Transactions and Isolation

# Isolation

## First Look at Isolation

As we know, one of the key elements of ACID transactions is **isolation** – the separation of multiple concurrent transactions that might be trying to use the same resources.

Let’s have our first look at that by creating a table in a transaction, and trying to view it in the object explorer before it is committed. We are going to use the TestDB database and create simplified versions of the Orders and OrderLines tables, which we will be using later.

NOTE: It is important that you execute all of the following steps in the same query window. We will see why later.

1. Open a new transaction:

BEGIN TRANSACTION;

1. Create our two new tables:

CREATE TABLE dbo.Orders (

OrderID INT IDENTITY PRIMARY KEY,

CustomerName NVARCHAR(50),

CustomerAddr NVARCHAR(100), -- Which normal form rule does this violate?

OrderDate DATE

);

CREATE TABLE dbo.OrderLines (

OrderLineID INT IDENTITY PRIMARY KEY,

OrderID INT,

ItemDesc NVARCHAR(50),

Quantity INT,

UnitPrice MONEY, -- When is this a 3NF violation? When is it not?

CONSTRAINT FK\_OrderLines\_Orders FOREIGN KEY ( OrderID ) REFERENCES dbo.Orders ( OrderID )

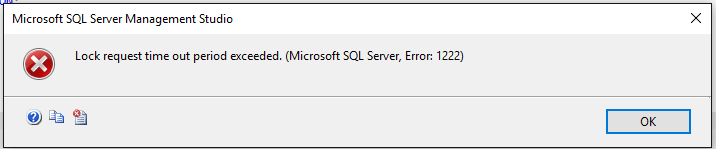
);

1. Refresh the TestDB database in the object explorer and try to expand the “Tables” tab.
2. Commit the transaction.

COMMIT TRANSACTION;

1. Refresh TestDB in the object explorer and try to expand “Tables” again.

If you wait long enough, you should see an error like this at step 3.



Why couldn’t we view the tables? What does it mean by “lock”? Why did OrderLines have no problem creating a foreign key referencing Orders, even though we couldn’t access the table anywhere else?

## Locks

The error thrown by the object explorer above was a lock wait timeout error.

When we created the tables but had not committed them, SQL Server automatically took a lock. The purpose of this lock was to prevent changes to the environment that may have affected our transaction. Similarly, when we tried to read the list of tables, the object explorer tried to take a lock to prevent changes that would affect the table list while that transaction was executing – it tired to maintain its isolation.

These two locks were incompatible, so the object explorer had to wait for our lock to be released. If we waited long enough, we timed out and got an error.

While there are many types of locks, for our purposes we will be looking at two main types: **shared** and **exclusive**.

**Exclusive Lock**

An exclusive lock is taken when a resource is modified, such as with INSERT, UPDATE, and DELETE. It signifies that the resource is dirty and should not be read until that change is committed. It is incompatible with other locks.

**Shared Lock**

A shared lock is taken when a resource is read, such as with a SELECT. It is compatible with other shared locks, but not with exclusive locks.

In our example above, our table create statement took an exclusive lock, which was incompatible with the shared lock that the object explorer was attempting to take.

## Locking Resources

To better see locking in action, we will need to maintain two simultaneous transactions. The easiest way for us to do that is to use two separate query windows. If it makes it easier for you to view, you are able to undock and move a query window by clicking and dragging the query’s tab.

In this demo, we will move back and forth between two open query tabs. I will be labelling each step as belonging to either “Query 1” or “Query 2”.

### Example 1

1. QUERY 1: Open a new transaction

BEGIN TRANSACTION;

1. QUERY 1: Insert a row into dbo.Orders. **Request exclusive lock.**

INSERT INTO dbo.Orders ( CustomerName, CustomerAddr, OrderDate )

VALUES ( 'Clark', '123 Main St.', GETDATE() );

1. QUERY 2: Attempt to select all rows from dbo.Orders. **Request shared lock.**

SELECT \* FROM dbo.Orders;

1. QUERY 1: Commit. **Release exclusive lock.**

COMMIT TRANSACTION;

You should notice two important things. In step 3, the query keeps running without ever completing. In step 4, as soon as we commit the select statement in query 2 immediately completes.

Query 1 holds an exclusive lock for the duration of its transaction. Query 2 attempts to take a shared lock, but that lock is not compatible with the exclusive lock that Query 1 already has, so it has to wait. When we commit query 1, its lock is released and query 2’s shared lock is granted.

### Example 2

1. QUERY 1: Open a new transaction

BEGIN TRANSACTION;

1. QUERY 1: Insert a row into dbo.Orders. **Request exclusive lock.**

INSERT INTO dbo.Orders ( CustomerName, CustomerAddr, OrderDate )

VALUES ( 'Bruce', '456 Main St.', GETDATE() );

1. QUERY 2: Attempt to select OrderID 1, the order we created in our last step. **Request shared lock.**

SELECT \* FROM dbo.Orders WHERE OrderID = 1;

1. QUERY 1: Commit. **Release exclusive lock.**

COMMIT TRANSACTION;

Why did query 2 succeed this time?

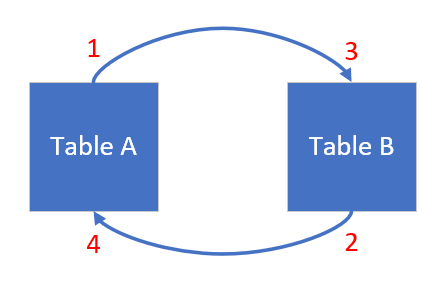
SQL Server will only attempt to lock the resources it needs. Query 1 only requires an exclusive lock on the row it has added, not on the entire table. This time, instead of trying to read all rows, query 2 is just looking for a specific row. Since that row is different than the one that is locked, query 2 can take its shared lock and complete successfully.

# Deadlocks

## What is a Deadlock

A deadlock happens when two or more processes are all waiting on each other to complete, resulting in none of them being able to finish. In a database, this most typically happens when two or more transactions take locks that the other needs to wait for before they can complete.

We can see this in the following simple diagram:



1. Transaction 1 takes a lock on Table A. There is no conflict, so the lock is granted.
2. Transaction 2 takes a lock on Table B. There is no conflict, so the lock is granted.
3. Transaction 1 attempts to take a lock on Table B. Transaction 2 already has a lock, so transaction 1 waits for that lock to be released.
4. Transaction 2 attempts to take a lock on Table A. Transaction 1 already has a lock, so transaction 2 waits for that lock to be released.

Both transactions are waiting for the other to release their locks, so neither transaction is able to complete and release their lock. As a result, both transactions get stuck waiting for each other to complete.

## Deadlock Example

A customer has ordered a bike. We get the paper copy of the order and input it into the system. We are careful to use a transaction, because an order without order lines is not meaningful in our system.

BEGIN TRANSACTION;

INSERT INTO dbo.Orders ( CustomerName, CustomerAddr, OrderDate )

VALUES ( 'Harry', '123 Fourth St.', GETDATE() );

INSERT INTO dbo.OrderLines ( OrderID, ItemDesc, Quantity, UnitPrice )

VALUES (SCOPE\_IDENTITY(), 'Red Bike', 1, 150.00 );

COMMIT TRANSACTION;

Someone notices an issue with the original form. Harry’s address should be 123 Fifth St. and he ordered 2 bikes, not one. They submit a new paper form to be added to our system. We set out to update it, being careful to use transactions because the order should either match the old order form or the new one, but should never be halfway between the two.

Unfortunately, a coworker is doing the same work, but doing it in the opposite order that we are:

|  |  |
| --- | --- |
| **QUERY 1** | **QUERY 2** |
| BEGIN TRANSACTION; | BEGIN TRANSACTION; |
| UPDATE dbo.OrderLines  SET Quantity = 2  WHERE ItemDesc = 'Red Bike'; | UPDATE dbo.Orders  SET CustomerAddr = '123 Fifth St.'  WHERE CustomerName = 'Harry' |
| UPDATE dbo.Orders  SET CustomerAddr = '123 Fifth St.'  WHERE CustomerName = 'Harry' | UPDATE dbo.OrderLines  SET Quantity = 2  WHERE ItemDesc = 'Red Bike'; |
| COMMIT TRANSACTION; | COMMIT TRANSACTION; |

We will get an error like this:  
Msg 1205, Level 13, State 51, Line 9

Transaction (Process ID 54) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

## Resolving and Preventing Deadlocks

SQL Server will automatically detect deadlocks. It will choose a deadlock victim to cancel, allowing the other transaction to proceed.

### Resolving Deadlocks

The simplest way to resolve a deadlock is to attempt to rerun the victim transaction. The transaction it conflicted with will have already successfully completed, so it often runs successfully without issue when retried.

### Preventing Deadlocks

Deadlocks are almost inevitable in any system with many concurrent transactions, but there are a few simple steps we can take to help prevent them:

1. **Control the flow**In our example above, our deadlock happened because one transaction altered Orders first then OrderLines, but the other transaction did the reverse. Controlling processes to ensure that they always affect data in one direction alleviates deadlocks. This is often (but not always) done by using stored procedures to access the database from an application.
2. **Keep Transactions Small**Make sure that transactions are encapsulating the smallest possible block of code. The longer and large the locks, the more likely a transaction is to cause a deadlock.
3. **Consolidate Code**As we learned earlier in this course, it is often possible to write our queries in a single statement instead of multiple independent statements with variables or temp tables. The more consolidated our code is, the better it is a taking locks and less likely it is to cause contention.

# Homework

1. Which of the following locks are compatible?
   1. A shared lock and an exclusive lock on the same resource.
   2. Two shared locks on the same resource.
   3. Two exclusive locks on the same resource.
   4. A shared lock and an exclusive lock on different resources.
   5. Three or more shared locks on the same resource.
2. What is a deadlock? How does the DBMS resolve it?
3. Could a shared lock ever cause a deadlock?