Transactions

Members:

1. Athiyo Chakma (2022118)
2. Aditya Yadav (2022041)
3. Aastha SIngh (2022009)

1. Four Non-Conflicting:

1. Display All Electronics Products from Products Table:

   START TRANSACTION;

   SELECT * FROM Products WHERE category = 'Electronics';

   COMMIT;

```
+------------+-------+--------------+-----------+-------------+--------------------+-----------------------------------------+
-------+
| product_id | stock | product_name | review_id | category    | subcategory        | description                             |
 price |
+------------+-------+--------------+-----------+-------------+--------------------+-----------------------------------------+
-------+
|          1 |     7 | Laptop       |         4 | Electronics | Computers          | High-performance laptop                 |
  1000 |
|          4 |    30 | LED TV       |         7 | Electronics | Television         | Smart LED TV with HD display            |
   700 |
|          7 |     5 | Headphones   |        10 | Electronics | Audio              | Wireless headphones with noise cancellation |
    50 |
|         10 |    12 | Rice Cooker  |        13 | Electronics | Kitchen Appliances | Multi-functional rice cooker            |
    80 |
+------------+-------+--------------+-----------+-------------+--------------------+-----------------------------------------+
-------+
4 rows in set (0.00 sec)
```

   Explanation:-
   This transaction only retrieves data from the Products table based on specific condition (category = ‘Electronics’) without modifying it, making it non-conflicting as it doesn't impact other transactions.

2. Update Stocks:

   START TRANSACTION;

   UPDATE Products SET stock = stock - 5 WHERE product_id = 1;

   COMMIT;

```
mysql> UPDATE Products SET stock = stock - 5 WHERE product_id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Explanation:-
This transaction updates the stock of a specific product, which is independent of other product updates, ensuring concurrency without conflicts.

3. Update Price:

START TRANSACTION;

UPDATE Products SET price = 1200 WHERE product_id = 4;

COMMIT;

```
mysql> UPDATE Products SET price = 1200 WHERE product_id = 4;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Explanation:-
This transaction updates the price of a different product, operating on separate data from the previous transaction, thus non-conflicting.

4. Display Customers from New York:

START TRANSACTION;

SELECT * FROM customer WHERE city = 'New York';

COMMIT;

```
mysql> SELECT * FROM customer WHERE city = 'New York';
+---------+--------------+-----------+------------+-----------+----------+---------+----------+-------+----------+-----------------------------+
| user_id | phone_number | password  | first_name | last_name | name     | flat_no | city     | state | zip_code | address                     |
+---------+--------------+-----------+------------+-----------+----------+---------+----------+-------+----------+-----------------------------+
|       1 | 1234567890   | password1 | John       | Doe       | John Doe | Apt 101 | New York | NY    | 10001    | Apt 101, New York, NY 10001 |
+---------+--------------+-----------+------------+-----------+----------+---------+----------+-------+----------+-----------------------------+
1 row in set (0.00 sec)
```

Explanation:-
This transaction retrieves customer data based on a specific condition (city = 'New York'), which is separate from product-related transactions, allowing it to execute concurrently without conflict.

2. Two Conflicting:

1. Two employees are updating the price of the same product at the same time without knowing of each other's actions.

   Transaction A (Promotional Discount) :

   START TRANSACTION;

SELECT price INTO @current_price FROM Products WHERE product_id = 1;

SET @new_price = @current_price * 0.90;

UPDATE Products SET price = @new_price WHERE product_id = 1;

COMMIT;

Transaction B (Cost Adjustment) :

START TRANSACTION;

SELECT price INTO @current_price FROM Products WHERE product_id = 1;

SET @new_price = @current_price + 5;

UPDATE Products SET price = @new_price WHERE product_id = 1;

COMMIT;

Explanation:

If these transactions are executed simultaneously and without proper isolation, one of the following might happen:

- **Employee A's** transaction executes first, setting the new price to $900. Then, **Employee B's** transaction, which reads the old price of $1000, executes and sets the new price to $1005, effectively overwriting the promotional discount.
- Or vice versa, where **Employee B** goes first, followed by **Employee A**, and the price set by **Employee B** is inadvertently reduced by the promotional discount, resulting in a price not reflecting the updated supplier cost.

This is a classic **lost update problem**. One transaction's changes are lost because another transaction overwrites the changes without knowing the first transaction's actions.

2. Read-Write Conflict (Dirty Read)

Transaction-C: Read product information of product-1 and then update its stock.

START TRANSACTION;

SELECT * FROM Products WHERE product_id = 1;

-- Simulated delay to allow another transaction to intervene

```
DO SLEEP(5);

UPDATE Products SET stock = stock - 3 WHERE product_id = 1;

COMMIT;
```

Transaction-D: Simultaneously update the same product's stock

```
START TRANSACTION;

UPDATE Products SET stock = stock + 8 WHERE product_id = 1;

COMMIT;
```

Explanation:-

In this scenario, **Transaction-C** initially reads product information for product-1 and then updates its stock after a delay. However, before **Transaction-C** completes its update, **Transaction-D** simultaneously updates the stock of the same product.

The conflict arises due to the possibility of **Transaction-D** updating the stock of product-1 while **Transaction-C** is still processing. If **Transaction-D**'s update occurs before **Transaction-C** completes its update, **Transaction-C** may act upon outdated data (i.e Previous Stock) , resulting in a "dirty read" where **Transaction-C** makes decisions based on stale information. This can lead to inconsistencies in the stock quantity, as **Transaction-C** may unintentionally overwrite or conflict with the changes made by **Transaction-D**, potentially resulting in incorrect stock levels or lost updates.

When the admin updates the products table a conflict may price on deciding the price for that we use lock in the products table to avoid such a conflict
Similar lock exist for payment table