Database Final Report

Chris Hughes

December 6, 2023

1 Design Choices

Usage

Assuming node/npm is installed `npm install` will install the SQLite and inquirer packages and then `node index.js` will run the main file to query the database. If the `data` directory or `final.db` files aren't present, then the user will be asked if they want to create an empty database or a database with sample data. The sample data should be sufficient to run the queries for the database final. If you want to start fresh, just delete the `final.db` file and running the app again will create a new one.

Node

I decided to use node for this project because I had already been working with the SQLite package for node and it was relatively easy to implement. I created a single database file and populated it with the tables required for the project. I used a series of functions that contain switch statements to sort user inputs and generated a query string, which is then sent to a query function that queries the database.

Orders

One issue I see with the orders table is that if a customer bought multiple books at the same time, then each book would need to be on a separate row with its own `order_id` since primary keys must be unique. perhaps you could use the `date` and `customer_id` together as a composite key to query all the books bought in a single purchase. If you record the exact time of the purchase, and input the that exact time for all books bought in a single purchase, then it should be unique as, even if you had a large volume of orders where its possible orders could take place at the exact same time, one customer should not be able to make multiple purchases simultaneously. If you had other information that you would like to store alongside groups of orders that took place together, you could use this to create another purchases table. A better approach might be to create a view for this. For example something along these lines might work.

```sql
CREATE VIEW Purchase as
SELECT c.first_name, c.last_name, o.order_date, sum(b.price)
FROM orders o
JOIN customers c ON o.customer_id=c.customer_id
JOIN books b ON o.book_id=b.book_id
GROUP BY o.customer_id, o.order_date;
```

Async

I had some issues getting things to line up the data from the queries and getting use input. I was having the data overwrite the inputs. What I eventually did was returned a promise inside the database callbacks and awaited them before allowing the main loop to continue.

Validation

In order to do some basic validation for the user inputs, I used `isNaN()` to check if the `parseFloat()` returned a value of NaN. If it is, I print an error and return to the main loop.

Deleting rows

It's safe to delete orders and books, but if you delete rows in other tables, it might cause issues. For example, if you delete an author, there may be books related to that author that will no longer point to an author, or may point to an incorrect author. If an author does not have any books assigned to it, it should be safe to delete the author. The authors with no books query should assist in this. Similarly, if a customer has placed orders, then orders for that customer would not point to the correct customer. If customer appears in the "Customers with no orders query", it should be safe to delete that customer.

Next steps

I think the most important next step would be to create some functions for updating entries. As it is currently, in order to update something I would need to remove it entirely and replace it with the correct data, but this would cause issues where the new ID would not match correctly for joins.

Another feature that could be nice is to add the delete warnings when attempting to delete something will cause data inconsistencies.

If the database were actually used for any real number of books/orders etc. it would be likely useful to have some kind of pager built in or better control on the limit of rows returned.

2 Normalization

First Normal Form

The table has a primary key

Each table has a primary key that uniquely identifies each row.

Each column has a single unique value.

The only possible exception to this is the date rows, but since date is a recognized data type in SQLite, there is no need to split up month, day, year into separate columns.

The nonprimary key columns are functionally dependent on the primary key.

All of the columns are directly related to the primary key. We know this because when we do joins, we can determine any column knowing only the primary key.

Second Normal Form

The tables conform to first normal form

Non-primary key attributes depend on all attributes of a composite key.

None of the tables rely on a composite key. You could consider using a composite key in the orders table using `customer_id` , `book_id` and `date`, as long as you are certain to be able to record the exact time of the purchase and not just the day of the purchase, but if multiple books were bought at the same time then this would not work.

Third Normal Form

The tables conform to first and second normal form

Each nonprimary key attribute in a row does not depend on the entry in another key column.

This is the case. no columns can be determined from any other combination of columns without outside information.

3 Queries

1. Retrieve the list of books with their authors.

This query is fairly straight forward. The book titles and author names are in different tables so you need to do a regular join of books on the authors table.

Query

```
SELECT b.title, a.author_name FROM books b
JOIN authors a ON b.author_id=a.author_id
```

Example output

```
{ title: 'The greatest book', author_name: 'Chris Hughes' }
{ title: "One Flew over the Cuckoo's nest", author_name: 'Ken Kesey' }
{ title: 'Crime and Punishment', author_name: 'Fyodor Dostoevsky' }
{ title: 'Of Human Bondage', author_name: 'W. Somerset Maugham' }
{ title: 'Heart of Darkness', author_name: 'Joseph Conrad' }
{ title: 'Siddartha', author_name: 'Herman Hesse' }
{ title: 'Candide', author_name: 'Voltaire' }
{ title: 'The Pearl', author_name: 'John Steinbeck' }
{ title: 'The Idiot', author_name: 'Fyodor Dostoevsky' }
```

2. Find the total sales (quantity * price) for each book.

In order to get the total sales, multiply the combined quantities of a `book_id`, by using the `SUM()` function, multiplying by price and then grouping by book id.

Query

```
SELECT b.title, SUM(o.quantity) * b.price AS "Total Sales"
FROM books b
JOIN orders o ON b.book_id=o.book_id
GROUP BY  b.book_id
```

Example output

```
{ title: "One Flew over the Cuckoo's nest", 'Total Sales': 77.94 }
{ title: 'Crime and Punishment', 'Total Sales': 19.35 }
{ title: 'Of Human Bondage', 'Total Sales': 719.59 }
{ title: 'Heart of Darkness', 'Total Sales': 229.77 }
{ title: 'Siddartha', 'Total Sales': 5.99 }
{ title: 'Candide', 'Total Sales': 41.98 }
```

3. Identify the top 3 bestselling genres.

I join the book and order table and aggregate genre so that I can `SUM()` the combined quantities based on the genre. I test to make sure the genre hasn't been left blank as well so that if there are a lot of instances where the genre has been left blank, genre: blank won't be a top result. When I first did this, I had accidentally put COUNT() instead of SUM(), which of course gave me the number of times the book had been ordered, but not the total quantity of orders.

Query

```
SELECT b.genre AS "Bestselling Genres",
SUM(o.quantity) AS "Books Sold"
FROM books b
JOIN orders o ON b.book_id=o.book_id
WHERE b.genre <> ''
GROUP BY b.genre
ORDER BY SUM(o.quantity) DESC
LIMIT 3
```

Example output

```
{ 'Bestselling Genres': 'Horror', 'Books Sold': 63 }
{ 'Bestselling Genres': 'True Crime', 'Books Sold': 7 }
{ 'Bestselling Genres': 'Classics', 'Books Sold': 2 }
```

4. List customers who have made at least two orders.

I aggregate customer IDs to combine the number of `order_id` and if that is greater than 1, we know they have placed multiple orders.

Query

```
SELECT c.customer_id,
c.first_name || " " || c.last_name AS "Customer Name",
COUNT(o.order_id) AS "Orders Placed"
FROM customers c
JOIN orders o ON c.customer_id=o.customer_id
GROUP BY c.customer_id
HAVING COUNT(o.order_id)>1
```

Example output

```
{
  customer_id: 2,
  'Customer Name': 'Ralph Johnston',
  'Orders Placed': 2
}
```

5. Update the price of all books in a specific genre by 10%.

I take the price of the desired genre and then multiply or divide by 1.1 and then I use the `ROUND()` function to round it to two decimal places in order to make it a suitable dollar amount.

Query

```
UPDATE books SET price=ROUND(price*1.1, 2)
WHERE genre="${genre}"
```

Example output

This is the results from listing the "True Crime" books

```
 {
  book_id: 6,
  title: 'Siddartha',
  genre: 'True Crime',
  price: 6.59,
  author: 'Herman Hesse'
}
{
  book_id: 1,
  title: 'The greatest book',
  genre: 'True Crime',
  price: 12.09,
  author: 'Chris Hughes'
}
```

And then after the Query. I double checked with a calculator to ensure the values are correct.

```
{
  book_id: 1,
  title: 'The greatest book',
  genre: 'True Crime',
  price: 13.3,
  author: 'Chris Hughes'
}
{
  book_id: 6,
  title: 'Siddartha',
  genre: 'True Crime',
  price: 7.25,
  author: 'Herman Hesse'
}
```

6. Calculate the average price of books for each author.

I aggregate by `author_id` so that I can get the average of the prices column. I round to 2 decimals to be a suitable dollar amount.

Query

```
SELECT a.author_id, a.author_name, ROUND(AVG(b.price), 2) AS "Average Price"
FROM authors a
JOIN books b ON a.author_id=b.author_id
GROUP BY a.author_id
```

Example output

```
{ author_id: 1, author_name: 'Chris Hughes', 'Average Price': 13.3 }
{ author_id: 2, author_name: 'Ken Kesey', 'Average Price': 15.72 }
{
  author_id: 3,
  author_name: 'Fyodor Dostoevsky',
  'Average Price': 19.57
}
{
  author_id: 4,
  author_name: 'W. Somerset Maugham',
  'Average Price': 17.99
}
{ author_id: 5, author_name: 'Joseph Conrad', 'Average Price': 9.99 }
{ author_id: 6, author_name: 'Herman Hesse', 'Average Price': 7.25 }
{ author_id: 7, author_name: 'Voltaire', 'Average Price': 20.99 }
{ author_id: 8, author_name: 'John Steinbeck', 'Average Price': 8 }
```

7. Retrieve the authors who have not published any books.

I use a left join to display the left column results where join doesn't find a match and only display the entries where the id is null.

Query

```sql
SELECT a.author_id, a.author_name
FROM authors a
LEFT JOIN books b ON a.author_id=b.author_id
WHERE b.author_id IS NULL
```

Example output

```
{ author_id: 9, author_name: 'Jesus Christ' }
```

8. Identify customers who have not placed any orders.

Essentially the same as the last query

Query

```sql
SELECT c.customer_id, c.first_name || " " || c.last_name As "Customer Name"
FROM customers c
LEFT JOIN orders o ON c.customer_id=o.customer_id
WHERE o.customer_id IS NULL
```

Example output

```
{ customer_id: 8, 'Customer Name': 'Barak Obama' }
```