Chapter 14

How to use transactions and locking

Objectives

Applied

- Given a set of SQL statements to be combined into a transaction, write a script that begins, commits, and rolls back the transaction.

Knowledge

- Describe the use of transactions.
- Describe the use of save points.
- Describe the way locking helps prevent concurrency problems.
- Describe the way the transaction isolation level affects concurrency problems and performance.
- Describe a deadlock.
- Describe three techniques that can reduce deadlocks.

A stored procedure with a transaction

```
DELIMITER //

CREATE PROCEDURE test()
BEGIN
  DECLARE sql_error TINYINT DEFAULT FALSE;

  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    SET sql_error = TRUE;

  START TRANSACTION;

  INSERT INTO invoices
  VALUES (115, 34, 'ZXA-080', '2015-01-18',
          14092.59, 0, 0, 3, '2015-04-18', NULL);

  INSERT INTO invoice_line_items
  VALUES (115, 1, 160, 4447.23, 'HW upgrade');
```

A stored procedure with a transaction (continued)

```
    INSERT INTO invoice_line_items
    VALUES (115, 2, 167, 9645.36, 'OS upgrade');

    IF sql_error = FALSE THEN
      COMMIT;
      SELECT 'The transaction was committed.';
    ELSE
      ROLLBACK;
      SELECT 'The transaction was rolled back.';
    END IF;
  END//
```

When to use transactions

- When you code two or more INSERT, UPDATE, or DELETE statements that affect related data.
- When you move rows from one table to another table by using INSERT and DELETE statements.
- Whenever the failure of an INSERT, UPDATE, or DELETE statement would violate data integrity.

A script that uses save points

```
USE ap;

START TRANSACTION;

SAVEPOINT before_invoice;

INSERT INTO invoices
VALUES (115, 34, 'ZXA-080', '2015-01-18',
        14092.59, 0, 0, 3, '2015-04-18', NULL);

SAVEPOINT before_line_item1;

INSERT INTO invoice_line_items
VALUES (115, 1, 160, 4447.23, 'HW upgrade');

SAVEPOINT before_line_item2;
```

A script that uses save points (continued)

```
INSERT INTO invoice_line_items
VALUES (115, 2, 167, 9645.36,'OS upgrade');

ROLLBACK TO SAVEPOINT before_line_item2;

ROLLBACK TO SAVEPOINT before_line_item1;

ROLLBACK TO SAVEPOINT before_invoice;

COMMIT;
```

Two transactions that retrieve and then modify the data in the same row

Transaction A

```
START TRANSACTION;

UPDATE invoices SET credit_total = credit_total + 100
WHERE invoice_id = 6;

-- the SELECT statement in Transaction B
--     won't show the updated data
-- the UPDATE statement in Transaction B
--     will wait for transaction A to finish

COMMIT;

-- the SELECT statement in Transaction B
--     will display the updated data
-- the UPDATE statement in Transaction B
       will execute immediately
```

Transaction B

```
START TRANSACTION;

SELECT invoice_id, credit_total
FROM invoices WHERE invoice_id = 6;

UPDATE invoices SET credit_total = credit_total + 200
WHERE invoice_id = 6;

COMMIT;
```

How to test these transactions

- Open a separate connection for each transaction.
- Execute one statement at a time, alternating between the two transactions.

The four types of concurrency problems that locking can prevent

- Lost updates
- Dirty reads
- Nonrepeatable reads
- Phantom reads

The concurrency problems prevented by each transaction isolation level

Isolation level	Problems prevented
READ UNCOMMITTED	None
READ COMMITTED	Dirty reads
REPEATABLE READ	Dirty reads, lost updates, nonrepeatable reads
SERIALIZABLE	All

The syntax of the SET TRANSACTION ISOLATION LEVEL statement

```
SET {GLOBAL|SESSION} TRANSACTION ISOLATION LEVEL
    {READ UNCOMMITTED|READ COMMITTED|
     REPEATABLE READ|SERIALIZABLE}
```

Set the level to SERIALIZABLE for the next transaction

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

Set the level to READ UNCOMMITTED for the current session

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

Set the level to READ COMMITTED for all sessions

```
SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED
```

UPDATE statements that illustrate deadlocking

Transaction A

```
START TRANSACTION;
UPDATE savings SET balance = balance - transfer_amount;
UPDATE checking SET balance = balance + transfer_amount;
COMMIT;
```

Transaction B (possible deadlock)

```
START TRANSACTION;
UPDATE checking SET balance = balance - transfer_amount;
UPDATE savings SET balance = balance + transfer_amount;
COMMIT;
```

Transaction B (prevents deadlocks)

```
START TRANSACTION;
UPDATE savings SET balance = balance + transfer_amount;
UPDATE checking SET balance = balance - transfer_amount;
COMMIT;
```

How to prevent deadlocks

- Don’t allow transactions to remain open for very long.
- Don’t use a transaction isolation level higher than necessary.
- Make large changes when you can be assured of nearly exclusive access.
- Consider locking when coding your transactions.