4 SQL 2 (The Sequel)

Lab Objective: *Since SQL databases contain multiple tables, retrieving information about the data can be complicated. In this lab we discuss joins, grouping, and other advanced SQL query concepts to facilitate rapid data retrieval.*

We will use the following database as an example throughout this lab, found in `students.db`.

MajorID	MajorName
1	Math
2	Science
3	Writing
4	Art

(a) MajorInfo

CourseID	CourseName
1	Calculus
2	English
3	Pottery
4	History

(b) CourseInfo

StudentID	StudentName	MajorID
401767594	Michelle Fernandez	1
678665086	Gilbert Chapman	NULL
553725811	Roberta Cook	2
886308195	Rene Cross	3
103066521	Cameron Kim	4
821568627	Mercedes Hall	NULL
206208438	Kristopher Tran	2
341324754	Cassandra Holland	1
262019426	Alfonso Phelps	NULL
622665098	Sammy Burke	2

(c) StudentInfo

StudentID	CourseID	Grade
401767594	4	C
401767594	3	B–
678665086	4	A+
678665086	3	A+
553725811	2	C
678665086	1	B
886308195	1	A
103066521	2	C
103066521	3	C–
821568627	4	D
821568627	2	A+
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D–
341324754	1	A–
103066521	4	A
262019426	2	B
262019426	3	C
622665098	1	A
622665098	2	A–

(d) StudentGrades

Table 4.1: Student database.

Joining Tables

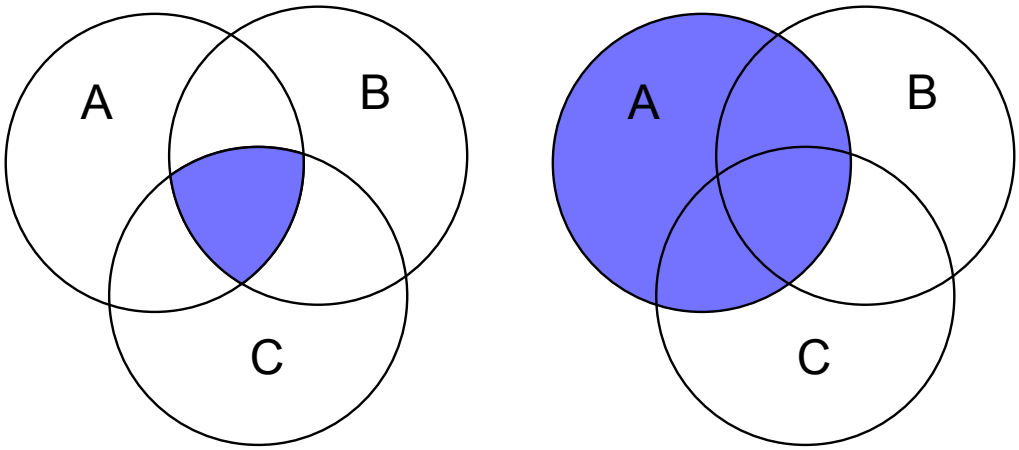
A *join* combines rows from different tables in a database based on common attributes. In other words, a join operation creates a new, temporary table containing data from 2 or more existing tables. Join commands in SQLite have the following general syntax.

```
SELECT <alias.column, ...>
    FROM <table> AS <alias> JOIN <table> AS <alias>, ...
    ON <alias.column> == <alias.column>, ...
    WHERE <condition>;
```

The `ON` clause tells the query how to join tables together. Typically if there are N tables being joined together, there should be $N - 1$ conditions in the `ON` clause.

Inner Joins

An *inner join* creates a temporary table with the rows that have exact matches on the attribute(s) specified in the `ON` clause. Inner joins **intersect** two or more tables, as in Figure 4.1a.



(a) An inner join of A, B, and C.

(b) A left outer join of A with B and C.

Figure 4.1

For example, Table 4.1c (`StudentInfo`) and Table 4.1a (`MajorInfo`) both have a `MajorID` column, so the tables can be joined by pairing rows that have the same `MajorID`. Such a join temporarily creates the following table.

StudentID	StudentName	MajorID	MajorID	MajorName
401767594	Michelle Fernandez	1	1	Math
553725811	Roberta Cook	2	2	Science
886308195	Rene Cross	3	3	Writing
103066521	Cameron Kim	4	4	Art
206208438	Kristopher Tran	2	2	Science
341324754	Cassandra Holland	1	1	Math
622665098	Sammy Burke	2	2	Science

Table 4.2: An inner join of `StudentInfo` and `MajorInfo` on `MajorID`.

Notice that this table is missing the rows where `MajorID` was `NULL` in the `StudentInfo` table. This is because there was no match for `NULL` in the `MajorID` column of the `MajorInfo` table, so the inner join throws those rows away.

Because joins deal with multiple tables at once, it is important to assign table aliases with the `AS` command. Join statements can also be supplemented with `WHERE` clauses like regular queries.

```
>>> import sqlite3 as sql
>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

>>> cur.execute("SELECT * "
...             "FROM StudentInfo AS SI INNER JOIN MajorInfo AS MI "
...             "ON SI.MajorID == MI.MajorID;").fetchall()
[(401767594, 'Michelle Fernandez', 1, 1, 'Math'),
 (553725811, 'Roberta Cook', 2, 2, 'Science'),
 (886308195, 'Rene Cross', 3, 3, 'Writing'),
 (103066521, 'Cameron Kim', 4, 4, 'Art'),
 (206208438, 'Kristopher Tran', 2, 2, 'Science'),
 (341324754, 'Cassandra Holland', 1, 1, 'Math'),
 (622665098, 'Sammy Burke', 2, 2, 'Science')]

# Select the names and ID numbers of the math majors.
>>> cur.execute("SELECT SI.StudentName, SI.StudentID "
...             "FROM StudentInfo AS SI INNER JOIN MajorInfo AS MI "
...             "ON SI.MajorID == MI.MajorID "
...             "WHERE MI.MajorName == 'Math';").fetchall()
[('Cassandra Holland', 341324754), ('Michelle Fernandez', 401767594)]
```

Problem 1. Write a function that accepts the name of a database file. Assuming the database to be in the format of Tables 4.1a–4.1d, query the database for the list of the names of students who have a B grade in any course (not a B– or a B+). Be sure to return a list of strings, not a list of tuples of strings.

Outer Joins

A *left outer join*, sometimes called a *left join*, creates a temporary table with **all** of the rows from the first (left-most) table, and all the "matched" rows on the given attribute(s) from the other relations. Rows from the left table that don't match up with the columns from the other tables are supplemented with `NULL` values to fill extra columns. Compare the following table and code to Table 4.2.

StudentID	StudentName	MajorID	MajorID	MajorName
401767594	Michelle Fernandez	1	1	Math
678665086	Gilbert Chapman	NULL	NULL	NULL
553725811	Roberta Cook	2	2	Science
886308195	Rene Cross	3	3	Writing
103066521	Cameron Kim	4	4	Art
821568627	Mercedes Hall	NULL	NULL	NULL
206208438	Kristopher Tran	2	2	Science
341324754	Cassandra Holland	1	1	Math
262019426	Alfonso Phelps	NULL	NULL	NULL
622665098	Sammy Burke	2	2	Science

Table 4.3: A left outer join of `StudentInfo` and `MajorInfo` on `MajorID`.

```
>>> cur.execute("SELECT * "
...             "FROM StudentInfo AS SI LEFT OUTER JOIN MajorInfo AS MI "
...             "ON SI.MajorID == MI.MajorID;").fetchall()
[(401767594, 'Michelle Fernandez', 1, 1, 'Math'),
 (678665086, 'Gilbert Chapman', None, None, None),
 (553725811, 'Roberta Cook', 2, 2, 'Science'),
 (886308195, 'Rene Cross', 3, 3, 'Writing'),
 (103066521, 'Cameron Kim', 4, 4, 'Art'),
 (821568627, 'Mercedes Hall', None, None, None),
 (206208438, 'Kristopher Tran', 2, 2, 'Science'),
 (341324754, 'Cassandra Holland', 1, 1, 'Math'),
 (262019426, 'Alfonso Phelps', None, None, None),
 (622665098, 'Sammy Burke', 2, 2, 'Science')]
```

Some flavors of SQL also support the `RIGHT OUTER JOIN` command, but `sqlite3` does not recognize the command since `T1 RIGHT OUTER JOIN T2` is equivalent to `T2 LEFT OUTER JOIN T1`.

Joining Multiple Tables

Complicated queries often join several different relations. If the same kind of join is being used, the relations and conditional statements can be put in list form. For example, the following code selects courses that Kristopher Tran has taken, and the grades that he got in those courses, by joining three tables together. Note that 2 conditions are required in the `ON` clause in this case.

```
>>> cur.execute("SELECT CI.CourseName, SG.Grade "
...             "FROM StudentInfo AS SI "           # Join 3 tables.
...                 "INNER JOIN CourseInfo AS CI, StudentGrades SG "
...             "ON SI.StudentID==SG.StudentID AND CI.CourseID==SG.CourseID "
...             "WHERE SI.StudentName == 'Kristopher Tran';").fetchall()
[('Calculus', 'C+'), ('English', 'A')]
```

To use different kinds of joins in a single query, append one join statement after another. The join closest to the beginning of the statement is executed first, creating a temporary table, and the next join attempts to operate on that table. The following example performs an additional join on Table 4.3 to find the name and major of every student who got a C in a class.

```
# Do an inner join on the results of the left outer join.
>>> cur.execute("SELECT SI.StudentName, MI.MajorName "
...              "FROM StudentInfo AS SI LEFT OUTER JOIN MajorInfo AS MI "
...              "ON SI.MajorID == MI.MajorID "
...              "INNER JOIN StudentGrades AS SG "
...              "ON SI.StudentID == SG.StudentID "
...              "WHERE SG.Grade == 'C';").fetchall()
[('Michelle Fernandez', 'Math'),
 ('Roberta Cook', 'Science'),
 ('Cameron Kim', 'Art'),
 ('Alfonso Phelps', None)]
```

In this last example, note carefully that Alfonso Phelps would have been excluded from the result set if an inner join was performed first instead of an outer join (since he lacks a major).

Problem 2. Write a function that accepts the name of a database file. Query the database for all tuples of the form `(Name, MajorName, Grade)` where `Name` is a student's name and `Grade` is their grade in Calculus. Only include results for students that are actually taking Calculus, but be careful not to exclude students who haven't declared a major.

Grouping Data

Many data sets can be naturally sorted into groups. The `GROUP BY` command gathers rows from a table and groups them by a certain attribute. The groups are then combined by one of the *aggregate functions* `AVG()`, `MIN()`, `MAX()`, `SUM()`, or `COUNT()`. Each of these functions accepts the name of the column to be operated on. Note that the first four of these require the column to hold numerical data. Since our database has no such data, we will delay examples of using the functions other than `COUNT()` until later.

The following code groups the rows in Table 4.1d by `studentID` and counts the number of entries in each group.

```
>>> cur.execute("SELECT StudentID, COUNT(*) "   # * means "all of the rows".
...              "FROM StudentGrades "
...              "GROUP BY StudentID;").fetchall()
[(103066521, 3),
 (206208438, 2),
 (262019426, 2),
 (341324754, 2),
 (401767594, 2),
 (553725811, 1),
 (622665098, 2),
 (678665086, 3),
 (821568627, 3),
 (886308195, 1)]
```

`GROUP BY` can also be used in conjunction with joins. The join creates a temporary table like Tables 4.2 or 4.3, the results of which can then be grouped.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) "
...             "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...             "ON SG.StudentID == SI.StudentID "
...             "GROUP BY SG.StudentID;").fetchall()
[('Cameron Kim', 3),
 ('Kristopher Tran', 2),
 ('Alfonso Phelps', 2),
 ('Cassandra Holland', 2),
 ('Michelle Fernandez', 2),
 ('Roberta Cook', 1),
 ('Sammy Burke', 2),
 ('Gilbert Chapman', 3),
 ('Mercedes Hall', 3),
 ('Rene Cross', 1)]
```

Just like the `WHERE` clause chooses rows in a relation, the `HAVING` clause chooses groups from the result of a `GROUP BY` based on some criteria related to the groupings. For this particular command, it is often useful (but not always necessary) to create an alias for the columns of the result set with the `AS` operator. For instance, the result set of the previous example can be filtered down to only contain students who are taking 3 courses.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) as num_courses "   # Alias.
...             "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...             "ON SG.StudentID == SI.StudentID "
...             "GROUP BY SG.StudentID "
...             "HAVING num_courses == 3;").fetchall()   # Refer to alias later↩
    .
[('Cameron Kim', 3), ('Gilbert Chapman', 3), ('Mercedes Hall', 3)]

# Alternatively, get just the student names.
>>> cur.execute("SELECT SI.StudentName "                             # No alias.
...             "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...             "ON SG.StudentID == SI.StudentID "
...             "GROUP BY SG.StudentID "
...             "HAVING COUNT(*) == 3;").fetchall()
[('Cameron Kim',), ('Gilbert Chapman',), ('Mercedes Hall',)]
```

Other Miscellaneous Commands

Ordering Result Sets

The `ORDER BY` command sorts a result set by one or more attributes. Sorting can be done in ascending or descending order with `ASC` or `DESC`, respectively. This is always the very last statement in a query.

```
>>> cur.execute("SELECT SI.StudentName, COUNT(*) AS num_courses "   # Alias.
```

```
...             "FROM StudentGrades AS SG INNER JOIN StudentInfo AS SI "
...             "ON SG.StudentID == SI.StudentID "
...             "GROUP BY SG.StudentID "
...             "ORDER BY num_courses DESC, SI.StudentName ASC;").fetchall()
[('Cameron Kim', 3),                # The results are now ordered by the
 ('Gilbert Chapman', 3),            # number of courses each student is in,
 ('Mercedes Hall', 3),              # then alphabetically by student name.
 ('Alfonso Phelps', 2),
 ('Cassandra Holland', 2),
 ('Kristopher Tran', 2),
 ('Michelle Fernandez', 2),
 ('Sammy Burke', 2),
 ('Rene Cross', 1),
 ('Roberta Cook', 1)]
```

Problem 3. Write a function that accepts a database file. Query the given database for tuples of the form `(MajorName, N)` where `N` is the number of students in the specified major. Sort the results in descending order by the count `N`, and then in alphabetic order by MajorName. Include **Null** majors.

Searching Text with Wildcards

The `LIKE` operator within a `WHERE` clause matches patterns in a `TEXT` column. The special characters `%` and `_` and called *wildcards* that match any number of characters or a single character, respectively. For instance, `%Z_` matches any string of characters ending in a Z then another character, and `%i%` matches any string containing the letter i.

```
>>> results = cur.execute("SELECT StudentName FROM StudentInfo "
...                       "WHERE StudentName LIKE '%i%';").fetchall()
>>> [r[0] for r in results]
['Michelle Fernandez', 'Gilbert Chapman', 'Cameron Kim', 'Kristopher Tran']
```

Case Expressions

A case expression maps the values in a column using boolean logic. There are two forms of a case expression: simple and searched. A *simple case expression* matches and replaces specified attributes.

```
# Replace the values MajorID with new custom values.
>>> cur.execute("SELECT StudentName, CASE MajorID "
...                 "WHEN 1 THEN 'Mathematics' "
...                 "WHEN 2 THEN 'Soft Science' "
...                 "WHEN 3 THEN 'Writing and Editing' "
...                 "WHEN 4 THEN 'Fine Arts' "
...                 "ELSE 'Undeclared' END "
...             "FROM StudentInfo "
```

```
...             "ORDER BY StudentName ASC;").fetchall()
[('Alfonso Phelps', 'Undeclared'),
 ('Cameron Kim', 'Fine Arts'),
 ('Cassandra Holland', 'Mathematics'),
 ('Gilbert Chapman', 'Undeclared'),
 ('Kristopher Tran', 'Soft Science'),
 ('Mercedes Hall', 'Undeclared'),
 ('Michelle Fernandez', 'Mathematics'),
 ('Rene Cross', 'Writing and Editing'),
 ('Roberta Cook', 'Soft Science'),
 ('Sammy Burke', 'Soft Science')]
```

A *searched case expression* involves using a boolean expression at each step, instead of listing all of the possible values for an attribute.

```
# Change NULL values in MajorID to 'Undeclared' and non-NULL to 'Declared'.
>>> cur.execute("SELECT StudentName, CASE "
...                 "WHEN MajorID IS NULL THEN 'Undeclared' "
...                 "ELSE 'Declared' END "
...             "FROM StudentInfo "
...             "ORDER BY StudentName ASC;").fetchall()
[('Alfonso Phelps', 'Undeclared'),
 ('Cameron Kim', 'Declared'),
 ('Cassandra Holland', 'Declared'),
 ('Gilbert Chapman', 'Undeclared'),
 ('Kristopher Tran', 'Declared'),
 ('Mercedes Hall', 'Undeclared'),
 ('Michelle Fernandez', 'Declared'),
 ('Rene Cross', 'Declared'),
 ('Roberta Cook', 'Declared'),
 ('Sammy Burke', 'Declared')]
```

Chaining Queries

The result set of any SQL query is really just another table with data from the original database. Separate queries can be made from result sets by enclosing the entire query in parentheses. For these sorts of operations, it is very important to carefully label the columns resulting from a subquery.

```
# Count how many declared and undeclared majors there are
# The subquery changes NULL values in MajorID to 'Undeclared' and
#    non-NULL to 'Declared'.
>>> cur.execute("SELECT majorstatus, COUNT(*) AS majorcount "
...             "FROM ( "                                    # Begin subquery.
...                 "SELECT StudentName, CASE "
...                 "WHEN MajorID IS NULL THEN 'Undeclared' "
...                 "ELSE 'Declared' END AS majorstatus "
...                 "FROM StudentInfo) "                     # End subquery.
...             "GROUP BY majorstatus "
```

```
...             "ORDER BY majorcount DESC;").fetchall()
[('Declared', 7), ('Undeclared', 3)]
```

Subqueries can also be joined with other tables, as in the following example. Note also that a subquery can be used to create numerical data out of non-numerical data, which can then be passed into any of the aggregate functions.

```
# Find the proportion of classes each student has an A+, A, or A- in.
# The inner query creates a column 'gradeisa' which is 1 if the student's grade
#    is A+, A, or A-, and 0 otherwise.
>>> cur.execute("SELECT SI.StudentName, AVG(SG.gradeisa) "
...             "FROM ("
...                 "SELECT StudentID, CASE Grade "
...                     "WHEN 'A+' THEN 1 "
...                     "WHEN 'A' THEN 1 "
...                     "WHEN 'A-' THEN 1 "
...                     "ELSE 0 END AS gradeisa "
...                 "FROM StudentGrades) AS SG "
...             "INNER JOIN StudentInfo AS SI "
...             "ON SG.StudentID == SI.StudentID "
...             "GROUP BY SG.StudentID;").fetchall()
[('Cameron Kim', 0.3333333333333333),
 ('Kristopher Tran', 0.5),
 ('Alfonso Phelps', 0.0),
 ('Cassandra Holland', 0.5),
 ('Michelle Fernandez', 0.0),
 ('Roberta Cook', 0.0),
 ('Sammy Burke', 1.0),
 ('Gilbert Chapman', 0.6666666666666666),
 ('Mercedes Hall', 0.3333333333333333),
 ('Rene Cross', 1.0)]
```

Problem 4. Write a function that accepts the name of a database file. Query the database for tuples of the form `(StudentName, N, GPA)` where `N` is the number of courses that the specified student is enrolled in and `GPA` is their grade point average based on the following point system:

A+, A = 4.0	B = 3.0	C = 2.0	D = 1.0
A− = 3.7	B− = 2.7	C− = 1.7	D− = 0.7
B+ = 3.4	C+ = 2.4	D+ = 1.4	

Order the results from greatest GPA to least.

Problem 5. The file `mystery_database.db` contains 4 tables called `table_1`, `table_2`, `table_3`, and `table_4` which contain information on over 5000 subjects. Hidden within these subjects is an obvious outlier. Use what you've learned about SQL to identify the outlier in this database. Return the outlier's name, ID number, eye color, and height as a list.
Hint: you may find that joining the tables is more difficult than it's worth; instead, try finding one clue at a time. Most of these subjects lived a long time ago in a galaxy far, far away... so a good place to start might be to find a subject who doesn't meet that criteria. Also, recall that the following commands can be used to get the column names of a specified table.

```
>>> with sql.connect("database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("SELECT * FROM specified_table;")
...     print([d[0] for d in cur.description])
...
['column_1', 'column_2', 'column_3']
```