Lab notes for MIDS 706 week 4 (SQLpart 2 – 9/14/2020 13:45 – 15:00)

Overview

In the following lab assignments, we'll endeavor to dive a bit more deeply and concretely into the topics we touched on during the lecture on Wednesday.

For most of the lab assignments outlined below, there's only one right *result*, but there may be multiple valid ways of getting it (and in fact, as you'll see, some of the exercises are designed to bring precisely that fact to the fore).

You'll need to have a working MySQL instance and to have installed the demonstration databases available alongside these notes in the GitHub respository in order to work through the lab exercises. The VCM host you should have created during Andy's section two weeks ago should be a fine place to work, although if you prefer to use a different MySQL instance running on a personal laptop or elsewhere, feel free to do so. The mysqldump'd databases in the "class" folder under the GitHub repository should load properly on any fairly current MySQL instance. Instructions for loading up the demonstration data are available in the README.md file within the GitHub repo. Given a running MySQL server, installing the data should take you no more than a few minutes, but I'd recommend doing so before we gather for the lab session so you can take the best advantage of the lab time.

The notes for this lab are aimed at your working with the MySQL command line client (just run "mysql -p" on your VCM instance while logged in as root, or "mysql -u root -p" while logged in as another user, and provide the (possibly blank) password you gave for your database 'root' user to bring up the client). If you have a preferred SQL client already set up to integrate with your MySQL instance, feel free to use that instead of the command line client, but try to avoid depending on special features of any GUI client you may use – we want to focus on looking closely at the SQL statements and the SQL responses we get, not at presentation layer a particular GUI client provides.

I'll break the notes into sections following the general outline of the lecture slides – feel free to refer back to the examples in the lecture slides and/or your notes when working through these exercises. Where possible, I'll try to include target results (or some indicator of success, at least) for each exercise, but since there are frequently different ways to approach the same data question with a SQL query, you may want to note what solution you finally arrive at for later discussion with your TAs or for other later reference.

Topic I: Data Modeling Exercises

In our lecture, we discussed two basic types of data models – ER and Relational models.

1) Refresher question: Think of one key difference between the ER model/ER Diagram mechanism and the Relational model. (Hint: What is cardinality?)

In MySQL, you can review the databases to which you have access with the command:

```
show databases;
```

Find the dewey_cheatham_howe database, and switch to its context with the mysql command:

```
use dewey_cheatham_howe;
```

[Remember that SQL statements must always end with a delimiter (which by default is the ";" character), and may span multiple lines.]

Exercises:

2) Retrieve the list of tables (show tables) in the database.
3) For each of the tables, inspect the table schema using the SQL 'describe' command.
4) Now, given what you know from the lecture about the relations involved and what the schema tells you about the tables, jot down (in whatever form you like) both a simple ER Diagram of the law office's data and a Relational model description for the data. Recognize the similarity between MySQL's 'describe table" output and the Relational model.
5) Do the same for the "orders" database.

Schemas and Normalization

In lecture, we talked about the so-called "star schema" and some of its advantages for simple use cases, as well as the issues it can introduce in dealing with higher-cardinality relations (as well as, in slides we didn't have a chance to discuss on Wednesday, security issues).

As we discussed, normalization **can** be formalized in mathematical terms, but for today, we'll stick to operational definitions. As such, the definitions are a bit weak, which is to say that there's room for some disagreement as to how they apply to specific cases.

To elaborate slightly on what we discussed: The first three "normal forms" (termed "1NF", "2NF", and "3NF") are **roughly** described as follows:

1NF: Column values are singular – that is, there is at most one value (or exactly one, if we consider NULL to be a valid value), there are no "repeating columnar groups" (eg., what we saw when we tried to split one column in the "star schema" table for student COVID data into two columns labeled "major1" and "major2", and there are no **fully duplicated** rows in the table.

2NF: Each table is in 1NF **and** has a primary key. The primary key may span multiple columns (as so-called compound primary key) and if it does, no other column can be **functionally dependent** on only one of the primary key columns. For example, in a table where the primary

key is the combination of a dorm name and a room number, a column for “street address” would depend solely on the dorm name (the name of the dorm uniquely determines the address of the building), and thus would strictly violate 2NF.

3NF: Each table is in 2NF, and additionally has no “transitive” functional dependencies (cases where there are second- or higher-order functional dependencies. For example, if a table relates a user id (which is the primary key) to a US zip code, a US state, and a US city, it may be in 2NF (the primary key in this case is singular), but not 3NF (because the relation from userid to zip can in turn determine the state and city for the user uniquely).

Exercise:

Look closely at the table schemas for the **normalized** database (the student COVID-19 testing database we normalized in the Wednesday lecture). The tables in that database include the star schema table (labeled “unnormalized”), the derived, “somewhat normalized” tables (student, tests, room_assignment, and majors) and the **views** described in the lecture notes. You may ignore the views for this exercise, and:

6) Look at the star schema table and identify whether and how it violates the constraints on each of the first three normal forms.
7) Look at the collection of more normalized tables (student, tests, room_assignment, and majors) and see if you can determine the highest normal form requirement each of them meets. Is the resulting collection of tables in 1NF, 2NF, or 3NF, and why?

Constraints, Keys, and Referential Integrity

In lecture, we discussed how database constraints (including key constraints) can be used to avoid data entry errors, enforce business rules on data, and also enable database engines to enforce **referential integrity** (ensuring that references between tables through **foreign keys** are at all times valid across a collection of interrelated tables).

We already identified the foreign keys in use in the child tables of the **normalized** database in the lecture, and we added a few useful constraints to them. Today, for exercise, we’ll add some more and see how they behave.

Recall the following syntactic facts:

- We can add a unique constraint to an existing table with an ALTER TABLE command of the form:
    - ALTER TABLE <tablename> ADD CONSTRAINT <constraint_title> UNIQUE(column_name[,…]);
- We can change the definition of an existing column in a table in MySQL with a similar ALTER TABLE command of the form:
    - ALTER TABLE <tablename> MODIFY <full field description including constraints>;

- We can see more details (including details about specific constraints) on a table by inspecting its "create" parameters using:
    - show create table <tablename>;
- We can add a LIMIT clause to the end of a SELECT statement to restrict the response to only N results, as:
    - <SELECT statement> LIMIT N;
- We can retrieve all columns in a table with a SELECT statement using "*" as the column definition, so the canonical "SELECT <something> FROM <something>" can be either "SELECT field1[,…] FROM <something>;" or "SELECT * FROM <something>;"
- We can add columns to a table with an INSERT statement, whose general form is:
    - INSERT INTO <tablename> (fieldname[,…]) VALUES (value[,…]);

Exercises:

Look at the schema for the majors table using "describe" and "show create table". Note the structure of the primary key and the foreign key that are defined, and keep those in mind (along with the implicit **referential integrity** constraints that will apply as a result) when you do the following exercises:

8) As currently built, the majors table schema will allow inserting an entry with a NULL major value. Does this seem like a good idea? Apply a constraint to the majors table to prevent someone from inadvertently inserting a row with a NULL major value. Verify that your constraint works, once applied, by trying to insert a new entry in the table with a null value for "major". Did you remember to address referential integrity requirements when performing your insert?
9) Consider the room_assignment table. Is it possible, in the current configuration, for one student to be assigned to more than one room in (potentially) more than one dorm? Let's make sure that doesn't happen by applying a UNIQUE constraint. Be careful how you think about the constraint, though – one **student** cannot be assigned to more than one room (that is, cannot appear in more than room assignment), but one **room** may house more than one student. (HINT: Since the table represents room *assignments*, consider how many *assignments* one student should be allowed to have). Verify that your constraint does what's expected by attempting to insert a row depicting a second room assignment for a given student.
10) Consider the tests table. The three non-key fields in the table are defined as follows:
    test_date is the date when a test was administered to the student
    test_returned is the date when the test's results were returned
    test_result is either "1" or "0" (TRUE or FALSE) for a positive or negative result
    All three are allowed to be NULL at the moment. This is appropriate for the latter two (test_returned and test_result may be NULL between when a test is administered and when its result is available), but not for test_date (what would be the meaning of a tests table row where no test has been administered?) Apply a constraint to the table so that test_date cannot be null.

SQL Subqueries

In lecture, we touched on the concept of a SQL subquery – where the result of a SELECT statement can be used as if it were, itself, a sort of ephemeral table, a list of values (ala the right side of an "IN" clause) or even a singular value in another SELECT (or, in fact, INSERT, UPDATE, etc.) SQL statement. We noted that this can be very powerful, and also can (in some cases) have an impact on query performance.

Exercises

In these exercises, we will answer some useful questions about student COVID tests using the normalized database tables we have and SQL subqueries. Note that subqueries are by no means the only way to answer these questions (and we'll see some examples of other methods in later exercises), so for this part of the lab, try to stick to using subqueries, even if they seem overly complicated or inefficient for the task at hand. Here, we're demonstrating exactly how subqueries operate.

Recall the following from previous SQL discussions:

- We can use "IN" clauses within WHERE clauses to match values against lists of values, eg., "SELECT x FROM y where a IN (1,2,3)" will find rows in y where field "a" has one of the three values 1, 2, or 3, and that subqueries can be treated as value lists in other queries.
- We can use subqueries across tables – that is, a subquery can reference one table while the query it's embedded within references another table.
- Remember that count(*) can easily return the number of rows in a result. A related trick is to use count(distinct <x>) to count the number of distinct values of <x> returned by some query, eg. "select count(distinct productid) from orders" will return the number of distinct products for which at least one order currently exists (in the orders database).

Keep those points in mind as you work through the following exercises:

11) Perform a simple query in the unnormalized table to find the number of students in the graduating class of 2022. Don't try to use a subquery of any sort – just directly query the table. If you've made no changes to the original test data, you should get back 3829 rows.
12) Now perform a simple query in the unnormalized table to find the first and last names of the students in the graduating class of 2022. Again, don't use a subquery of any sort – just use a WHERE clause in a direct query. You should get back the same number of rows (3829).
13) Now consider that any combination of a source and a WHERE clause (and possible extensions to it) can be replaced with an equivalent select statement and used as the source of another SELECT statement. That is "SELECT x FROM y" is the same (logically)

as “SELECT x FROM (SELECT x FROM y)”. The latter is almost always less performant, but the two are equivalent. Use that fact to perform the same work you did in (12) above using a simple subselect. Verify that you get back the same number of results (3829).

14) Now switch to the “orders” database and try running the following query:
    a. SELECT name FROM accounts WHERE accounted in (SELECT ordering_account from orders) AND email LIKE ‘%gmail.com’;

    What does that subquery (SELECT ordering_account from orders) retrieve? What does the whole query do?

15) Now switch back to the “normalized” database. This time, ignore the star schema table, and focus only on the students, room_assignment, majors, and tests tables. Using a strategy similar to the one in (14) above (where a subquery SELECT in one table is used to generate a list into which a value in another table can be matched) run a query to answer the question:

    What are the names of the students who have tested positive for COVID-19?

    (Hint: Note that the tests table’s “userid” is actually a foreign key that matches the “userid” field in the student table).

    How many results do you receive? Was it 1355?

    Note down the query you used and the number of results – we’ll refer back to it in a later section.

16) Now take a look at the tests table and query it for how many rows represent positive COVID tests. Do you get the same number as in (15) above?
17) Try looking at the number of distinct **userid** values in the tests table for students who tested positive for COVID. Does that number match the number of results you got in (15) above? Why might that be? (Hint: What’s the meaning of “WHERE x IN (1,1,2,2,3,3)”?
18) SQL subqueries can be nested – that is, a subquery can contain another subquery. Formulate query using a nested set of SQL subqueries to answer the following question:
    a. What are the names of students living on the first floor of their dorms (which we will define as their room_number values being between 100 and 200) have tested positive for COVID?

    (Hint: You can break this question into a series of “WHERE x in y” queries that nest with one another. Try thinking of it in terms of “the students where userid is in (list of userids where room is between 100 and 200 and userid is in (list of userids of students with positive COVID tests))” You’ll need two sub-selects and three tables to make this work. The result should be 52, by my counting, if you haven’t changed the test data.)

    Record your result count and the query you use – we’ll refer back to it in a later section in this lab.

JOIN statements

In the lecture, we noted that JOIN statements can be used to construct virtual tables by "merging" other tables in a database together. This merging or joining can be performed in three (or up to five, depending on how you count) ways:

- Inner joins – where the resulting virtual table contains one row for every row where the two joined tables intersect on some field (which may or may not be a key field, but is often a primary key in one table and a foreign key in another). The canonical syntax for an inner join statement looks like:
    - <x> INNER JOIN <y> ON x.field1 = y.field2

  Which will produce a virtual table with all the fields from <x> followed by all the fields from <y> for every row in <x> and <y> where the value of field1 in <x> matches the value of field2 in <y>. In the special case where field1 and field2 have the same name in both tables, this can be shortened to the commonly used:
    - <x> INNER JOIN <y> USING(field)

  In set theoretic terms, the inner join returns the intersection of the two sets of rows (<x> and <y>) overlapping in the match field.
- Outer joins – where the resulting virtual table contains all the rows from at least one of the joined tables, along with matching entries from the other table where they exist (and NULL values where no match exists between the tables. Outer joins come in two (or in some SQL implementations, three flavors):
    - Left outer joins - <x> LEFT JOIN <y> ON x.field1 = y.field2 – return a virtual table with all the rows from <x> and all corresponding data from rows in <y> (or NULLs)
    - Right outer joins - <x> RIGHT JOIN <y> on x.field1 = y.field2 – return a virtual table with all the rows from <y> and all corresponding data from rows in <x> (or NULLs)
    - Full outer joins (not supported in MySQL and some other RDBMS) - <x> FULL JOIN <y> on x.field1 = y.field2 – return the logical union of the left and right outer joins of <x> and <y> on the join condition.
- Cross joins – where every row in one table is matched (without any matching criteria) to every single row in the other table, producing the **cartesian product** of the two tables. The cross join syntax looks like:
    - <x> CROSS JOIN <y>

  And note that there is no criterion specified – the cross join doesn't need one.

Exercises:

In these exercises, we'll try to answer some even more sophisticated questions about COVID statuses and student relationships in our normalized student testing data. Recall the following facts about the testing data:

- Entries exist in the tests table only for students who have had tests administered
- Entries exist in the room_assignment table only for students who live on campus
- There may be more than one entry in the majors table for any student if the student has more than one major
- There may be more than one entry in the room_assignment table that refers to a given (dorm,room_number) combination, if the room has more than one inhabitant

And the following facts about SQL statements:

- GROUP BY clauses can be added to SQL statements to "roll up" results based on matching values in a field – eg "SELECT a,count(*) FROM <x> GROUP BY a" will return the distinct values of "a" in the table along with the number of rows that have that value of a.
- GROUP BY clauses can be extended with HAVING statements to restrict the groups enumerated to those that (as a group) match the HAVING statement, eg. "SELECT a FROM <x> GROUP BY a HAVING count(*) > 200" will return only those values of "a" for which the number of grouped rows is greater than 200.

Keep those facts in minds as you work through the following exercises:

19) In preparation for off-season commencement, the chair of the Sociology department would like a report generated of all the class of 2020 sociology majors that shows their first and last names, their class years, and the department name (so she can feel confident she's gotten the right result). Construct a simple SELECT using a join between two tables to produce the report for her. You should get back 511 results.
20) The chair liked your report. You're now in for more work. ☺ She writes back later in the day and notes that the Provost is concerned about COVID transmission during the Sociology mixer the night before commencement. She requests that you add COVID test results to the report for her so she can prepare for a call with the Provost. Expand the JOIN you used in (19) above to produce a similar report that includes test results. For this round, only produce a record in the report if the student has had a test taken. You should get back 125 results. (Hint: Remember that you can group a join with parentheses and then use it as the left or right side of another join – "(a join b) join c" – and remember the meaning of inner versus outer joins). Consider adding an "ORDER BY" to your query to organize the report by last name, and note if you observe any students who may have been administered multiple tests.
21) Oh joy – she really liked that. But she's concerned that she's not seeing everything she expects. She's only seeing 125 entries in the second report and wants reassurance that that's the right number. She notes that there are some students appearing more than once in the report, and that some students don't seem to have test results (even though she thought you were restricting the list to just students with tests administered). Modify the join you used in (20) above to produce a list of *all* the 2020 sociology majors that shows all the same data, but includes *all* the 2020 sociology students and

adds the test_date and test_return fields to help her understand where students have outstanding tests, where students have not had tests administered at all, and where one student may have been tested multiple times. You should get back 516 rows. (Question: Why is the number 516 and not 511 as in (19) above?) KEEP THIS QUERY HANDY – we will use it in the next exercise…

Views

In lecture, we saw that views are essentially a way to "can" useful queries and treat the results of the queries as if they were persistent tables (even though they're not).

Recall that any SELECT statement can be turned into a VIEW using a CREATE VIEW statement like:

CREATE VIEW <view_name> AS <SELECT statement>

"view_name" then looks and behaves just like any other table, but with content that's dynamically determined when the table is queried.

Exercises:

22) The Sociology chair is quite happy with her report, and asks you to generate it for her on a regular basis – once every morning and once every evening – for the remainder of the semester so she can keep apprised of any changes she needs to make to her plans for commencement in November. Obviously, you could run her a query every 12 hours for the rest of the semester, but you have other matters to attend to, and you'd prefer not to be running the same query over and over for two months. As a defensive maneuver, institutionalize the query from (21) above by creating a view named "sociology_2020_testing" that replicates it. Verify that you have the right number of rows returned into the view by doing a "SELECT count(*)" on it.
23) Two weeks pass, and you receive a call from the chair of the English department. He has talked to the Sociology chair, and was very excited to hear what you did for his colleague. He asks that you produce him a similar view with two differences: Select students with the major 'English', and include the major in the view. Create a view named 'english2020_tests' to meet his specifications. You should find that it contains 796 rows.
24) We noted that views can be used as though they were tables. In fact, you may query them like tables and even use those queries to construct further views – views of views, as it were. The English chair is not the most tech-savvy faculty member, and while he's proficient enough with SQL to perform simple "SELECT * FROM english2020_tests" queries to get what he usually needs, he's not prepared to do more sophisticated querying. He's asked by the Provost to identify soon-to-graduate English majors who may have been tested for COVID but may not yet have received their results, and asks you to help him. Construct a view named "pending_english2020_tests" from the "english2020_tests" view by SELECTing on the first view to find students whose test

results are not yet recorded. (Hint: These will be students where test_date is not null but test_returned and test_result are). Verify that the pending_english2020_tests view contains 43 rows.

Triggers and Stored Procedures

In lecture, we briefly touched on the fact that SQL triggers allow you to extend the behavior of certain SQL operations by establishing "tripwires" on those events to execute or "trigger" other things to be done whenever those events occur. The events in question may be the execution of a particular sort of SQL statement in a particular table (eg., every time an INSERT statement is run or every time a SELECT is run)

We reviewed a stored procedure and a trigger in the very simple "running_total" database the purpose of which is to log a "running total" of the values in the "entries" table every time a new entry is added, writing the log (a datestamp along with the new running total) in the "running_total" table.

Stored procedures are a very complex subject beyond the scope of today's lab, and as their syntax is highly RDBMS specific, we won't attempt to delve into them further in this section, except to note that they are global properties stored above the level of individual databases (but scoped to individual databases). As such, stored procedures are not replicated when a database is backed up – and as a result, your copies of our test databases do not contain the stored procedure from the slides. You can list stored procedures (in MySQL) with the command:

```
show procedure status
```

We will, however, note that triggers are stored as properties of **databases** rather than **tables** (they individually refer to tables, but they're considered database properties rather than table properties). That being the case, you can use the command:

```
show triggers from <database>
```

to list any active triggers, and the command:

```
drop trigger <database-name>.<trigger-name>
```

to remove a specific trigger from a database.

Exercise:

25) Use "show triggers from <database>" and "show procedure status" to verify that your running_totals database has a trigger referencing (via a CALL instruction) a non-existent

stored procedure. That being the case, remove the trigger using the "drop trigger" command outlined above.

Performance

In lecture, we noted at various points that there are often many different ways to solve the same problem with a SQL statement (or a set of SQL statements) in a given set of data, and that sometimes some of those options may be more or less performant than others. We also noted that there are some common mistakes made in designing SQL queries that can hinder performance or even cause performance issues for the database itself (and any other users of it), or in some cases result in inaccurate or undesired results. We also mentioned the use of indexes to speed up SELECTs in certain situations, and saw how one of the common mistakes can interfere with the use of indexes in otherwise more performant queries.

Recall that the common mistakes we discussed were:

- Using improper joins or using implicit joins improperly (eg. "select … from table1,table2,table3" rather than specifically joining the three tables explicitly).
- Failing to use parentheses to override default precedence orders in SQL queries.
- Failing to create needed indexes or using leading "%" and "_" wildcards in LIKE matches.
- Using overly complex queries when simpler queries would suffice
- "Fixing up" (typically improper) join results by over-using "distinct" (or "unique", depending on the RDBMS in use)

Exercises:

In the following exercises, we will execute a query that either returns an invalid result or takes appreciably longer than required due to violating one or more of the rules above, and then attempt to submit an alternative query that does what the original query appeared to try to do, but does it more efficiently and/or more correctly. We will be working in the "normalized" database again, so if you've moved to a different database, remember to "use normalized" to return to the normalized database for this set of exercises.

26) The Dean of Physical Sciences has requested that your new intern (Rupa) get him a list of all Physics and Chemistry students. Rupa comes to you, perplexed, and says that her SQL query is not behaving properly. She reports that she's using:

```
select firstname.lastname from (student join majors) where major = 'Physics' or
major = 'Chemistry';
```

and not getting the results she expects. She's tried running:

```
select count(*) from (student join majors) where major = 'Physics' or major =
'Chemistry';
```

and she's terrified of the number she sees.

Run the "select count(*)" query Rupa is using to observe what she's reporting. Debug her query and explain her error, giving her one that does what the Dean needs. (Hint: The Dean should get a list of 4237 students back if my understanding is correct).

27) The Chair of the Psychology department has requested that your other intern (Lon) give her a similar report, this one for just Psychology students graduating in 2021 or 2022. Lon is using the query:

```
select firstname,lastname from unnormalized where major = 'Psychology' and
class = 2021 or class = 2022;
```

and getting back more results than he believes he should. Explain to him why that is, and correct his query to return the proper number of results (in this case, 1081). Also, point out to him that we have deprecated the use of the star schema table, and show him a proper query against the normalized student and majors tables that accomplishes the same thing.

28) Rupa has been working with the Engineering Dean for a while to get data together about COVID tests for Engineering students. She decided to start with a subquery approach and use:

```
SELECT firstname,lastname from student where userid in
(SELECT userid from majors where major LIKE '%Engineering')
```

to get Engineering majors. Once she was confident in that result, she tried extending the approach to include a select to get the students with covid test results, and came up with:

```
SELECT firstname,lastname from student where userid in
(SELECT userid from majors where major LIKE '%Engineering'
AND userid in (select userid from tests where test_result is not null))
```

Which she was somewhat satisfied with, until she realized she needed to include the actual majors and test results in the report for the dean. She finally tried then running:

```
SELECT student.firstname, student.lastname, majors.major, tests.test_result
FROM student,majors,tests where userid in (select userid from majors where
major like '%Engineering' and userid in (select userid from tests where
test_result is not null))
```

She says she's been waiting for a response from that last query for over an hour, and

she just got an angry call from someone in Database Administration saying she had exceeded her cpu quota limit (whatever that was) for the month.

First, see if you can determine what may be causing her query to "hang". (Hint: That list of tables is an implicit join, isn't it?)

Help her come up with a more performant and less complex query that does what she needs. (Hint: You might consider whether to restructure her subqueries into a single join to get both the results and the data she needs).

Miscellany

We won't operationally address architectural and security issues in the lab today, but you're urged to look back over the lecture slides and answer the following thought questions:

- If you know that your database is replicated (fully – that is, that there are multiple copies of your data in multiple places) are you comfortable with not backing it up at all? (Hint: What happens if you type "drop table" instead of "drop constraint", or you type "drop database" instead of "drop table"?)
- Given what you've seen in the lab today, would you say that the mysqldump backups used to provide you with test data for the lab were adequate for backing up the test databases? What if there had been a number of stored procedures mixed in with the data? (Note: It *is* possible to back up stored procedures using mysqldump, but the restoration can result in more changes than you might expect).
- How might you use views to solve some of the deans' requests in the earlier lab sections? How might that be used to apply useful security restrictions in similar cases?