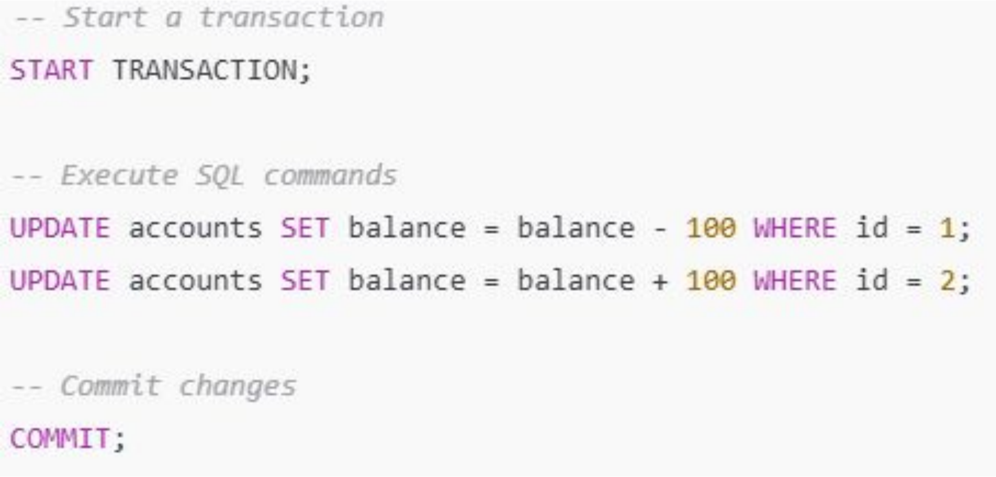
LESSON 10

Transactions and ACID Properties

Ihor Liutak

Introduction to Transactions

Definition: A transaction is a unit of work that is performed against a database. It consists of one or more SQL operations.



```
-- Start a transaction
START TRANSACTION;

-- Execute SQL commands
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;

-- Commit changes
COMMIT;
```

Why Transactions?

To ensure data consistency.

To handle errors gracefully.

Example Scenarios:

Bank transfers.

Booking systems (e.g., flights, hotels).

ACID Properties

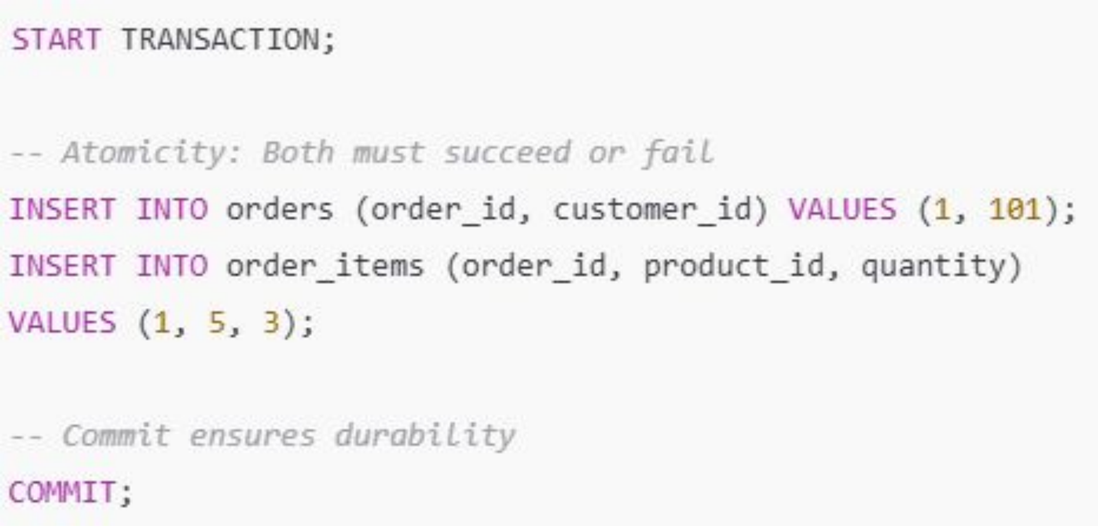
ACID Principles:

Atomicity: Entire transaction succeeds or fails.

Consistency: Data remains in a valid state.

Isolation: Transactions do not affect each other.

Durability: Changes persist even after a crash



```
START TRANSACTION;

-- Atomicity: Both must succeed or fail
INSERT INTO orders (order_id, customer_id) VALUES (1, 101);
INSERT INTO order_items (order_id, product_id, quantity)
VALUES (1, 5, 3);

-- Commit ensures durability
COMMIT;
```

Transaction Management Commands

START TRANSACTION / BEGIN:

Begins a new transaction.

COMMIT:

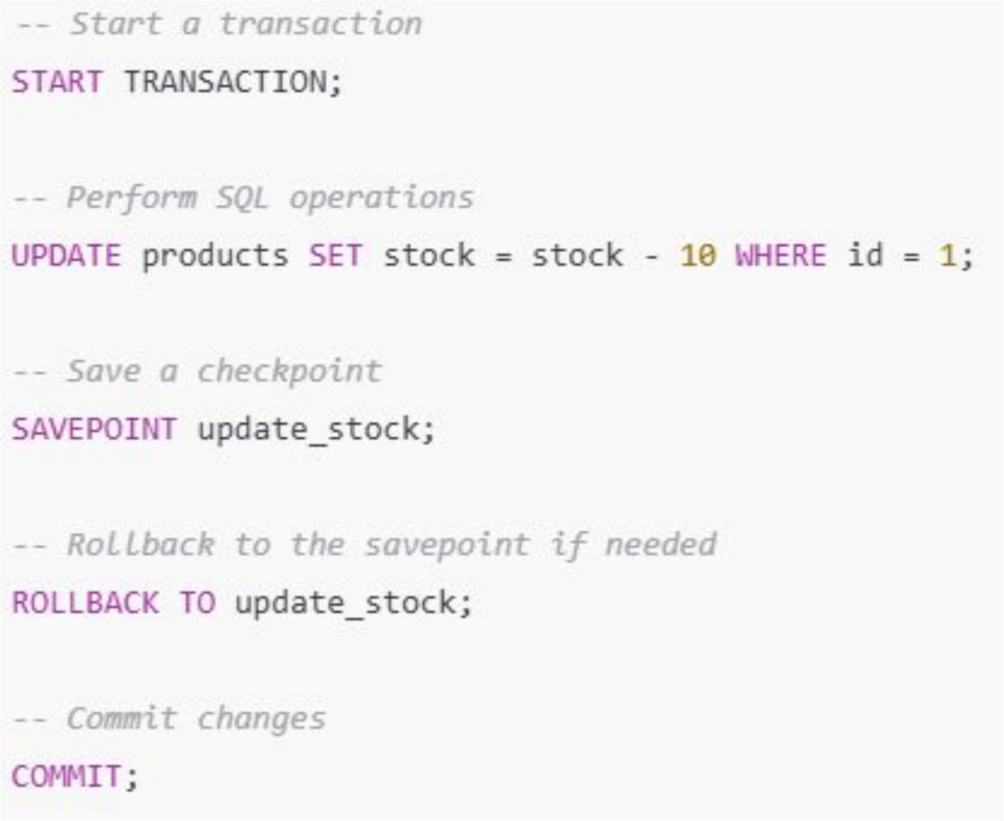
Saves all changes in the transaction.

ROLLBACK:

Undoes changes made during the transaction.

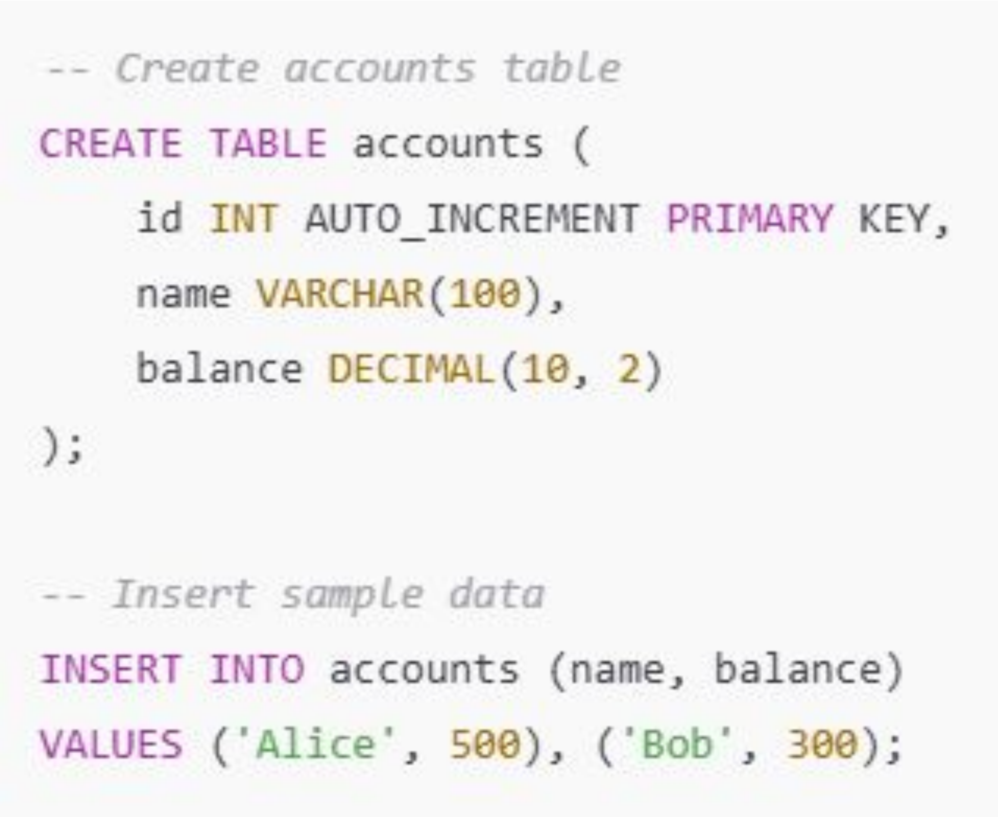
SAVEPOINT:

Sets a checkpoint to partially roll back.



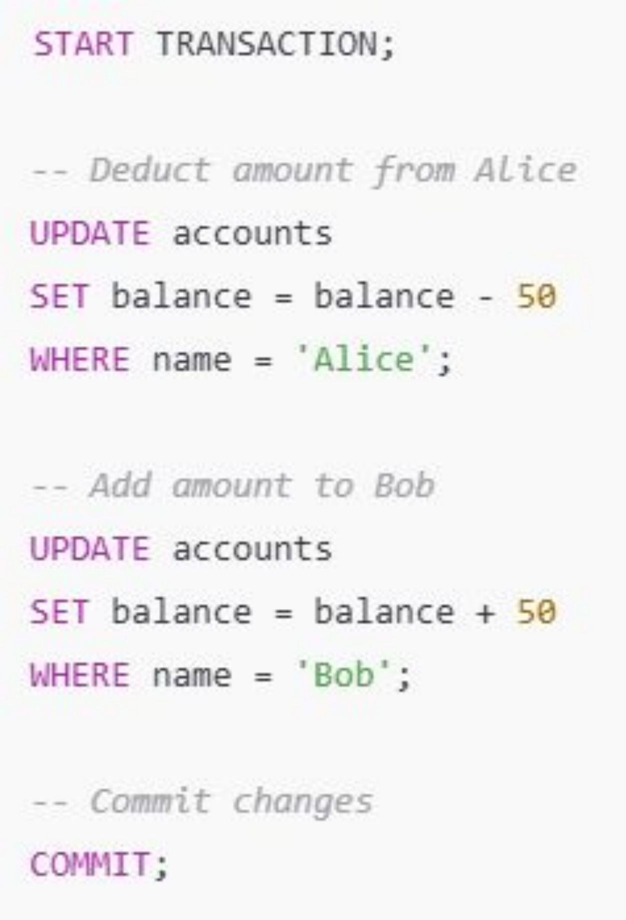
```sql
-- Start a transaction
START TRANSACTION;

-- Perform SQL operations
UPDATE products SET stock = stock - 10 WHERE id = 1;

-- Save a checkpoint
SAVEPOINT update_stock;

-- Rollback to the savepoint if needed
ROLLBACK TO update_stock;

-- Commit changes
COMMIT;
```

Lab: Setup



```sql
-- Create accounts table
CREATE TABLE accounts (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    balance DECIMAL(10, 2)
);

-- Insert sample data
INSERT INTO accounts (name, balance)
VALUES ('Alice', 500), ('Bob', 300);
```

Practice Basic Transaction Commands

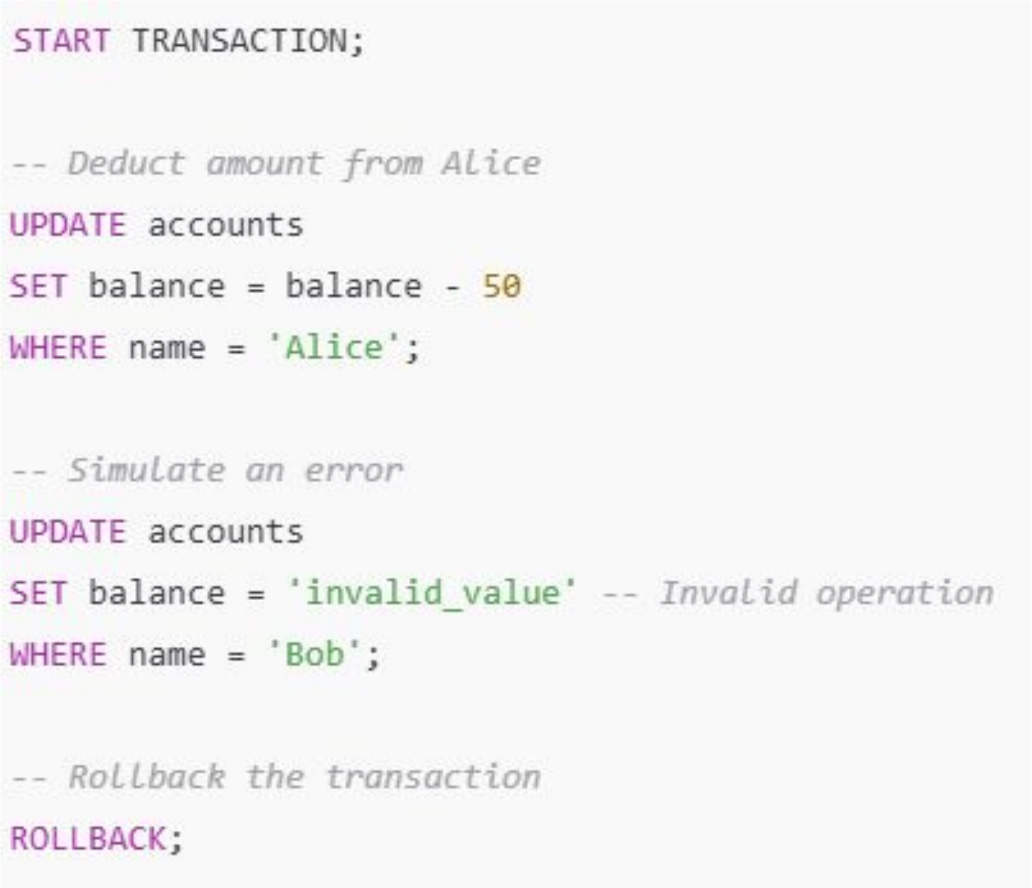


```sql
START TRANSACTION;

-- Deduct amount from Alice
UPDATE accounts
SET balance = balance - 50
WHERE name = 'Alice';

-- Add amount to Bob
UPDATE accounts
SET balance = balance + 50
WHERE name = 'Bob';

-- Commit changes
COMMIT;
```

Implement a Transfer

Deduct from one account and add to another.

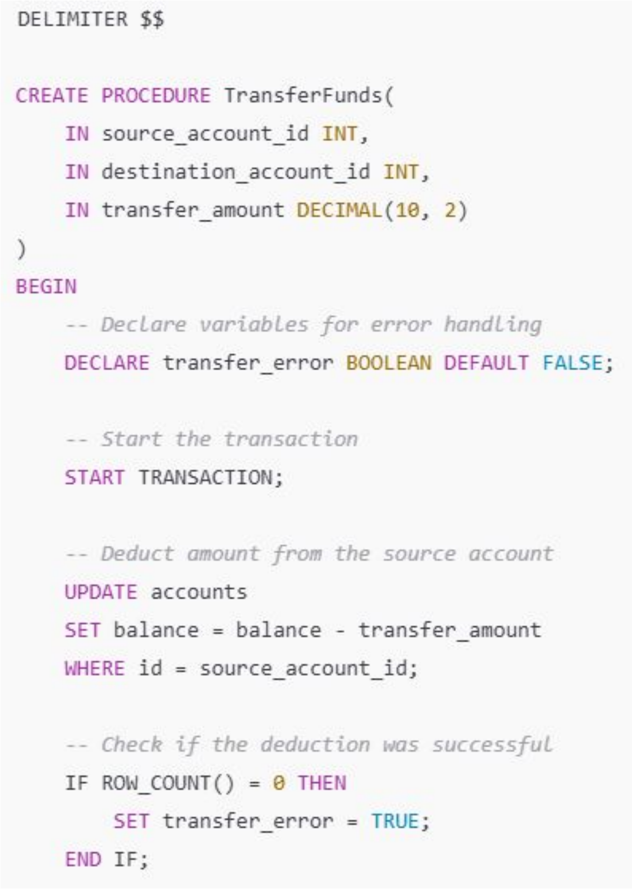
Simulate an Error



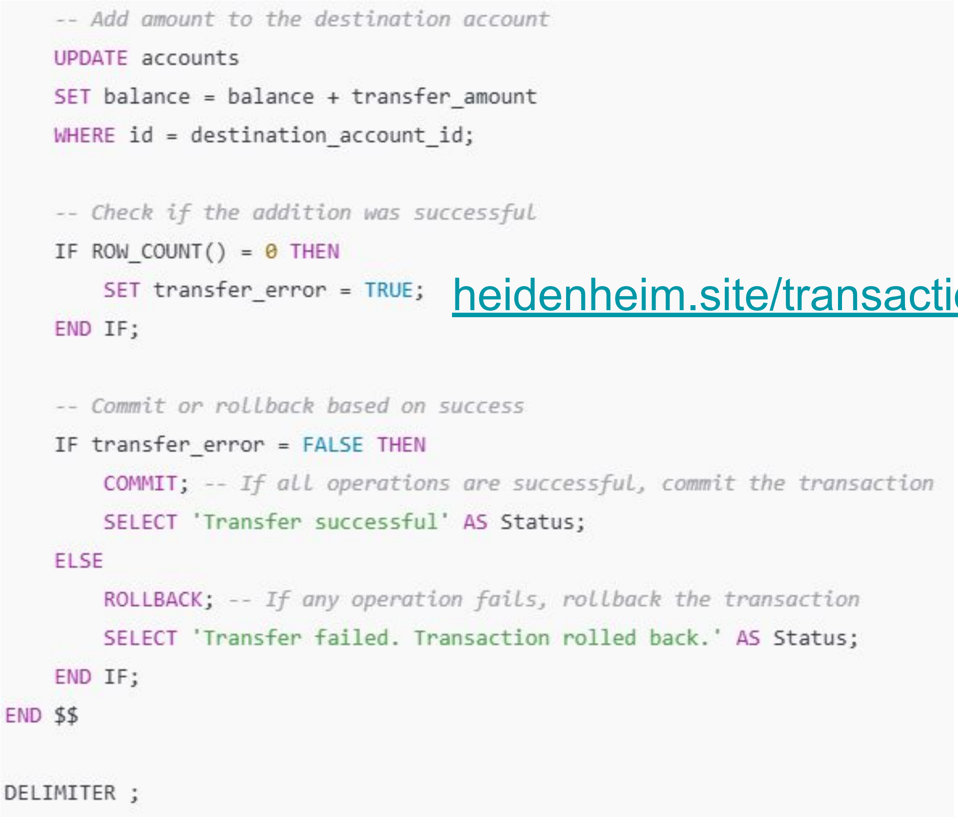
```sql
START TRANSACTION;

-- Deduct amount from Alice
UPDATE accounts
SET balance = balance - 50
WHERE name = 'Alice';

-- Simulate an error
UPDATE accounts
SET balance = 'invalid_value' -- Invalid operation
WHERE name = 'Bob';

-- Rollback the transaction
ROLLBACK;
```

Use ROLLBACK to handle an error scenario.

Stored procedure for transferring funds between two accounts

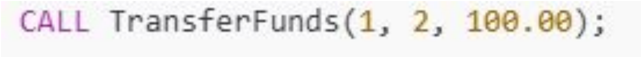


```sql
DELIMITER $$

CREATE PROCEDURE TransferFunds(
    IN source_account_id INT,
    IN destination_account_id INT,
    IN transfer_amount DECIMAL(10, 2)
)
BEGIN
    -- Declare variables for error handling
    DECLARE transfer_error BOOLEAN DEFAULT FALSE;

    -- Start the transaction
    START TRANSACTION;

    -- Deduct amount from the source account
    UPDATE accounts
    SET balance = balance - transfer_amount
    WHERE id = source_account_id;

    -- Check if the deduction was successful
    IF ROW_COUNT() = 0 THEN
        SET transfer_error = TRUE;
    END IF;
```



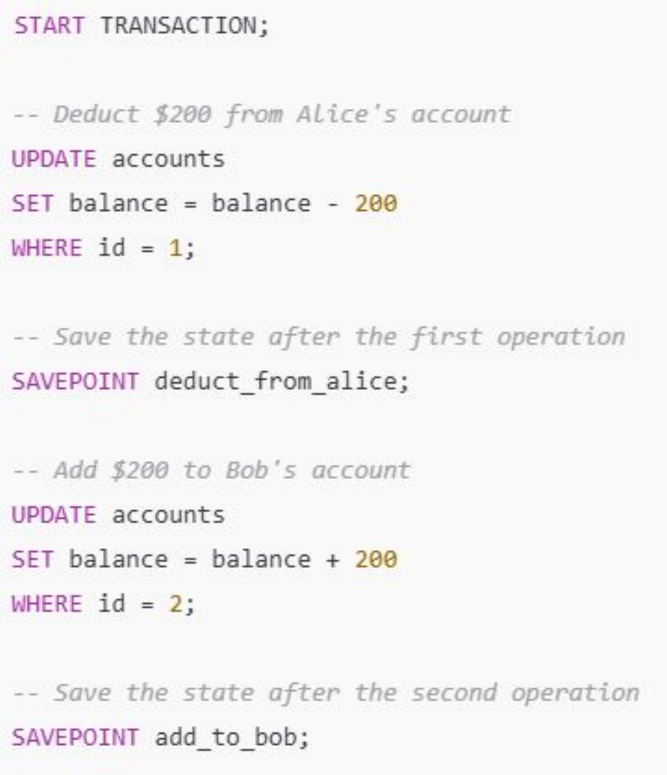
```sql
    -- Add amount to the destination account
    UPDATE accounts
    SET balance = balance + transfer_amount
    WHERE id = destination_account_id;

    -- Check if the addition was successful
    IF ROW_COUNT() = 0 THEN
        SET transfer_error = TRUE;
    END IF;

    -- Commit or rollback based on success
    IF transfer_error = FALSE THEN
        COMMIT; -- If all operations are successful, commit the transaction
        SELECT 'Transfer successful' AS Status;
    ELSE
        ROLLBACK; -- If any operation fails, rollback the transaction
        SELECT 'Transfer failed. Transaction rolled back.' AS Status;
    END IF;
END $$

DELIMITER ;
```

heidenheim.site/transaction.sql

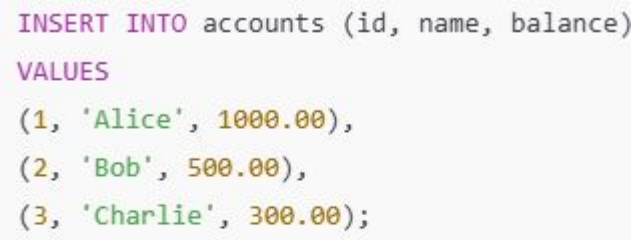


```sql
CALL TransferFunds(1, 2, 100.00);
```

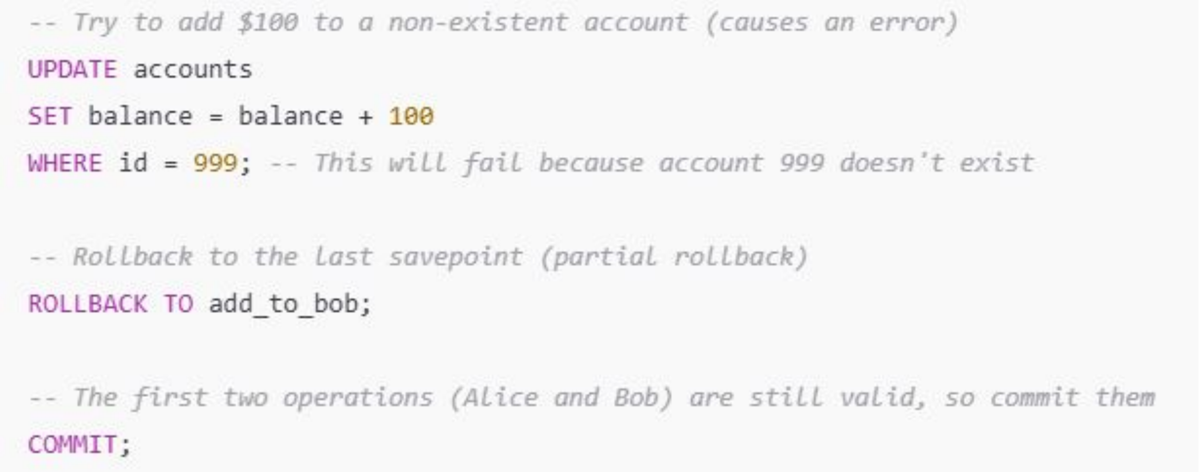
Partial rollback



```sql
START TRANSACTION;

-- Deduct $200 from Alice's account
UPDATE accounts
SET balance = balance - 200
WHERE id = 1;

-- Save the state after the first operation
SAVEPOINT deduct_from_alice;

-- Add $200 to Bob's account
UPDATE accounts
SET balance = balance + 200
WHERE id = 2;

-- Save the state after the second operation
SAVEPOINT add_to_bob;
```

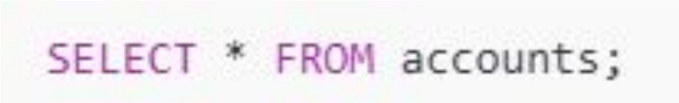


```sql
INSERT INTO accounts (id, name, balance)
VALUES
(1, 'Alice', 1000.00),
(2, 'Bob', 500.00),
(3, 'Charlie', 300.00);
```

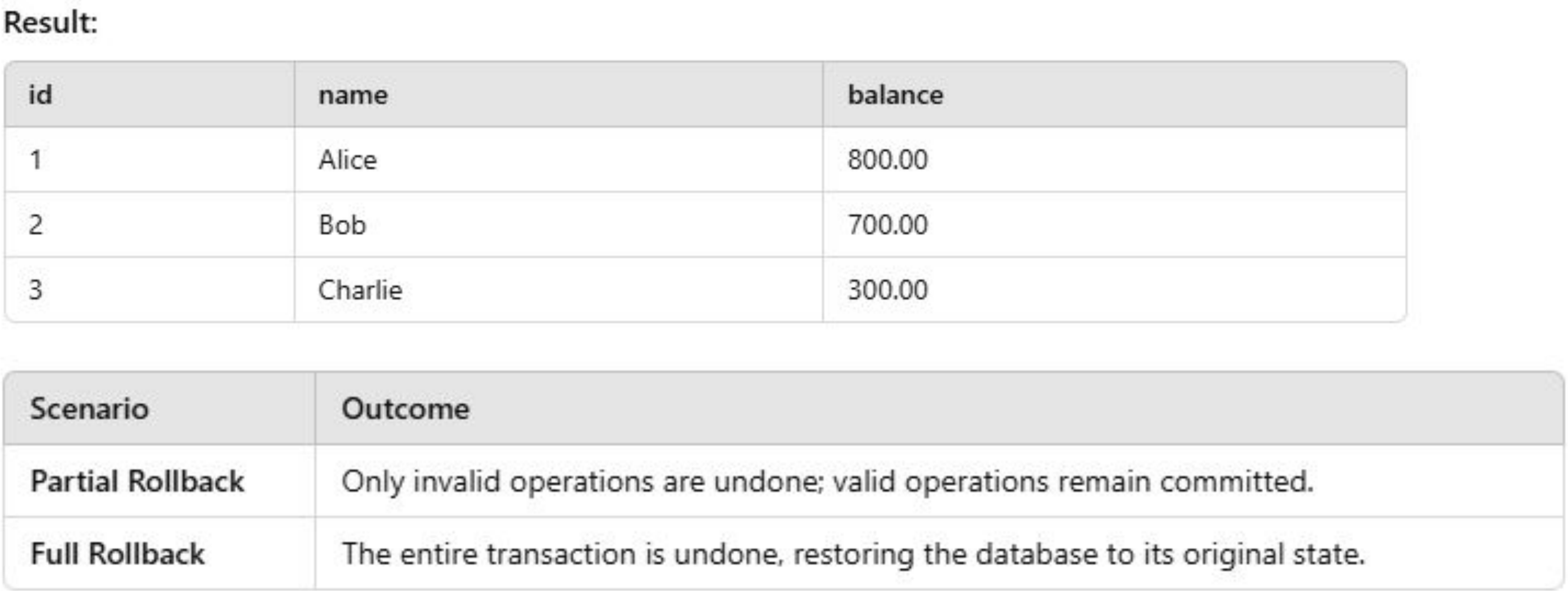


```sql
-- Try to add $100 to a non-existent account (causes an error)
UPDATE accounts
SET balance = balance + 100
WHERE id = 999; -- This will fail because account 999 doesn't exist

-- Rollback to the last savepoint (partial rollback)
ROLLBACK TO add_to_bob;

-- The first two operations (Alice and Bob) are still valid, so commit them
COMMIT;
```

Output: After the Transaction



```sql
SELECT * FROM accounts;
```



Result:

id	name	balance
1	Alice	800.00
2	Bob	700.00
3	Charlie	300.00

Scenario	Outcome
Partial Rollback	Only invalid operations are undone; valid operations remain committed.
Full Rollback	The entire transaction is undone, restoring the database to its original state.

Pros and Cons of Transactions

Pros of Transactions

Data Consistency:

Ensures the database remains in a valid state by following ACID properties.

Example: Prevents half-completed operations like deducting money from one account without adding it to another.

Error Handling:

Allows rolling back changes when errors occur, ensuring no invalid or partial data is saved.

Atomicity:

Transactions are "all or nothing" — either all operations succeed, or none are applied.

Concurrency Control:

Ensures data integrity when multiple users or processes access the database simultaneously.

Ease of Recovery:

Transactions ensure the database can be restored to a consistent state in case of failure (e.g., crashes).

Controlled Workflow:

Complex operations (e.g., financial transfers, batch updates) can be managed step by step, with checkpoints (`SAVEPOINT`).

Cons of Transactions

Performance Overhead:

Managing transactions adds processing overhead, especially for high-volume operations.

Locking Issues:

Concurrent transactions can cause locking conflicts, reducing database performance in multi-user environments.

Complexity:

Writing transaction-based logic requires additional effort and careful planning to handle edge cases.

Resource Usage:

Open transactions consume memory and other resources, which can lead to bottlenecks if not managed properly.

Deadlocks:

Improper transaction design may cause circular waits, where two or more transactions are stuck waiting for each other to release resources.

Requires Careful Management:

Forgetting to commit or rollback can leave transactions open, leading to unexpected behavior or resource exhaustion.

SURVEY

This survey is designed to test your understanding of MySQL concepts and features, including subqueries, joins, stored procedures, normalization, relationships, and ACID properties. It's a fun and interactive way to assess your knowledge and explore your expertise in relational database management systems.

Please note:

> **This survey does not save your answers or track your results.** It is purely for educational and self-assessment purposes.
>
> **Feel free to try it as many times as you like** to enhance your learning experience.
>
> After completing the survey, you'll see instant feedback, with correct answers highlighted for your review.
>
> Remember, the goal is to challenge yourself and improve your understanding, so relax and enjoy the process!

https://heidenheim.site/survey.html