CS50 Lecture 7: SQL

Jordan Mandel

2021_10_22

Counting Titles in Python

- recall that a set is like a list that doesn't have any duplicates.
- canonicalize means reformat
- schema is a way we've stored the data
- `string.strip().upper()`

```
titles = {}

with open("Facorite TV Shows - Form Responses 1.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        title = row["title"].strip().upper()
        if title in titles:
            titles[title] += 1 # gets error if it doesn't exitst
        else:
            titles[title] = 1

def f(title):
    return titles[title]
# could also do key = lambda title: titles[title]
for title in sorted(titles, key = f, reverse = True):
    print(title, titles[title])
```

above can also do:

```
if title not in titles:
    titles[title] = 0
titles[title] += 1
```

Harder Problem in Python: Searching

- This is not time efficient it is `O(n)`

```
import csv

title = input("Title: ").strip().upper()

with open ("Facorite TV Shows blah blah.csv", "r) as file:
    reader = csv.DictReader(file)
    counter = 0
    for row in reader:
        if row["title"].strip().upper() == title:
            counter += 1
```

```
print(counter)
```

Relational Databases

- Much more time efficient than spreadsheets or csvs but require more memory

```
.mode csv
.import 'filename.csv' varname
```

There are four main things we can do: - create `INSERT` - read `SELECT` - select `SELECT column FROM table`; - `UPDATE` - `DELETE`

We can check the schema of our table with `.schema`: lists the commands used to make the table.

Select column with:

```
SELECT title FROM shows:
SELECT title, Timestamp FROM shows;
SELECT * FROM shows;
```

Some functions that we can use include

```
AVG
COUNT
DISTINCT
LOWER
MAX
MIN
UPPER
```

so we can go:

```
SELECT DISTINCT(UPPER(title)) FROM shows;
SELECT title FROM shows WHERE title = "The Office";
```

There is a `LIKE` keyword we can take a look at. It gets similar strings with the use of wildcards.

```
SELECT title FROM shows WHERE title LIKE "%Office%"
```

The `%` means zero or more other characters.

We can order by something:

```
SELECT DISTINCT(UPPER(title)) FROM shows ORDER BY UPPER(title);
```

We can group and count.

```
SELECT UPPER(title), COUNT(title) FROM shows GROUP BY UPPER(title);
```

Then we can order the counts, make it go in descending order, and limit to ten, and trim whitespace: - In the below we have to consider whether I should have pu `trim` around title the second time.

```
SELECT UPPER(trim(title)), COUNT(title) FROM shows GROUP BY UPPER(trim(title)) ORDER BY COUNT(title) DE
```

can save

```
.save shows.db
```

More Details on Tables

We can insert data into a table manually `INSERT INTO table (column, ...) VALUES(value, ...);`

```
INSERT INTO shows (Timestamp, title, genres) VALUES("now", "The Mupper Show", "Comedy, Musical");
```

There is also `UPDATE table SET column = value WHERE condition;`

```
UPDATE shows SET genres = "Comedy, Drama, Musical" WHERE title = "The Muppet Show"
```

Or can go `DELETE FROM table WHERE condition`

```
DELETE FROM shows WHERE title LIKE "Friends";
```

Ok given this info we can have multiple tables; one associating the show with an id, and another with multiple entries per show.

There are data types:

- `BLOB`, binary large object that might represent a file
- `INTEGER`
- `NUMERIC` number like but not a number; possibly a date or time
- `REAL` for floating point values
- `TEXT` like strings.

There are also other properties a column can have.

- `NOT NULL` there has to be something
- `UNIQUE` value must be different for every row
- `PRIMARY KEY`
- `FOREIGN KEY`

Can use `SQL` Function in `CS50` to make Queries

```
import csv

from cs50 import SQL

open("shows.db", "w").close()
db = SQL("sqlite:///shows.db")

db.execute("CREATE TABLE shows (id INTEGER, title TEXT, PRIMARY KEY(id))")
db.execute("CREATE TABLE genres (show_id INTEGER, genre TEXT, FOREIGN KEY(show_id) REFERENCES shows(id)


with open("Favorite TV Shows - Form Responses 1.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        title = row["title"].strip().upper()

        id = db.execute("INSERT INTO shows (title) VALUES(?)", title)'

        for genre in row["genres"].split(", "):
            db.execute("INSERT INTO genres (show_id, genre) VALUES(?, ?)", id, genre)
```

Now we have two tables, show and genre with a primary key ID.

We can select everything:

```
SELECT * FROM shows;
SELECT * FROM genres;
```

We can select specific things:

```
SELECT show_id FROM genres WHERE genre = "Musical"
```

Can nest the things as well:

```
SELECT DISTINCT(genre) FROM genres WHERE show_id IN(SELECT id FROM shows WHERE title = "THE OFFICE") ORI
```

Right now we have

- a table with `[id, title]`, and `[id, genre]`
- but this stores genres many times. a better system might be `[movie_id, title]` `[genre_id, genre_name]` `[movie_id, genre_id]`
- now to change a genre's name we just have to update one row of a table rather than many

more subtypes

INTEGER

- with fewer bits
- integer
- bigint, with more bits

NUMERIC

- boolean
- date
- datetimey
- numeric(scale,precision), with a fixed number of digits
- time
- timestamp

REAL

- real
- double precision, with twice as many bits

TEXT

- char(n), a fixed number of characters
- varchar(n), a variable number of characters, up to some limit n
- text, a string with no limit

IMDB

It is a complicated movie database. With more than 150,000 shows! We have to make an index which is a `B-tree` which is like a binary tree. Making it takes time but things go much faster afterwards.

```
CREATE INDEX title_index ON shows (title)
```

we can do joins

```
SELECT title FROM people
JOIN stars ON people.id = stars.person_id
JOIN shows ON stars.show_id = shows.id
WHERE name = "Steve Carell";
```

Security

SQL Injection

```
f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
```

but we make the rest a comment like:

```
 f"SELECT * FROM users WHERE username = 'malan@harvard.edu'--' AND password = '{password}'"
```

this would be safer:

```
rows = db.execute("SELECT * FROM users WHERE username = ? AND password = ?", username, password)
```

Race Conditions

```
rows = db.execute("SELECT likes FROM posts WHERE id = ?", id);
likes = rows[0]["likes"]
db.execute("UPDATE posts SET likes = ? WHERE id = ?", likes + 1, id);
```

If this happens on two different servers we could have a problem! They might get the same number of likes.

To solve this we have `BEGIN TRANSACTION`, `COMMIT` and `ROLLBACK`. We can go like this:

```
db.execute("BEGIN TRANSACTION")
rows = db.execute("SELECT likes FROM posts WHERE id = ?", id);
likes = rows[0]["likes"]
db.execute("UPDATE posts SET likes = ? WHERE id = ?", likes + 1, id);
db.execute("COMMIT")11
```

Video Short

INSERT

```
INSERT INTO <table>(<columns>) VALUES(<values>)
INSERT INTO users(username, pasword, fullname) VALUES('newman', 'USMAIL', 'Newman')
INSERT INTO moms(username, mother) VALUES('kramer', 'Babs Kramer')
```

SELECT

- We didn't include id number but that is OK!

```
SELECT <columns> FROM <table> WHERE <predicate> ORDER BY <column>
SELECT idnum,fullname FROM users
SELECT idnum FROM users WHERE idnum<12
SELECT * FROM moms WHERE username=jerry
```

SELECT JOIN

```
SELECT <columns> FROM <table1> JOIN <table2> ON <predicate>
SELECT users.fullname, moms.mother FROM users JOIN moms ON users.username = moms.username
```

UPDATE

```
UPDATE <table> SET <column> = <value> WHERE <predicate>
UPDATE users SET password = 'yadayada' WHERE idnum=10
```

DELETE

```
DELETE FROM <table> WHERE <predicate>
DELETE FROM users WHERE name=newman
```