Chapter 16

How to manage transactions and locking

Applied objective

- Given a set of statements to be combined into a transaction, insert the Transact-SQL statements to explicitly begin, commit, and roll back the transaction.

Knowledge objectives

- Describe the use of implicit transactions.
- Describe the use of explicit transactions.
- Describe the use of the COMMIT TRAN statement and the @@TRANCOUNT function within nested transactions.
- Describe the use of save points.
- Define these types of concurrency problems: lost updates, dirty reads, nonrepeatable reads, and phantom reads.
- Describe the way locking and the transaction isolation level help to prevent concurrency problems.
- Describe the way SQL Server manages locking in terms of granularity, lock escalation, shared locks, exclusive locks, and lock promotion.
- Describe deadlocks and the way SQL Server handles them.
- Describe four coding techniques that can reduce deadlocks.

INSERT statements that work with related data

```
DECLARE @InvoiceID int;
INSERT Invoices
    VALUES (34,'ZXA-080','2020-03-01',14092.59,
            0,0,3,'2020-03-31',NULL);
SET @InvoiceID = @@IDENTITY;
INSERT InvoiceLineItems
    VALUES (@InvoiceID,1,160,4447.23,'HW upgrade');
INSERT InvoiceLineItems
    VALUES (@InvoiceID,2,167,9645.36,'OS upgrade');
```

The same statements coded as a transaction

```
DECLARE @InvoiceID int;
BEGIN TRY
    BEGIN TRAN;
    INSERT Invoices
      VALUES (34,'ZXA-080','2020-03-01',14092.59,
              0,0,3,'2020-03-31',NULL);
    SET @InvoiceID = @@IDENTITY;
    INSERT InvoiceLineItems
        VALUES (@InvoiceID,1,160,4447.23,'HW upgrade');
    INSERT InvoiceLineItems
        VALUES (@InvoiceID,2,167,9645.36,'OS upgrade');
    COMMIT TRAN;
END TRY
BEGIN CATCH
    ROLLBACK TRAN;
END CATCH;
```

When to use explicit transactions

- When you code two or more action queries that affect related data
- When you update foreign key references
- When you move rows from one table to another table
- When you code a SELECT query followed by an action query and the values inserted in the action query are based on the results of the SELECT query
- When a failure of any set of SQL statements would violate data integrity

The SQL statements for processing transactions

```
BEGIN {TRAN|TRANSACTION}
SAVE {TRAN|TRANSACTION} save_point
COMMIT [TRAN|TRANSACTION]
ROLLBACK [[TRAN|TRANSACTION] [save_point]]
```

A script that performs a test before committing the transaction

```
BEGIN TRAN;

DELETE Invoices
WHERE VendorID = 34;
IF @@ROWCOUNT > 1
    BEGIN
        ROLLBACK TRAN;
        PRINT 'More invoices than expected. ' +
            'Deletions rolled back.';
    END;
ELSE
    BEGIN
        COMMIT TRAN;
        PRINT 'Deletions committed to the database.';
    END;
```

The response from the system

```
(3 rows affected)
More invoices than expected. Deletions rolled back.
```

How nested transactions work

- If you commit a transaction when @@TRANCOUNT is equal to 1, all of the changes made to the database during the transaction are committed and @@TRANCOUNT is set to zero.
- If you commit a transaction when @@TRANCOUNT is greater than 1, @@TRANCOUNT is simply decremented by 1.
- The ROLLBACK TRAN statement rolls back all active transactions regardless of the nesting level where it's coded. It also sets the value of @@TRANCOUNT back to 0.

A script with nested transactions (part 1)

```
BEGIN TRAN;
PRINT 'First Tran  @@TRANCOUNT: ' +
    CONVERT(varchar,@@TRANCOUNT);
DELETE Invoices;
  BEGIN TRAN;
    PRINT 'Second Tran @@TRANCOUNT: ' +
        CONVERT(varchar,@@TRANCOUNT);
    DELETE Vendors;
  COMMIT TRAN;       -- This COMMIT decrements @@TRANCOUNT.
                     -- It doesn't commit 'DELETE Vendors'.
  PRINT 'COMMIT      @@TRANCOUNT: ' +
      CONVERT(varchar,@@TRANCOUNT);
ROLLBACK TRAN;
PRINT 'ROLLBACK   @@TRANCOUNT: ' +
    CONVERT(varchar,@@TRANCOUNT);
```

A script with nested transactions (part 2)

```
PRINT ' ';
DECLARE @VendorsCount int, @InvoicesCount int;
SELECT @VendorsCount = COUNT (*) FROM Vendors;
SELECT @InvoicesCount = COUNT (*) FROM Invoices;
PRINT 'Vendors Count:  ' +
    CONVERT (varchar , @VendorsCount);
PRINT 'Invoices Count: ' +
    CONVERT (varchar , @InvoicesCount);
```

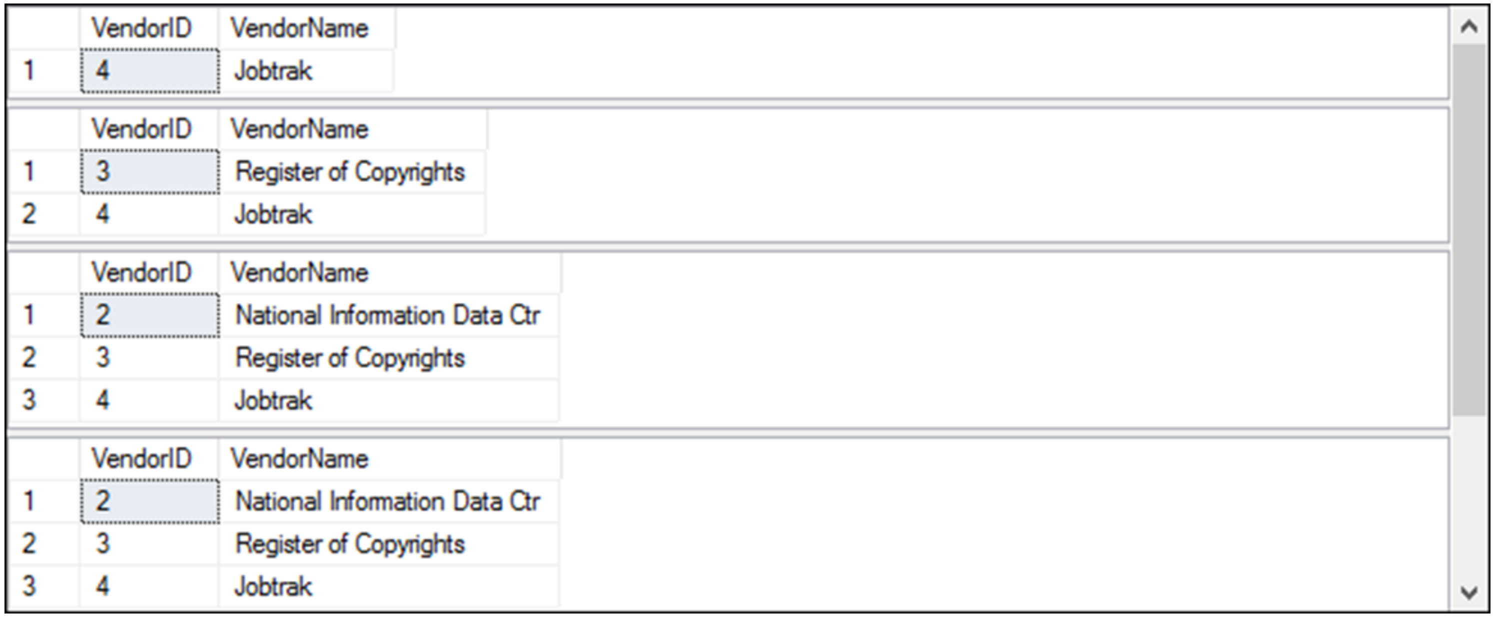
The response from the system

```
First Tran  @@TRANCOUNT: 1

(114 rows affected)
Second Tran @@TRANCOUNT: 2

(122 rows affected)
COMMIT     @@TRANCOUNT: 1
ROLLBACK   @@TRANCOUNT: 0

Vendors count:  122
Invoices count: 114
```

A transaction with two save points

```
IF OBJECT_ID('tempdb..#VendorCopy') IS NOT NULL
    DROP TABLE tempdb.. #VendorCopy;
SELECT VendorID, VendorName
INTO #VendorCopy
FROM Vendors
WHERE VendorID < 5;
BEGIN TRAN;
  DELETE #VendorCopy WHERE VendorID = 1;
  SAVE TRAN Vendor1;
    DELETE #VendorCopy WHERE VendorID = 2;
    SAVE TRAN Vendor2;
      DELETE #VendorCopy WHERE VendorID = 3;
      SELECT * FROM #VendorCopy;
    ROLLBACK TRAN Vendor2;
    SELECT * FROM #VendorCopy;
  ROLLBACK TRAN Vendor1;
  SELECT * FROM #VendorCopy;
COMMIT TRAN;
SELECT * FROM #VendorCopy;
```

The response from the system



	VendorID	VendorName
1	4	Jobtrak

	VendorID	VendorName
1	3	Register of Copyrights
2	4	Jobtrak

	VendorID	VendorName
1	2	National Information Data Ctr
2	3	Register of Copyrights
3	4	Jobtrak

	VendorID	VendorName
1	2	National Information Data Ctr
2	3	Register of Copyrights
3	4	Jobtrak

Terms related to transactions

- Transaction
- Commit a transaction
- Roll back a transaction
- Autocommit mode
- Save point

Two transactions that retrieve and then modify the same row (part 1)

Transaction A

```
BEGIN TRAN;
DECLARE @InvoiceTotal money, @PaymentTotal money,
    @CreditTotal money;
SELECT @InvoiceTotal = InvoiceTotal,
       @CreditTotal = CreditTotal,
       @PaymentTotal = PaymentTotal
FROM Invoices WHERE InvoiceID = 112;
UPDATE Invoices
  SET InvoiceTotal = @InvoiceTotal,
      CreditTotal = @CreditTotal + 317.40,
      PaymentTotal = @PaymentTotal WHERE InvoiceID = 112;
COMMIT TRAN;
```

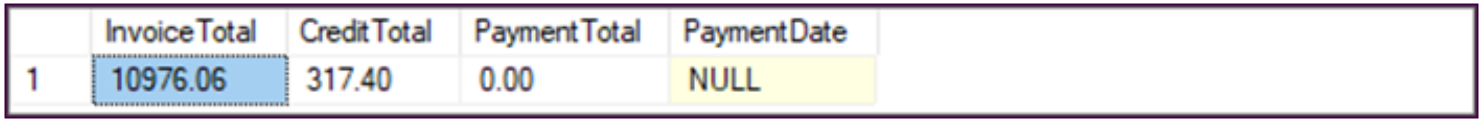
Two transactions that retrieve and then modify the same row (part 2)

Transaction B

```
BEGIN TRAN;
DECLARE @InvoiceTotal money, @PaymentTotal money,
        @CreditTotal money;
SELECT @InvoiceTotal = InvoiceTotal,
       @CreditTotal = CreditTotal,
       @PaymentTotal = PaymentTotal
FROM Invoices WHERE InvoiceID = 112;
UPDATE Invoices
  SET InvoiceTotal = @InvoiceTotal,
      CreditTotal = @CreditTotal,
      PaymentTotal = @InvoiceTotal - @CreditTotal,
      PaymentDate = GetDate() WHERE InvoiceID = 112;
COMMIT TRAN;
```

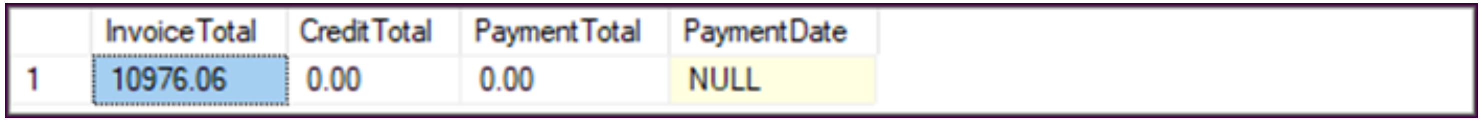
Two transactions that retrieve and then modify the same row (part 3)

The values after transaction A executes



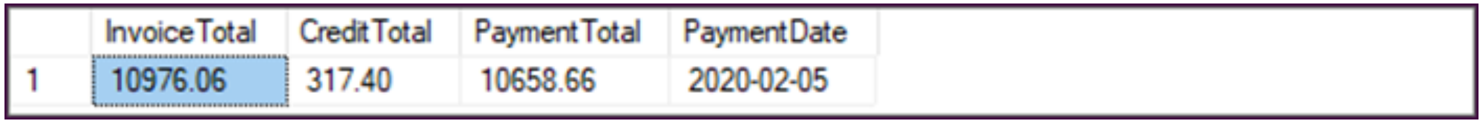
	InvoiceTotal	CreditTotal	PaymentTotal	PaymentDate
1	10976.06	317.40	0.00	NULL

The initial values for the row



	InvoiceTotal	CreditTotal	PaymentTotal	PaymentDate
1	10976.06	0.00	0.00	NULL

The values after transaction B executes



	InvoiceTotal	CreditTotal	PaymentTotal	PaymentDate
1	10976.06	317.40	10658.66	2020-02-05

The four types of concurrency problems

Problem	Description
Lost updates	Occur when two transactions select the same row and then update the row based on the values originally selected.
Dirty reads (uncommitted dependencies)	Occur when a transaction selects data that isn't committed by another transaction.
Nonrepeatable reads (inconsistent analysis)	Occur when two SELECT statements of the same data result in different values because another transaction has updated the data in the time between the two statements.
Phantom reads	Occur when you perform an update or delete on a set of rows when another transaction is performing an insert or delete that affects one or more rows in that same set of rows.

The syntax of the SET TRANSACTION ISOLATION LEVEL statement

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

Concurrency problems prevented by each transaction isolation level

Isolation level	Dirty reads	Lost updates	Nonrepeatable reads	Phantom reads
READ UNCOMMITTED	Allows	Allows	Allows	Allows
READ COMMITTED	Prevents	Allows	Allows	Allows
REPEATABLE READ	Prevents	Prevents	Prevents	Allows
SNAPSHOT	Prevents	Prevents	Prevents	Prevents
SERIALIZABLE	Prevents	Prevents	Prevents	Prevents

Terms related to concurrency

- Concurrency
- Locks
- Transaction isolation level

10 levels of lockable resources (coarse to fine)

Resource	Locks…
Database	An entire database.
Allocation unit	A collection of pages that contains a particular type of data.
Metadata	The data in the system catalog.
File	An entire database file.
Table	An entire table, including indexes.
Heap or B-tree	The index pages (B-tree) for a table with a clustered index or the data pages (heap) for a table with no clustered index.
Extent	A contiguous group of eight pages.
Page	One page (8 KB) of data.
Key	A key or range of keys in an index.
Row	A single row within a table.

Common SQL Server lock modes

Category	Lock mode	What the lock owner can do
Shared	Schema Stability (Sch-S)	Compile a query
	Intent Shared (IS)	Read but not change data
	Shared (S)	Read but not change data
	Update (U)	Read but not change data until promoted to an Exclusive (X) lock
Exclusive	Shared with Intent Exclusive (SIX)	Read and change data
	Intent Exclusive (IX)	Read and change data
	Exclusive (X)	Read and change data
	Bulk Update (BU)	Bulk-copy data into a table
	Schema Modification (Sch-M)	Modify the database schema

Terms related to locking

- Lockable resources
- Coarse-grain lock
- Fine-grain lock
- Lock manager
- Lock escalation
- Lock mode
- Lock promotion

Compatibility between lock modes

		Requested lock mode								
Current lock mode		**Sch-S**	**IS**	**S**	**U**	**SIX**	**IX**	**X**	**BU**	**Sch-M**
Schema Stability	**Sch-S**	√	√	√	√	√	√	√	√	
Intent Shared	**IS**	√	√	√	√	√	√			
Shared	**S**	√	√	√	√					
Update	**U**	√	√	√						
Shared w/Intent Exclusive	**SIX**	√	√							
Intent Exclusive	**IX**	√	√				√			
Exclusive	**X**	√								
Bulk Update	**BU**	√								
Schema Modification	**Sch-M**									

Two transactions that deadlock: Transaction A

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

   DECLARE @InvoiceTotal money;

   BEGIN TRAN;
     SELECT @InvoiceTotal = SUM(InvoiceLineItemAmount)
     FROM InvoiceLineItems
     WHERE InvoiceID = 101;

   WAITFOR DELAY '00:00:05';

     UPDATE Invoices
     SET InvoiceTotal = @InvoiceTotal
     WHERE InvoiceID = 101;

   COMMIT TRAN;
```

The response from the system

```
(1 row affected)
```

Two transactions that deadlock: Transaction B

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

   DECLARE @InvoiceTotal money;

   BEGIN TRAN;
       SELECT @InvoiceTotal = InvoiceTotal
       FROM Invoices
       WHERE InvoiceID = 101;

       UPDATE InvoiceLineItems
       SET InvoiceLineItemAmount = @InvoiceTotal
       WHERE InvoiceID = 101 AND InvoiceSequence = 1;

   COMMIT TRAN;
```

The response from the system

```
Msg 1205, Level 13, State 51, Line 11
Transaction (Process ID 53) was deadlocked on lock
resources with another process and has been chosen
as the deadlock victim. Rerun the transaction.
```

How the deadlock occurs

1. Transaction A requests and acquires a shared lock on the InvoiceLineItems table.
2. Transaction B requests and acquires a shared lock on the Invoices table.
3. Transaction A tries to acquire an exclusive lock on the Invoices table to perform the update. Since transaction B already holds a shared lock on this table, transaction A must wait for the exclusive lock.
4. Transaction B tries to acquire an exclusive lock on the InvoiceLineItems table, but must wait because transaction A holds a shared lock on that table.

Terms related to deadlocks

- Deadlock
- Deadlock victim

Coding techniques that prevent deadlocks (part 1)

Don’t allow transactions to remain open for very long

- Keep transactions short.
- Keep SELECT statements outside of the transaction except when absolutely necessary.
- Never code requests for user input during an open transaction.

Use the lowest possible transaction isolation level

- The default level of READ COMMITTED is almost always sufficient.
- Reserve the use of higher levels for short transactions that make changes to data where integrity is vital.

Coding techniques that prevent deadlocks (part 2)

Make large changes when you can be assured of nearly exclusive access

- If you need to change millions of rows in an active table, don't do so during hours of peak usage.
- If possible, give yourself exclusive access to the database before making large changes.

Consider locking when coding your transactions

- If you need to code two or more transactions that update the same resources, code the updates in the same order in each transaction.

UPDATE statements that transfer money between two accounts

From savings to checking

```
UPDATE Savings SET Balance = Balance - @TransferAmt;
UPDATE Checking SET Balance = Balance + @TransferAmt;
```

From checking to savings

```
UPDATE Checking SET Balance = Balance - @TransferAmt;
UPDATE Savings SET Balance = Balance + @TransferAmt;
```

From checking to savings in reverse order to prevent deadlocks

```
UPDATE Savings SET Balance = Balance + @TransferAmt;
UPDATE Checking SET Balance = Balance - @TransferAmt;
```