Relational Division and SQL

Robert Soulé

1 Example Relations and Queries

As a motivating example, consider the following two relations: `Taken(Student,Course)` which contains the courses that each student has completed, and `Required(Course)`, which contains the courses that are required to graduate. The instances for this example are shown below:

Taken	Student	Course
	Robert	Databases
	Robert	Programming Languages
	Susie	Databases
	Susie	Operating Systems
	Julie	Programming Languages
	Julie	Machine Learning
	Emilie	Operating Systems

Required	Course
	Databases
	Programming Languages

Suppose we are asked the following two queries:

1. Find all students who have taken a required course.
2. Find all students who can graduate (i.e., who have taken all required courses).

2 Asking About Some

We have already seen how to do Query 1 using the standard `SELECT` ... `FROM` ... `WHERE` clauses. To remove duplicates from our result, we can use the SQL keyword `DISTINCT`. In terms of relational algebra, we use a selection (σ), to filter rows with the appropriate predicate, and a projection (π) to get the desired columns.

```
SELECT DISTINCT Student
FROM Taken
WHERE Course = ’Databases’
   or Course = ’Programming Languages’;
```

If we want to be slightly more general, we can use a *sub-query*:

```
SELECT DISTINCT Student
FROM Taken
WHERE Course IN (SELECT Course FROM Required);
```

Informally, we might read this query as “give me the set of students who have taken a course that appears in the set of required courses”, or “the set of students whose courses contain *at least one* course that is required.”

3 Asking About All

Query 2 is more difficult. Just by looking at this small instance, it is easy to see that the answer we want is:

CanGraduate	Student
	Robert
	Robert

There is a relational operator that directly gives us this result. The operator is *division*, written $R \div S$.

Unfortunately, there is no direct way to express division in SQL. We can write this query, but to do so, we will have to express our query through double negation and existential quantifiers. We will produce this query in stages. Our roadmap is as follows

1. Find all students
2. Find all students and the courses required to graduate
3. Find all students and the required courses that they have not taken
4. Find all students who can not graduate
5. Find all students who can graduate

All Students First, we create a set of all student that have taken courses. We can express this positively using a selection and projection:

```
CREATE TEMP TABLE AllStudent as
SELECT Student
FROM Taken ;
```

AllStudent	Student
	Robert
	Susie
	Julie
	Emilie

All Students and Required Classes Next, we will create a set of students and the courses they need to graduate. We can express this as a Cartesian product, creating the pairs of the form (student,course):

```
CREATE TABLE StudentsAndRequired AS
SELECT DISTINCT AllStudent.student, Required.course
FROM AllStudent, Required ;
```

StudentsAndRequired	Student	Course
	Robert	Databases
	Robert	Programming Languages
	Susie	Databases
	Susie	Programming Languages
	Julie	Databases
	Julie	Programming Languages
	Emilie	Databases
	Emilie	Programming Languages

All Students and Required Classes Not Taken This is where our query starts to get tricky. We want to find the subset of the relation we just produced that includes the students and the required courses that they *have not taken*. We are doing this as a first step towards finding the students who cannot graduate.

The intuition is that we want to find all (student,course) pairs that are in the relation `StudentsAndRequired`, but not in the relation `Taken`. This should give us the set of students who cannot graduate, with the courses that they still need to take:

```
CREATE TEMP TABLE StudentsAndRequiredNotTaken AS
SELECT * FROM StudentsAndRequired
WHERE NOT EXISTS (SELECT *
FROM Taken
WHERE StudentsAndRequired.student = Taken.student AND StudentsAndRequired.Course = Taken.Course);
```

StudentsAndRequiredNotTaken	Student	Course
	Susie	Programming Languages
	Julie	Databases
	Emilie	Databases
	Emilie	Programming Languages

Students Who Can Not Graduate From the previous relation, we can apply a projection to get the set of students who cannot graduate.

```
CREATE TEMP TABLE CannotGraduate AS
SELECT Student FROM StudentsAndRequiredNotTaken;
```

CannotGraduate	Student
	Susie
	Julie
	Emilie
	Emilie

Students Who Can Graduate This is the second tricky part of our query. We find the subset of students who can graduate by looking at the students in `AllStudents` who are not in the set of `CannotGraduate`. Put another way, the set of all students except the students who cannot graduate:

```
CREATE TEMP TABLE CanGraduate AS
SELECT * FROM AllStudents
WHERE NOT EXISTS (SELECT *
FROM CannotGraduate
WHERE CannotGraduate.Student = AllStudents.Student);
```

CanGraduate	Student
	Robert
	Robert

Re-write As a Single Query We can re-write the above set of queries into a single query that does not use the temporary tables:

```
SELECT DISTINCT x.Student
FROM taken AS x
WHERE NOT EXISTS (
    SELECT *
    FROM required AS y
    WHERE NOT EXISTS (
        SELECT *
        FROM taken AS z
        WHERE (z.Student=x.Student)
        AND (z.Course=y.Course)));
```

Re-write Using Except (or Minus) We can re-write the above to use `EXCEPT` instead of `NOT EXISTS`:

```
SELECT Student FROM Taken
EXCEPT
SELECT Student FROM (
SELECT Student,Course
FROM (select Student FROM Taken), Required
EXCEPT
SELECT Student,Course FROM Taken);
```

Re-write Using Aggregations We can write the query in an (arguably) simpler version, using set membership (i.e., `IN`), `GROUP BY`, `COUNT` aggregations, and `HAVING`:

```
SELECT Student
From Taken
WHERE Course IN (SELECT Course FROM Required)
GROUP BY Student
HAVING COUNT(*) = (SELECT COUNT(*) FROM Required);
```