Flat and Nested Distributed Transaction

Prepared by:
Anish Raj Manandhar(THA077BCT010)
Nabin Shrestha(THA077BCT026)
Prayush Bhattarai(THA077BCT036)

Prepared for:
Department of Electronics
And Computer Engineering

Introduction to Transaction

A transaction is a series of object operations that must be done in an ACID-compliant manner.

- **Atomicity**

The transaction is completed entirely or not at all.

- **Consistency**

It is a term that refers to the transition from one consistent state to another.

- **Isolation**

It is carried out separately from other transactions.

- **Durability**

Once completed, it is long lasting.

For Example

Transaction	YES	NO
Atomicity		 OR 
Consistency	 	 
Isolation	 	 
Durability	 	 

Why Distributed Transaction?

- A distributed transaction is a database transaction in which two or more network hosts are involved.
- Some properties are harder to implement, that cannot be implemented with simple transactions. Many times basic single-system techniques are not sufficient.
- Distributed transactions are required when there is a need to quickly update data that is related and spread across the multiple databases or nodes connected in a network.

Fig: Un-Distributed Transaction

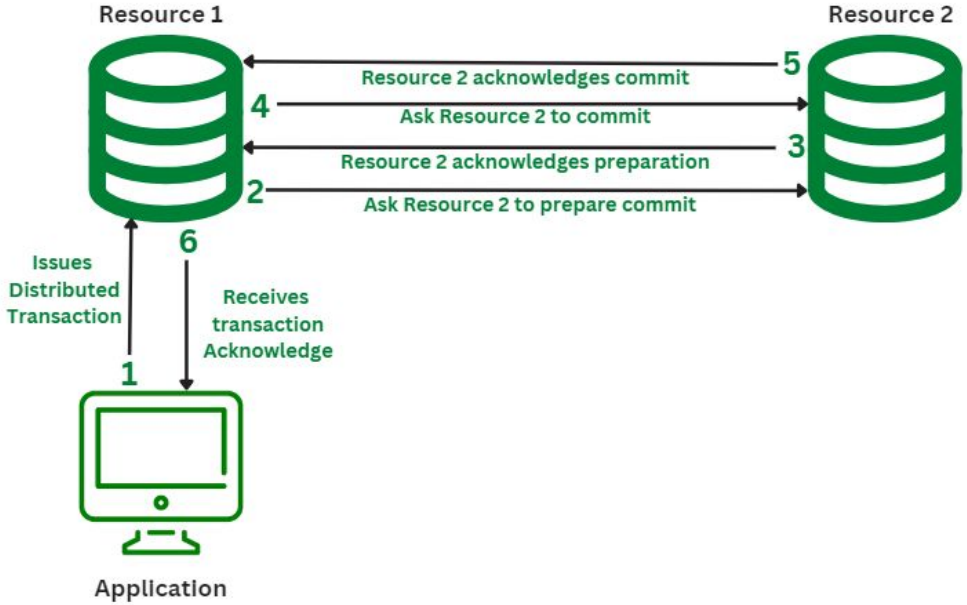
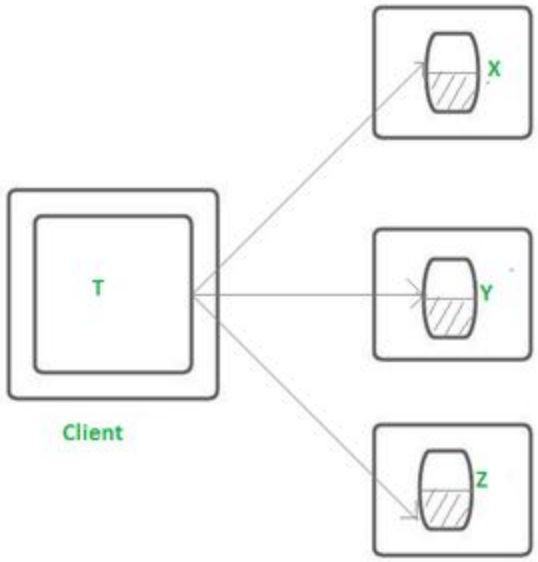


Fig: Distributed Transaction

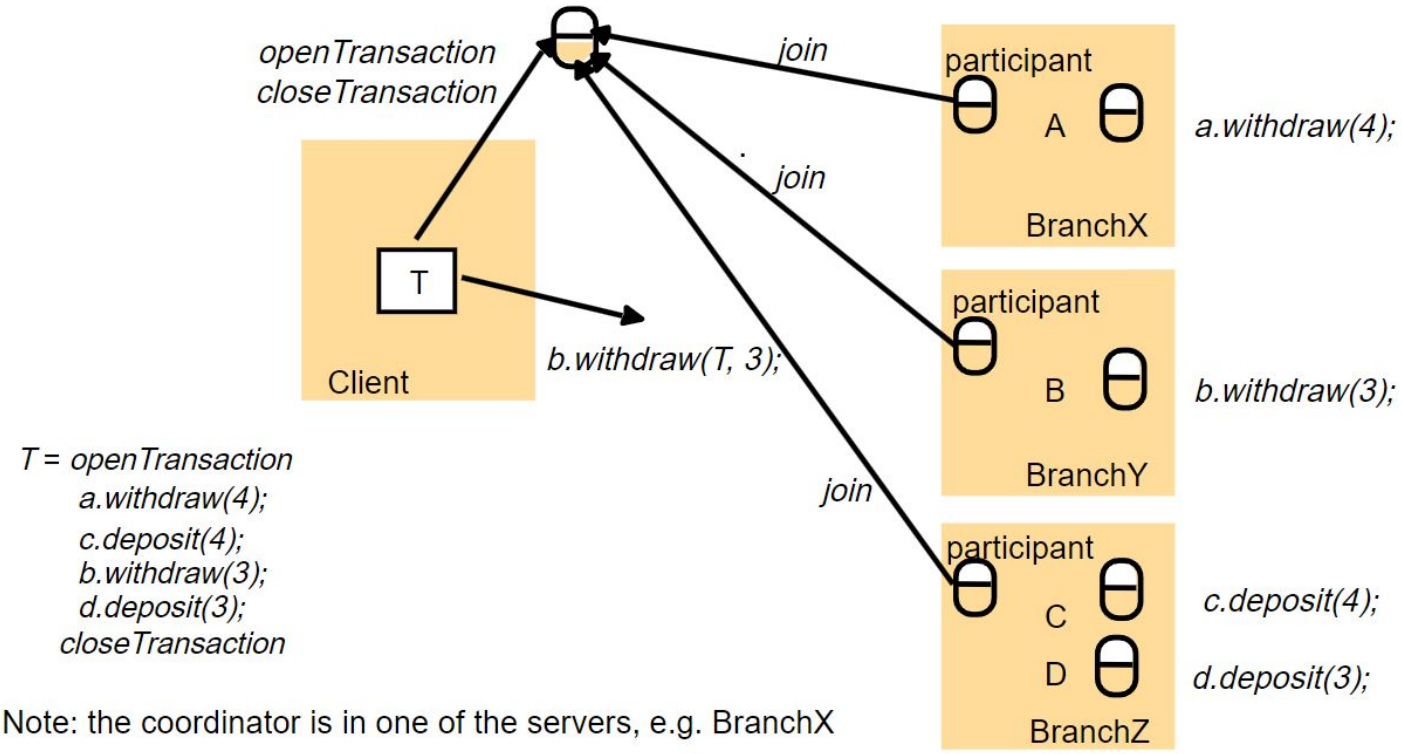
Flat Distributed Transactions[1]

- Has a single initiating point(begin) and a single end point(Commit or abort).
- Usually very simple and generally used for short activities rather than larger ones.

- A client makes requests to multiple servers in a flat transaction.
- For eg. T is a flat transaction that performs operations on objects in servers X, Y, and Z.
- Before moving on to the next request, a flat transaction completes the previous one. As a result, each transaction visits the server object in order.



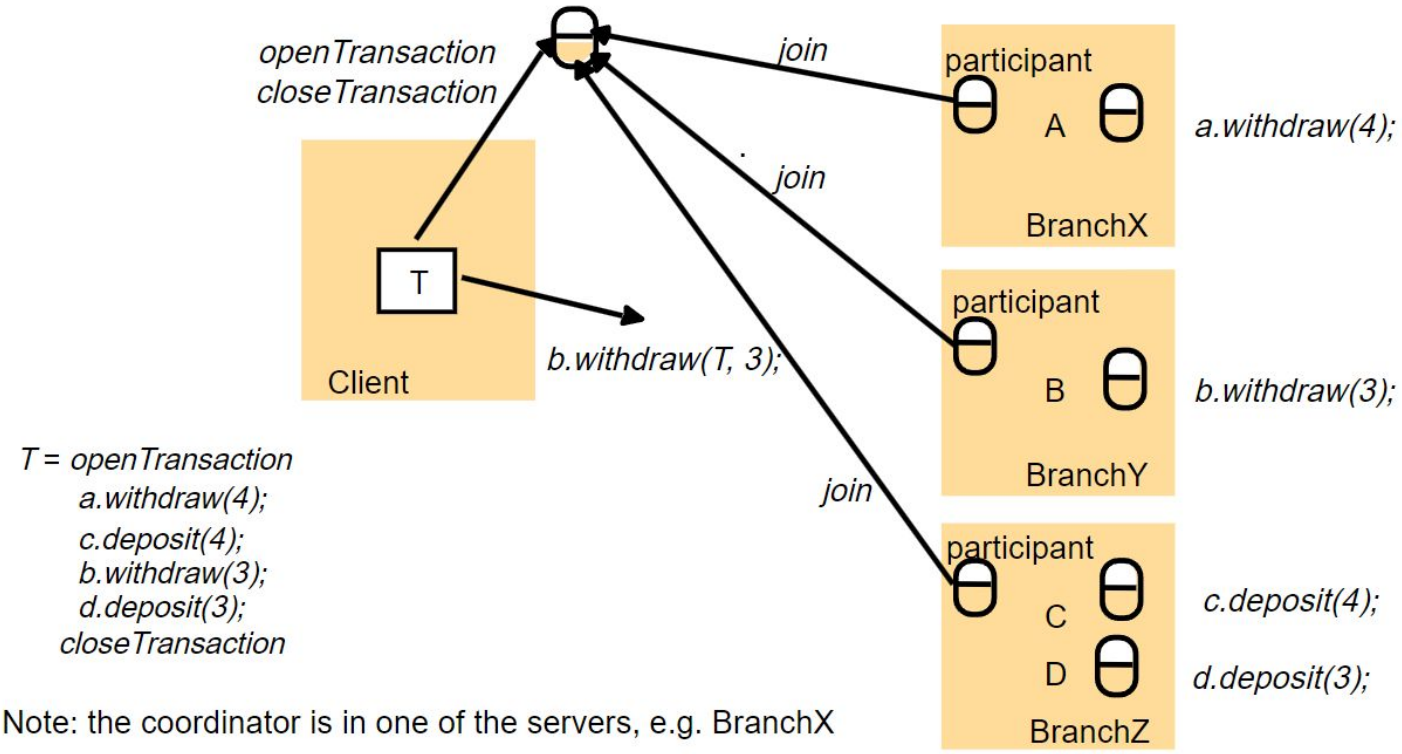
Flat Distributed Transactions[2]



Servers execute requests in a distributed transaction.

- When it commits they must communicate with one another to coordinate their actions.
- A Client starts a transaction by sending an *openTransaction* request to a coordinator in any server.
  - It returns a TID unique in the distributed system (eg. serverID)
  - At the end, it will be responsible for committing or aborting it.

Flat Distributed Transactions[3]



- Each server managing an object accessed by the transaction is a participant – it joins the transaction.
  - A participant keeps track of objects involved in the transaction.
  - At the end it cooperates with the coordinator in carrying out the commit protocol.
- Note that a participant can call *abortTransaction* in coordinator.

For Example

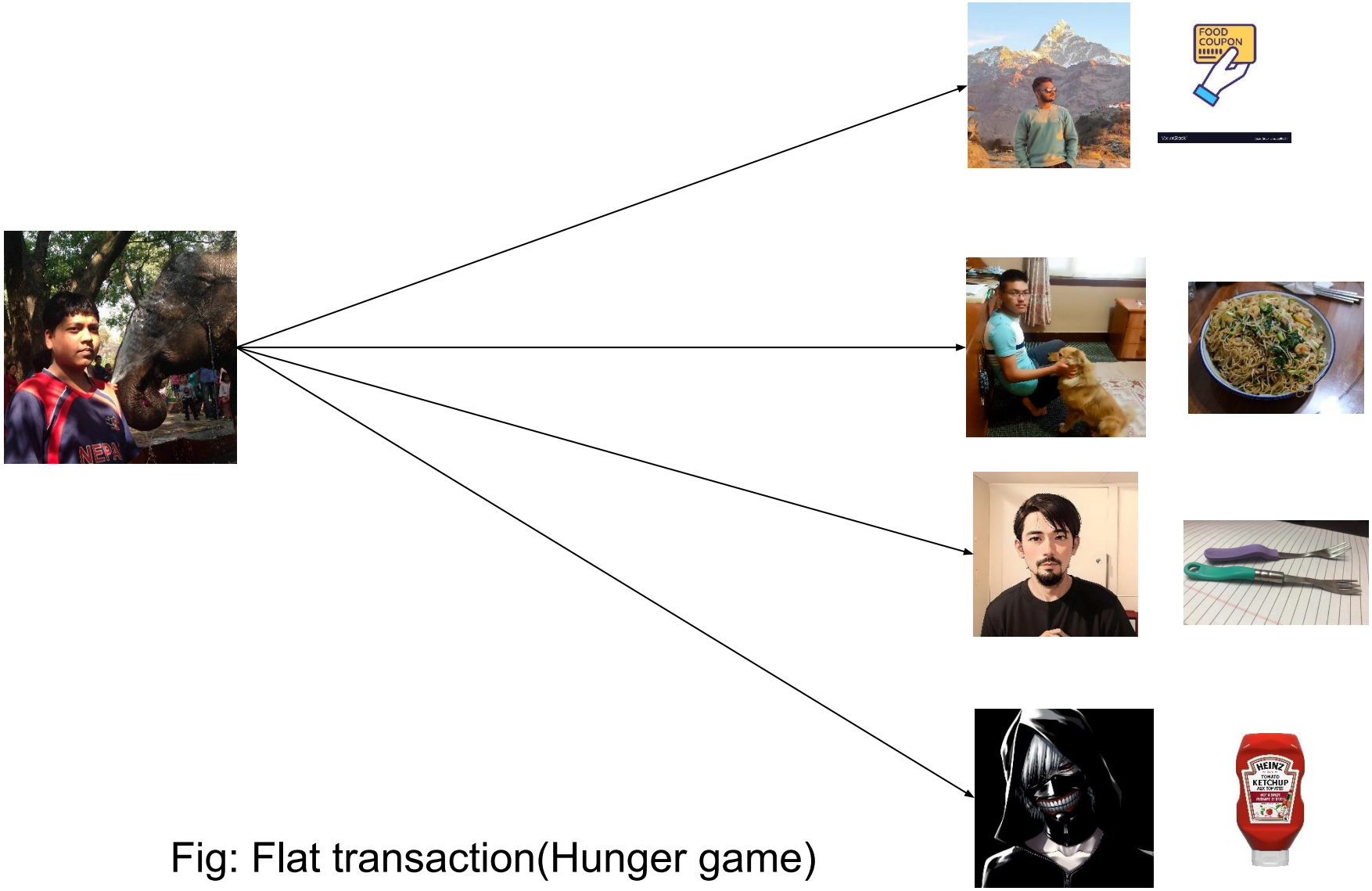
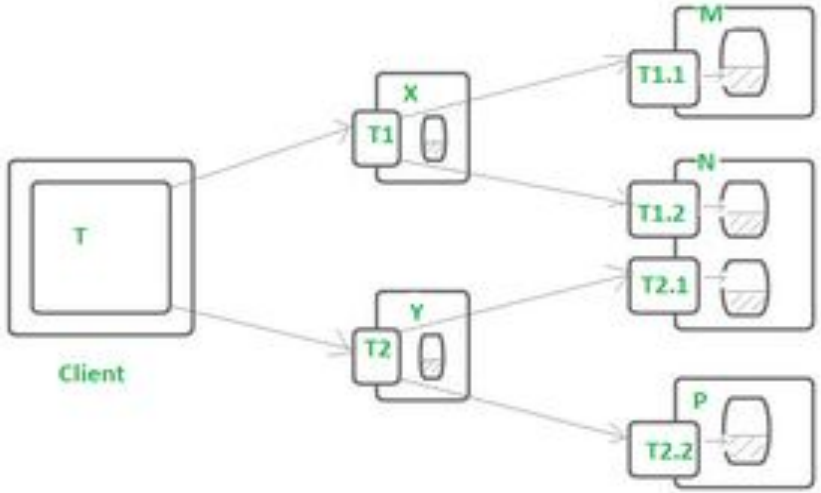


Fig: Flat transaction(Hunger game)

Nested Transaction-[1]

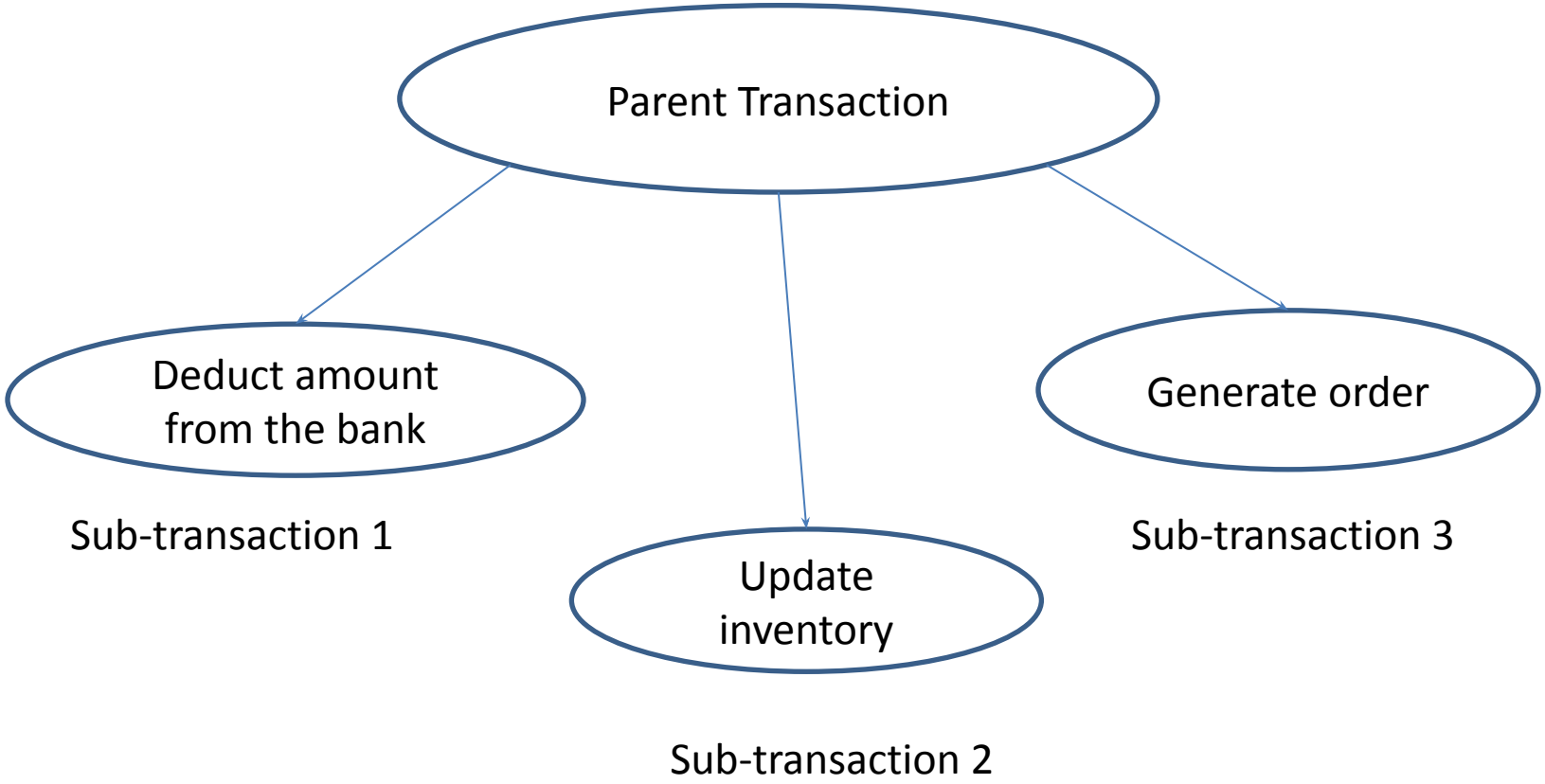
- A transaction contains other transactions.
- The nested transactions here are called sub-transactions.
- The top-level transaction in a nested transaction can open sub-transactions, and each sub-transaction can open more sub-transactions down to any depth of nesting.

Nested Transaction-[2]



- Concurrent Execution of the Sub-transactions is done which are at the same level.
- Here, in the above diagram, T1 and T2 invoke objects on different servers and hence they can run in parallel and are therefore concurrent.

Nested Transaction-[3]
Example: E-commerce



Comparison

Flat Transactions:

- Single-level transactions without sub-transactions
- All operations must succeed or fail as a single unit
- Simpler structure, harder to manage complex scenarios

Nested Transactions:

- Hierarchical transactions with sub-transactions
- Each sub-transaction can commit or rollback independently
- Better suited for complex, distributed operations

Implementation (Flat Transaction)

```python
import mysql.connector
from mysql.connector import errorcode

def transfer_money(cursor, from_account, to_account, amount):
    try:
        # Check if the from_account has enough balance
        cursor.execute("SELECT balance FROM accounts WHERE name = %s", (from_account,))
        from_balance = cursor.fetchone()[0]

        if amount <= 0:
            raise Exception("Check Input field!!!")

        if from_balance < amount:
            raise Exception("Insufficient funds")

        # Deduct the amount from from_account
        cursor.execute("UPDATE accounts SET balance = balance - %s WHERE name = %s", (amount, from_account))

        # Add the amount to to_account
        cursor.execute("UPDATE accounts SET balance = balance + %s WHERE name = %s", (amount, to_account))

    except Exception as e:
        print(f"Transaction failed: {e}")
        return False

    return True
```

Implementation (Flat Transaction)

id	name	balance
1	Alice	1500.00
2	Bob	500.00

Committed Test Case

```
conn.start_transaction()

# Perform the transfer
if transfer_money(cursor, 'Alice', 'Bob', 100.00):
    # Commit the transaction
    conn.commit()
    print("Transaction committed successfully")
else:
    # Rollback the transaction
    conn.rollback()
    print("Transaction rolled back")
```

```
Transaction committed successfully
```

id	name	balance
1	Alice	1400.00
2	Bob	600.00

Rollback Test Case

```
conn.start_transaction()

# Perform the transfer
if transfer_money(cursor, 'Alice', 'Bob', 0.00):
    # Commit the transaction
    conn.commit()
    print("Transaction committed successfully")
else:
    # Rollback the transaction
    conn.rollback()
    print("Transaction rolled back")
```

```
Transaction failed: Check Input field!!!
Transaction rolled back
```

Implementation (Nested Transaction)

```python
# Function to update inventory
def update_inventory(conn, product_name, quantity):
    try:
        cursor = conn.cursor()

        # Check current inventory
        cursor.execute("SELECT quantity FROM inventory WHERE product_name = %s", (product_name,))
        current_quantity = cursor.fetchone()[0]

        # Check if sufficient quantity is available
        if current_quantity < quantity:
            raise Exception(f"Insufficient quantity available for {product_name}")

        # Update inventory
        cursor.execute("UPDATE inventory SET quantity = quantity - %s WHERE product_name = %s", (quantity, product_name))
        cursor.close()

    except Exception as e:
        raise e
```

Implementation (Nested Transaction)

```python
# Function to process payment
def process_payment(conn, order_id, amount):
    try:
        cursor = conn.cursor()
        # Simulate payment failure
        if amount <= 0:
            raise Exception("Invalid payment amount")

        # Process payment (example: insert into payments table)
        cursor.execute("INSERT INTO payments (order_id, amount) VALUES (%s, %s)", (order_id, amount))
        cursor.close()
    except Exception as e:
        raise e
# Function to record shipping information
def record_shipping(conn, order_id, shipping_address):
    try:
        cursor = conn.cursor()

        # Simulate shipping failure
        if not shipping_address:
            raise Exception("Invalid shipping address")

        # Record shipping information (example: insert into shipping table)
        cursor.execute("INSERT INTO shipping (order_id, address) VALUES (%s, %s)", (order_id, shipping_address))

        cursor.close()

    except Exception as e:
        raise e
```

Implementation (Nested Transaction)

```python
# Start a transaction
conn.start_transaction()

# Process each step within the transaction
try:
    # Step 1: Update inventory
    update_inventory(conn, 'Sneakers', 5)

    # Step 2: Process payment
    process_payment(conn, 100, 4000)

    # Step 3: Record shipping information
    record_shipping(conn, 100, 'Lazimpat')

    # Commit the transaction if all steps succeed
    conn.commit()
    print("Order processed successfully.")

except Exception as e:
    conn.rollback()
    print(f"Transaction failed: {e}")
```

id	product_name	quantity
100	Sneakers	15

```
Order processed successfully.
```

id	product_name	quantity
100	Sneakers	10

THANK YOU