Practice Exam Notes -- 5 December 2016

See the Dec. 5 Lecture Notes for a list of topics that will be covered on the exam.

Modeling Questions

Question 1: Recall that a "scalar" value means a *single* value / atomic value, AKA, not a list or other complex value type; not a nested objects. Since the JSON from the Tweet is such a complex-valued object (it's got lots of name-value pairs), *Tweet.raw_tweet* violates 1NF, so the table is not in 1st Normal Form.

Question 2: The only table with a composite key is *Enrollment*. But the columns *grade* and *grade_pts* depend on both of *eid* and *unique* because a grade is assigned to a student for a particular class, i.e., a unique combination of student and class. So there are no violations of 2NF in this table (besides *Tweet.raw_tweet*, which violates 1NF, which is required for 2NF). Note if we added a *student_name* column to *Enrollment*, we would have a violation, because *student_name* depends only on *eid*, not the combination of *eid* and *unique*.

Question 3: In the *Major* table, we see that *college_name* is functionally dependent on *college_code*, which is not the primary key, so this table violates 3NF. Note that if we wanted to change the *Major* table to be in 3NF, we would "split off" a new table *College* containing the rows *college_code* and *college_name*, and remove *college_name* from *Major*. In this new model, *college_code* in *Major* would be a foreign key to the new *College* table.

Question 4: If a student can have multiple majors, we will now have a many-to-many relationship between *Student* and *Major*, because a student can have multiple majors and a major can have multiple students. So we need a junction table between these two tables, which we will call *StudentMajor*. This new table will have a composite primary key of *eid, major_code*. So the table definition, in the given formatting, will look like

StudentMajor(**eid**, **major_code**) .

Note that each of these columns is a foreign key back to their original table.

Question 5: We will need a new table called *Prereq* that is a junction table between *Class* and itself. This will look like: Prereq(**unique**, **prereq_unique**, min_grade) . The composite key is made up of two course unique numbers, which each reference the *Class* table as a foreign key.

SQL Questions

Question 1: This is just a basic insert statement.

Question 2: This is just a regular old update statement. These take the form

UPDATE [table] SET [col1] = [val1], [col2] = [val2], … WHERE [col]=[val]

Note that normally we use **is null** instead of **= null**, but when we're setting a value to **null**, like we're doing here, we use **= null.**

Questions 3 & 4: These are just basic delete statements. Question 4 makes sure you know the standard date formatting. For Question 3, note that in case that class has enrollments in *Enrollment*, we need to delete the child rows in *Enrollment* before deleting the parent row in *Class*. Otherwise, we will not be able to delete the class from *Class*.

Question 5: Basic query using **distinct** to get only the unique values, and using an **order by** to sort the results.

Question 6: Note we want to use an inner join here because we only want the domestic students. We join on the *eid*s and filter out the "in-state" rows, i.e., those with *Domestic_Student.state* = 'TX'. That is, we keep rows where *Domestic_Student.state* <> *'TX'* (<> means not equal).

Question 7: Here, we need the tables *Student, Enrollment,* and *Class* to get all the necessary columns. So we need to join these three tables. We **join** *Student* to *Enrollment* using *eid* and we **join** *Enrollment* to *Class* using *unique*. We use a **where** to filter so that we only get the rows where the major is math, the class name is 'Elements of Databases' and the grade is null. Note that both joins are inner joins.

Question 8: We want to find the number of classes requiring each prerequisite class. This means we want to group by the prerequisite column, *prereq_unique*. Since we want "the number of classes that use it as a pre-requisite", we want to use the **count()** aggregation. We order the results descending with **order by** to get the highest values first. Since we only want the top 10 results, we use **limit**. Note that in the select we can either use *count(*)* or *count(prereq_unique)*, because *prereq_unique* is a required column, so these counts will be equivalent.

Question 9: We want an outer join here to see which instructors have a record in *Instructor* but no records in *Class*. If this is the case, then *Class.unique* will be null. We check *Class.unique* instead of one of the other rows, because *unique* is a required column -- the other columns may be null for reasons other than that they have no corresponding *Instructor.eid*.

Question 10: We also want to do a join on *Instructor* and *Class* here, but this time we want an inner join instead of an outer join, because we want instructors who have taught multiple classes in this case. (An outer join would work here, but we don't need it and it will be less efficient.) We **group by** *Instructor.eid* because we want a count for each unique instructor eid. We use *count(c.unique)* because *unique* is a required column. We use a **having** clause to filter for rows corresponding to an instructor having taught more than 3 classes. We order by these

counts from greatest to lowest. We use a **where** to filter out the rows that aren't for classes in 2016. Note we use the alias *count(c.unique) as count* because we cannot use the aggregation in the **order by** clause.