Mimic a Lock With Transactions

NOTE: To see the underlining properly, turn off the proofreading. Go to “File -> Options -> Proofing and uncheck the spell check boxes”

In this example, we will see how SQL Server implements the concept of two-phase locking. First, we need to know that by default an update, select or delete statement in SQL Server causes a lock known as an exclusive lock.

This type of lock doesn’t allow nay other operations to acquire a lock on the table while it is being worked on.

We will learn how to mimic locks with explicit transaction in SQL server.

Step. 1: Table Creation

Connect to a database in a new query pad (we will refer to this as Query1) and create the following Employee table and populate with data.

```
create table Employee
(
  pkId int primary key identity(1,1) ,
  empName varchar(50)
)

insert into Employee
select 'John'
union
select 'Steve'
```

Execute just this script and you should now have the table and data setup.

Step 2: Open a transaction

In the same Query1, we are going to open an explicit transaction without committing the result. We do this by doing BEGIN TRAN and the transaction statement

```
--Incomplete Transaction
--Run a transaction and update the table without committing the
transaction, which will
--cause this process maintain a lock on Employee
Begin tran
update Employee set empName = 'John Smith'
where pkId = 1
```

Execute this statement and after it executes, Open a new Query pad Window (we will refer to this as Query2)

Step 3: Query the table being worked on by the open transaction

When you open Query2, this is a different connection to the database from Query1. This lets us simulate one user updating (Query1) and another user trying to select (Query2).

Try to execute the following query in Query2:

```
--Query the table that is being locked by our update
--without (nolock) hint. It is waiting for another process to release
the lock

select * from Employee
```

You'll notice that the query never returns anything. In fact, it seems to be running and doesn't complete its operation.

This is because Query1 still has a lock on the Employee table from the open transaction.

Step 4: In Query1, complete the transaction

In the Query 1 tab, execute this statement:

```
Commit tran
```

This will commit the update statement that you began in Step 2 and will release any locks on the Employee table.

In turn, this will cause the select statement you were running in Query2 to return results. Check Query 2 and you will see that it has returned records.

Step 5: Using the (nolock) hint

Repeat step 2 again (but change the empname to something else) in Query 1. This should cause the lock to occur on Employee table again.

Now, in Query2 run the following statement:

```
--Query the table that is being locked by our update
--with a (nolock) hint, which can give back uncommited data (Dirty
Read)

select * from Employee (nolock)
```

You'll see that you get data right away. Why?

Even though Query 1 has a lock on the employee table, you have used a (nolock) hint on your query.

(Nolock) essentially tells the select statement, not to acquire an exclusive lock and allow the select statement to read what is currently in memory for the table (known as a shared lock).

Therefore, you should see the empname to be whatever value you placed for the empName update statement.

HOWEVER, using NOLOCK can allow you to read uncommitted data. Remember your step 2 transaction is uncommitted. So, if the user was to “rollback”, the value you see in your query will be different.

Step 6: Rolling back a transaction

Go back to Query1. This time instead of committing the open transaction, we are going to roll it back or undo it from committing the value.

To rollback an open transaction use:

```
rollback tran
```

This will undo your update, and if you run your select statement in Query 2, you will see that it has a the previous value.