ACID Properties:

1. **Atomicity:**
   - Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all its operations are executed, or none are.

   Example (using SQL):
   ```sql
   -- SQL Server example
   BEGIN TRANSACTION;
   UPDATE Account SET Balance = Balance - 100 WHERE AccountID = 123;
   INSERT INTO TransactionHistory (AccountID, Amount) VALUES (123, -100);
   COMMIT;
   ```

   In this example, the transaction deducts $100 from the account balance and records the transaction in the history. If any part of the transaction fails (e.g., due to an error), the entire transaction will be rolled back, and the database will remain unchanged.

2. **Consistency:**
   - Consistency ensures that a transaction brings the database from one valid state to another. The database must satisfy a set of integrity constraints before and after the transaction.

   Example (using Python and a fictional database library):
   ```python
   import db_library

   def transfer_funds(source_account, target_account, amount):
       db = db_library.connect()

       try:
           db.begin_transaction()

           # Check if source account has sufficient balance
           if db.get_balance(source_account) < amount:
               raise ValueError("Insufficient funds")

           # Perform the transfer
           db.update_balance(source_account, db.get_balance(source_account) - amount)
           db.update_balance(target_account, db.get_balance(target_account) + amount)

           db.commit()
   ```

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

    finally:
        db.close()
```

In this Python example, the `transfer_funds` function ensures that the database remains in a consistent state by checking for sufficient funds before transferring money between accounts.

3. **Isolation:**
   - Isolation ensures that concurrent execution of transactions does not result in interference. Each transaction should be isolated from others until it is committed.

   Example (using Java and JDBC):
   ```java
   Connection connection = DriverManager.getConnection(url, user, password);
   connection.setAutoCommit(false); // Disable auto-commit

   try {
       // Transaction 1
       Statement statement1 = connection.createStatement();
       statement1.executeUpdate("UPDATE Products SET Stock = Stock - 10 WHERE ProductID = 101");

       // Simulate a delay in the second transaction
       Thread.sleep(5000);

       // Transaction 2
       Statement statement2 = connection.createStatement();
       statement2.executeUpdate("UPDATE Products SET Stock = Stock + 10 WHERE ProductID = 102");

       connection.commit();

   } catch (SQLException | InterruptedException e) {
       connection.rollback();
       e.printStackTrace();
   } finally {
       connection.close();
   }
   ```

In this Java example, two transactions are executed concurrently. The delay introduced between them simulates potential concurrency issues. The database connection is set to manual commit mode, allowing explicit control over when to commit or rollback.

4. **Durability:**
   - Durability ensures that once a transaction is committed, its changes are permanent and will survive subsequent failures.

   Example (using a web application with a database):
   ```python
   # Flask web application example
   from flask import Flask, request
   import db_library

   app = Flask(__name__)

   @app.route('/make_purchase', methods=['POST'])
   def make_purchase():
       try:
           # Process purchase transaction
           purchase_amount = request.json['amount']
           user_id = request.json['user_id']

           db = db_library.connect()
           db.begin_transaction()

           # Deduct funds from the user's account
           db.update_balance(user_id, db.get_balance(user_id) - purchase_amount)

           # Record the purchase in the transaction history
           db.record_purchase(user_id, purchase_amount)

           db.commit()

           return "Purchase successful"

       except Exception as e:
           db.rollback()
           return f"Transaction failed: {e}"

       finally:
           db.close()
   ```

In this Flask web application example, the purchase transaction is committed only if all the steps (deducting funds and recording the purchase) are successful. If any part fails, the entire transaction is rolled back to maintain durability.

Certainly! Let's continue with examples of the COMMIT and ROLLBACK statements, which are used to either confirm or undo the changes made during a transaction.

COMMIT Statement:

The COMMIT statement is used to permanently save the changes made during a transaction.

1. **Example in SQL:**
   ```sql
   -- SQL Server example
   BEGIN TRANSACTION;

   UPDATE Employees SET Salary = Salary + 500 WHERE Department = 'IT';

   -- Check if the update is correct before committing
   -- If satisfied, commit the transaction
   COMMIT;
   ```

   In this SQL example, a transaction updates the salary of employees in the IT department. The COMMIT statement is used to make the changes permanent only if the update is correct and meets the desired criteria.

2. **Example in Python (using a fictional database library):**
   ```python
   import db_library

   def add_new_employee(employee_data):
       db = db_library.connect()

       try:
           db.begin_transaction()

           # Insert a new employee into the database
           db.insert_employee(employee_data)

           # If all steps are successful, commit the transaction
           db.commit()
   ```

```
        print("Employee added successfully")

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

    finally:
        db.close()
```

In this Python example, the `add_new_employee` function inserts a new employee into the database. If the insertion is successful, the COMMIT statement is used to make the addition permanent.

ROLLBACK Statement:

The ROLLBACK statement is used to undo the changes made during a transaction.

1. **Example in SQL:**
   ```sql
   -- SQL Server example
   BEGIN TRANSACTION;

   UPDATE Inventory SET Quantity = Quantity - 20 WHERE ProductID = 101;

   -- If there's an issue, rollback the entire transaction
   ROLLBACK;
   ```

   In this SQL example, a transaction attempts to update the inventory quantity for a product. If there's an issue, the ROLLBACK statement is used to undo the changes made during the transaction.

2. **Example in Java (using JDBC):**
   ```java
   Connection connection = DriverManager.getConnection(url, user, password);
   connection.setAutoCommit(false); // Disable auto-commit

   try {
       // Transaction to transfer funds
       Statement statement = connection.createStatement();
       statement.executeUpdate("UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 123");
```

```
        statement.executeUpdate("UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 456");

        // If a condition is not met, rollback the entire transaction
        if (someConditionNotMet()) {
            connection.rollback();
        } else {
            connection.commit();
        }

    } catch (SQLException e) {
        connection.rollback();
        e.printStackTrace();
    } finally {
        connection.close();
    }
```

In this Java example using JDBC, a transaction updates the balance in two accounts. If a specific condition is not met (simulated by the `someConditionNotMet` function), the ROLLBACK statement is used to undo the changes made during the transaction. Otherwise, the COMMIT statement is used to confirm the changes.

These examples illustrate how COMMIT and ROLLBACK statements are used in different programming languages and database systems to control the outcome of transactions.