Tutorial: Transactions

This tutorial is made from notes and using a database provided by Tomoko Okochi

A **transaction** is a group of SQL queries that all succeed or fail together; they're used when queries in a group are related to one another and it's important that they all get executed together. If one query fails or if some other failure occurs during execution (maybe the power on the database server goes out) all queries in the transaction are **rolled back**, which means the database is returned to its previous state, before the transaction began. Either ALL queries are executed, or NO queries are executed.

You can probably think of a lot of cases in which this would be useful, but the classic example is transferring funds from one back account to another. This operation could have two parts:

- The account balance of one person is decreased;
- The account balance of the other person is increased, by the same amount.

If the program were to crash after the first operation but before the second, you can see how there would be a problem: one person had funds withdrawn, but the other person never received the funds they were owed! It would be as if the money disappeared. Obviously this is not acceptable; a **transaction** ensures that if an error or failure occurs at any point during the transaction, the database simply reverts to its previous state.

Transaction must have all of the following properties:

- **Atomicity**: this means the group of queries cannot be broken into smaller groups; all queries must be executed together!
- **Consistency:** this means that after the transaction, all constraints of the schema must be met.
- **Isolation:** when multiple transactions happen at the same time (this is common, maybe due to different users interacting with the database) the transactions don't interfere with each other.
    - For example, one wouldn't want to purchase a product from an online store (and pay for it) just as the last item is sold to another customer!
- **Durability:** completed transactions should be fully committed and permanent, even if there's a failure after completion.

The acronym used for the above properties is **ACID**.

The syntax for transactions in SQLite is pretty basic (despite being ACID). To start a transaction, the statement is:

BEGIN TRANSACTION;

All queries that follow are part of the transaction until it is ended with the following statement:

COMMIT;

Committing the transaction finalizes everything and makes the effects of the queries permanent.

SQLite will automatically rollback a transaction in the following cases:

- An error occurs during the transaction;
- The database connection is terminated during the transaction.

This is great - it means that without any effort on our part (other than defining the start and end of the transaction) SQLite is keeping our database safe from incomplete updates!

If we want to manually initiate a rollback, we can do so with the following statement:

ROLLBACK;

Often, we'll use error handling logic to figure out when a transaction isn't working out roll it back.

Task 1: Setup

In the starter files for this tutorial, I've provided the following:

- **bank.db**: this is a very simple database with a single table for recording the account balances of customers. Use **SQLite Viewer** to see the rows, and **.schema** in **sqlite3** to see the table schema.
    - Notice that the **balance** column has a **CHECK** constraint that prevents it from taking a negative value. This is important, because we want an error to occur rather than the account becoming overdrawn!
- **reset-accounts.sql:** this script can be used to reset the accounts of customers Bob and Alice back to their original state.

Import both files into a blank Codespace, then create a JavaScript called **transactions.js;** in this script, we'll write a simple program that transfers some money from Alice's account to Bob's account, using a transaction.

Paste the following code into the file:

```
import { DatabaseSync } from 'node:sqlite';
import path from 'node:path';

const db = new DatabaseSync(path.join(import.meta.dirname, '/bank.db'));
```

This code simply opens the database and stores the connection in the variable **db.**

To run this script, use the following command in the terminal:

```
node transactions.js
```

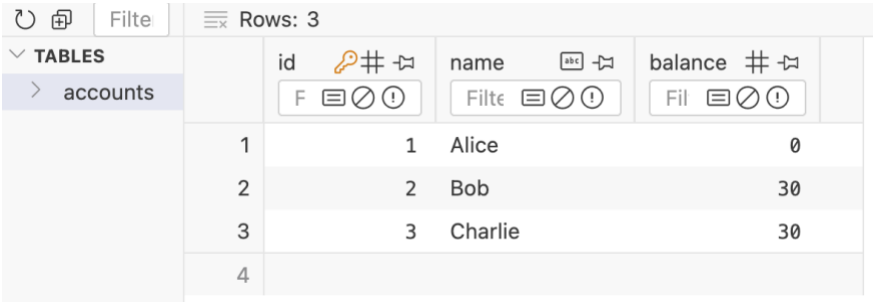
Task 2: Creating the Transaction

Using prepared statements, we can write a very simple implementation of this program:

```
db.prepare(`
    BEGIN TRANSACTION;
    `)
    .run();

let transferAmount = 5;

db.prepare(`
    UPDATE accounts
    SET balance = balance - ?
    WHERE id = 1;
    `)
    .run(transferAmount);

db.prepare(`
    UPDATE accounts
    SET balance = balance + ?
    WHERE id = 2;
    `)
    .run(transferAmount);

db.prepare(`
    COMMIT;
    `)
    .run();
```

As you can see, an **UPDATE** statement is being used to decrement the account balance of Alice (with **id = 1**) by the amount of the variable **transferAmount.** The variable **transferAmount** is being embedded into the query at the position of the **?** question mark placeholder. A second **UPDATE** statement increments Bob's account (**id = 2**) by the same amount. Before either of these updates occurs, the transaction is begun, and after, it is committed.

You can run this script a few times to confirm that the transfer works. The third time you run it, an error will occur due to the **CHECK** constraint preventing the balance from becoming negative. SQLite will automatically roll back the transaction, and you should find the database in the same state as it was before the last run:



This is good, because if the rollback didn't occur, Bob's balance would have been increased to 35 and it would have been as if money was created from thin air! (I guess it's not good for Bob.)

In a new terminal, start the **sqlite3** command-line interface and run the script **reset-accounts.sql** to restore the original state of the database. Let's now try introducing a different kind of error; we can simulate a server crash by terminating the Node.js program after the first update with the following code:

```
process.exit();
```

Paste this code between the two **UPDATE** statements, and run the script. The program will terminate prematurely, but once again, you should find that the transaction has been rolled back and the no changes have been made to the database. Comment out the **process.exit();** line.

Task 3: Rolling Back

Rather than relying on transactions only in error and failure cases, we can write programs that intentionally roll back the transaction when conditions are not right for it to continue. The following code handles this is with a **try catch** clause, responding to errors—gracefully and without a crash—that are unpredictable or unexpected:

```
db.prepare(`
    BEGIN TRANSACTION;
    `)
    .run();

let transferAmount = 5;

try {
    db.prepare(`
        UPDATE accounts
        SET balance = balance - ?
        WHERE id = 1;
        `)
        .run(transferAmount);

    db.prepare(`
        UPDATE accounts
        SET balance = balance + ?
        WHERE id = 2;
        `)
        .run(transferAmount);

    db.prepare(`
        COMMIT;
        `)
        .run();
}
catch (error) {
    db.prepare('ROLLBACK;').run();
    console.log("Something went wrong; transaction rolled back.");
}
```

As you can see, the UPDATE statements (which may be prone to errors and constraint violations) are safely contained within the **try** block, while the **catch** block takes over and rolls back in the case of an error.