CSCB20 Worksheet – More MySQL

1 Queries Cont...

Selecting Columns with an Aggregate Function

- Find the total number of instructors who taught a course in the Spring 2010 semester.
  `SELECT COUNT(id) FROM teaches WHERE semester='Spring' AND year=2010;`

  This gives us the wrong answer...want `DISTINCT` instructors.

  ```
  SELECT COUNT(DISTINCT id) FROM teaches WHERE semester='Spring' AND year=2010;
  ```

  Use `DISTINCT` whenever you do not want the duplicates to be included. With the `AVG` function we want to include all duplicates, with `COUNT` sometimes we don't.

The `HAVING` Clause

- Select the department name and average salary for instructors for each department. Include only those departments whose salaries are greater than $42000.

  ```
  SELECT dept_name, AVG(salary) AS avg_salary FROM instructor GROUP BY dept_name
  HAVING AVG (salary) > 42000;
  ```

  The HAVING cause is a condition that applies to *groups* rather than *tuples*. The HAVING clause is applied *after* the groups have been formed.

Q. In what order are the SQL commands applied to a relation to create the smaller relation?

1. FROM
2. The predicate in the WHERE clause (if present) is applied to tuples satisfying the FROM clause.
3. Tuples satisfying the WHERE predicate are then placed into groups by the GROUP BY clause (if present).
4. The HAVING clause (if present) is applied to each group. Groups not satisfying the HAVING clause are removed.
5. The SELECT clause is applied to the remaining groups, applying aggregate functions to get the resulting tuple for each group.

- For each course section offered in 2009, find the average total credits (tot_cred) of all students enrolled in the section, **if** the section had at least 2 students.

  ```
  SELECT course_id, semester, year, sec_id, AVG(tot_cred) FROM takes NATURAL
  JOIN student WHERE year = 2009 GROUP BY course_id, semester, year, sec_id HAVING
  COUNT(ID) >= 2;
  ```

The `IN` and `NOT IN` Clauses

- Find all the courses taught in both the Fall 2009 and Spring 2010 semesters.
  ```
  SELECT course_id FROM section WHERE semester='Spring' AND year='2010' AND
  course_id IN (SELECT course_id FROM section WHERE semester='Fall' AND year='2009');
  ```

- Find all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester.

  ```
  SELECT course_id FROM section WHERE semester='Spring' AND year='2010' AND
  course_id NOT IN (SELECT course_id FROM section WHERE semester='Fall' AND year='2009')
  ```

  `IN` or `NOT IN` can be use to check if an attribute belongs to a tuple. For example:

- Find the names of the instructors whose names are neither 'Mozart' nor 'Einstein'.

  ```
  SELECT name FROM instructor WHERE name NOT IN ('Mozart', 'Einstein');
  ```

  We can also check whether a tuple of attributes belongs in a relation using `IN` or `NOT IN`.

- Find the total number of distinct students who have taken course sections taught by the instructor with `ID 10101`.

  ```
  SELECT COUNT (DISTINCT ID) FROM takes WHERE
  (course_id, sec_id, semester, year) IN (SELECT course_id, sec_id, semester,
  year FROM teaches WHERE teaches.ID = 10101);
  ```

Using `LIKE` and `NOT LIKE`

- Find all the students whose names contain "an" in them.

  ```
  SELECT name FROM student WHERE
  name LIKE '%an%';
  ```

- Find all the students whose names are 4 characters long.

  ```
  SELECT name FROM student WHERE
  name LIKE '_ _ _ _';
  ```

  Use the `LIKE` or `NOT LIKE` condition to search for strings that match. The % matches anything and the _ matches exactly one character.

Using Subqueries.

- We know that we can SELECT ... FROM <*any relation*>. This means that the FROM clause can be the result of a SELECT statement itself.

  Find the average instructors' salaries of those departments where the average salary is greater than $42,000 (*without using a* `HAVING` clause).

  First find the depart names and their average salaries.

  ```
  SELECT dept_name, AVG(salary) AS avg_salary FROM instructor GROUP BY dept_name
  ```

  Now we can `SELECT` from it.

  ```
  SELECT dept_name, avg_salary
  FROM (SELECT dept_name, AVG(salary) AS avg_salary FROM instructor GROUP BY
  dept_name) AS T
  WHERE T.avg_salary > 42000;
  ```

2 Creating and Updating Tables

Let's create a new table called `accounts` that contains a persons `ID, name, address` and `phone`.

```
CREATE TABLE account(ID VARCHAR(5), name VARCHAR(20) NOT NULL, address VARCHAR(50),
phone CHAR(12));
```

And now let's insert a sample row:

```
INSERT INTO account VALUES ('11111', 'Anna Bretscher', '1265 Military Trail, Toronto',
'416-978-7572');
```

We can `DROP` this table if we don't need it any more using:

```
DROP account;
```

Now lets insert into the **student** table a new student who studies 'Music', has ID '99999', name 'Star' and 150 credits.

```
INSERT INTO student VALUES('99999', 'Star', 'Music', 150);
```

Now we can try inserting into the instructor table using a INSERT with SELECT statement.

```
INSERT INTO instructor
SELECT ID, name, dept_name, 18000
FROM student
WHERE dept_name='Music' AND tot_cred > 144;
```

We can use `UPDATE` to give the instructors a 5% pay raise.

```
UPDATE instructor
SET salary = salary*1.05;
```

Increase the salary by 5% for all instructors whose salary is less than $60,000.

```
UPDATE instructor
SET salary = salary*1.05
WHERE salary < 60000;
```

Increase the salary by 5% for all instructors whose salary is less than or equal to $60,000 and by 3% if the salary is greater than $60000.

```
UPDATE instructor
SET salary = CASE WHEN salary <= 60000 THEN salary*1.05
ELSE salary*1.03 END;
```

3 Views

Suppose we wish to display each student ID and a CGPA. We can break this into steps by first creating a `VIEW` . You may assume that each letter grades translates to a GPA value as shown by the table `grade_points`.

```
+-------+--------+
| grade | points |
+-------+--------+
| A     |    4.0 |
| A+    |    4.0 |
```

```
| A-    |    3.7 |
| B     |    3.0 |
| B+    |    3.3 |
| B-    |    2.7 |
| C     |    2.0 |
| C+    |    2.3 |
| C-    |    1.7 |
| D     |    1.0 |
| D+    |    1.3 |
| D-    |    0.7 |
| F     |    0.0 |
+-------+--------+
```

The CGPA is defined as

$$\frac{\sum_i (credit_i * points_i)}{\sum_i credits_i}$$

Which tables do we need columns from?

`course, takes, grade_points`

We simplify the problem by creating a view with which columns?

ID, course_id, grade, credits

The `VIEW` statement is then:

```
create view student_marks AS (select id, takes.course_id, grade, credits from takes natural join course);
```

and the final `SELECT` statement is then:

```
select id, sum(credits*points)/sum(credits) as gpa from student_marks natural join grade_points group by id;
```

4 Outer Joins

- Find all students and the courses they have taken - include students who have not taken any courses yet.

  ```
  SELECT * FROM student NATURAL LEFT OUTER JOIN takes;
  ```

  Notice the difference with `SELECT * FROM student NATURAL JOIN takes;`.

  Student Snow is missing.

  Can we rewrite the LEFT OUTER JOIN using an equivalent RIGHT OUTER JOIN?

  yes. switch relations order so `SELECT * FROM takes NATURAL RIGHT OUTER JOIN student;`

  What if we would like to find all students who have *not* take a course?

  ```
  SELECT * FROM student NATURAL LEFT OUTER JOIN takes WHERE course_id IS NULL;
  ```

- Display a list of all students in the Comp. Sci. department, along with the course sections, if any, that they have taken in Spring 2009; all course sections from Spring 2009 must be displayed, even if no student from the Comp. Sci. department has taken the course section.

  Let's first think of this in words. We want to select all the students in Comp. Sci. and natural full outer join with all the spring 2009 courses. In MySQL this is the union of the left outer join with the right outer join.

  ```
  SELECT * FROM
  (SELECT * FROM student WHERE dept_name='Comp.  Sci.')  A
  NATURAL FULL OUTER JOIN
  (SELECT * FROM takes WHERE semester='Spring' AND year=2009) B;
  ```

  which is in MySQL:

  ```
  (select A.*, B.* from (select * from student where dept_name='Comp. Sci.') A
   LEFT OUTER JOIN
   (select * from takes where semester='Spring' and year=2009) B ON A.id=B.id)

  UNION

  (select A.*, B.* from (select * from student where dept_name='Comp. Sci.') A
   RIGHT OUTER JOIN
   (select * from takes where semester='Spring' and year=2009) B ON A.id=B.id);
  ```

- Another `JOIN` example. Select all student names and their advisor names include those students who do not have advisors.

SELECT student.name AS 'student name', instructor.name AS 'instructor name' FROM (student LEFT JOIN advisor ON student.id = s_id) LEFT JOIN instructor ON advisor.i_id = instructor.id;

- What if we'd like to see the instructors who don't have students to advise?

```
(SELECT student.name AS 'student name', instructor.name AS 'Instructor Name'
FROM (student LEFT JOIN advisor ON student.id = s_id)
LEFT JOIN instructor ON advisor.i_id = instructor.id)
UNION
 (SELECT student.name AS 'student name', instructor.name AS 'Instructor Name'
  FROM (student  JOIN advisor ON student.id = s_id)
RIGHT JOIN instructor ON advisor.i_id = instructor.id);
```

5 University Relations

Relations and their schemas:

classroom(*building*, *room_number*, *capacity*)
department(*dept_name*, *building*, *budget*)
course(*course_id*, *title*, *dept_name*, *credits*)
instructor(*ID*, *name*, *dept_name*, *salary*)
section(*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
student(*ID*, *name*, *dept_name*, *tot_cred*)
takes(*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
advisor(*s_ID*, *i_ID*)
time_slot(*time_slot_id*, *day*, *start_time*, *end_time*)
prereq(*course_id*, *prereq_id*)

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Teaches

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Instructor

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Section

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Prereq

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Department

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
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

Course