Question 1 of 3

Recall that, in lecture, we introduced SQL databases as a way to organize and manage data. We've previously seen that we can also use CSV (comma-separated values) files to store data as well. Why might we prefer to use a SQL database instead of a CSV file to store data?

Answer

Answers may vary. Possibilities include:

- In a CSV file, to search for a particular entry, we need to linear search through the entire file. With a SQL database, we have the ability to search for data more efficiently.
- In a CSV file, it's more difficult to perform complex queries: filtering on a particular column, grouping related entries together, or updating and deleting collections of entries at once, for example. In a SQL database, we can use SQL queries to be able to perform those operations on databases efficiently, rather than needing to write, for example, a Python script to generate the results.
- If a CSV file is storing a lot of information about different types of entities (TV shows and genres, for instance), it might make more logical sense to store information in a SQL database instead, such that we could store data across multiple different tables in the database.

Question 2 of 3

Consider the following SQL code to generate a table for storing data about a music library.

```
CREATE TABLE playlists (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT
);

CREATE TABLE songs (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT,
    artist TEXT,
    album TEXT,
    year NUMERIC,
    playlist_id INTEGER,
    FOREIGN KEY(playlist_id) REFERENCES playlists(id)
);
```

Critique the design of this database, as by proposing and explaining at least two ways in which its design could be improved.

Answer

Answers may vary. Possibilities include any two of the following (among others):

- This database assumes every song can only belong to a single playlist, since there is a `playlist_id` column in the `songs` table that means any one song can only have a single `playlist_id`. We could instead create a separate `playlists` table, and then have a join table that maps songs to the playlists that they belong in (as a many-to-many relationship, instead of a one-to-many relationship).
- To make it easier to store information about artists, we could move `artists` to be a separate table, and have a join table that maps songs to artists. This would allow us to store other information about artists more easily — like their birth year, hometown, etc. It would also let us map a song to multiple artists if a song has multiple artists.

- We might want to separate `albums` into a separate table. This would allow us to store additional information about the album in the `albums` table, but also store information like the order in which a song appears in a particular album.
- Depending on the type of SQL database we're using, we might want to instead store our `TEXT` fields as `VARCHAR` fields instead. `VARCHAR` fields can be queried more quickly than `TEXT` fields for certain SQL databases like MySQL or PostgreSQL, and since the titles of songs, names of artists, and names of playlists likely have some reasonable upper bound on length, `VARCHAR` is a reasonable type.
- We might want to separate an artist's name into the first name and last name (for artists where it is applicable to do so), in case we ever wanted to order our artists by their last name.
- We might want some of our columns, like `playlists.name`, `songs.title`, or `songs.artist` to be `NOT NULL`, to enforce the constraint that playlists must have a name, songs must have a title, and songs must have an artist.

Question 3 of 3

Recall that, by creating an index to a column of a SQL table, we can speed up `SELECT` queries on that column. Why, then, should we not just always create indexes on every column in a SQL table?

Answer

Answers may vary. Possibilities include:

- When we create an index, SQLite creates a B-tree to store that index. That B-tree takes up memory, and will take up more memory if there's more data in the table. So creating an index on every column, even the ones we aren't likely to query on, will result in our database taking up more memory, without much of a speed improvement. Therefore, it's generally preferable to only create indexes on the columns that we're likely to perform queries on.
- While creating an index does speed up SELECT queries, it also has the effect of slowing down INSERT, UPDATE, and DELETE queries - since for each of those queries, the index will need to be updated, which requires additional time. Therefore, if we're expecting to do many more INSERTs, UPDATEs, and DELETEs than we are SELECTs, adding an index might not be advantageous.
- Creating a SQL index also takes a nonzero amount of time. If you're never going to query on a particular column, or you're only going to do so once or a few times, then it might not be worth the upfront time cost of creating an index to perform the query. Creating the SQL index is likely worthwhile only when you're going to query the field multiple times.