

Assignment 2: The Structured Query Language (100pts)

Due: Friday, Jan. 22nd 2AM PST (**morning**)

The following two sections contain formatting requirements for your homework submission. **Please read both sections; failure to follow these guidelines will result in point deductions.**

General Formatting Guidelines

All the rules about good coding style apply to SQL statements as well, because they also need to be maintained over time. Here are some of the most important ones:

- Keep lines to 80 characters or less. (We're serious about this one.)
- Always give aggregates and other computed results a clear and meaningful name when they are referenced by other parts of the query, or when they appear in a view.
- Don't wrap long SQL statements just anywhere! It's best to wrap SQL statements at the start of a major clause, or at the end of a component of a given clause. Make it easy to understand.
- Similarly, follow a uniform indentation style that clearly indicates the various parts of SQL queries. For example, if you have a large number of tables in a FROM clause, or you have a derived relation, make sure everything in the FROM clause is indented to show its extent.
- Make sure indentation is done with **spaces and not tabs**, as tabs are not necessarily the same width from machine to machine, or editor to editor.
- You are strongly encouraged to capitalize SQL keywords and use lower-case table/column names (as shown in lecture), but this is not required. Whatever you end up doing, be consistent.

We have provided a style-checking tool for you to run against your files. While you are not required to run it prior to submitting your SQL, it is highly recommended, as we will be using the same style-checker to deduct points for style errors. You can run it as follows (make sure you use Python 3 instead of Python 2):

```
# Presuming you have done "chmod u+x check.py" first:  
check.py file1.sql file2.sql ...
```

```
# If you don't want to make check.py executable:  
python3 check.py file1.sql file2.sql ...
```

Submission Formatting Requirements

We use an automated tool to perform basic testing of your SQL submissions. Your answers for the problems need to be put into various SQL files; the specific filenames will be indicated in each section. (A template is provided; you can fill it in with your answers.)

We are using CodePost for code-based homework submissions this quarter. There will be 3 different CodePost assignments to submit, corresponding to the 3 parts in A2: Part A, A2: Part B, and A2: Part C. You can submit the file for each part on <https://codepost.io/> (you should be able to see these logged in using your @caltech.edu email). Late submissions are not accepted, and will need to be submitted as a rework.

For full credit, you also need to follow specific guidelines for naming and formatting your SQL files.

The answer for each problem must begin with a comment like this:

```
-- [Problem 1a]
```

In this case, "1a" is the number of the specific problem you are solving. Each question will specify what to put as the problem number, e.g. "5" or "6e".

Any descriptive comments should follow this annotation, for example:

```
-- [Problem 1a]
-- Comments for this problem.
```

The SQL statement can be anywhere after the problem annotation and the comment, like this:

```
-- [Problem 1a]
-- Selects all rows from t with value for a greater than 4.
SELECT * FROM t WHERE a > 4;
```

Your comment does not need to read the statement verbatim, but should have a brief English description of what the query does. Of course, you can put blank lines between any of these portions to improve readability. **However, do not put any code before the first "-- [Problem . .]" annotation! Any code before this annotation will not be run and may cause the auto-grader to mark all your responses as incorrect!**

Using MySQL

This term we will be using MySQL 8.0 - you may use either **mysql** on the command line or a graphical tool like [phpMyAdmin](#) or [MySQL Workbench](#), which may be of interest to students looking to explore features in such database software with a nice GUI. However, note that your solutions will be tested using **mysql** on the command line. Also note that most graphical tools have features to manage databases and tables using the GUI, but you should solve this assignment using the mysql console or a SQL query tab, and save your answers as **.sql** files. **DO NOT export any SQL generated by software.** If you don't know what that means, don't worry about it.

You are encouraged to consult the MySQL online documentation¹ if you have any questions about syntax or functionality. You will be exploring many of MySQL's features, so it is a good idea to become familiar with the manual now. Also note that there are many interesting properties/features of MySQL, and many nuances. You are encouraged to ask questions in lectures for this unit as you get started with MySQL.

Graphical Tools (Optional): MySQL Workbench or phpMyAdmin

The MySQL Workbench and phpMyAdmin are both very helpful graphical tools for working with your database. You can download the latest version of the workbench here:

<https://dev.mysql.com/downloads/workbench/> or phpMyAdmin here:
<https://www.phpmyadmin.net/>.

Once downloaded, you can create and save a connection to your database, so that it is very easy to connect and do your work.

¹ <https://dev.mysql.com/doc/refman/8.0/en/>

MySQL Console

You can also use the MySQL console client which is available when you install MySQL on your machine. This will be used often in lectures. We have written a guide on installing and using MySQL on the command line which you can find [here](#). For students on Windows, we recommend installing Ubuntu for Windows - we have written a separate guide for using Ubuntu WSL with MySQL [here](#).

If you are working at the console, you can save the details of your interactions using the MySQL client's "tee" feature, and then load the results into an editor later. Just run the client as follows:

```
mysql [other arguments] --tee=hw2out.txt
```

As you use the MySQL client, every single command you type, and its output, will be stored into the file `hw2out.txt`.

Setting Configurations

MySQL has various default configurations set when installed. Sometimes, we want to change these configurations depending on restrictions we would like to have on a hosted server or locally. Since you are using MySQL locally, you will need to set the following configuration settings on your own, which will be important to ensure you don't run into issues with MySQL's non-standard grouping/aggregation support or its non-standard support of double-quoted strings. *Before* running any of the provided .sql files, execute the following command in the mysql prompt:

```
mysql> SET GLOBAL sql_mode = 'ANSI,ONLY_FULL_GROUP_BY,STRICT_ALL_TABLES';
```

You can check the current global configurations using the following:

```
SELECT @@GLOBAL.sql_mode;
```

You should see the following result:

```
+-----+
| @@GLOBAL.sql_mode |
+-----+
| REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,ONLY_FULL_GROUP_BY,ANSI,STRICT_ALL_TABLES |
+-----+
```

If you don't, feel free to ask on Discord. You can find more information about configurations in MySQL [here](#). You can similarly set the mode with the query tab in MySQL Workbench or phpMyAdmin.

Setting Up Database Schemas

Once you have connected to the database server using your own username and password, you can load and run the script specified in each section to create the schema for that part of the assignment.

- If you are using the command-line client then use the **source** command, as shown below for the setup file for Part A. Note that this isn't an actual SQL statement; it is simply a command the client provides for importing SQL files. Use the **help** command to learn about **source** and the other client commands you can use.

```
mysql> source make-university.sql;
```

- If you are using the MySQL Workbench, you can choose “Open SQL Script...” from the File menu, load the file, and then run it from the workbench. For phpMyAdmin, you can use the import option to load scripts (after creating a database on the left menu).

If the file runs correctly, you should see no errors. You may see a few warnings for the very first commands; these are because the SQL schema file will try to drop tables if they already exist; if they don't exist, you get a little warning.

You can verify that these steps have completed successfully by running these simple commands:

```
SHOW TABLES ;  
SHOW TABLE STATUS ;
```

(MySQL Workbench and phpMyAdmin have graphical representations of the database schema, so you don't need to do this with the graphical UI.)

The **SHOW** commands are very useful for finding out what is in a database schema. Look at the **SHOW TABLE STATUS** output in particular; it shows some of the underlying details of the tables in your database. For example, we can see that the database is using the InnoDB storage engine, which provides us with both transactions and referential integrity. (The default MyISAM storage engine provides neither of these important capabilities, but it is much faster and is frequently used for loading and analyzing very large datasets in MySQL.)

Alright, now to jump in to use SQL!

Part A: Book Problems (40 points)

These problems are based on those in the “Exercises” section of Chapter 3 of the textbook, with a few changes to expose you to more SQL features. These problems require you to write SQL statements to perform various queries or modifications to a database. Make sure to follow good SQL coding style. Make sure you adhere only to features supported by [MySQL 8.0](#).

The schema for the university database is in **make-university.sql**.

The schema for the library database is in **make-library.sql**.

All of your work for this problem will go into the `bookproblems.sql` file.

Problems:

1. [\[Exercise 3.11\]](#) (*Parts a-b are worth 3 points, c-d are worth 2 points; total is 10 points.*)
Write the following queries in SQL, using the university schema (**described at the [end of this document](#)**).
 - a) Find the names of all students who have taken at least one Comp. Sci. course; make sure there are no duplicate names in the result.
 - b) For each department, find the maximum salary of instructors in that department. You may assume that every department has at least one instructor.
 - c) Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query. You should actually include the query from part c; you can't just pretend it has a name. Do not use the **WITH** clause for this query.

- d) The **WITH** clause can be used to simplify SQL queries that perform aggregates of aggregates, or that use the same subquery in different parts of a SQL statement. This functionality is frequently also called “Common Table Expressions,” or CTEs for short; it is like being able to define a view for use within a single statement. Rewrite your answer from part c using a **WITH** clause to perform the innermost aggregation. For details on the **WITH** clause, refer to the [MySQL Reference Manual](#).

2. [Exercise 3.12] (Parts a-b are worth 2 points, c-f are worth 3 points; total is 16 points.)

- a) Create a new course “CS-001”, titled “Weekly Seminar”, with 3 credits. The course ID is “CS-001”, which is stored as a string.
- b) Create a section of this course in Winter 2021, with *sec_id* of 1.
- c) Enroll every student in the Comp. Sci. department in the above section. Don't forget the very useful **INSERT . . . SELECT** statement.
- d) Delete enrollments in the above section where the student's name is Chavez. You would normally also want to update the student's *tot_cred* “total credits” value, but since CS-001 is worth 0 credits, just concentrate on removing the appropriate rows from the *takes* table, and ignore the other part.
- e) Delete the course CS-001. What will happen if you run this delete statement without first deleting offerings (sections) of this course?
- f) Delete all *takes* tuples corresponding to any section of any course with the word “database” as a part of the title; ignore case when matching the word with the title.

3. [Exercise 3.21] (Part a is worth 2 points; b-e are worth 3 points; total is 14 points.)

Consider the following library database schema:

member (*memb_no*, *name*, *age*)
book (*isbn*, *title*, *authors*, *publisher*)
borrowed (*memb_no*, *isbn*, *date*)

- a) Retrieve the names of members who have borrowed any book published by “McGraw-Hill”.
- b) Retrieve the names of members who have borrowed all books published by “McGraw-Hill”.
- c) For each publisher, retrieve the names of members who have borrowed more than five books of that publisher.

- d) Compute the average number of books borrowed per member. Do not use the **WITH** clause in your answer. Take into account that if a member does not borrow any books, then that member does not appear in the *borrowed* relation at all. To be more specific, members who borrow no books should bring down the average, as you would expect, since the number of books they have borrowed is 0.
- e) Rewrite your answer for part d using a **WITH** clause.

Part B: Relational Algebra and SQL (14 points)

In this problem you will translate relational algebra expressions into corresponding SQL queries. Note that these are not just fragments of SQL; you should write a complete SQL query that would produce the exact same results as the corresponding relational algebra expression.

Write your solutions in a file called `algebra.sql`.

Given two relation schemas $R = (A, B, C)$, and $S = (D, E, F)$, and two relations $r(R)$ and $s(S)$, write SQL queries that produce identical results to the following relational algebra expressions. Recall that r and s are sets of tuples, not multisets, and the results of these operations should also be sets of tuples. Therefore you should eliminate duplicates where necessary, but only do this where it's actually necessary! Feel free to explain your rationale in the comments.

1. $\Pi_A(r)$
2. $\sigma_{B=42}(r)$
3. $r \times s$
4. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

For these problems, $R = (A, B, C)$ as before, and let there be two relations $r_1(R)$ and $r_2(R)$. Write complete SQL queries (not just fragments) that produce results identical to these relational algebra expressions. Again, consider that the inputs are sets of tuples; results must also be sets of tuples. Feel free to use any standard-compliant SQL. (MySQL doesn't support all of these operations.)

5. $r_1 \cup r_2$
6. $r_1 \cap r_2$
7. $r_1 - r_2$

(Scoring: Each part is worth 2 points, for a total of 14 points.)

Part C: MySQL Database Exercises (46 points)

The remaining exercises in this assignment use the SQL schema for the bank database that we have discussed so far. **Note that the schema in many of the lecture slides is different from the schema in the SQL file, so review the SQL file before trying these exercises.**

All of your work for this part will be done in the `banking.sql` file.

The banking schema is provided in the file `make-banking.sql`. For reference, here is the banking database schema:

```
branch (branch_name, branch_city, assets)
customer (customer_name, customer_street, customer_city)
loan (loan_number, branch_name, amount)
borrower (customer_name, loan_number)
account (account_number, branch_name, balance)
depositor (customer_name, account_number)
```

Problem 0: Warm-up (1 point each, 2 points)

In this assignment, we are practicing writing SQL queries using provided SQL datasets with table schemas given to you. But when designing your own tables, it is important to consider some of the trade-offs with the domains you are specifying and be able to justify your design decisions (you'll get more practice with this in upcoming assignments). For the following questions, write your answers as SQL comments following the appropriate question label.

1. The *balance* attribute in *account* is defined as *NUMERIC*, which defines exact numbers. We can also specify floating-point values in SQL with *FLOAT* or *DOUBLE*. What could go wrong if we were to use *FLOAT* or *DOUBLE* instead of *NUMERIC* to represent bank account balances?
2. Identify one attribute in the *account* relation which could reasonably be represented with a type different than *VARCHAR*. What would the type be, and what would some advantages be with this alternative?

Problem 1 (5 parts, 3 points per part; total is 15 points)

Here are some warm-up exercises for you. Write a SQL query for each of these questions. Make sure your query runs against the database and generates a correct result.

- a) Retrieve the loan-numbers and amounts of loans with amounts of at least \$1000, and at most \$2000.
- b) Retrieve the loan-number and amount of all loans owned by Smith. Order the results by increasing loan number.
- c) Retrieve the city of the branch where account A-446 is open.
- d) Retrieve the customer name, account number, branch name, and balance, of accounts owned by customers whose names start with "J". Order the results by customer name in increasing alphabetical order..
- e) Retrieve the names of all customers with more than five bank accounts.

Problem 2 (6 parts, 4 points each; 24 points total)

Here are some more challenging problems to try against the banking database.

- a) Generate a list of all cities that customers live in, where there is no bank branch in that city. Make sure that the results are distinct; no city should appear twice. Also, sort the cities in increasing alphabetical order.
- b) Are there any customers who have neither an account nor a loan? Write a SQL query that reports the name of any customers that have neither an account nor a loan. Note that MySQL does not support the **EXCEPT** operator! But there is more than one way...
- c) The bank decides to promote its branches located in the city of Horseneck, so it wants to make a \$75 gift-deposit into all accounts held at branches in the city of Horseneck. Write the SQL **UPDATE** command for performing this operation.

Do not use MySQL-specific extensions for this **UPDATE** operation. You may only use the standard “**UPDATE tblname SET ... WHERE ...**” form, where you can only specify one table in the **UPDATE** clause, and all references to other tables must appear in the **WHERE** clause.

- d) MySQL also supports a non-standard, multiple table version of **UPDATE**, of the form:

UPDATE tbl1, tbl2, ... SET ... WHERE ...

In this form, you can refer to columns from any of the specified tables in the **SET** and **WHERE** clauses, and MySQL will figure out what to update from what you write. Write another answer to part c, using this syntax. Note that you can also give the tables in the **UPDATE** clause aliases, as usual.

- e) Retrieve all details (*account_number*, *branch_name*, *balance*) for the largest account at each branch. Implement this query as a join against a derived relation in the **FROM** clause.
- f) Implement the same query as in the previous problem, this time using an **IN** predicate with multiple columns, e.g. “**(branch_name, balance) IN ...**”

Problem 3: Rank (5 points)

A *rank* of a collection of records assigns a 1 to the top-most value (it is in 1st place), a 2 to the second highest value (2nd place), and so forth. If multiple records have the same value, they are assigned the same rank, but there is a corresponding gap to the next rank value. For example, if you had the following values:

- 350
- 190
- 470
- 350

Their rank would be as follows:

- 1: 470
- 2: 350
- 2: 350
- 4: 190

In particular, note that no value has a rank of 3, because two values have a rank of 2.

If a value has a rank of 2, this means there is one other value that is greater. If a value has a rank of 4, there are three other values that are greater. Given this, it should be evident that you need to compare the set of records against itself in order to properly compute the rank.

Compute the rank of all bank branches, based on the amount of assets that each branch holds. The result schema should be *(branch_name, assets, rank)*, where the *rank* value is the rank of the branch. Order the results by decreasing rank, and secondarily by branch name (ascending), in case there are ties.

Hint 1: There are several approaches you can use to compute this result, but all of them will come down to the same general theme: joining the branches table against itself with a non-equality condition. We call these “non-equijoins.”

Hint 2: There are two branches with the same assets; make sure the rank values are correct for those branches and for the adjacent branches in the ranking.

Feedback Survey (+3 bonus points)

Complete the feedback survey for this assignment on Canvas, and 3 points will be added to your score (max of 100/100).

University Database

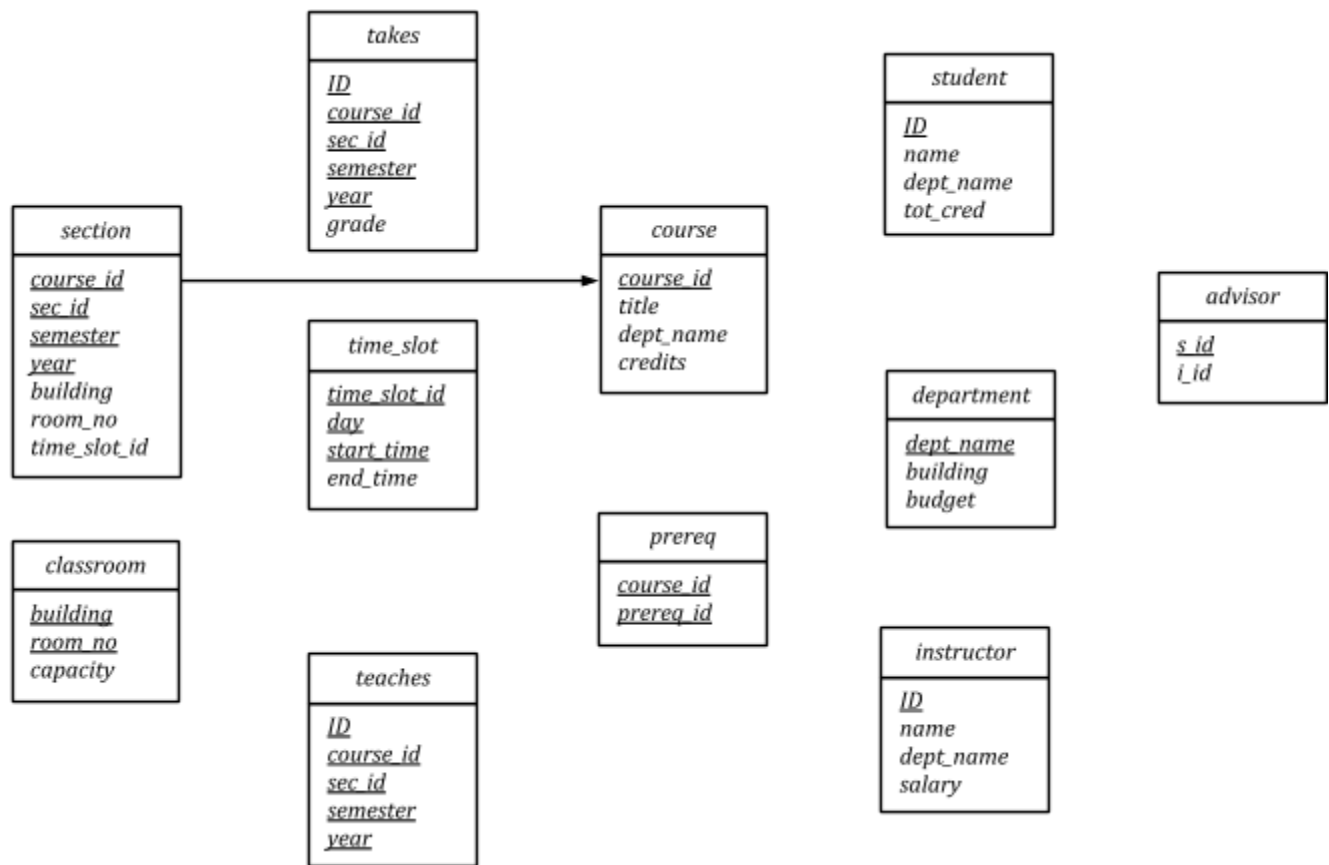


Figure 2.8 Schema diagram for the university database.

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)

Figure 2.9 Schema of the university database.

```
create table department (  
    dep_name    varchar (20),  
    building    varchar (15),  
    budget      numeric (12,2),  
    primary key (dept_name));  
  
create table course (  
    course_id    varchar (7),  
    title        varchar (50),  
    dept_name    varchar (20),  
    credits      numeric (2,0),  
    primary key (course_id),  
    foreign key (dept_name) references department);  
  
create table instructor (  
    ID           varchar (5),  
    name         varchar (20) not null,  
    dept_name    varchar (20),  
    salary       numeric (8,2),  
    primary key (ID),  
    foreign key (dept_name) references department);  
  
create table section (  
    course_id    varchar (8),  
    sec_id       varchar (8),  
    semester     varchar (6),  
    year         numeric (4,0),  
    building     varchar (15),  
    room_number  varchar (7),  
    time_slot_id varchar (4),  
    primary key (course_id, sec_id, semester, year),  
    foreign key (course_id) references course);  
  
create table teaches (  
    ID           varchar (5),  
    course_id    varchar (8),  
    sec_id       varchar (8),  
    semester     varchar (6),  
    year         numeric (4,0),  
    primary key (ID, course_id, sec_id, semester, year),  
    foreign key (course_id, sec_id, semester, year) references section,  
    foreign key (ID) references instructor);
```

Figure 3.1 SQL data definition for part of the university database.