Alexander Taniguchi(Wiegman)
08-174510

Homework Week 4: More Database SQL with My Courses

Logic

The assignment is to modify a database of my courses and show a few more operations on such a database. We will base this assignment off the same ‘courses.db’ that was created in the previous week. While I see that there is a worry for redundancy, this is altogether too unlikely (at least from a personal standpoint) and was not something I was giving much consideration towards. As a reminder, this is the database table as I currently have it.

```
Administrator: KAMOKU - sqlite3 courses.db

C:\Users\IceWobs\Documents\sqlite>sqlite3 courses.db
SQLite version 3.23.1 2018-04-10 17:39:29
Enter ".help" for usage hints.
sqlite> select * from KAMOKU;
Professional Ethics|2|2017|S|ITAMI|1|1
Environmental Risk Management|2|2017|A|MATSUDA|1|1
Contemporary Environmental Issues|2|2017|A|HIRONO|1|1
Simulation Methods|2|2017|S|KAKIMURA|1|0
Environmental Measurement II|2|2017|A|TAKEUCHI|1|1
Chemistry for Environmental Studies|2|2017|A|WOODWARD|1|1
Earth System Science III|2|2017|A|OKABE|1|1
Advanced Energy Science and Engineering|2|2017|S|TSUTSUMI|1|1
Food Safety and Risk Analysis|2|2017|A|YAMAKAWA|1|1
Human Population and Dynamics [Environmental Sciences]|2|2017|S|E.COHEN|1|1
Urban Planning Technology II|2|2017|A|MEGURO|1|1
Scientific Writing and Presentation Skills (a)|1|2017|S|WOODWARD|1|1
Scientific Writing and Presentation Skills (b)|1|2017|S|SHEFFERSON|1|1
Fieldwork and Case-Studies for Environmental Sciences I|1|2017|A|NAGATA|1|1
Materials Chemistry|2|2017|S|UCHIDA|1|1
Experiments in Environmental Sciences I|2|2017|S|SATO|1|1
Science, Technology, and Society|2|2017|A|GIRAUDOU|1|1
Introduction to Philosophy of Science and Technology|2|2017|A|NOBUHARA|0|1
Introduction to Applied Ethics [Science and Technology Studies]|2|2017|A|SUZUKI|0|1
Philosophy of Science and Technology IV (Seminar)|2|2017|A|ODEA|0|1
Science and Technology Studies I (Seminar)|2|2017|S|OKAMOTO|0|1
Applied Ethics III (Seminar) [Science and Technology Studies]|2|2017|S|ISHIHARA|0|1
Mathematical and Information Sciences IV|2|2017|A|MORIHATA|0|0
Information Engineering VI|2|2017|S|KOSHIZUKA|0|1
Statistics [Informatics]|2|2017|A|SHIMADA|0|2
Special Lecture on Informatics I|1|2017|A|ASO|0|1
Special Lecture on Informatics IV|1|2017|S|KOBAYASHI|0|1
Civilization and Technologies|2|2018|S|HAGIWARA|1|2
Law and the Environment (3)|2|2018|S|GIRAUDOU|1|2
Biodiversity and Ecosystems|2|2018|S|SHEFFERSON|1|2
Modeling and Simulation|2|2018|S|MAEDA|1|2
Information Engineering III|2|2018|S|YAMAGUCHI|0|2
```

In this table, I have created the columns for the ‘course TITLE’ as the ‘name’, ‘CREDITS obtained’ as an integer, ‘YEAR course was taken’ as a date, ‘SEMESTER course was taken (summer or autumn)’ as a one-letter ‘s’ or ‘a’ textual representation, ‘PROFESSOR teaching the course’ using their ‘LAST_NAME’, ‘REQUIRED course or elective’ as ‘BOOLEAN’ value (0 is elective, 1 is required for major), and ‘PASS’ as an integer representation where 0 is not pass (fail), 1 is pass, and 2 is “didn’t sit exam / drop / other”.

With this simple table, we will now move out of logic and into usage.

Usage

Mathematical Operations

I think it could be possible to count my total credits. This can be done with relative simplicity using the *sum* command.

```
sqlite> select sum(CREDITS) from KAMOKU;
59
```

But quickly, I realize this is wrong. Why?

Sometimes, the course is not one we are able to count. It is one that I did not pass for a various reason, or one that I am currently taking. Hence, we must employ the *where* command to further constrict our criteria.

```
sqlite> select sum(CREDITS) from KAMOKU
   ...> where PASS=1;
43
```

This looks much better now.

[I am still behind and need to take many more classes T_T!]

A more interesting calculation would involve taking an average of any single course's credit hours from the courses that I have passed. Still, like all mathematical commands in programming languages, a relatively simple thing to have output.

```
sqlite> select avg(CREDITS) from KAMOKU where PASS=1;
1.79166666666667
```

~ 1.8 credits per course.

Considering my faculties are made of 1 or 2 credit courses, I am taking an efficient load of mostly 2 credit courses, and not very many 1 credit courses.

Adding a Column in SQL for adding some extra information

A question I had from a previous week wondered how I could easily edit the table in the database I have created to make an additional column – for example, I liked how one student noted the number of students in a course, or perhaps to give the course a rating from 1-5.

```
sqlite> .tables
KAMOKU
sqlite> ALTER TABLE KAMOKU ADD COLUMN NumStudents
   ...> ;
```

After confirming the name of the table, we use the '*ALTER TABLE*' command and the '*ADD COLUMN*' command to select, mark for change, and add the column '*NumStudents*' which will list the number of students present in a course.

```
sqlite> UPDATE KAMOKU SET NumStudents=26 WHERE TITLE="Professional Ethics";
sqlite> select * from KAMOKU;
Professional Ethics|2|2017|S|ITAMI|1|1|26
```

Success! Now, we see an entry for 26 students (and indeed in the following entries, there is a column-space of blank entries). The | shows the existence of the column.

```
Professional Ethics|2|2017|S|ITAMI|1|1|26
Environmental Risk Management|2|2017|A|MATSUDA|1|1|
Contemporary Environmental Issues|2|2017|A|HIRONO|1|1|
```

However, this is where I noticed I also messed up in last week's assignment big-time. It would have been well to have included an ID for each course (row) so that I would be able to uniquely identify a course by ID = 1… n (or even using the '08xxx' designation that Komaba Campus uses), instead of TITLE = "(name)" in a manual-labor intensive row recursion.

Update from a few days later: A re-read of the course material shows me that SQLITE uses the command '*rowid*' to individually number each row.

```
sqlite> select * from KAMOKU where rowid=1;
Professional Ethics|2|2017|S|ITAMI|1|1|26
```

Indeed, this is much better.

Querying Data from the Table

- Show me all courses and if I passed them that I have taken in 2017, where results are listed in alphabetical order and only list a 2-credit course.

```
sqlite> select title, PASS from KAMOKU where YEAR=2017 group by TITLE having min(CREDITS) > 1;
Advanced Energy Science and Engineering|1
Applied Ethics III (Seminar) [Science and Technology Studies]|1
Chemistry for Environmental Studies|1
Contemporary Environmental Issues|1
Earth System Science III|1
Environmental Measurement II|1
Environmental Risk Management|1
Experiments in Environmental Sciences I|1
Food Safety and Risk Analysis|1
Human Population and Dynamics [Environmental Sciences]|1
Information Engineering VI|1
Introduction to Applied Ethics [Science and Technology Studies]|1
Introduction to Philosophy of Science and Technology|1
Materials Chemistry|1
Mathematical and Information Sciences IV|0
Philosophy of Science and Technology IV (Seminar)|1
Professional Ethics|1
Science and Technology Studies I (Seminar)|1
Science, Technology, and Society|1
Simulation Methods|0
Statistics [Informatics]|2
Urban Planning Technology II|1
```

This happens to be a very good use of the '*having*' qualifier-type command.

- While I don't happen to have a second table, if you had two people compare the courses they took in 2017 and output all courses in common. Output the common courses in any order.

```
sqlite> CREATE TABLE ALFREDCOURSES (title text, credits integer, year date, semester text, professor text, required bool
ean, pass integer, NumStudents integer)
```

First, I initialize a table for my friend's courses.

```
sqlite> INSERT INTO ALFREDCOURSES (title, credits, year, semester, professor, required, pass) VALUES
   ...> ('Professional Ethics', 2, 2017, 'S', 'ITAMI', 1, 1),
   ...> ('Algebraic Optimization', 2, 2018, 'S', 'HASHIMOTO', 1, 1),
   ...>
```

Next, I insert some values into the table.

```
sqlite> select distinct * from KAMOKU intersect select distinct * from ALFREDCOURSES;
```

The '*distinct*' command will help avoid redundancy and the intersect command finds the common courses of the two sets. However, this will output the whole line. Hence, we change the wildcard to a specific entry for course TITLE(s).

```
sqlite> select distinct TITLE from KAMOKU intersect select distinct TITLE from ALFREDCOURSES;
```

- Non-redundantly display a list of professors from courses I took. Show results alphabetically. Further, query the total count of professors I have met through taking their course.

  The key here is a command called '*distinct*', helping us avoid 2+ *Woodward* entries, for example.

```
sqlite> select distinct PROFESSOR from KAMOKU GROUP by PROFESSOR;
ASO
E.COHEN
GIRAUDOU
HAGIWARA
HIRONO
ISHIHARA
ITAMI
KAKIMURA
KOBAYASHI
KOSHIZUKA
MAEDA
MATSUDA
MEGURO
MORIHATA
NAGATA
NOBUHARA
ODEA
OKABE
OKAMOTO
SATO
SHEFFERSON
SHIMADA
SUZUKI
TAKEUCHI
TSUTSUMI
UCHIDA
WOODWARD
YAMAGUCHI
YAMAKAWA
```

That looks like a small library of professors. Now, let's envelope the whole command in a '*count*' statement and see if I get the correct number of professors.

```
sqlite> select distinct count(PROFESSOR) from KAMOKU;
32
```

This is incorrect.

```
sqlite> select count(distinct PROFESSOR) from KAMOKU;
29
```

This is correct.

The *order of keywords*, as we can see, is equally as important as the use of an *accurate* term!

(End of Week 4 Assignment)