SQL with Python

It turns out that we'll be able to write Python code that automates this, so we can imagine building web applications that can programmatically store and look up user data, online shopping orders, and more

We can write a program that asks the user for a show title and then prints its popularity:

```
import csv

from cs50 import SQL

db = SQL("sqlite:///favorites.db")

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

rows = db.execute("SELECT COUNT(*) AS counter FROM favorites WHERE title LIKE ?", title)

row = rows[0]

print(row["counter"])
```

We'll use the cs50 library to run SQL commands more easily, and open the favorites.db database we created earlier.

We'll prompt the user for a title, and then execute a command. A ? in the command will allow us to safely substitute variables in our command

The results are returned in a list of rows, and COUNT(*) returns just one row. In our command, we'll add AS counter, so the count is returned in the row (which is a dictionary) with the column name counter

And we can tweak our program to print all the rows that match:

```
import csv

from cs50 import SQL

db = SQL("sqlite:///favorites.db")

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

rows = db.execute("SELECT title FROM favorites WHERE title LIKE ?", title)

for row in rows:
    print(row["title"])
```

Since LIKE is case-insensitive, we see all the various ways the titles were capitalized

IMDb

IMDb, or the Internet Movie Database, has datasets available for download as TSV (tab-separated values) files

We’ll open a database that the staff has created beforehand:

```
$ sqlite3 shows.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE shows (
                id INTEGER,
                title TEXT NOT NULL,
                year NUMERIC,
                episodes INTEGER,
                PRIMARY KEY(id)
            );
CREATE TABLE genres (
                show_id INTEGER NOT NULL,
                genre TEXT NOT NULL,
                FOREIGN KEY(show_id) REFERENCES shows(id)
            );
CREATE TABLE stars (
            show_id INTEGER NOT NULL,
            person_id INTEGER NOT NULL,
            FOREIGN KEY(show_id) REFERENCES shows(id),
            FOREIGN KEY(person_id) REFERENCES people(id)
        );
CREATE TABLE writers (
            show_id INTEGER NOT NULL,
            person_id INTEGER NOT NULL,
            FOREIGN KEY(show_id) REFERENCES shows(id),
            FOREIGN KEY(person_id) REFERENCES people(id)
        );
CREATE TABLE ratings (
            show_id INTEGER NOT NULL,
            rating REAL NOT NULL,
            votes INTEGER NOT NULL,
            FOREIGN KEY(show_id) REFERENCES shows(id)
        );
CREATE TABLE people (
            id INTEGER,
            name TEXT NOT NULL,
            birth NUMERIC,
            PRIMARY KEY(id)
        );
```

Notice that we have multiple tables, each of which has columns of various data types

In both the `stars` and `writers` table, for example, we have a `show_id` column that references the `id` of some row in the `shows` table, and a `person_id` column that references the `id` of some row in the `people` table. Effectively, they link shows and people by their `id`

It turns out that SQL, too, has its own data types:

- **BLOB**, for "binary large object", raw binary data that might represent files
- **INTEGER**
- **NUMERIC**, number-like but not quite a number, like a date or time
- **REAL**, for floating-point values
- **TEXT**, like strings

Columns can also have additional attributes:

- **PRIMARY KEY**, like the id columns above that will be used to uniquely identify each row
- **FOREIGN KEY**, like the show_id column above that refers to a column in some other table

We can see that there are millions of rows in the people table:

```
sqlite> SELECT * FROM people;
```

But like before, we can search for just one row:

```
sqlite> SELECT * FROM people WHERE name = "Steve Carell";
```

It turns out that there are a few shows titled "The Office":

```
sqlite> SELECT * FROM shows WHERE title = "The Office";
```

The most popular one, with 188 episodes, is the one we want, so we can get just that one:

```
sqlite> SELECT * FROM shows WHERE title = "The Office" and year = "2005";
```

We can turn on a timer and see that our original command took about 0.02 seconds to run:

```
sqlite> .timer on
sqlite> SELECT * FROM shows WHERE title = "The Office";
```

```
---------+------------+------+----------+
|   id    |   title    | year | episodes |
+---------+------------+------+----------+
| 112108  | The Office | 1995 | 6        |
| 290978  | The Office | 2001 | 14       |
| 386676  | The Office | 2005 | 188      |
| 1791001 | The Office | 2010 | 30       |
| 2186395 | The Office | 2012 | 8        |
| 8305218 | The Office | 2019 | 28       |
+---------+------------+------+----------+
Run Time: real 0.021 user 0.016419 sys 0.004117
```

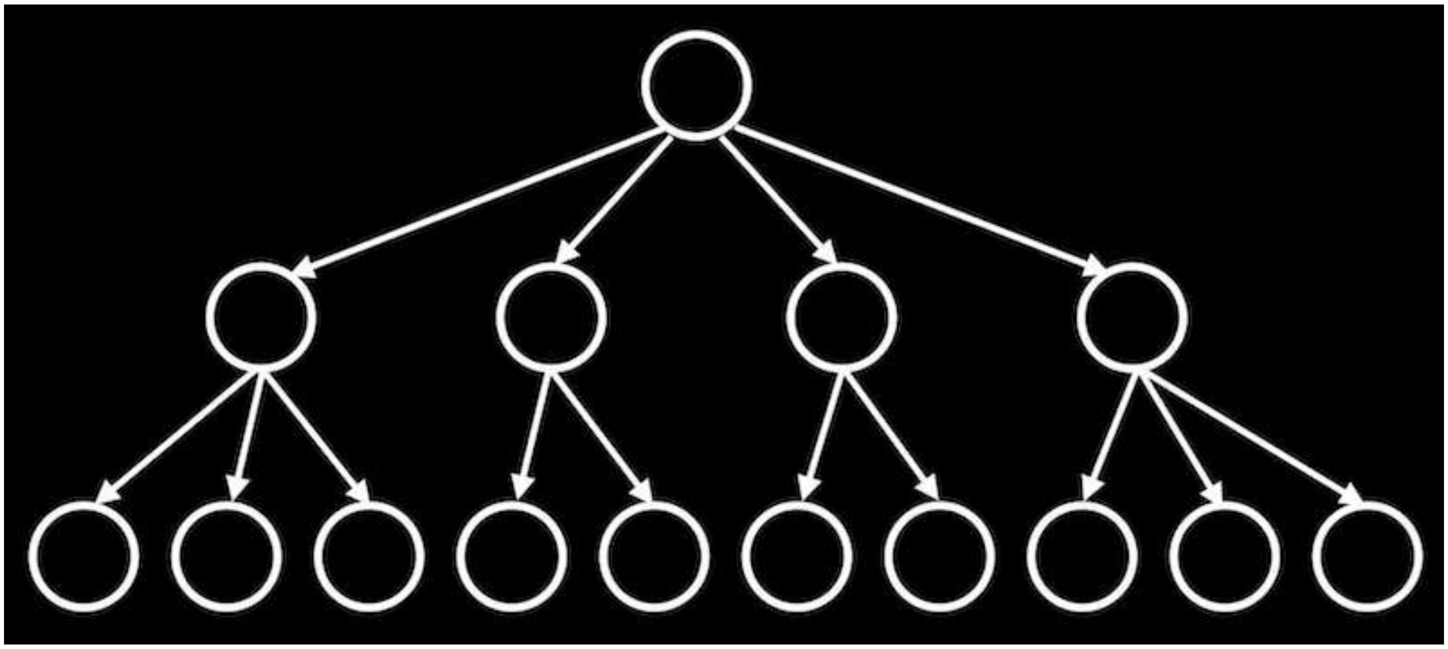
We can create an index, or additional data structures that our database program will use for future searches:

```
sqlite> CREATE INDEX "title_index" ON "shows" ("title");
Run Time: real 0.349 user 0.195206 sys 0.051217
```

Now, our search command takes nearly no time:

```
sqlite> SELECT * FROM shows WHERE title = "The Office";
+---------+------------+------+----------+
|   id    |   title    | year | episodes |
+---------+------------+------+----------+
| 112108  | The Office | 1995 | 6        |
| 290978  | The Office | 2001 | 14       |
| 386676  | The Office | 2005 | 188      |
| 1791001 | The Office | 2010 | 30       |
| 2186395 | The Office | 2012 | 8        |
| 8305218 | The Office | 2019 | 28       |
+---------+------------+------+----------+
Run Time: real 0.000 user 0.000104 sys 0.000124
```

It turns out that these data structures are generally **B-trees**, like binary trees we've seen in C but with more children, with nodes organized such that we can search faster than linearly:



Creating an index takes some time up front, perhaps by sorting the data, but afterwards we can search much more quickly.

With our data spread among different tables, we can nest our queries to get useful data. For example, we can get all the titles of shows starring a particular person:

```
sqlite3> SELECT title FROM shows WHERE id IN (SELECT show_id FROM stars WHERE person_id = (SELECT id FROM people WHERE name = "Steve Carell"));
```

We'll SELECT the title from the shows table for shows with an id that matches a list of show_ids from the stars table. Those show_ids, in turn, must have a person_id that matches the id of Steve Carell in the people table

Our query runs pretty quickly, but we can create a few more indexes:

```
sqlite> CREATE INDEX person_index ON stars (person_id);
Run Time: real 0.890 user 0.662294 sys 0.097505
sqlite> CREATE INDEX show_index ON stars (show_id);
Run Time: real 0.644 user 0.469162 sys 0.058866
sqlite> CREATE INDEX name_index ON people (name);
Run Time: real 0.840 user 0.609600 sys 0.088177
```

Each index takes almost a second to build, but afterwards, our same query takes very little time to run

It turns out that we can use **JOIN** commands to combine tables in our queries:

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

With the JOIN syntax, we can virtually combine tables based on their foreign keys, and use their columns as though they were one table. Here, we're matching the people table with the stars table, and then with the shows table

We can format the same query a little better by listing the tables we want to use all at once:

```
sqlite> SELECT title FROM people, stars, shows
   ...> WHERE people.id = stars.person_id
   ...> AND stars.show_id = shows.id
   ...> AND name = "Steve Carell";
```

The downside to having lots of indexes is that each of them take up some amount of space, which might become significant with lots of data and lots of indexes

Problems

One problem in SQL is called a **SQL injection attack**, where an someone can inject, or place, their own commands into inputs that we then run on our database

We might encounter a login page for a website that asks for a username and password, and checks for those in a SQL database

Our query for searching for a user might be:

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

if len(rows) == 1:
    # Log user in
```

By using the `?` symbols as placeholders, our SQL library will escape the input, or prevent dangerous characters from being interpreted as part of the command

In contrast, we might have a SQL query that's a formatted string, such as:

```
rows = db.execute(f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'")

if len(rows) == 1:
    # Log user in
```

If a user types in `malan@harvard.edu'--` as their input, then the query will end up being:

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

This query will actually select the row where `username = 'malan@harvard.edu'`, without checking the password, since the single quotes end the input, and `--` turns the rest of the line into a comment in SQL

The user could even add a semicolon, `;`, and write a new command of their own, that our database will execute

Another set of problems with databases are **race conditions**, where shared data is unintentionally changed by code running on different devices or servers at the same time

One example is a popular post getting lots of likes. A server might try to increment the number of likes, asking the database for the current number of likes, adding one, and updating the value in the database:

```
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);
```

Two different servers, responding to two different users, might get the same starting number of likes since the first line of code runs at the same time on each server

Then, both will use `UPDATE` to set the *same* new number of likes, even though there should have been two separate increments

To solve this problem, SQL supports **transactions**, where we can lock rows in a database, such that a particular set of actions are **atomic**, or guaranteed to happen together

For example, we can fix our problem above with:

```
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")
```

The database will ensure that all the queries in between are executed together

But the more transactions we have, the slower our applications might be, since each server has to wait for other servers' transactions to finish