**Exercise 1: Simple Transaction**

1. Create a table called **bank\_accounts** with columns: **account\_id**, **account\_holder**, and **balance**.
2. Insert two rows into the **bank\_accounts** table with sample data.
3. Start a transaction to update the balance of both accounts. Deduct $100 from one account and add $100 to the other.
4. Commit the transaction.

**Answer 1: Simple Transaction**

-- Create the table

CREATE TABLE bank\_accounts (

account\_id INT PRIMARY KEY,

account\_holder VARCHAR(255),

balance DECIMAL(10, 2)

);

-- Insert sample data

INSERT INTO bank\_accounts (account\_id, account\_holder, balance) VALUES

(1, 'John Doe', 1000.00),

(2, 'Jane Smith', 750.00);

-- Start a transaction

BEGIN;

-- Update balances

UPDATE bank\_accounts SET balance = balance - 100 WHERE account\_id = 1;

UPDATE bank\_accounts SET balance = balance + 100 WHERE account\_id = 2;

-- Commit the transaction

COMMIT;

**Exercise 2: Rollback Transaction**

1. Start a transaction.
2. Insert a new row into the **bank\_accounts** table.
3. Attempt to update a non-existent account.
4. Rollback the transaction.

**Answer 2: Rollback Transaction**

-- Start a transaction

BEGIN;

-- Insert a new row

INSERT INTO bank\_accounts (account\_id, account\_holder, balance) VALUES (3, 'Alice Johnson', 1200.00);

-- Attempt to update a non-existent account

UPDATE bank\_accounts SET balance = balance - 100 WHERE account\_id = 4;

-- Rollback the transaction

ROLLBACK;

**Exercise 3: Savepoint**

1. Start a transaction.
2. Insert a new row into the **bank\_accounts** table.
3. Create a savepoint named "insert\_savepoint."
4. Attempt to update a non-existent account.
5. Rollback to the "insert\_savepoint."
6. Commit the transaction.

**Answer 3: Savepoint**

-- Start a transaction

BEGIN;

-- Insert a new row

INSERT INTO bank\_accounts (account\_id, account\_holder, balance) VALUES (4, 'Bob Williams', 1500.00);

-- Create a savepoint

SAVEPOINT insert\_savepoint;

-- Attempt to update a non-existent account

UPDATE bank\_accounts SET balance = balance - 100 WHERE account\_id = 5;

-- Rollback to the savepoint

ROLLBACK TO insert\_savepoint;

-- Commit the transaction

COMMIT;

**Exercise 4: Nested Transactions**

1. Start a transaction.
2. Insert a new row into the **bank\_accounts** table.
3. Start a nested transaction.
4. Attempt to update a non-existent account in the nested transaction.
5. Rollback the nested transaction.
6. Commit the outer transaction.

**Answer 4: Nested Transactions**

-- Start a transaction

BEGIN;

-- Insert a new row

INSERT INTO bank\_accounts (account\_id, account\_holder, balance) VALUES (5, 'Eva Lee', 1800.00);

-- Start a nested transaction

SAVEPOINT outer\_savepoint;

-- Attempt to update a non-existent account in the nested transaction

UPDATE bank\_accounts SET balance = balance - 100 WHERE account\_id = 6;

-- Rollback the nested transaction

ROLLBACK TO outer\_savepoint;

-- Commit the outer transaction

COMMIT;

**Exercise 5: Concurrent Transactions**

1. Open two MySQL sessions.
2. In the first session, start a transaction and update the balance of an account.
3. In the second session, try to update the same account within its own transaction.
4. Commit the first session's transaction.
5. Commit the second session's transaction.

**Answer 5: Concurrent Transactions**

Session 1:

-- Start a transaction in Session 1

BEGIN;

-- Update the balance in Session 1

UPDATE bank\_accounts SET balance = balance - 50 WHERE account\_id = 1;

-- Commit the transaction in Session 1

COMMIT;

Session 2:

-- Start a transaction in Session 2

BEGIN;

-- Try to update the same account in Session 2

UPDATE bank\_accounts SET balance = balance - 30 WHERE account\_id = 1;

-- Commit the transaction in Session 2

COMMIT;

**Exercise 6: Handling Deadlocks**

1. Open two MySQL sessions.
2. In the first session, start a transaction and update the balance of account 1.
3. In the second session, start a transaction and update the balance of account 2.
4. In the first session, try to update the balance of account 2.
5. In the second session, try to update the balance of account 1.
6. Handle the deadlock situation using **ROLLBACK** and **RETRY** or other appropriate methods.

**Answer 6: Handling Deadlocks**

Session 1:

-- Start a transaction in Session 1

BEGIN;

-- Update the balance of account 1 in Session 1

UPDATE bank\_accounts SET balance = balance - 50 WHERE account\_id = 1;

-- Try to update the balance of account 2 in Session 1 (may result in a deadlock)

-- Handle deadlock (e.g., rollback and retry)

ROLLBACK;

Session 2:

-- Start a transaction in Session 2

BEGIN;

-- Update the balance of account 2 in Session 2

UPDATE bank\_accounts SET balance = balance - 30 WHERE account\_id = 2;

-- Try to update the balance of account 1 in Session 2 (may result in a deadlock)

-- Handle deadlock (e.g., rollback and retry)

ROLLBACK;

**Exercise 7: Transaction Isolation Levels**

1. Set the isolation level to **READ UNCOMMITTED**.
2. In one session, start a transaction and select data from the **bank\_accounts** table.
3. In another session, start a transaction and update a row in the **bank\_accounts** table.
4. Observe the behavior of the first session in terms of reading uncommitted changes.

**Answer 7: Transaction Isolation Levels** Session 1:

-- Set isolation level to READ UNCOMMITTED

SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

-- Start a transaction in Session 1

BEGIN;

-- Select data from the bank\_accounts table in Session 1

SELECT \* FROM bank\_accounts;

-- Observe the uncommitted changes made in Session 2

Session 2:

-- Start a transaction in Session 2

BEGIN;

-- Update a row in the bank\_accounts table in Session 2

UPDATE bank\_accounts SET balance = balance - 20 WHERE account\_id = 3;

-- Commit the transaction in Session 2

COMMIT;

In this exercise, the first session will observe the uncommitted changes made in the second session due to the **READ UNCOMMITTED** isolation level.

**Exercise 8: Transaction Timeout**

1. Start a transaction with a timeout of 10 seconds.
2. Run a long-running query within the transaction.
3. Wait for the timeout to occur.
4. Rollback the transaction due to the timeout.

**Answer 8: Transaction Timeout**

-- Start a transaction with a timeout of 10 seconds

SET SESSION innodb\_lock\_wait\_timeout = 10;

START TRANSACTION;

-- Run a long-running query (e.g., a sleep command)

SELECT SLEEP(20);

-- Wait for the timeout to occur (transaction will be automatically rolled back)

In this exercise, the transaction will be automatically rolled back when the timeout (10 seconds) is reached.

**Exercise 9: Explicit Transaction ROLLBACK**

1. Start a transaction.
2. Insert a new row into the **bank\_accounts** table.
3. Rollback the transaction explicitly.

**Answer 9: Explicit Transaction ROLLBACK**

-- Start a transaction

START TRANSACTION;

-- Insert a new row

INSERT INTO bank\_accounts (account\_id, account\_holder, balance) VALUES (6, 'Michael Brown', 2000.00);

-- Rollback the transaction explicitly

ROLLBACK;

**Exercise 10: Simulate Locking and Deadlock Detection**

1. Start a transaction.
2. Select data from the **bank\_accounts** table for an account.
3. In another session, attempt to update the same account.
4. Simulate a deadlock situation.
5. Observe MySQL's deadlock detection and resolution mechanism.

**Answer 10: Simulate Locking and Deadlock Detection**

Session 1:

-- Start a transaction in Session 1

START TRANSACTION;

-- Select data from the bank\_accounts table for an account

SELECT \* FROM bank\_accounts WHERE account\_id = 1;

-- In another session (Session 2), attempt to update the same account

-- Simulate a deadlock by not committing or rolling back in Session 1

Session 2:

-- In Session 2, attempt to update the same account

-- MySQL will detect the deadlock and automatically resolve it by rolling back one of the sessions

In this exercise, MySQL's built-in deadlock detection and resolution mechanism will handle the deadlock situation.