Bài thực hành số 4

Phần 1: Mức ưu tiên có sẵn trong cơ sở dữ liệu

Exercise 1: Acquire Locks by Using the Read Committed Isolation Level

1. Open Microsoft SQL Server Management Studio, and connect to an instance of SQL Server 2005.
2. In a new query window, which will be referred to as Connection 1, type and execute the following SQL statements to create the *TestDB* database, the Test schema, and the table that you will use in this exercise:

```
-- Connection 1 - Session ID: <put @@SPID result here>
/* Leave the above line to easily see that this query window
belongs to Connection 1. */
SELECT @@SPID;
GO
CREATE DATABASE TestDB;
GO
USE TestDB;
GO
CREATE SCHEMA Test;
GO
CREATE TABLE Test.TestTable (
     Col1 INT NOT NULL
     ,Col2 INT NOT NULL
);
INSERT Test.TestTable (Col1, Col2) VALUES (1,10);
INSERT Test.TestTable (Col1, Col2) VALUES (2,20);
INSERT Test.TestTable (Col1, Col2) VALUES (3,30);
INSERT Test.TestTable (Col1, Col2) VALUES (4,40);
INSERT Test.TestTable (Col1, Col2) VALUES (5,50);
INSERT Test.TestTable (Col1, Col2) VALUES (6,60);
```

3. Open another query window, which will be referred to as Connection 2, and type and execute the following SQL statement to prepare the connection

```
-- Connection 2 - Session ID: <put @@SPID result here>
/* Leave the above line to easily see that this query window
belongs to Connection 2. */
SELECT @@SPID;
GO
USE TestDB;
```

4. Open a third query window, which will be referred to as Connection 3, and type and execute the following SQL statement to prepare the connection:

```
---Connection 3
/* Leave the above line to easily see that this query window
belongs to Connection 3. */
```

```
USE TestDB;
```

5. In Connection 1, execute the following SQL statements to start a transaction in the read committed transaction isolation level, and read a row from the test table (but do not commit the transaction!).

```
-- Connection 1
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN;
      SELECT * FROM Test.TestTable
      WHERE Col1 = 1;
```

6. To see which locks have been acquired by the transaction in Connection 1, open Connection 3, and execute the following SELECT statement. In the line of code that contains *@@SPID of Connection 1*>, be sure to replace this with the ID value returned by the code executed in step 2 of this exercise.

```
SELECT
      resource_type
      ,request_mode
      ,request_status
FROM sys.dm_tran_locks
WHERE resource_database_id = DB_ID('TestDB')
      AND request_session_id = <@@SPID of Connection 1>
      AND request_mode IN ('S', 'X')
      AND resource_type <> 'DATABASE';
```

Why doesn't Connection 1 have a shared lock on the row that it read using the SELECT statement?

7. In Connection 1, execute the following SQL statement to end the started transaction:

```
---Connection 1
COMMIT TRAN;
```

8. In Connection 2, execute the following SQL statements to start a transaction, and acquire an exclusive lock on one row in the test table.

```
-- Connection 2
BEGIN TRAN;
      UPDATE Test.TestTable SET Col2 = Col2 + 1
      WHERE Col1 = 1;
```

9. In Connection 1, execute the following transaction to try to read the row that has been updated (but not committed) by Connection 2. After you execute the code in this step, move on to the next step, as this connection will now be blocked.

```
-- Connection 1
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN;
```

```
      SELECT * FROM Test.TestTable
      WHERE Col1 = 1;
-- This SELECT statement will be blocked!
```

10. To see which locks have been acquired by the transaction in Connection 1, open Connection 3, and execute the following SELECT statement. In the line of code that contains *@@SPID of Connection 1*, be sure to replace this with the ID value returned by the code executed in step 2 of this exercise.

```
SELECT
      resource_type
      ,request_mode
      ,request_status
FROM sys.dm_tran_locks
WHERE resource_database_id = DB_ID('TestDB')
      AND request_session_id = <@@SPID of Connection 1>
      AND request_mode IN ('S', 'X')
      AND resource_type <> 'DATABASE';
```

Here you can see that Connection 1 tries to acquire a shared lock on the row.

11. In Connection 2, execute the following SQL statements to end the transaction started earlier.

```
--Connection 2
COMMIT TRAN;
```

12. Now, first have a look in Connection 1 and note that the SELECT statement has been completed. Switch to Connection 3, and execute its SELECT statement again to see which locks are now acquired by the transaction in Connection 1. In the line of code that contains *@@SPID of Connection 1*>, be sure to replace this with the ID value returned by the code executed in step 2 of this exercise.

```
SELECT
      resource_type
      ,request_mode
      ,request_status
FROM sys.dm_tran_locks
WHERE resource_database_id = DB_ID('TestDB')
      AND request_session_id = <@@SPID of Connection 1>
      AND request_mode IN ('S', 'X')
      AND resource_type <> 'DATABASE';
```

You should now see that no locks are acquired by Connection 1. This is because, after acquiring the lock on the row, Connection 1 released the lock.

13. Close the three query windows for Connections 1, 2, and 3. Open a new query window, and execute the following SQL statement to clean up after this exercise:

```
USE master;
DROP DATABASE TestDB;
```

Exercise 2: Acquire Locks by Using the Read Committed Snapshot Isolation Level

In this exercise, you execute the same type of transactions as in the previous exercise, but use the read committed snapshot transaction isolation level.

1. Open SQL Server Management Studio, and connect to an instance of SQL Server 2008.
2. In a new query window, which will be referred to as Connection 1, type and execute the following SQL statements to create the *TestDB* database, the Test schema, and the table that will be used in this exercise:

```
-- Connection 1
/*  Leave the above line to easily see that this query window
belongs to Connection 1. */
CREATE DATABASE TestDB;
GO
ALTER DATABASE TestDB SET READ_COMMITTED_SNAPSHOT ON;
GO
USE TestDB;
GO
CREATE SCHEMA Test;
GO
CREATE TABLE Test.TestTable (
     Col1 INT NOT NULL
     ,Col2 INT NOT NULL
);
INSERT Test.TestTable (Col1, Col2) VALUES (1,10);
INSERT Test.TestTable (Col1, Col2) VALUES (2,20);
INSERT Test.TestTable (Col1, Col2) VALUES (3,30);
INSERT Test.TestTable (Col1, Col2) VALUES (4,40);
INSERT Test.TestTable (Col1, Col2) VALUES (5,50);
INSERT Test.TestTable (Col1, Col2) VALUES (6,60);
```

3. Open another query window, which will be referred to as Connection 2, and type and execute the following SQL statement to prepare the connection:

```
--Connection 2
>/* Leave the above line to easily see that this query window
belongs to Connection 2. */
USE TestDB;
```

4. Open a third query window, which will be referred to as Connection 3, and type and execute the following SQL statement to prepare the connection:

```
--Connection 3
/* Leave the above line to easily see that this query window
belongs to Connection 3. */
USE TestDB;
```

5. In Connection 2, execute the following SQL statements to start a transaction, and acquire an exclusive lock on one row in the test table.

```
-- Connection 2
BEGIN TRAN;
     UPDATE Test.TestTable SET Col2 = Col2 + 1
     WHERE Col1 = 1;
```

6. In Connection 1, execute the following transaction to try to read the row that has been updated (but not committed) by Connection 2.

```
-- Connection 1
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN;
     SELECT * FROM Test.TestTable
     WHERE Col1 = 1;
```

   Why wasn't the SELECT statement blocked by Connection 2? Which values were returned by the query, the values that existed before or after the update?

7. To see which locks have been acquired by the transaction in Connections 1 and 2, open Connection 3, and execute the following SELECT statement:

```
SELECT
     resource_type
     ,request_mode
     ,request_status
FROM sys.dm_tran_locks
WHERE resource_database_id = DB_ID('TestDB')
     AND request_mode IN ('S', 'X')
     AND resource_type <> 'DATABASE';
```

8. To see if any row versions are available for the *TestDB* database, execute the following query in Connection 3:

```
SELECT * FROM sys.dm_tran_version_store
WHERE database_id = DB_ID('TestDB');
```

9. In Connection 2, execute the following SQL statements to end the transaction started earlier.

```
--Connection 2
COMMIT TRAN;
```

10. In the open transaction in Connection 1, execute the SELECT statement again.

```
SELECT * FROM Test.TestTable
WHERE Col1 = 1;
```

   Which values are now returned, the values that existed before or after the update? Did this SELECT statement return dirty reads? Did the first SELECT statement in Connection 1 return dirty reads?

11. Close the three query windows for Connection 1, 2, and 3. Open a new query window, and execute the following SQL statement to clean up after this exercise:

```
USE master;
DROP DATABASE TestDB;
```

Phần 2: Thiết kế giao dịch và quản lý các khóa trong giao dịch

Exercise 1: Use the Default Isolation Level

In this exercise, you create the draft for the stored procedure and use the read committed transaction isolation level.

1. Open SQL Server Management Studio, and connect to an instance of SQL Server 2005.
2. Open a new query window, and type and execute the following SQL statements. This will create the *TestDB* database, the Test schema, and the tables that are used in this exercise: you will also create the Test.spAccountReset stored procedure. You can execute this procedure to reset the data in the tables if you need to restart the exercise.

```
CREATE DATABASE TestDB;
GO
USE TestDB;
GO
CREATE SCHEMA Test;
GO
CREATE TABLE Test.Accounts (
      AccountNumber INT PRIMARY KEY
);
CREATE TABLE Test.AccountTransactions (
      TransactionID INT IDENTITY PRIMARY KEY
      ,AccountNumber INT NOT NULL REFERENCES Test.Accounts
      ,CreatedDateTime DATETIME NOT NULL DEFAULT
CURRENT_TIMESTAMP
      ,Amount DECIMAL(19, 5) NOT NULL
);
GO
CREATE PROC Test.spAccountReset
AS
BEGIN
      SET NOCOUNT ON;
      DELETE Test.AccountTransactions;
      DELETE Test.Accounts;
      INSERT Test.Accounts (AccountNumber) VALUES (1001);
      INSERT Test.AccountTransactions (AccountNumber, Amount)
      VALUES (1001, 100);
      INSERT Test.AccountTransactions (AccountNumber, Amount)
      VALUES (1001, 500);
      INSERT Test.AccountTransactions (AccountNumber, Amount)
```

```
        VALUES (1001, 1400);
        SELECT AccountNumber, SUM(Amount) AS Balance
        FROM Test.AccountTransactions
        GROUP BY AccountNumber;
END
```

3. Open another query window, and type and execute the following SQL statements to create the Test.spAccountWithdraw stored procedure:

```
USE TestDB;
GO
CREATE PROC Test.spAccountWithdraw
@AccountNumber INT
,@AmountToWithdraw DECIMAL(19, 5)
AS
BEGIN
        SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
        BEGIN TRY
                IF(@AmountToWithdraw <= 0)
                        RAISERROR('@AmountToWithdraw must be > 0.', 16,
1);
                BEGIN TRAN;
                -- Verify that the account exists...
                        IF NOT EXISTS(
                                        SELECT *
                                        FROM Test.Accounts
                                        WHERE AccountNumber = @AccountNumber
                                )
                                RAISERROR('Account not found.', 16, 1);
                -- Verify that the account will not be overdrawn...
                        IF (@AmountToWithdraw > (
                                        SELECT SUM(Amount)
                                        FROM Test.AccountTransactions
                                        WHERE AccountNumber = @AccountNumber)
                                )
                                RAISERROR('Not enough funds in account.',
16, 1);
                -- ** USED TO TEST CONCURRENCY PROBLEMS **
                        RAISERROR('Pausing procedure for 10 seconds...',
10, 1)
                                WITH NOWAIT;
                        WAITFOR DELAY '00:00:30';
                        RAISERROR('Procedure continues...', 10, 1) WITH
NOWAIT;
                        -- Make the withdrawal...
                        INSERT Test.AccountTransactions (AccountNumber,
Amount)
                                VALUES (@AccountNumber, -
@AmountToWithdraw);
                        -- Return the new balance of the account:
                        SELECT SUM(Amount) AS BalanceAfterWithdrawal
```

```
                    FROM Test.AccountTransactions
                    WHERE AccountNumber = @AccountNumber;
              COMMIT TRAN;
        END TRY
        BEGIN CATCH
              DECLARE @ErrorMessage NVARCHAR(2047);
              SET @ErrorMessage = ERROR_MESSAGE();
              RAISERROR(@ErrorMessage, 16, 1);
              -- Should also use ERROR_SEVERITY() and
ERROR_STATE()...
              IF(XACT_STATE() <> 0)
                    ROLLBACK TRAN;
        END CATCH
END
```

4. Open another query window, which will be referred to as Connection 1, and type and execute the following SQL statement to prepare the connection:

```
Connection 1
/* Leave the above line to easily see that this query window
belongs to Connection 1. */
USE TestDB
GO
Reset/generate the account data
EXEC Test.spAccountReset;
```

5. Open another query window, which will be referred to as Connection 2, and type and execute the following SQL statement to prepare the connection:

```
--Connection 2
/* Leave the above line to easily see that this query window
belongs to Connection 2. */
USE TestDB
GO
```

6. In this step, you will execute two batches at the same time to try to test for concurrency problems. In both the Connection 1 and Connection 2 query windows, type the following SQL statements without executing them yet. The statements will first retrieve the current account balance and then attempt to empty the account.

```
SELECT SUM(Amount) AS BalanceBeforeWithdrawal
FROM Test.AccountTransactions
WHERE AccountNumber = 1001;
GO
EXEC Test.spAccountWithdraw @AccountNumber = 1001,
                    @AmountToWithdraw = 2000;
```

To get a better view of what will happen, press Ctrl+T in SQL Server Management Studio to set results to be returned as text instead of grids. Do this for both query windows.

Now, start the execution in both query windows simultaneously and wait for both batches to finish execution. (This should take approximately 30 seconds because of the WAITFOR DELAY statement in the Test.spAccountWithdraw stored procedure.) Both connections' batches should return two result sets; the first result set will contain the current account balance (which should be 2,000 for both batches), and the second result set will contain the account balance after the withdrawal. What was the result of the two withdrawals? Was the account overdrawn? What kind of concurrency problem occurred (if any)?

7. Close all open query windows except one, and in that query window, type and execute the following SQL statements to clean up after this exercise:

```
>USE master;
GO
DROP DATABASE TestDB;
```

Exercise 2: Use a Locking Hint

In the previous exercise, you encountered the "phantom reads" concurrency problem. In this exercise, you re-create the stored procedure, but this time, you will use the serializable locking hint to protect against phantom reads.

1. Open SQL Server Management Studio, and connect to an instance of SQL Server 2005.

2. Open a new query window, and type and execute the following SQL statements. This will create the *TestDB* database, the Test schema, and the tables that you will use in this exercise. You will also create the Test.spAccountReset stored procedure. You can execute this procedure to reset the data in the tables if you need to restart the exercise.

```
CREATE DATABASE TestDB;
GO
USE TestDB;
GO
CREATE SCHEMA Test;
GO
CREATE TABLE Test.Accounts (
      AccountNumber INT PRIMARY KEY);
CREATE TABLE Test.AccountTransactions (
      TransactionID INT IDENTITY PRIMARY KEY
      ,AccountNumber INT NOT NULL REFERENCES Test.Accounts
      ,CreatedDateTime DATETIME NOT NULL DEFAULT
CURRENT_TIMESTAMP
      ,Amount DECIMAL(19, 5) NOT NULL
);
GO
CREATE PROC Test.spAccountReset
AS
BEGIN
```

```
	SET NOCOUNT ON;
	DELETE Test.AccountTransactions;
	DELETE Test.Accounts;
	INSERT Test.Accounts (AccountNumber) VALUES (1001);
	INSERT Test.AccountTransactions (AccountNumber, Amount)
	VALUES (1001, 100);
	INSERT Test.AccountTransactions (AccountNumber, Amount)
	VALUES (1001, 500);
	INSERT Test.AccountTransactions (AccountNumber, Amount)
	VALUES (1001, 1400);
	SELECT AccountNumber, SUM(Amount) AS Balance
	FROM Test.AccountTransactions
	GROUP BY AccountNumber;
END
```

3. Open another query window, and type and execute the following SQL statements to create the Test.spAccountWithdraw stored procedure:

```
USE TestDB;
GO
CREATE PROC Test.spAccountWithdraw
@AccountNumber INT
,@AmountToWithdraw DECIMAL(19, 5)
AS
BEGIN
	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
	BEGIN TRY
		IF(@AmountToWithdraw <= 0)
		RAISERROR('@AmountToWithdraw must be > 0.', 16, 1);
		BEGIN TRAN;
		-- Verify that the account exists...
			IF NOT EXISTS(
					SELECT *
					FROM Test.Accounts
					WHERE AccountNumber = @AccountNumber
				)
				RAISERROR('Account not found.', 16, 1);
		-- Verify that the account will not be overdrawn...
			IF (@AmountToWithdraw > (
					SELECT SUM(Amount)
					FROM Test.AccountTransactions
WITH(SERIALIZABLE)
					WHERE AccountNumber = @AccountNumber)
				)
				RAISERROR('Not enough funds in account.',
16, 1);
			-- ** USED TO TEST CONCURRENCY PROBLEMS **
			RAISERROR('Pausing procedure for 10 seconds...',
10, 1)
				WITH NOWAIT;
			WAITFOR DELAY '00:00:30';
```

```
                RAISERROR('Procedure continues...', 10, 1) WITH
NOWAIT;
                -- Make the withdrawal...
                INSERT Test.AccountTransactions (AccountNumber,
Amount)
                     VALUES (@AccountNumber, -
@AmountToWithdraw);

                -- Return the new balance of the account:
                SELECT SUM(Amount) AS BalanceAfterWithdrawal
                FROM Test.AccountTransactions
                WHERE AccountNumber = @AccountNumber;
           COMMIT TRAN;
     END TRY
     BEGIN CATCH
           DECLARE @ErrorMessage NVARCHAR(2047);
           SET @ErrorMessage = ERROR_MESSAGE();
           RAISERROR(@ErrorMessage, 16, 1);
           -- Should also use ERROR_SEVERITY() and
ERROR_STATE()...
           IF(XACT_STATE() <> 0)
                ROLLBACK TRAN;
     END CATCH
END
```

4. Open another query window, which will be referred to as Connection 1, and type and execute the following SQL statement to prepare the connection:

```
--Connection 1
/* Leave the above line to easily see that this query window
belongs to Connection 1. */
USE TestDB;
GO
--Reset/generate the account data
EXEC Test.spAccountReset;
```

5. Open another query window, which will be referred to as Connection 2, and type and execute the following SQL statement to prepare the connection:

```
--Connection 2
/* Leave the above line to easily see that this query window
belongs to Connection 2. */
USE TestDB;
GO
```

6. In this step, you will execute two batches at the same time to try to test for concurrency problems. In both the Connection 1 and Connection 2 query windows, type the following SQL statements without executing them yet. The statements will first retrieve the current account balance and then attempt to empty the account.

```
SELECT SUM(Amount) AS BalanceBeforeWithdrawal
FROM Test.AccountTransactions
WHERE AccountNumber = 1001;
GO
EXEC Test.spAccountWithdraw @AccountNumber = 1001,
                            @AmountToWithdraw = 2000;
```

To get a better view of what will happen, press Ctrl+T in SQL Server Management Studio to set results to be returned as text instead of grids. Do this for both query windows. Now, start the execution in both query windows simultaneously and wait for both batches to finish execution. (This should take approximately 30 seconds because of the WAITFOR DELAY statement in the Test.spAccountWithdraw stored procedure.) Both connections' batches should return two result sets; the first result set will contain the current account balance (which should be 2,000 for both batches), and the second result set will contain the account balance after the withdrawal. What was the result of the two withdrawals? Was the account overdrawn? What kind of concurrency problem occurred (if any)? Was there any other problem with this implementation?

7. Close all open query windows except one, and in that query window, type and execute the following SQL statements to clean up after this exercise:

```
USE master;
GO
DROP DATABASE TestDB;
```

Exercise 3: Use an Alternative Solution

In Exercise 2, the account was not overdrawn, and you didn't experience any concurrency problems. The connections were instead deadlocked. In this exercise, you re-create the stored procedure to protect against both phantom reads and deadlocks by changing the implementation slightly.

1. Open SQL Server Management Studio, and connect to an instance of SQL Server 2005.

2. Open a new query window, and type and execute the following SQL statements. This will create the *TestDB* database, the Test schema, and the tables that will be used in this exercise: you will also create the Test.spAccountReset stored procedure. You can execute this procedure to reset the data in the tables if you need to restart the exercise.

```
CREATE DATABASE TestDB;
GO
USE TestDB;
GO
CREATE SCHEMA Test;
GO
CREATE TABLE Test.Accounts (
      AccountNumber INT PRIMARY KEY
);
CREATE TABLE Test.AccountTransactions (
      TransactionID INT IDENTITY PRIMARY KEY
      ,AccountNumber INT NOT NULL REFERENCES Test.Accounts
      ,CreatedDateTime DATETIME NOT NULL DEFAULT
CURRENT_TIMESTAMP
      ,Amount DECIMAL(19, 5) NOT NULL
);
GO
CREATE PROC Test.spAccountReset
AS
BEGIN
      SET NOCOUNT ON;
      DELETE Test.AccountTransactions;
      DELETE Test.Accounts;
      INSERT Test.Accounts (AccountNumber) VALUES (1001);
      INSERT Test.AccountTransactions (AccountNumber, Amount)
      VALUES (1001, 100);
      INSERT Test.AccountTransactions (AccountNumber, Amount)
      VALUES (1001, 500);
      INSERT Test.AccountTransactions (AccountNumber, Amount)
      VALUES (1001, 1400);
      SELECT AccountNumber, SUM(Amount) AS Balance
      FROM Test.AccountTransactions
      GROUP BY AccountNumber;
END
```

3. Open another query window, and type and execute the following SQL statements to create the Test.spAccountWithdraw stored procedure:

```
USE TestDB;
GO
CREATE PROC Test.spAccountWithdraw
@AccountNumber INT
,@AmountToWithdraw DECIMAL(19, 5)
AS
BEGIN
      SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
      BEGIN TRY
            IF(@AmountToWithdraw <= 0)
                  RAISERROR('@AmountToWithdraw must be > 0.', 16,
1);
            BEGIN TRAN;
                  -- Verify that the account exists
                  -- and LOCK the account from access by other
queries
      -- that will write to the account or its transactions.
      -- Note that SELECT statements against the account
                  -- will still be allowed.
                  IF NOT EXISTS(
                                    SELECT *
                                    FROM Test.Accounts WITH (UPDLOCK)
                                    WHERE AccountNumber = @AccountNumber
                               )
                               RAISERROR('Account not found.', 16, 1);
       -- Verify that the account will not be overdrawn...
                        IF (@AmountToWithdraw > (
                                    SELECT SUM(Amount)
                                    FROM Test.AccountTransactions /* NO
LOCKING HINT */
                                    WHERE AccountNumber = @AccountNumber)
                               )
                               RAISERROR('Not enough funds in account.',
16, 1);
                        -- ** USED TO TEST CONCURRENCY PROBLEMS **
                        RAISERROR('Pausing procedure for 10 seconds...',
10, 1)
                               WITH NOWAIT;
                        WAITFOR DELAY '00:00:30';
                        RAISERROR('Procedure continues...', 10, 1) WITH
NOWAIT;
                        -- Make the withdrawal...
                        INSERT Test.AccountTransactions (AccountNumber,
Amount)
                               VALUES (@AccountNumber, -
@AmountToWithdraw);
                        -- Return the new balance of the account:
                        SELECT SUM(Amount) AS BalanceAfterWithdrawal
                        FROM Test.AccountTransactions
                        WHERE AccountNumber = @AccountNumber;
              COMMIT TRAN;
       END TRY
       BEGIN CATCH
              DECLARE @ErrorMessage NVARCHAR(2047);
              SET @ErrorMessage = ERROR_MESSAGE();
              RAISERROR(@ErrorMessage, 16, 1);
       -- Should also use ERROR_SEVERITY() and ERROR_STATE()...
              IF(XACT_STATE() <> 0)
                        ROLLBACK TRAN;
       END CATCH
END
```

4. Open another query window, which will be referred to as Connection 1, and type and execute the following SQL statement to prepare the connection: