More advanced MySQL constructs

Reading / References

- Concise Guide to Databases: A Practical Introduction[1] chapter 4
- Introduction to SQL: Mastering the Relational Database Language[2]
- MySQL[3]
- Stanford free online course on databases[4]

Notes

In this section we will consider some of the more advanced SQL constructs:

- **Subordinate SELECT queries** allow us to use a whole SELECT query in the WHERE clause of another query. They are useful in some situations where the condition for the WHERE clause depends on the specific element values. But they are also costly to perform, so they should only be used when needed.
- **Joins** are an extremely important construction that brings together tables along some related values.
- Specialized SQL **functions** can be used for basic and not-so-basic operations. They are often used in conjunction with **GROUP BY**, which allows us to obtain summary information from our tables based on some grouping of the rows.
- **DELETE** and **UPDATE** operations have a slightly different syntax with its own challenges.

Joins Many times in the earlier sections we have had the need to consolidate different tables across foreign keys. Let us look at one more example, where we want to show all enrollments in terms of student name and course info. This task brings together information from different tables, namely the student, course and enrollment tables. We will do this task in steps.

First, we will try to look at enrollments but instead of seeing student ids, we want to see all their information. Our first attempt would be this:

```
SELECT *
FROM students s, enrollments e;
```

Try it out and look at the result. You should see first the columns for the students, and then the ones from the enrollments. But take a closer look at the id column, holding the student id, and the student_id column, holding the student id stored in the enrollment. They don't always match! Right now we have every student listed with

[1] https://acm.skillport.com/skillportfe/main.action#summary/BOOKS/RW$31069:_ss_book:76983

[2] https://www.safaribooksonline.com/library/view/introduction-to-sql/0321305965/

[3] https://www.safaribooksonline.com/library/view/mysql-fifth-edition/9780133038552/

[4] https://lagunita.stanford.edu/courses/Home/Databases/Engineering/about

every enrollment, not only theirs but of all the other students as well! That's clearly wrong. We need to make sure to only see the pair of a student and an enrollment if the enrollment corresponds to *that* student. We need a WHERE clause for that:

```
SELECT *
FROM students s, enrollments e
WHERE s.id = e.student_id;
```

Any time we bring together tables like that, it is called a **join**. SQL offers us an alternative way to describe such joins, like so:

```
SELECT *
FROM students s
JOIN enrollments e ON e.student_id = s.id;
```

If we had omitted the ON part it would have given us all pairs of students and enrollments. The ON part is the analog of the WHERE clause before.

This is often called an "inner" join. We can also have something called a "left join". Try it out and see the difference:

```
SELECT *
FROM students s
LEFT JOIN enrollments e ON e.student_id = s.id;
```

You see that this includes a row for a student that is not enrolled in any classes. There is also a "right join" that would have instead done the same thing for "enrollments", but that would not have given us anything new. There should also be a "full join" that preserves rows for values that appear in only one side, without a match on the other side, but MySQL does not support it. There are ways to emulate it however, and if you find yourself needing it just search online for the many answers.

Here is a further example to incorporate the course info in the table:

```
SELECT first, last, prefix, no
FROM students as s
LEFT JOIN enrollments as e ON s.id = e.student_id
LEFT JOIN courses as c ON c.id = e.course_id
ORDER BY prefix, no;
```

Let us practice some more with joins. You can do these problems via either joins or just using a WHERE clause where appropriate.

- Find all pairs of students and courses where the student is enrolled in the course.
- Find all pairs of students and CS courses where the student is enrolled in the course.
- Find all pairs of ids for students that are in the same class. This would require joining two enrollments tables (i.e. joining the enrollments table with another copy of itself). You should not include pairs that consist of the same student twice.
- Find all pairs of students with the same last name but different first names.
- In the two problems above, find a way to make it so that pairs only appear once, i.e. if we have students s and t we would NOT see both the pair (s,t) and the pair (t,s).
- Find all pairs of courses that have a student in common.

Subordinate SELECT queries Let's take it up one more notch. We want to find all students that are not enrolled in any computer science courses, and enroll them into the introductory CS course. Let's start by finding out those students' ids. This is tricky. Here is a verbal description:

We want to select those students from s, whose id cannot be found in any enrollment where the course has prefix CS. So for each student we want to ask: "Is there an enrollment in a course for this student and where the course is a CS course"?

```
SELECT s.id
FROM students AS s
WHERE NOT EXISTS (SELECT prefix
                  FROM enrollments AS e, courses AS c
                  WHERE e.course_id = c.id
                  AND e.student_id = s.id
                  AND c.prefix = "CS"
                 );
```

Note the inner SELECT query, where we look for pairs of enrollments and courses, and we tie them together via the IDs.

You might have been tempted to do the following instead, and it would have been wrong:

```
SELECT student_id
FROM enrollments AS e, courses AS c
WHERE e.course_id = c.id
AND c.prefix <> "CS";
```

This would not have been right: It looks for all enrollments of students in courses. So it would include a student as long as they are enrolled in at least one non-CS course, even if the same student is also enrolled in a CS course.

Practice: Write a query to get back pairs of a student id and a course prefix, where the student is not enrolled in any courses with that prefix.

Now that we got the list of the student IDs, we need to use that whole thing inside an INSERT query. This is a common practice: Work out the SELECT first, and when you have that working then put it in the INSERT clause. In fact we'll take one more step before the insert:

```
SELECT s.id, c2.id
FROM students AS s, courses AS c2
WHERE c2.prefix = "CS"
AND c2.no = 220
AND NOT EXISTS (SELECT prefix
                  FROM enrollments AS e, courses AS c
                  WHERE e.course_id = c.id
                  AND e.student_id = s.id
                  AND c.prefix = "CS"
                 );
```

We needed to add some extra steps to get the correct course. We could have looked at the courses list, found the id and used that directly, but this way is a bit more elegant.

Finally, adding the INSERT:

```
INSERT INTO enrollments (student_id, course_id)
SELECT s.id, c2.id
FROM students AS s, courses AS c2
WHERE c2.prefix = "CS"
AND c2.no = 220
AND NOT EXISTS (SELECT prefix
                FROM enrollments AS e, courses AS c
                WHERE e.course_id = c.id
                AND e.student_id = s.id
                AND c.prefix = "CS"
               );
```

SQL functions SQL contains a number of built-in functions, and even allows you to create your own, though that is a more advanced process. You can find a full list of the available mySQL functions here[5]. We will highlight a few:

- AVG() returns the average
- CONCAT() concatenates strings
- COUNT() and COUNT(DISTINCT) return counts of matches
- MAX() and MIN() find maximum and minimum values
- RAND() produces a random floating point value
- SUM() adds up all the values

Here is a simple example returning the concatenated first-last names of the students:

```
SELECT CONCAT(first, " ", last)
FROM students;
```

PRACTICE: Use the same approach to produce course names that look like so: "CS220: Fundamentals ..."

Let's learn a bit about how to use COUNT now. Here's a way to count all the students:

```
SELECT COUNT(*)
FROM students;
```

Let us look at how many students each course has. In order to do that we need to also learn about grouping. Look at the following:

```
SELECT course_id, COUNT(*)
FROM enrollments
GROUP BY course_id;
```

OK! So it shows us the three courses that have students enrolled in them, as well as how many there are there. GROUP BY tells it to group all those rows with the same course_id, and perform the described operation to them. In such a case you are restricted into what you can put in the SELECT part. They must be either the entries in the GROUP_BY clause or functions that aggregate across the entire list of entries, like COUNT.

[5] http://dev.mysql.com/doc/refman/5.7/en/func-op-summary-ref.html

More advanced queries This is a bit unsatisfactory however. First, we would like to see the course prefix plus number, not just the ids. Second, we would like to see the courses with 0 students included. This is trickier.

```
SELECT prefix, no, (SELECT COUNT(student_id)
                    FROM enrollments
                    WHERE course_id = id) AS enrollment
FROM courses;
```

This is a different example of a subordinate SELECT query. This time the whole SELECT query goes into one of the "column" in the outer SELECT. Within its form we can use the specific id for the course in that row. So what we are saying here is basically "for each course, count the number of enrollments with that course id, and these numbers form the basis for the enrollment column".

We may further want to order by that new column:

```
SELECT prefix, no, (SELECT COUNT(student_id)
                    FROM enrollments
                    WHERE course_id = id) AS enrollment
FROM courses
ORDER BY enrollment DESC;
```

DELETE We can use DELETE to remove entries. For instance let's remove all students with first name Peter from all classes. As this is a destructive operation, you want to do a SELECT query first to make sure you have the right cases. You should do this yourself: Find all enrollments where the student's name is "Joe".

```
SELECT * FROM enrollments
WHERE student_id IN (SELECT id from students
                     WHERE first = "Joe");
```

Now we change this into a DELETE. Remember we just want to delete the enrollment, not the student record:

```
DELETE FROM enrollments
WHERE student_id IN (SELECT id from students
                     WHERE first = "Joe");
```

Oops, did you see the error message? The system refused to do this. By default mysql will discourage you from deleting in any situation where the WHERE clause doesn't include conditions on all key attributes. In this case that would have had to be the combined student_id and course_id columns, but we did not use the course_id ones. This is called "safe update mode" and it is there to protect us from making too many changes by accident. One way around this is to disable the mode altogether, then reconnect to the server. Another is to add a clause for the missing key, with a condition that is always true:

```
DELETE FROM enrollments
WHERE student_id IN (SELECT id from students
                     WHERE first = "Joe")
AND course_id > 0;
```

UPDATE We can use the UPDATE method to set values within particular records. Some of the clauses in it are similar to that in a SELECT clause, where we need to identify which rows to act on. Here is an example. Let us say that we want to update the entry in a student's file that shows the number of credits the student has. This would entail adding up the credits from the various courses that the student is enrolled in. Let us first work out the SELECT query that would get that information for us. Do it yourself before reading on.

```
SELECT s.id, (SELECT SUM(c.credits)
              FROM courses AS c
              JOIN enrollments AS e on e.course_id = c.id
              WHERE e.student_id = s.id)
FROM students s;
```

When you run the above query you will notice that some of the sums are NULL. That is what happens for students that are not enrolled in any courses. We would rather have 0s there. We could do this using the function COALESCE that returns the first non-null argument in its list:

```
SELECT s.id, (SELECT COALESCE(SUM(c.credits), 0)
              FROM courses AS c
              JOIN enrollments AS e on e.course_id = c.id
              WHERE e.student_id = s.id)
FROM students AS s;
```

We could now turn this into an UPDATE query:

```
UPDATE students AS s
SET s.credits = (SELECT COALESCE(SUM(c.credits), 0)
                 FROM courses AS c
                 JOIN enrollments AS e on e.course_id = c.id
                 WHERE e.student_id = s.id)
WHERE s.id > 0;
```

The last line is needed for the same "safe update" reasons we ran into when we were trying to do a delete.

Let us modify the above query a bit. We should only count students as having credits in courses in which they have received a letter grade. This would require us to change the inner SELECT query:

```
UPDATE students AS s
SET s.credits = (SELECT COALESCE(SUM(c.credits), 0)
                 FROM courses AS c
                 JOIN enrollments AS e on e.course_id = c.id
                 WHERE e.student_id = s.id
                 AND e.letter_grade IS NOT NULL)
WHERE s.id > 0;
```

We should find all students to have 0 credits now, as there are no assigned grades so far.

For your homework, you will create a new table to hold the numeric correspondence of letter grades to numbers, and then you would be able to use that to update the gpa entries.