CS108: Advanced Database

- Database Programming

Lecture 09:

Transactions and Concurrency

Overview

- **Transactions**
- Locks and Blocking
  - Locks and Lock Types
  - Troubleshooting Blocking
- Isolation Levels
- Deadlocks

Transactions

- A transaction is a unit of work that might include multiple activities that query and modify data.
- We define a transaction by:
  - the beginning of a transaction explicitly with
    - a *BEGIN TRAN* (or *BEGIN TRANSACTION*) statement.
  - and the end of a transaction explicitly with
    - *COMMIT TRAN* statement if we want to confirm it
    - *ROLLBACK TRAN* (or *ROLLBACK TRANSACTION*) statement if we do not want to confirm it (undo the changes)

Transactions

- By default, SQL Server automatically commits the transaction at the end of each individual statement.
- We can change the way SQL Server handles implicit transactions with IMPLICIT_TRANSACTIONS.
  - When this option is ON, we do not have to specify the BEGIN TRAN statement to mark the beginning of a transaction
  - We have to mark the transaction's end with a COMMIT TRAN or a ROLLBACK TRAN statement

```
BEGIN TRAN;
INSERT INTO T1(keycol, col1, col2) VALUES(4, 101, 'C');
INSERT INTO T2(keycol, col1, col2) VALUES(4, 201, 'X');
COMMIT TRAN;
```

Transactions

- Transactions have four properties - atomicity, consistency, isolation, and durability - ACID.
  - *Atomicity* - A transaction is an atomic unit of work.
  - *Consistency* - The state of the data that the RDBMS gives we access to as concurrent transactions modify and query it.
  - *Isolation* - A mechanism used to control access to data and ensure that transactions access data only if the data is in the level of consistency.
  - *Durability* - Data changes are always written to the database's transaction log on disk before they are written to the data portion of the database on disk.

Transaction Example

```
BEGIN TRAN

    -- Declare a variable
    DECLARE @neworderid AS INT;

    -- Insert a new order into the Sales.Orders table
    INSERT INTO Sales.SalesOrderHeader
        (RevisionNumber, OrderDate, DueDate, ShipDate,
         Status, OnlineOrderFlag, PurchaseOrderNumber, AccountNumber,
         CustomerID, SalesPersonID, TerritoryID, BillToAddressID,
         ShipToAddressID, ShipMethodID, CreditCardID,
         CreditCardApprovalCode, CurrencyRateID, SubTotal, TaxAmt,
         Freight, Comment)
    VALUES
        (8, '2014-06-30', '2014-07-12',  '2014-07-07', 5, 1,
         NULL, '10-4030-018759', 18759, NULL, 6, 14024, 14024, 1,
         10084, '230370Vi51970', NULL, 189.97, 15.1976, 4.7493, NULL)

    -- Save the new order ID in a variable
    SET @neworderid = SCOPE_IDENTITY();
```

Transaction Example

```sql
    -- Save the new order ID in a variable
    SET @neworderid = SCOPE_IDENTITY();

    -- Return the new order ID
    SELECT @neworderid AS neworderid;

    -- Insert order lines for the new order into Sales.OrderDetails
    INSERT INTO Sales.SalesOrderDetail
        (SalesOrderID, CarrierTrackingNumber, OrderQty,
         ProductID, SpecialOfferID, UnitPrice, UnitPriceDiscount)
    VALUES
        (@neworderid, '4911-403C-98', 1, 776, 1, 2024.994,  0.00),
        (@neworderid, '4911-403C-98', 1, 776, 1, 2024.994,  0.00),
        (@neworderid, '4911-403C-98', 1, 776, 1, 2024.994,  0.00),
        (@neworderid, '4911-403C-98', 1, 776, 1, 2024.994,  0.00),
        (@neworderid, '4911-403C-98', 1, 776, 1, 2024.994,  0.00),
        (@neworderid, '4911-403C-98', 1, 776, 1, 2024.994,  0.00);

COMMIT TRAN;
```

Overview

- Transactions
- **Locks and Blocking**
  - **Locks and Lock Modes**
  - **Troubleshooting Blocking**
- Isolation Levels
- Deadlocks

Locks and Lock Modes

- SQL Server uses *locks* to enforce the *isolation* property of transactions.
- Locks are control resources obtained by a transaction to guard data resources, preventing conflicting or incompatible access by other transactions.
- There are two types of lock – *shared* and *exclusive*.
  - *Shared lock* - If one or more shared locks already exist, exclusive locks cannot be obtained.
  - *Exclusive lock* - If an object is exclusively locked, shared locks cannot be obtained.

Lock Modes and Compatibility

- When we try to modify data, our transaction requests an *exclusive lock* on the data resource, regardless of the isolation level.
  - *Exclusive locks* – a resource cannot obtain an exclusive lock if another transaction is holding any lock mode on the resource.
- When we try to read data (READ COMMITTED isolation level), by default our transaction requests a *shared lock* on the data resource and releases the lock as soon as the read statement is done with that resource
  - *Shared locks* – multiple transactions can hold shared locks on the same data resource simultaneously.

Lock Modes and Compatibility

- Lock interaction between transactions is known as lock compatibility.

Requested Mode	Granted Exclusive (X)	Granted Shared (S)
Grant request for exclusive?	No	No
Grant request for shared?	No	Yes

- A “No” in the intersection means that the locks are incompatible and the requested mode is denied; the requester must wait.
- A “Yes” in the intersection means that the locks are compatible and the requested mode is accepted.

Lockable Resource Types

- SQL Server can lock different types of resources. The types of resources that can be locked include
  - RIDs or keys (row), pages, objects (for example, tables), databases, and others.
- SQL Server determines dynamically which resource types to lock.
  - For ideal concurrency, it is best to lock only what needs to be locked, namely only the affected rows.
  - SQL Server might first acquire fine-grained locks (e.g., page locks), and in certain circumstances, try to escalate the fine-grained locks to more coarse-grained locks (e.g., table locks).

Troubleshooting Blocking

- When one transaction holds a *lock* on a data resource and another transaction requests an incompatible lock on the same resource, the request is *blocked* and the requester enters a *wait* state.
- *Blocking* is normal in a system as long as requests are satisfied within a reasonable amount of time.
- If some requests end up *waiting* too long, we might need to *troubleshoot* the *blocking* situation and see whether we can do something to prevent such long latencies.

Example: Troubleshooting Blocking

- We will demonstrates a blocking situation and walks we through the process of troubleshooting it.
- Note that this demonstration assumes that we're connected to a SQL Server instance and using the READ COMMITTED isolation level, meaning that by default SELECT statements will request a *shared lock*.

- STEP 1: Open three separate query windows in SQL Server Management Studio.
- STEP 2: Run the following code in Connection 1 to update a row in the Production.Product table, adding 1.00 to the current unit price of 0.00 for product 2.

```
BEGIN TRAN;

UPDATE Production.Product
SET    ListPrice += 1.00
WHERE  ProductId = 2;
```

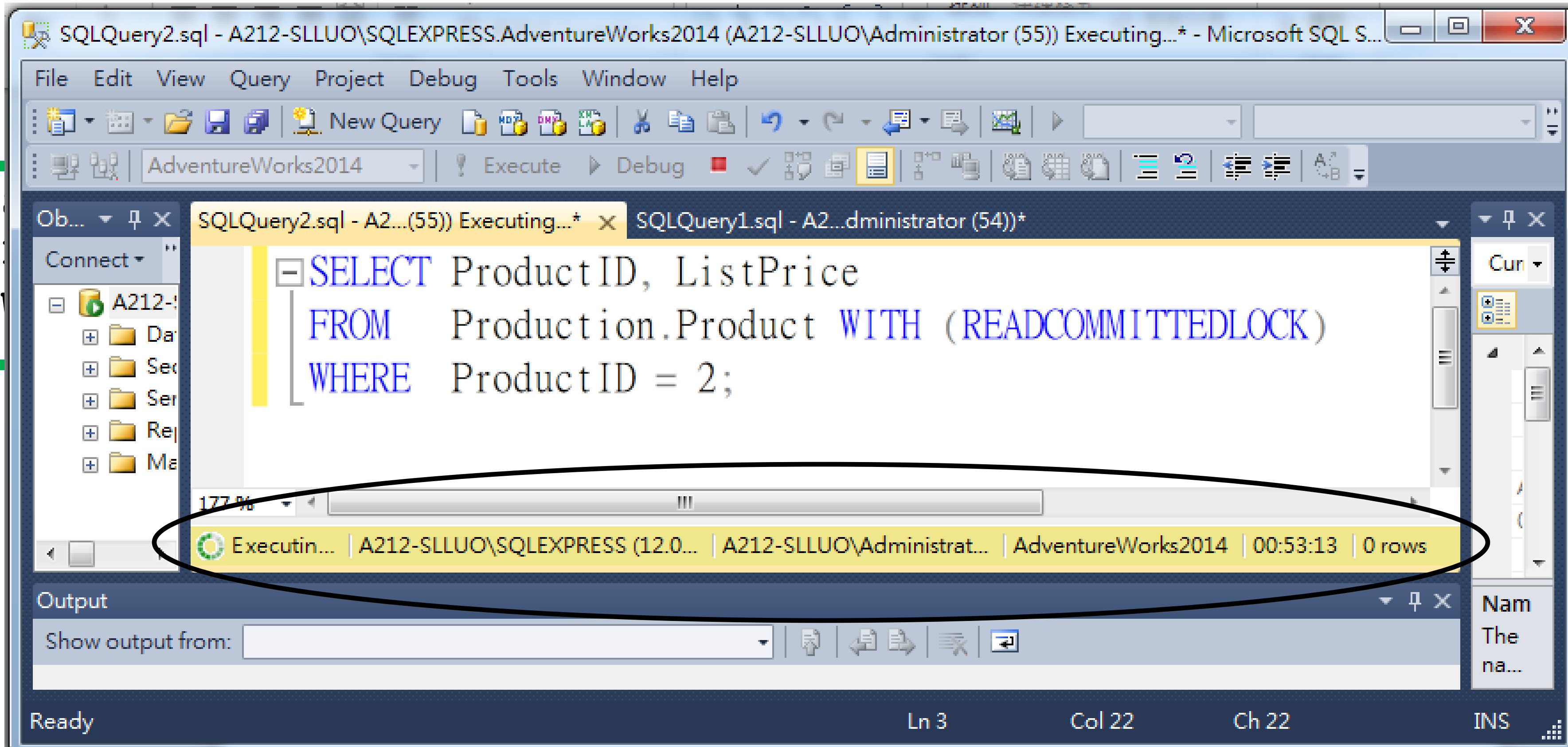
- Recall that *exclusive locks* are kept until the *end* of the transaction, and because the transaction remains open, the *lock* is still held.

- STEP 3: Run the following code in Connection 2 to try to query the same row (uncomment the hint WITH (READCOMMITTEDLOCK))

```
SELECT ProductID, ListPrice
FROM   Production.Product --WITH (READCOMMITTEDLOCK)
WHERE  ProductID = 2;
```

- Our session needs a *shared lock* to read the data, but because the row is exclusively locked by the other session, and a *shared lock* is incompatible with an *exclusive lock*, our session is blocked and has to *wait*.

- STEP 3: Run the following code in Connection 2 to try to query the same row (uncomment the hint WITH



our session is blocked and has to wait.

- STEP 4: Assuming that such a *blocking* situation happens in our system, and the blocked session ends up *waiting* for a long time, we probably want to *troubleshoot* the situation.
- We use queries against dynamic management objects, including views and functions, that we should run from Connection 3 when we troubleshoot the blocking situation.
- To get lock information, including both locks that are currently granted to sessions and locks that sessions are waiting for, query the dynamic management view (DMV) sys.dm_tran_locks in Connection 3.

```sql
SELECT -- use * to explore other available attributes
    request_session_id                AS spid,
    resource_type                     AS restype,
    resource_database_id              AS dbid,
    DB_NAME(resource_database_id)     AS dbname,
    resource_description              AS res,
    resource_associated_entity_id     AS resid,
    request_mode                      AS mode,
    request_status                    AS status
FROM sys.dm_tran_locks;
```

```
spid restype     dbid dbname                  res                         resid               mode  status
--------------------------------------------------------------------------------------------------------------
53   DATABASE    5    ReportServer$SQLEXPRESS                             0                   S     GRANT
55   DATABASE    7    AdventureWorks2014                                  0                   S     GRANT
54   DATABASE    7    AdventureWorks2014                                  0                   S     GRANT
54   METADATA    7    AdventureWorks2014      xml_collection_id = 65539   0                   Sch-S GRANT
55   PAGE        7    AdventureWorks2014      1:925                       72057594047365120   IS    GRANT
54   PAGE        7    AdventureWorks2014      1:925                       72057594047365120   IX    GRANT
54   METADATA    7    AdventureWorks2014      xml_collection_id = 65540   0                   Sch-S GRANT
54   METADATA    7    AdventureWorks2014      schema_id = 7               0                   Sch-S GRANT
55   OBJECT      7    AdventureWorks2014                                  1973582069          IS    GRANT
54   OBJECT      7    AdventureWorks2014                                  1973582069          IX    GRANT
54   KEY         7    AdventureWorks2014      (61a06abd401c)              72057594047365120   X     GRANT
55   KEY         7    AdventureWorks2014      (61a06abd401c)              72057594047365120   S     WAIT
```


	spid	restype	dbid	dbname	res	resid	mode	status
12	55	KEY	7	AdventureWorks2014	(61a06abd401c)	72057594047365120	S	WAIT

spid	restype	dbid	dbname	res	resid	mode	status
53	DATABASE	5	ReportServer$SQLEXPRESS		0	S	GRANT
55	DATABASE	7	AdventureWorks2014		0	S	GRANT
54	DATABASE	7	AdventureWorks2014		0	S	GRANT
54	METADATA	7	AdventureWorks2014	xml_collection_id = 65539	0	Sch-S	GRANT
55	PAGE	7	AdventureWorks2014	1:925	72057594047365120	IS	GRANT
54	PAGE	7	AdventureWorks2014	1:925	72057594047365120	IX	GRANT
54	METADATA	7	AdventureWorks2014	xml_collection_id = 65540	0	Sch-S	GRANT
54	METADATA	7	AdventureWorks2014	schema_id = 7	0	Sch-S	GRANT
55	OBJECT	7	AdventureWorks2014		1973582069	IS	GRANT
54	OBJECT	7	AdventureWorks2014		1973582069	IX	GRANT
54	KEY	7	AdventureWorks2014	(61a06abd401c)	72057594047365120	X	GRANT
55	KEY	7	AdventureWorks2014	(61a06abd401c)	72057594047365120	S	WAIT

- As we can see in the output of the query against sys.dm_tran_locks, three sessions (53, 54, 55) are currently holding locks. We can see the following:
  - The resource type that is *locked* (for example, KEY for a row in an index)
  - The ID of the database in which it is locked, which we can translate to the database name by using the DB_NAME function
  - The resource and resource ID
  - The lock mode
  - Whether the lock was granted or the session is waiting for it

- STEP 5: The sys.dm_tran_locks view only gives we information about the IDs of the processes involved in the blocking chain and nothing else.
- To get information about the connections associated with the processes involved in the blocking chain, query a view called sys.dm_exec_connections, and filter only the SPIDs that are involved.

```
SELECT -- use * to explore
       session_id AS spid,
       connect_time,
       last_read,
       last_write,
       most_recent_sql_handle
FROM  sys.dm_exec_connections
WHERE session_id IN(54, 55);
```

```
spid        connect_time            last_read
------------------------------------------------------------------
54          2016-11-10 12:13:16.963 2016-11-10 12:15:14.367
55          2016-11-10 12:17:06.790 2016-11-10 12:17:59.450

spid        last_write
------------------------------------------------------------------
54           2016-11-10 12:15:14.367
55           2016-11-10 12:17:10.703

spid        most_recent_sql_handle
------------------------------------------------------------------
54          0x01000700823 8FB24D053E1D201000000000000000000000000
55          0x02000003B22321B0FD81B5667957BBF96D0AD048759778500
```

- The information that this query gives us about the connections includes:
  - The time they connected.
  - The time of their last read and write.
  - A binary value holding a handle to the most recent SQL batch run by the connection. We provide this handle as an input parameter to a table function called sys.dm_exec_sql_text, and the function returns the batch of code represented by the handle.

- STEP 6: We query the table function passing the binary handle explicitly, and apply the table function to each connection row like this (run in Connection 3) to see the process.

```
SELECT session_id, text
FROM   sys.dm_exec_connections
CROSS APPLY sys.dm_exec_sql_text(most_recent_sql_handle) AS ST
WHERE session_id IN(54, 55);
```

```
session_id  text
----------  -------------------------------------------------------
54          BEGIN TRAN;

           UPDATE Production.Product
           SET    ListPrice += 1.00
           WHERE  ProductId = 2;

55          (@1 tinyint)SELECT [ProductID],[ListPrice] FROM
            [Production].[Product] WITH(readcommittedlock) WHERE [ProductID]=@1
```

- The blocked process – 55, shows the query that is waiting because that’s the last thing that the process ran.

- STEP 7: We will probably find very useful for troubleshooting blocking situations by sys.dm_exec_requests.

```
SELECT -- use * to explore
       session_id AS spid,
       blocking_session_id,
       command,
       sql_handle,
       database_id,
       wait_type,
       wait_time,
       wait_resource
FROM   sys.dm_exec_requests
WHERE  blocking_session_id > 0;
```

```
spid blocking_session_id command
------------------------------------
55   54                  SELECT

spid sql_handle                                          database_id
--------------------------------------------------------------------
55   0x02000000FB9954025FF409245270C42E53393D9094EB342  7

spid wait_type wait_time   wait_resource
--------------------------------------------------------------------
53   LCK_M_S   1383760     KEY: 7:72057594047365120 (61a06abd401c)
```

- STEP 8: We can terminate the blocker by kill <spid>. To terminate the update transaction in Connection 1, run the following code from Connection 3.

```
KILL 54;
```

- This statement causes a rollback of the transaction in Connection 1, meaning that the price change of product 2 from 0.00 to 1.00 is undone.

```
ProductID   ListPrice
----------- ---------------------
2            0.00
```

Overview

- Transactions
- Locks and Blocking
  - Locks and Lock Types
  - Troubleshooting Blocking
- **Isolation Levels**
- Deadlocks

Isolation Levels

- Isolation levels determine the behavior of concurrent users who read or write data.
  - A reader is any statement that selects data, using a shared lock by default.
  - A writer is any statement that makes a modification to a table and requires an exclusive lock.
- We *cannot* control the way writers behave in terms of the locks that they acquire and the duration of the locks, but we can control the way readers behave.
- We do so by setting the isolation level, either at the session level with a session option or at the query level with a table hint.

Isolation Levels

- SQL Server supports four traditional isolation levels that are based on pessimistic concurrency control (locking):
  - READ UNCOMMITTED,
  - READ COMMITTED (the default in SQL Server instances),
  - REPEATABLE READ, and
  - SERIALIZABLE.
- SQL Server also supports two isolation levels that are based on optimistic concurrency control (row versioning):
  - SNAPSHOT and
  - READ COMMITTED SNAPSHOT.

Isolation Levels

- We can set the isolation level of the whole session by using

```
SET TRANSACTION ISOLATION LEVEL <isolation name>;
```

- We can use a table hint to set the isolation level of a query.

```
SELECT ... FROM <table> WITH (<isolationname>);
```

- The higher the isolation level
  - The tougher the locks that readers request and the longer their duration;
  - The higher the consistency and the lower the concurrency.

The READ UNCOMMITTED Isolation Level

- READ UNCOMMITTED is the lowest available isolation level.
- In this isolation level, a reader doesn’t ask for a shared lock.
- A reader that doesn’t ask for a shared lock can never be in conflict with a writer that is holding an exclusive lock.
- The reader can read uncommitted changes (also known as dirty reads).

- Example: To see how an uncommitted read (dirty read) works, open two query windows (Connection 1 and Connection 2).
- STEP 1: Run the following code in Connection 1 to open a transaction, update the unit price of product 2 by adding 1.00 to its current price (0.00), and then query the product’s row.

```
BEGIN TRAN;
    UPDATE Production.Product
    SET    ListPrice = ListPrice + 1.00
    WHERE  ProductId = 2;

    SELECT ProductId, ListPrice
    FROM   Production.Product
    WHERE  ProductId = 2;
```

- STEP 1: Note that the transaction remains open, meaning that the product's row is locked exclusively by Connection 1.

```
BEGIN TRAN;
     UPDATE Production.Product
     SET    ListPrice = ListRice + 1.00
     WHERE  ProductId = 2;

     SELECT ProductId, ListPrice
     FROM   Production.Product
     WHERE  ProductId = 2;
```

```
ProductId   ListPrice
----------- ---------------------
2           1.00
```

- STEP 2: In Connection 2, run the following code to set the isolation level to READ UNCOMMITTED and query the row for product 2.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT ProductId, ListPrice
FROM   Production.Product
WHERE  ProductId = 2;
```

```
ProductId   ListPrice
----------- ---------------------
2           1.00
```

- STEP 3: Keep in mind that Connection 1 might apply further changes to the row later in the transaction or even roll back at some point. For example

```
ROLLBACK TRAN;
```

- This rollback undoes the update of product 2, changing its price back to 0.00. The value 1.00 that the reader got was never committed.

- That’s an example of a dirty read.

```
ProductId   ListPrice
----------- ---------------------
2           0.00
```

The READ COMMITTED Isolation Level

- If we want to prevent readers from reading uncommitted changes, we need to use a stronger isolation level.
- The lowest isolation level that prevents dirty reads is READ COMMITTED, which is also the default isolation level in a SQL Server installation.
- It prevents uncommitted reads by requiring a reader to obtain a shared lock.
- If a writer is holding an exclusive lock, the reader's shared lock request will be in conflict with the writer, and it has to wait.

- Example: The READ COMMITTED Isolation Level. To see how this works, open two query windows (Connection 1 and Connection 2).
- STEP 1: Run the following code in Connection 1 to open a transaction, update the unit price of product 2 by adding 1.00 to its current price (0.00), and then query the product's row.

```
BEGIN TRAN;
    UPDATE Production.Product
    SET    ListPrice = ListPrice + 1.00
    WHERE  ProductId = 2;

    SELECT ProductId, ListPrice
    FROM   Production.Product
    WHERE  ProductId = 2;
```

- STEP 1: Connection 1 now locks the row for product 2 exclusively.

```
BEGIN TRAN;
    UPDATE Production.Product
    SET    ListPrice = ListPrice + 1.00
    WHERE  ProductId = 2;

    SELECT ProductId, ListPrice
    FROM   Production.Product
    WHERE  ProductId = 2;
```

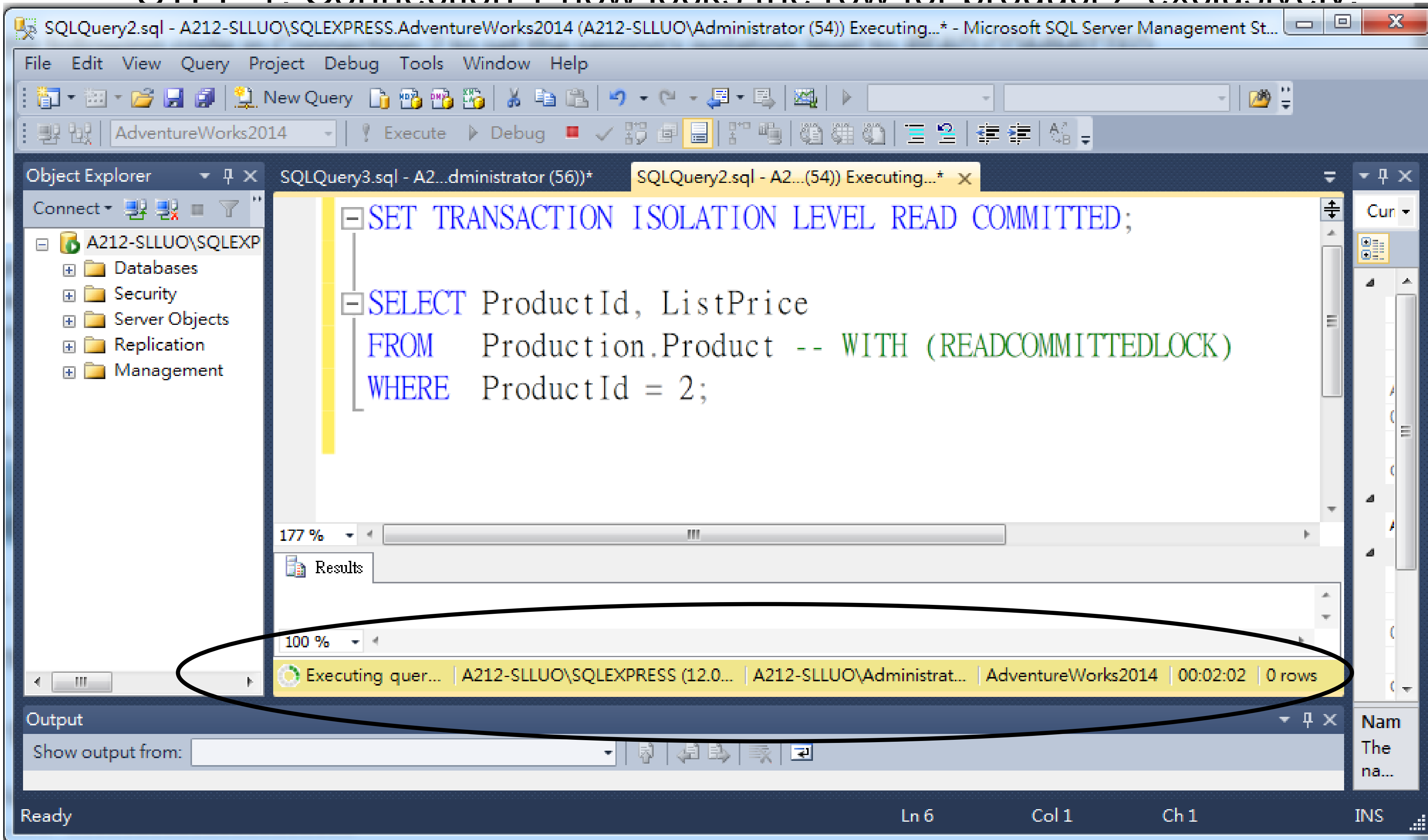
```
ProductId   ListPrice
----------- --------------------
2           1.00
```

- STEP 2: Run the following code in Connection 2 to set the session's isolation level to READ COMMITTED and query the row

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT ProductId, ListPrice
FROM   Production.Product -- WITH (READCOMMITTEDLOCK)
WHERE  ProductId = 2;
```

- The SELECT statement is currently blocked because it needs a shared lock to be able to read, and this shared lock request is in conflict with the exclusive lock held by the writer in Connection 1.

- STEP 1: Connection 1 now locks the row for product 2 exclusively.



shared lock to be able to read, and this shared lock request is in conflict with the exclusive lock held by the writer in Connection 1.

- STEP 3: Next, run the following code in Connection 1 to commit the transaction.

```
COMMIT TRAN;
```

- Now go to Connection 2 and notice that we get the following output.

SQLQuery3.sql - A2...dministrator (56))* | SQLQuery2.sql - A2...dministrator (54))* | SQLQuery1.sql - A2...dministrator (53))*

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT ProductId, ListPrice
FROM   Production.Product -- WITH (READCOMMITTEDLOCK)
WHERE  ProductId = 2;
```

```
ProductId   ListPrice
----------- ---------------------
2           1.00
```

100 %

Query executed successfully. | A212-SLLUO\SQLEXPRESS (12.0... | A212-SLLUO\Administrat... | AdventureWorks2014 | 00:03:21 | 1 rows

The READ COMMITTED Isolation Level

- Unlike in READ UNCOMMITTED, in the READ COMMITTED isolation level, we don't get dirty reads.
- In the READ COMMITTED isolation level, a reader only holds the shared lock until it is done with the resource.
  - It doesn't keep the lock until the end of the transaction; it doesn't even keep the lock until the end of the statement.
  - In between two reads of the same data resource in the same transaction, no lock is held on the resource.

The READ COMMITTED Isolation Level

- The READ COMMITTED Isolation Level doesn't keep the lock until the end of the transaction.
  - Another transaction can modify the resource in *between* those two reads, and the reader might get different values in each read.
  - This phenomenon is called non-repeatable reads or inconsistent analysis.
- For many applications, non-repeatable reads is acceptable, but for some it isn't.

The REPEATABLE READ Isolation Level

- If we want to ensure that no one can change values in *between* reads that take place in the same transaction, we need to move up in the isolation levels to REPEATABLE READ.
- In this isolation level, not only does a reader need a shared lock to be able to read, but it also holds the lock until the end of the transaction.
- We're guaranteed to get repeatable reads, or consistent analysis.

- Example: The REPEATABLE READ Isolation Level
- STEP 1: Run the following code in Connection 1 to set the session's isolation level to REPEATABLE READ, open a transaction, and read the row for product 2.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRAN;
    SELECT ProductId, ListPrice
    FROM   Production.Product
    WHERE  ProductId = 2;
```

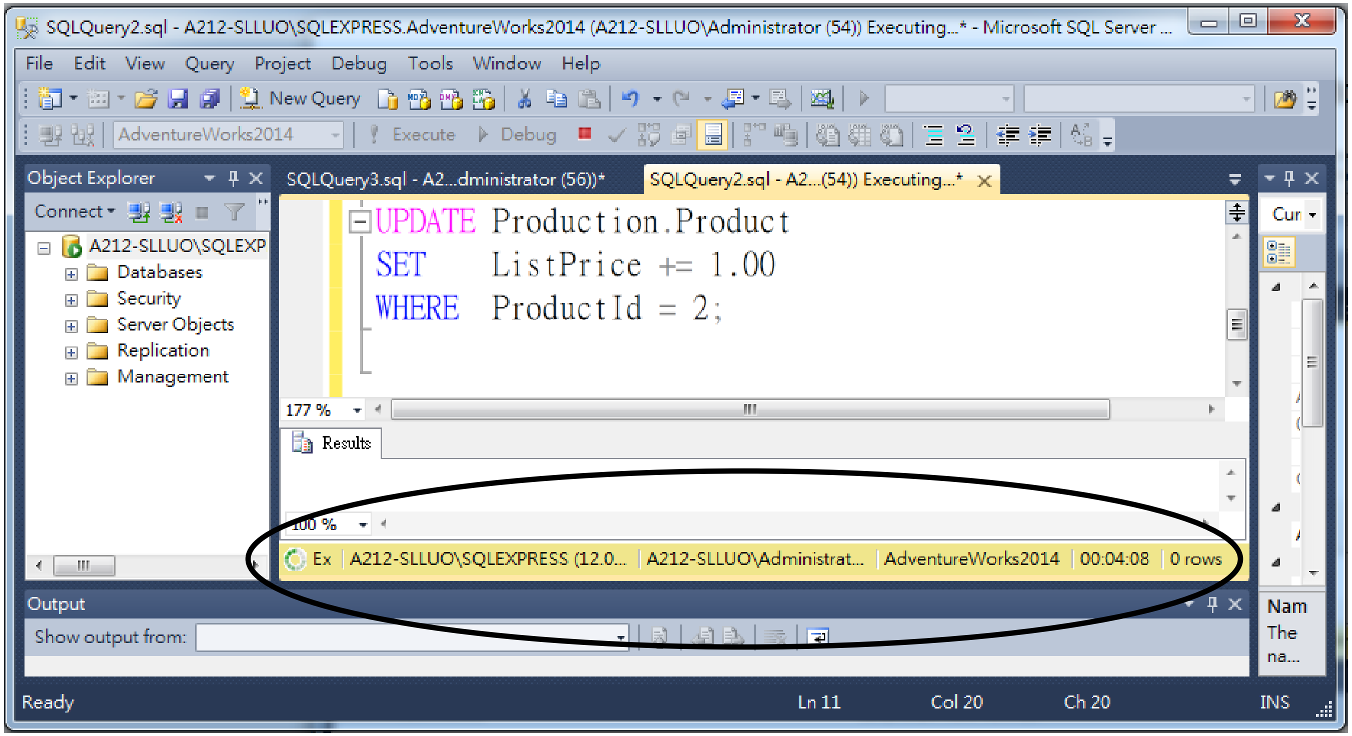
```
ProductId   ListPrice
----------- ---------------------
2           0.00
```

- Connection 1 still holds a shared lock on the row for product 2 because in REPEATABLE READ, shared locks are held until the end of the transaction.

- STEP 2: Run the following code from Connection 2 to try to modify the row for product 2.

```
UPDATE Production.Product
SET    ListPrice = ListPrice + 1.00
WHERE  ProductId = 2;
```

- Notice that the attempt is blocked because the modifier's request for an exclusive lock is in conflict with the reader's granted shared lock.



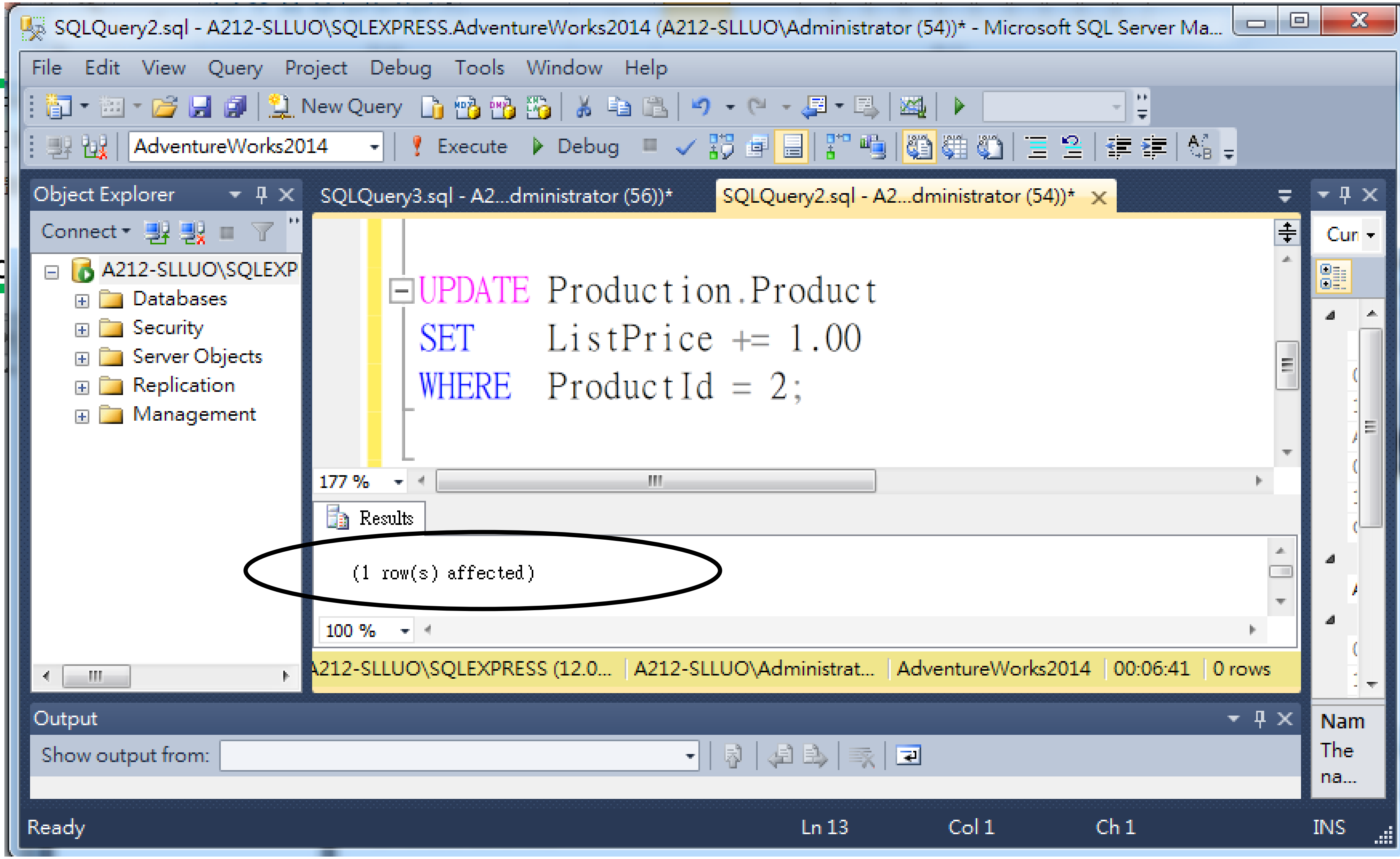
- STEP 3: Back in Connection 1, run the following code to read the row for product 2 a second time and commit the transaction.

```
    SELECT ProductId, ListPrice
    FROM   Production.Product
    WHERE  ProductId = 2;

COMMIT TRAN;
```

```
ProductId   ListPrice
----------- ---------------------
2           0.00
```

- Notice that the second read got the *same* unit price for product 2 as the first read.
- Now that the reader's transaction has been committed and the shared lock is released, the modifier in Connection 2 can obtain the exclusive lock it was waiting for and update the row.

- STEP 3: Back in Connection 1, run the following code to read the



Lost Update Prevented by REPEATABLE READ

- Another phenomenon prevented by REPEATABLE READ but not by lower isolation levels is called a *lost update*
- In isolation levels lower than REPEATABLE READ
  - No lock is held on the resource *after* the read
  - Transactions can update the value, will *overwrite* the other transaction's update
- In REPEATABLE READ, both sides keep their shared locks after the first read, so neither can acquire an exclusive lock later in order to update
  - A deadlock situation occur, and the update conflict is prevented

Phantom Reads

- The REPEATABLE READ keeps readers shared locks until the end of the transaction to guarantee to get a repeatable read of the rows.
- The transaction locks resources (for example, rows) that the query found the first time it ran, may not rows that weren’t there when the query ran.
  - A second read in the same transaction might return *new rows* as well. Those new rows are called phantoms, and such reads are called *phantom reads*.
  - This happens if, in between the reads, another transaction adds new rows that qualify for the reader’s query filter.

The SERIALIZABLE Isolation Level

- To prevent phantom reads, we need to move up in the isolation levels to SERIALIZABLE.
- The SERIALIZABLE isolation level
  - Similarly to REPEATABLE READ: requires a reader to obtain a shared lock to be able to read, and keeps the lock until the end of the transaction; and
  - Also lock the whole range of keys that qualify for the query's filter – it blocks attempts made by other transactions to add rows that qualify for the reader's query filter.

- Example: The SERIALIZABLE isolation level prevents phantom reads.
- STEP 1: Run the following code in Connection 1 to set the transaction isolation level to SERIALIZABLE, open a transaction, and query all products with category 1.

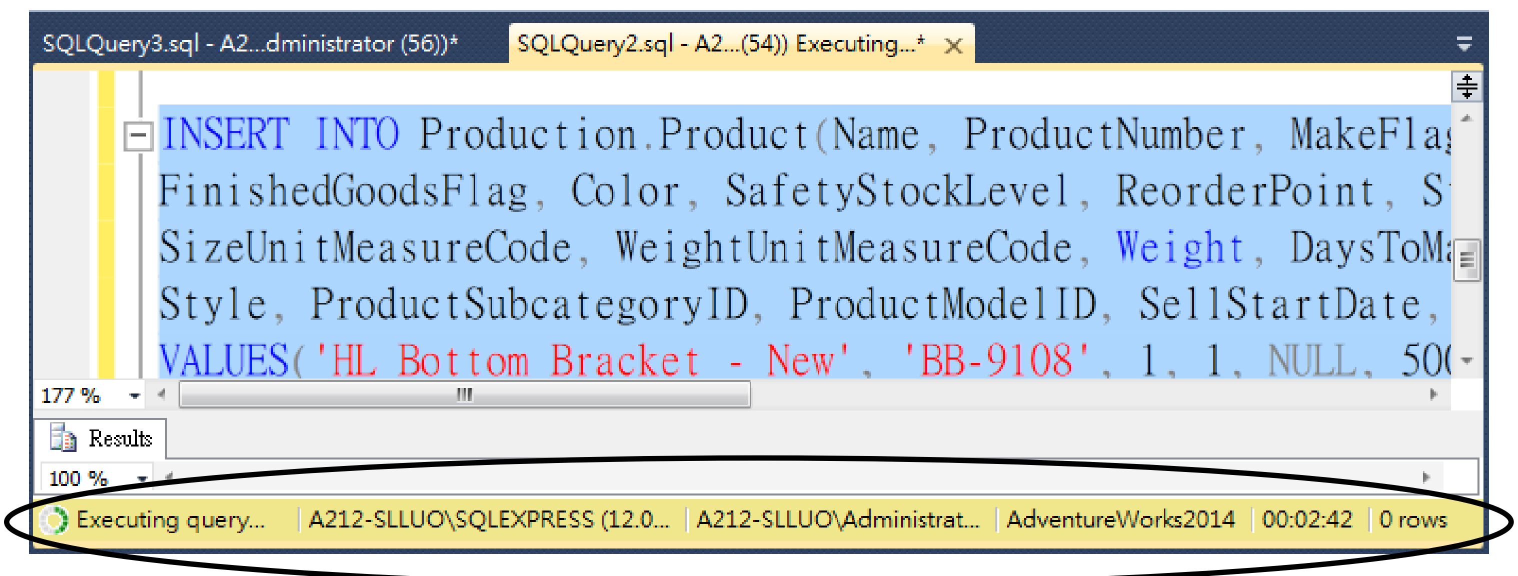
```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRAN;
    SELECT ProductId, Name, ProductSubcategoryID, ListPrice
    FROM   Production.Product
    WHERE  ProductSubcategoryID = 5;
```

```
ProductId   Name                   ProductSubcategoryID     ListPrice
----------- ---------------------- ------------------------ -------------------
994         LL Bottom Bracket      5                        53.99
995         ML Bottom Bracket      5                        101.24
996         HL Bottom Bracket      5                        121.49
```

- STEP 2: From Connection 2, run the following code in an attempt to insert a new product with category 5.

```sql
INSERT INTO Production.Product(Name, ProductNumber, MakeFlag,
FinishedGoodsFlag, Color, SafetyStockLevel, ReorderPoint, StandardCost, ListPrice, Size,
SizeUnitMeasureCode, WeightUnitMeasureCode, Weight, DaysToManufacture, ProductLine, Class,
Style, ProductSubcategoryID, ProductModelID, SellStartDate, SellEndDate, DiscontinuedDate)
VALUES('HL Bottom Bracket - New', 'BB-9108', 1, 1, NULL, 500, 375, 53.9416, 121.49, NULL,
NULL, 'G', 170.00, 1, NULL, 'H', NULL, 5, 97, '2013-05-30 00:00:00.000', NULL, NULL);
```

- In all isolation levels that are lower than SERIALIZABLE, such an attempt would have been successful. In the SERIALIZABLE isolation level, the attempt is blocked.



- STEP 3: Back in Connection 1, run the following code to query products with category 1 a second time and commit the transaction.

```
    SELECT ProductId, Name, ProductSubcategoryID, ListPrice
    FROM   Production.Product
    WHERE  ProductSubcategoryID = 5;

COMMIT TRAN;
```

```
ProductId   Name                   ProductSubcategoryID      ListPrice
----------- ---------------------- ------------------------- --------------------
994         LL Bottom Bracket      5                         53.99
995         ML Bottom Bracket      5                         101.24
996         HL Bottom Bracket      5                         121.49
```

- We get the same output as before, with no phantoms. Now that the reader's transaction is committed, and the shared key-range lock is released, the modifier in Connection 2 can obtain the exclusive lock it was waiting for and insert the row.

Summary: Isolation Level

Isolation Level	Allows Uncommitted Reads?	Allows Non-repeatable Reads?	Allows Lost Updates?	Allows Phantom Reads?
READ UNCOMMITTED	YES	YES	YES	YES
READ COMMITTED	NO	YES	YES	YES
REPEATABLE READ	NO	NO	NO	YES
SERIALIZABLE	NO	NO	NO	NO

Overview

- Transactions
- Locks and Blocking
  - Locks and Lock Types
  - Troubleshooting Blocking
- Isolation Levels
- **Deadlocks**

Deadlocks

- A deadlock is a situation in which two or more processes block each other.
- SQL Server detects the deadlock and intervenes by terminating one of the transactions.
- SQL Server chooses to terminate the transaction that did the least work, because it is cheapest to roll that transaction’s work back.

- Example: A simple deadlock.
- STEP 1: Run the following code in Connection 1 to open a new transaction, update a row in the Sales.SalesOrderDetails table for product 897, and leave the transaction open.

```
BEGIN TRAN;

    UPDATE Sales.SalesOrderDetail
    SET    UnitPrice = UnitPrice + 1.00
    WHERE  ProductID = 897;
```

- STEP 2: Run the following code in Connection 2 to open a new transaction, update a row in the Production.Product table for product 897, and leave the transaction open.

```
BEGIN TRAN;

    UPDATE Production.Product
    SET    ListPrice = ListPrice + 1.00
    WHERE  ProductId = 897;
```

- Connection 1 is holding an exclusive lock on in the Sales.SalesOrderDetails table, and the Connection 2 is now holding locks in the Production.Product table. Both queries succeed, and no blocking has occurred yet.
- STEP 3: Run the following code in Connection 1 to attempt to query the rows for product 897 in the Production.Product table and commit the transaction

```
    SELECT ProductId, ListPrice
    FROM   Production.Product -- WITH (READCOMMITTEDLOCK)
    WHERE  ProductId = 897;

COMMIT TRAN;
```

- Connection 1 needs a shared lock to be able to perform its read. Because the other transaction holds an exclusive lock on the same resource, Connection 1 is blocked.
- At this point, we have a blocking situation, not yet a deadlock.

- STEP 4: Run the following code in Connection 2 to attempt to query the row for product 897 in the Sales.SalesOrderDetail table and commit the transaction.

```
    SELECT SalesOrderDetailID, ProductId, UnitPrice
    FROM   Sales.SalesOrderDetail -- WITH (READCOMMITTEDLOCK)
    WHERE  ProductId = 897;

COMMIT TRAN;
```

- Connection 2 needs a shared lock in the Sales.SalesOrderDetail table, this request is now in conflict with the exclusive lock held on the same resource by Connection 1. Each of the processes blocks the other - we have a deadlock.
- SQL Server identifies the deadlock (typically within a few seconds), chooses one of the two processes as the deadlock victim, and terminates its transaction with the following error.

```
Msg 1205, Level 13, State 51, Line 38
Transaction (Process ID 53) was deadlocked on lock resources with another
process and has been chosen as the deadlock victim. Rerun the transaction.
```

```
Msg 1205, Level 13, State 51, Line 38
Transaction (Process ID 53) was deadlocked on lock resources with another
process and has been chosen as the deadlock victim. Rerun the transaction.
```

- In this example, SQL Server chose to terminate the transaction in Connection 1 (shown here as process ID 53).
- Deadlocks are expensive because they involve undoing work that has already been done.
- Obviously, the longer the transactions are, the longer locks are kept, increasing the probability of deadlocks. We should try to keep transactions as short as possible.
- A deadlock happens when transactions access resources in inverse order. By swapping the order in one of the transactions, we can prevent this type of deadlock from happening - assuming that it makes no logical difference to our application.