title, dept_name, credits (σ prereq.prereq_id=null (course ⟖
prereq))

π course.course_id → course_id, course.title → title, course.dept_name → dept_name,
course.credits → credits table
```

Execution:


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



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

Execution time: 4 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

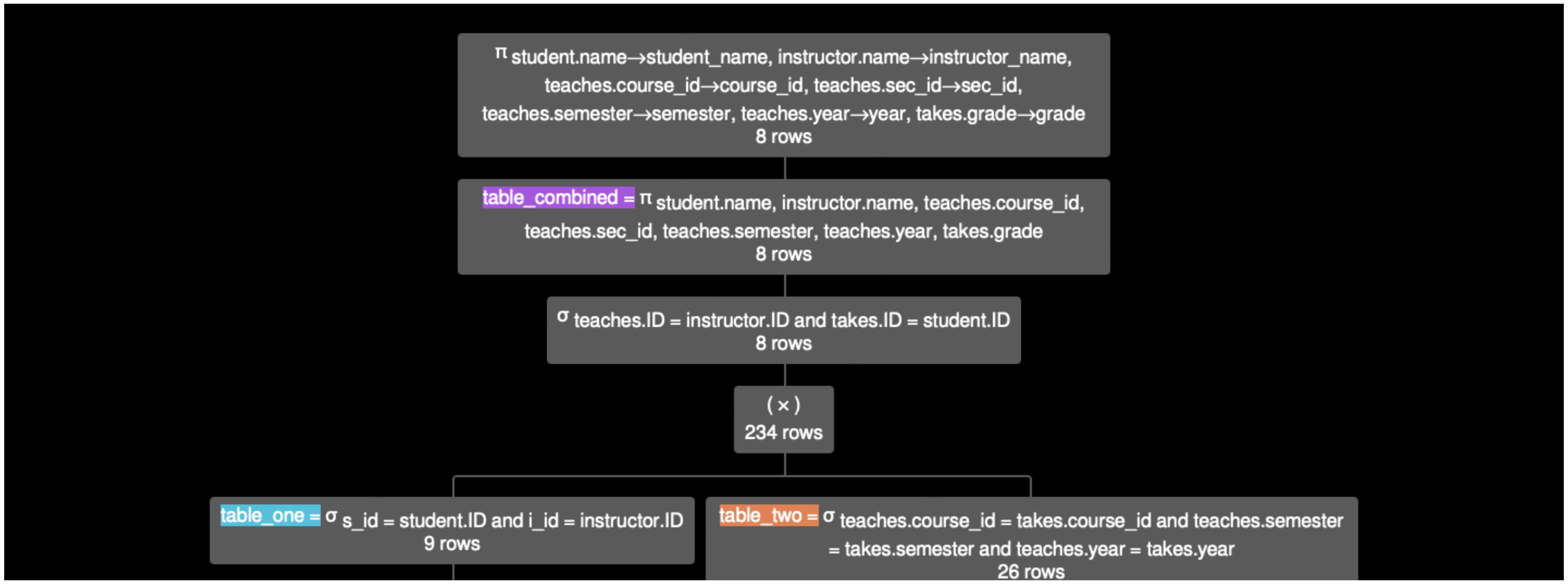
Algebra statement:

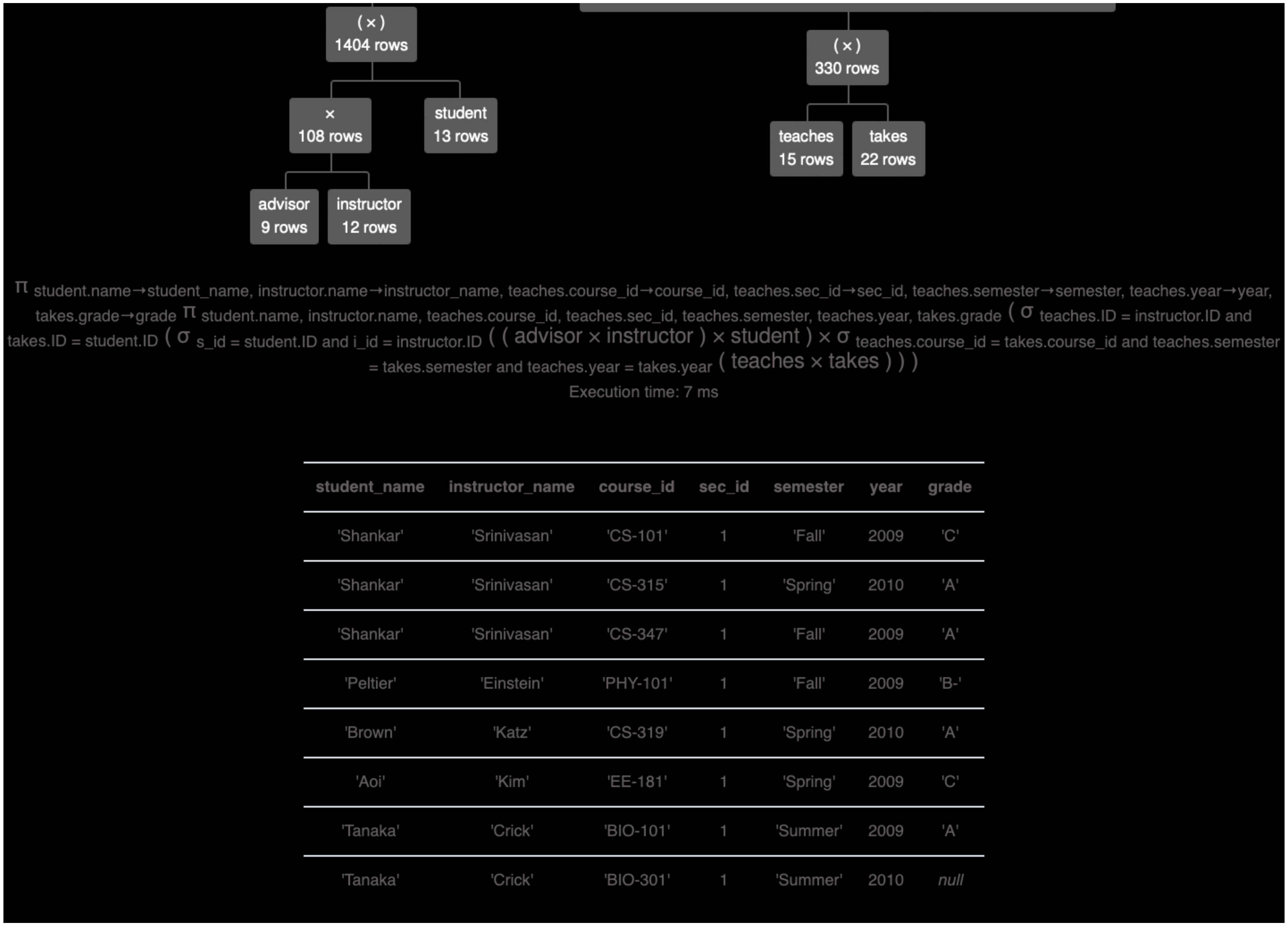
```
table_one = σ s_id=student.ID ∧ i_id=instructor.ID (advisor × instructor × student)
table_two = σ teaches.course_id=takes.course_id ∧
    teaches.semester=takes.semester ∧
    teaches.year=takes.year
    (teaches × takes)

table_combined = π student.name, instructor.name, teaches.course_id, teaches.sec_id,
teaches.semester, teaches.year, takes.grade (σ teaches.ID=instructor.ID ∧
takes.ID=student.ID (table_one  × table_two))

π student.name → student_name, instructor.name → instructor_name, teaches.course_id →
course_id, teaches.sec_id → sec_id, teaches.semester → semester, teaches.year → year,
takes.grade → grade table_combined
```

Execution:


π student.name→student_name, instructor.name→instructor_name, teaches.course_id→course_id, teaches.sec_id→sec_id, teaches.semester→semester, teaches.year→year, takes.grade→grade
8 rows
table_combined = π student.name, instructor.name, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year, takes.grade
8 rows
σ teaches.ID = instructor.ID and takes.ID = student.ID
8 rows
(×)
234 rows
table_one = σ s_id = student.ID and i_id = instructor.ID
9 rows
table_two = σ teaches.course_id = takes.course_id and teaches.semester = takes.semester and teaches.year = takes.year
26 rows



π student.name→student_name, instructor.name→instructor_name, teaches.course_id→course_id, teaches.sec_id→sec_id, teaches.semester→semester, teaches.year→year, takes.grade→grade π student.name, instructor.name, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year, takes.grade (σ teaches.ID = instructor.ID and takes.ID = student.ID (σ s_id = student.ID and i_id = instructor.ID ((advisor × instructor) × student) × σ teaches.course_id = takes.course_id and teaches.semester = takes.semester and teaches.year = takes.year (teaches × takes)))

Execution time: 7 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'
'Peltier'	'Einstein'	'PHY-101'	1	'Fall'	2009	'B-'
'Brown'	'Katz'	'CS-319'	1	'Spring'	2010	'A'
'Aoi'	'Kim'	'EE-181'	1	'Spring'	2009	'C'
'Tanaka'	'Crick'	'BIO-101'	1	'Summer'	2009	'A'
'Tanaka'	'Crick'	'BIO-301'	1	'Summer'	2010	*null*

R2 Execution Result

R3

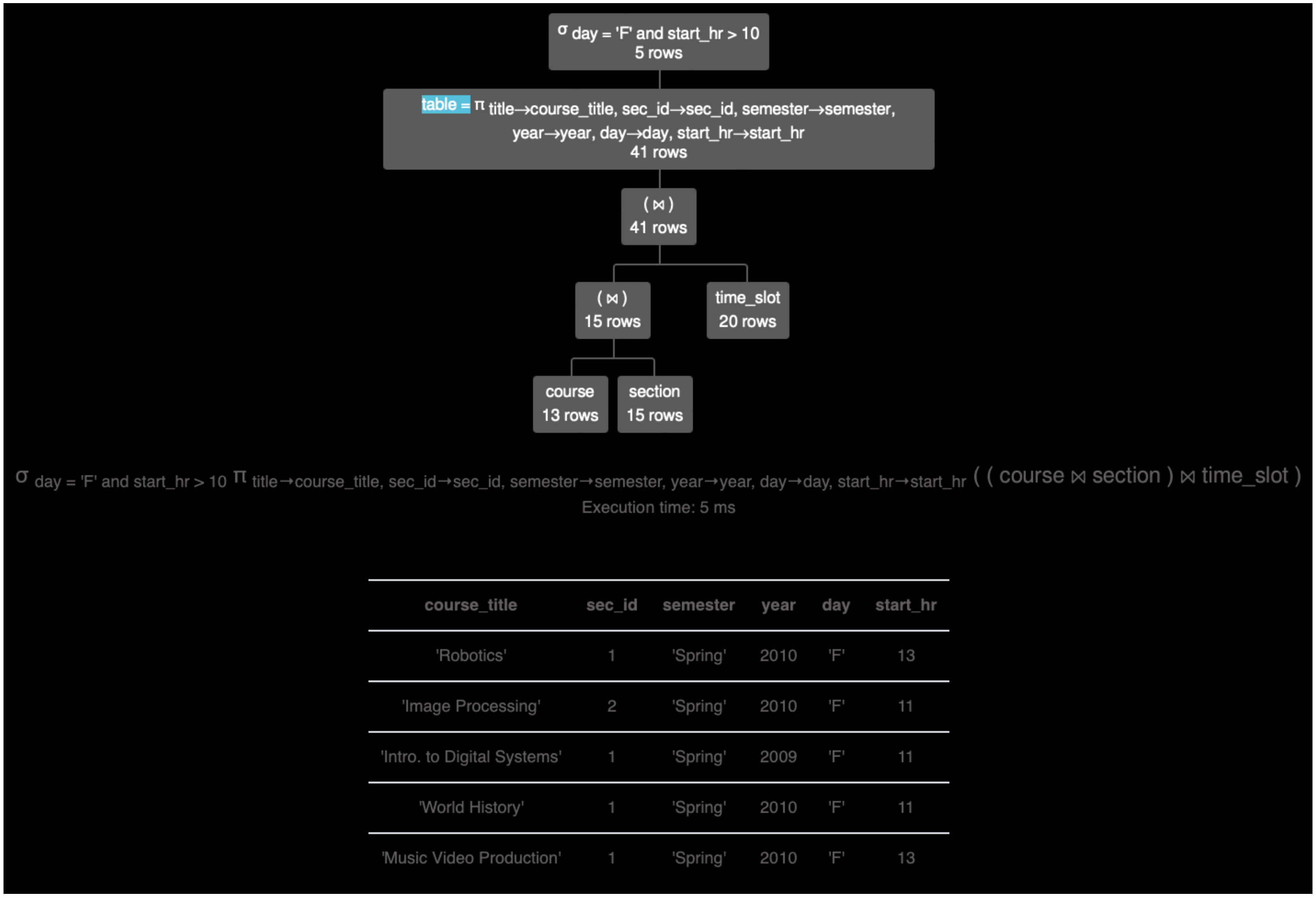
- Write a relational algebra query that produces a relation showing **sections that occur on Friday and start after 10 AM**
- Your output should have the following columns (names should match exactly; there should be no prefixes):
    - `course_title`
    - `sec_id`
    - `semester`
    - `year`
   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

Algebra statement:

```
table = π title → course_title, sec_id → sec_id, semester → semester, year → year, day →
day, start_hr → start_hr ((course ⨝ section) ⨝ time_slot)

σ day='F' ∧ start_hr>10 table
```

Execution:


σ day = 'F' and start_hr > 10
5 rows
table = π title→course_title, sec_id→sec_id, semester→semester, year→year, day→day, start_hr→start_hr
41 rows
(⋈)
41 rows
(⋈)
15 rows
time_slot
20 rows
course
13 rows
section
15 rows

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

Execution time: 5 ms

course_title	sec_id	semester	year	day	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 [ ]:
```