Lab 5

SQL 2 (The SQL Sequel)

Lab Objective: *Learn more of the advanced and specialized features of SQL.*

Database Normalization

Normalizing a database is the process of organizing tables and columns to minimize the amount of redundant information in the database. For example, a non-normalized database might have a table that stores customer contact information and a table that contains all of the products a company has sold. However, they might want to track who buys what products in case they need to contact them later. To do so, they store all the contact information of a particular buyer along with every item they purchased. Now, two tables store the customer contact information. If we needed to update a customer's phone number, we have to update two tables. While that may not be bad for small databases, larger databases would be near impossible to update correctly. The idea of normalizing a database allows us to store all customer contact information in one place in the database. All other tables that might need a customer's name, phone number, or address would reference the contact information table. When an update needs to be performed, we only need to update the contact information table. Then any table that references this information is also automatically up to date.

To properly normalize a database, we need to discuss the types of relations tables might have.

One to One

This is the simplest relation to model. A single table can be used to express this relation. The relation is between one record and at most one other record. An example of this relationship is an employee and their organization. One employee works at one organization. Another example would be that a driver's license belongs to only one person.

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

Table 5.1: students

ID	Name
1	Math
2	Science
3	Writing
4	Art

Table 5.2: fields

One to Many

This relationship and its inverse must be modeled with at least two tables. The general approach is to use a unique ID. Note that a relationship that appears one to one may actually be a one to many relationship. Many people will, therefore, use the same unique ID approach on one to one relationships too in the case it turns out to be a one to many relationship. An example of a one to many relationship would be between an department and its employees. The department would receive a unique ID and then each employee in that department would be tagged with that ID.

Many to Many

This relationship requires at least three tables. A many to many relationship can be visualized as two, separate one to many relationships. The records in each of the two tables receive a unique ID. A third table then serves as a map between IDs of table to IDs of the other table. An example of a many to many relationship is doctors and patients. One doctor can have several patients and one patient can have several doctors.

For the rest of the lab, we will be using the following tables: 5.1, 5.2, 5.3, and 5.4.

Problem 1. Classify the relations between the various records in these tables: 5.1, 5.2, 5.3, and 5.4.

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

Table 5.3: grades

ClassID	Name
1	Calculus
2	English
3	Pottery
4	History

Table 5.4: classes

Classify each relation as either one to one, one to many, or many to many. Identify the tables used in each relationship.

Note

There are instances where you would not want a completely normalized database. Whether to normalize your database depends on your specific needs. Usually, though, the decision to denormalize a database is a last-resort attempt to improve performance.

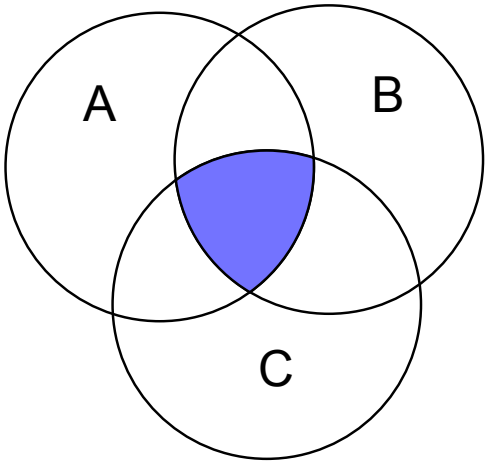

Figure 5.1: An inner joining of tables A, B, and C.

Joining Tables

We can use SQL to join two or more tables together for a query. This is a very powerful tool. SQLite supports three types of standard table joins.

Joining tables is a common practice to collect data from different parts of the database into a single table. Joins are absolutely essential in a normalized database since data is split between multiple tables.

Inner Join

This is often the default join operation in SQL. An inner join can be depicted as an intersection of two or more tables. When performing an inner join on tables, the result will only be those records that match across all tables. For example, this join will select all the students' names along with their major as long as the students.majorcode matches the majors.id. If the students.majorcode is missing or null, then they won't be selected.

```
SELECT students.name, majors.name FROM students JOIN majors ON students.majorcode↩
    =majors.id;
```

An inner join is equivalent to the following pseudo-loop in Python

```
for row_s in students:
    for row_m in majors:
        if predicates(row_s, row_m):
            yield columns(row_s, row_m)
```

Left Outer Join

A left outer join will return all relations from the left table even if they don't match any relation on the joined tables. An illustration of a left outer join is given in figure 5.2.

students.name	majors.name
Michelle Fernandez	Math
Roberta Cook	Science
Rene Cross	Writing
Cameron Kim	Art
Kristopher Tran	Science
Cassandra Holland	Math
Sammy Burke	Science

Table 5.5: An inner join of students and majors

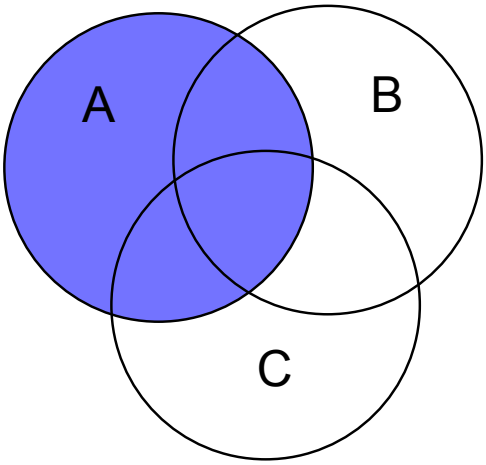

Figure 5.2: A left outer table join of A with tables B and C.

A Pythonesque loop that illustrates how to perform a left outer join is

```
for row_s in students:
    for row_m in majors:
        if predicates(row_s, row_m):
            yield columns(row_s, row_m)
        else:
            yield columns(row_s)
```

The following left outer join will result in the table shown in table 5.6.

```
SELECT students.name, majors.name FROM students LEFT OUTER JOIN majors ON ↩
    students.majorcode=majors.id;
```

Cross Join

Essentially a Cartesian product of tables. Care must be taken when using cross join because of the size of the joined table. A cross join should only be used on small tables. It matches each relation in one table with every other possible combination of relations in the joined tables. For example, if you were to run a cross join on the above join. You would get every student matched with every major. This doesn't really make much sense but can be useful for other applications. Use with caution.

students.name	majors.name
Michelle Fernandez	Math
Gilbert Chapman	None
Roberta Cook	Science
Rene Cross	Writing
Cameron Kim	Art
Mercedes Hall	None
Kristopher Tran	Science
Cassandra Holland	Math
Alfonso Phelps	None
Sammy Burke	Science

Table 5.6: A left outer join of students and majors

Function	Description
MIN()	Retrieve the smallest numeric value of a column
MAX()	Retrieve the largest numeric value of a column
SUM()	Sum the numeric values of a column
AVG()	Retrieve the average numeric value of the column
COUNT()	Retrieve the total number of matching records in a column

Table 5.7: SQL aggregation functions

Advanced Selections

Aggregate functions are useful for summarizing the data in a column. The functions can be found in 5.7.

We can count the number of students by executing the following SQL statement.

```
SELECT COUNT(*) FROM students;
```

Ordering and Grouping Relations

The `ORDER BY` keyword can be used to sort the result set by columns. We can sort in ascending order or descending order.

```
SELECT name FROM students ORDER BY name ASC;
SELECT name FROM students ORDER BY name DESC;
```

Another useful SQL keyword is the `GROUP BY` keyword. It is used along with an aggregating function to group the result set by columns.

```
SELECT grade, COUNT(studentid) FROM grades GROUP BY grade;
```

The result set is given in table 5.8.

grade	COUNT(studentid)
None	5
A	4
A+	1
A-	1
B	2
B-	1
C	3
C+	1
C-	1
D	1
D-	1

Table 5.8: Grouping of students by grade.

Problem 2. Create a database containing tables 5.1, 5.2, 5.3, and 5.4. Write a SQL query to count how many students belong to each major, including students that don't have a major. Sort your results in ascending order by name. Your result set should be table 5.9

None	3
Art	1
Math	2
Science	3
Writing	1

Table 5.9: Result set

Another important keyword is the `HAVING` keyword. This is necessary because the `WHERE` clause does not support aggregate functions. A `HAVING` clause requires a `GROUP BY` clause. The following will not work.

```
SELECT grade FROM grades GROUP BY grade WHERE COUNT(*)=1;
```

Since `COUNT` is an aggregating function, the following is required.

```
SELECT grade FROM grades GROUP BY grade HAVING COUNT(*)=1;
```

This SQL query returns all the grades that occur only once in the table. A simple way to remember the difference is *`WHERE` operates on individual records and `HAVING` operates on groups of records.*

Problem 3. Select all the students who have received grades (non-null grades) in more than two classes. How many grades did he receive?

Case Expression

A case expression allows you to temporarily modify records from a select operation. There are two forms of the case expression; simple and searched. The simple form of the expression is a match and replace on a specified column. A simple case expression is demonstrated below. This code will return the name of the student along with their majorcode. But instead of integers, the names of the majors will be returned.

```
SELECT name,
CASE majorcode
    WHEN 1 THEN 'Math'
    WHEN 2 THEN 'Science'
    WHEN 3 THEN 'Writing'
    WHEN 4 THEN 'Art'
    ELSE 'Undeclared'
    END AS major
FROM students;
```

A searched case expression using a boolean expression for the `WHEN` clauses.

```
SELECT name,
CASE
    WHEN majorcode IS NULL THEN 'Undeclared'
    ELSE majorcode
    END AS major,
CASE
    WHEN minorcode IS NULL THEN 'Undeclared'
    ELSE minorcode
    END AS minor
FROM students;
```

Problem 4. Find the overall GPA of all the students in the school. Use a regular 4.0 scale (A=4.0, B=3.0, C=2.0, D=1.0). Any pluses or minuses are dropped so an A- becomes an A.

Your result set should be one column and one row with average of all GPAs of all the students taking classes. Your solution should return a single floating point number.

Use the `ROUND()` function in SQL to round your result to the nearest hundredth.

Like Command

The SQL keyword, `LIKE`, allows us to match patterns in a column. For example,

```
SELECT name, studentid FROM students WHERE studentid LIKE '\%4';
```

will return all the students that have a student ID that ends with the digit 4. Use '\%' before the '4' to signify that there can be any number of characters before the '4.' If you only want one character, you can use an underscore. For example, if you were to search a database of words in the english dictionary and you entered the following command:

```
SELECT word FROM englishDictionary WHERE word LIKE 'i_';
```

You would get words like 'is,' 'it,' or 'in.'

Problem 5. Write a SQL statement that will find all students with a last name that begins with the letter 'C' and return their names and majors. Your returned records should be

Gilbert Chapman	None
Roberta Cook	Science
Rene Cross	Writing