**Seguridad:**

Inicio de Sesión

CREATE LOGIN [EXPRESS\Usuario] FROM WINDOWS

CREATE LOGIN SQLUser WITH PASSWORD='Pa$$w0rd'

Crear Usuarios de Base de Datos

USE DemoMSDN  
GO  
CREATE USER Usuario FOR LOGIN [EXPRESS\Usuario]

**Esquemas de Base de Datos**

USE MASTER  
GO  
  
-- Creamos un Inicio de Sesión de SQL Server  
CREATE LOGIN Alumno WITH PASSWORD='Pa$$w0rd'  
GO  
  
-- Creamos una Base de Datos para pruebas  
CREATE DATABASE TestDB  
GO  
  
-- Nos conectamos a la Base de Datos  
USE TestDB  
GO  
  
-- Creamos el Schema Ventas  
CREATE SCHEMA Ventas  
GO  
  
--Creamos la Tabla Pedidos en el Esquema Ventas  
CREATE TABLE Ventas.Pedidos (idPedido int primary key, FechaPedido smalldatetime, idCliente int, Estado tinyint)  
GO  
  
-- Creamos el usuario alumno enlazado con el Inicio de Sesión Alumno  
CREATE USER Alumno FOR LOGIN Alumno  
GO  
  
-- Otorgamos el permiso de Select al usuario Alumno sobre el Esquema Ventas.  
GRANT SELECT ON SCHEMA::Ventas TO Alumno

Nota: Para ejecutar este ejemplo, la instancia de SQL Server Express debe de estar ejecutándose en modo Autenticación Mixta.

**Funciones**

Ejemplo de Función de concatenación

SET NOCOUNT ON;

USE Northwind;

GO

IF OBJECT\_ID('dbo.fn\_ConcatOrders') IS NOT NULL

DROP FUNCTION dbo.fn\_ConcatOrders;

GO

CREATE FUNCTION dbo.fn\_ConcatOrders

(@cid AS NCHAR(5)) RETURNS VARCHAR(8000)

AS

BEGIN

DECLARE @orders AS VARCHAR(8000);

SET @orders = '';

SELECT @orders = @orders + CAST(OrderID AS VARCHAR(10)) + ';'

FROM dbo.Orders

WHERE CustomerID = @cid;

RETURN @orders;

END

GO

Inline Table-Valued UDFs

SET NOCOUNT ON;

USE Northwind;

GO

IF OBJECT\_ID('dbo.fn\_GetCustOrders') IS NOT NULL

DROP FUNCTION dbo.fn\_GetCustOrders;

GO

CREATE FUNCTION dbo.fn\_GetCustOrders

(@cid AS NCHAR(5)) RETURNS TABLE

AS

RETURN

SELECT OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate,

ShippedDate, ShipVia, Freight, ShipName, ShipAddress, ShipCity,

ShipRegion, ShipPostalCode, ShipCountry

FROM dbo.Orders

WHERE CustomerID = @cid;

GO

Luego:

SELECT O.OrderID, O.CustomerID, OD.ProductID, OD.Quantity

FROM dbo.fn\_GetCustOrders(N'ALFKI') AS O

JOIN [Order Details] AS OD

ON O.OrderID = OD.OrderID;

**Exec (SQL Dinámico):**

DECLARE @sql AS VARCHAR(52);

SET @sql = 'SELECT \* FROM dbo.Orders WHERE OrderID = 10248;';

EXEC(@sql);

DECLARE @i AS INT;

SET @i = 10248;

DECLARE @sql AS VARCHAR(52);

SET @sql = 'SELECT \* FROM dbo.Orders WHERE OrderID = '

+ CAST(@i AS VARCHAR(10)) + N';';

EXEC(@sql);

**Tempdb (Tabla Temporal)**

Temporary tables are created in tempdb, regardless of the database context of your session.

SET NOCOUNT ON;

USE tempdb;

GO

IF OBJECT\_ID('dbo.proc1') IS NOT NULL

DROP PROC dbo.proc1;

GO

CREATE PROC dbo.proc1

AS

CREATE TABLE #T1(col1 INT NOT NULL);

INSERT INTO #T1 VALUES(1);

SELECT \* FROM #T1;

GO

**Transacciones**

## Niveles de aislamiento

Si nos detenemos en la definición de aislamiento podemos observar que no es muy concreta, es decir que da lugar a distintas interpretaciones o implementaciones de ese aislamiento. Esos niveles y exactamente lo que significa el aislamiento es lo que vamos a tratar en los siguientes apartados.

### READ UNCOMMITED

Lecturas no confirmadas, realmente lo que sucede en este nivel de aislamiento es que los usuarios pueden leer datos que aún no están confirmados, y que por tanto pueden llegar a no existir nunca. Los problemas que presenta este nivel de aislamiento son los siguientes:

* Lecturas sucias: Lee datos que no han llegado a validarse.
* Lecturas no repetibles: Dos sentencias SELECT iguales y consecutivas podrían devolver datos diferentes.
* Datos fantasma: En dos sentencias SELECT iguales y consecutivas podrían aparecer y desaparecer filas.

Hay que ser muy cuidadoso si se utiliza este nivel de aislamiento ya que podría hacer que los usuarios se basen en datos que realmente no han existido nunca y que por tanto son erróneos. En muchas ocasiones se usa para evitar bloqueos, si así lo hacemos hemos de estar seguros que las consecuencias son admisibles.

### READ COMMITED

Lecturas confirmadas, este es el nivel de aislamiento por defecto en SQL Server, no lee datos que no estén confirmados, sino que esperaría (se quedaría bloqueado) a que esos datos estén confirmados. Los bloqueos que establece mientras se lee información, tan solo permanecen activos durante el tiempo de la ejecución no durante toda la transacción y no se ve bloqueado por lecturas. Los problemas que tiene este nivel de aislamiento son:

* Lecturas no repetibles: Dos sentencias SELECT iguales y consecutivas podrían devolver datos diferentes.
* Datos fantasma: En dos sentencias SELECT iguales y consecutivas podrían aparecer y desaparecer filas.

En este nivel de aislamiento las lecturas solo se ven bloqueadas por las escrituras, y las escrituras rara vez se ven bloqueadas por lecturas y además solo durante un muy corto periodo de tiempo de tal forma que no suelen producirse muchos interbloqueos.

### REPETEABLE READ

Lecturas repetibles. Este nivel de aislamiento garantiza que dos select consecutivas dentro de una transacción devolverán la misma información, y lo hace creando bloqueos compartidos sobre los registros que lee de tal forma que no pueden ser modificados. El único problema que presenta es:

* Datos fantasma: En dos sentencias SELECT iguales y consecutivas podrían aparecer filas.

En este nivel de aislamiento las lecturas solo se ven bloqueadas por las escrituras, pero las escrituras se ven bloqueadas por lecturas durante el tiempo que dura la transacción que lee de tal forma que es más frecuente encontrar problemas de bloqueos.

### SERIALIZABLE

Serializable. Este nivel de aislamiento no tiene ni siquiera el problema de datos fantasma, por que cuando realiza un select crea bloqueos compartidos no solamente sobre los registros que existen sino sobre los nuevos que pudiesen llegar (inserts) de tal forma que dos instrucciones select consecutivas dentro de la misma transacción devolverán exactamente los mismos datos. En muchas ocasiones los desarrolladores eligen este nivel por que es lo más parecido a “estar solo” en el sistema, sin embargo este nivel de aislamiento necesita crear un número considerable de bloqueos para poder garantizar estas lecturas repetibles evitando datos fantasma.  
  
En este nivel de aislamiento las lecturas solo se ven bloqueadas por las escrituras, pero las escrituras se ven bloqueadas por lecturas incluso escrituras de tipo INSERT que en el nivel de aislamiento REPEATABLE READ no se verían afectas.

Este es el nivel máximo de aislamiento y también genera el nivel máximo de bloqueos, analice si realmente necesita estas características antes de usarlo.

## Nuevo nivel de aislamiento de instantánea

El nuevo nivel de aislamiento de instantánea (el SNAPSHOT Isolation Level), cumple los mismos requisitos que el serializable es decir no tiene ninguno de los problemas de lecturas sucias, ni lecturas no repetibles ni lecturas fantasma, pero no se basa en una estrategia de bloqueo para conseguirlo sino en una estrategia de versiones de las filas. Esta estrategia permite evitar todos los problemas descritos, sin necesidad de bloquear las filas, de tal forma que una sentencia SELECT devolverá exactamente los mismos datos cada vez que se ejecute.

Dentro de estos nuevos niveles de aislamiento existen dos versiones, el propiamente llamado SNAPSHOT ISOLATION y el READ COMMITTED SNAPSHOT.  
  
Para habilitarlos debemos usar el comando ALTER DATABASE <Base de Datos> SET ALLOW\_SNAPSHOT\_ISOLATION ON o ALTER DATABASE <Base de Datos> SET READ\_COMMITTED\_SNAPSHOT ON.  
  
El nivel de aislamiento de instantáneas puede mostrar conflictos en actualizaciones, el motivo es sencillo, si las sentencias select devuelven siempre la misma información pero no han impedido que otros usuarios cambien la información, en el momento en que la transacción actual vaya a cambiar algún dato, debe comprobar que los datos que están validados en la base de datos son los mismos que existían cuando fueron leídos y versionados la primera vez, en caso contrario podríamos sobrescribir modificaciones de otros usuarios pero basados en datos que ya no existen.

**Isolation Levels**

Isolation levels allow you to control the consistency level that you will get when manipulating data, bearing in mind that multiple processes might be running concurrently. SQL Server 2000 gave you four isolation levels (*read uncommitted, read committed, repeatable read*, and *serializable*). SQL Server 2005 adds two new isolation levels (*snapshot* and *read committed snapshot*).

The different isolation levels control the level of consistency that you will get when manipulating data mainly by controlling the way readers behave. You can improve consistency at the cost of worsening concurrency, and vice versa. Technically, isolation levels improve consistency by increasing the duration of locks. Of course, the other side of the coin is that processes will need to wait longer.

Your choice of isolation level will determine which of the following types of consistency-related problems can or cannot happen:

* **Dirty reads** A read of uncommitted data. One process changes data but does not commit the change before another process reads the same data. The second process ends up reading an inconsistent state of the data.
* **Lost updates** One process reads data, makes some calculations based on the data, and later updates the data based on those calculations. If two processes first read the data and later update it based on what they read, one process might overwrite the other's update.
* **Nonrepeatable reads (also known as inconsistent analysis)** In two separate reads within the same transaction, the process gets different values when reading the same resource. This can happen if a second process changed the data in between the reads made by the first process.
* **Phantoms** Phantoms have to do with one process performing an action against a range of rows, while another process performs an incompatible action on rows in the same range. For example, one process deletes all rows based on some filter, and during the delete transaction (call it Tran1), another process inserts a new row that satisfies that filter. The new row is considered to be a *phantom row*. Issuing a SELECT query in the same transaction (Tran 1) with the same filter as the delete filter will return that phantom row—such a read is considered a *phantom read*.

Another aspect of concurrency that is determined by your choice of isolation level is the concurrency model, which can be either pessimistic or optimistic. In a pessimistic concurrency model, you lock a resource to guarantee that you will be able to perform an action you're planning later. In between the time you lock the resource and the time you perform the action later, no one can perform an incompatible action against the data. Obviously, this mode has the potential to hurt concurrency.

In an optimistic model, you don't lock the resource. Rather, you have the means to identify whether an incompatible action took place in between your activities. This model can potentially improve concurrency.

In the following sections, I'll describe the different isolation levels supported by SQL Server 2005 and which concurrency problems each does or doesn't allow.

SQL Server allows you to set the isolation level either at the session level or at the query level. You set the isolation level at the session level by issuing the following statement:

SET TRANSACTION ISOLATION LEVEL <isolation level>;

In this statement, *<isolation level>* can be one of the following: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE, or SNAPSHOT.

Or you can set the isolation level of a query by using a table hint (READUNCOMMITTED, READCOMMITTED, REPEATABLEREAD, or SERIALIZABLE). The hint NOLOCK is equivalent to READUNCOMMITTED.

|  |  |  |
| --- | --- | --- |
|  | Tip | SQL Server also provides you with a table hint called READPAST. This hint causes your process to skip locked rows rather than being blocked or getting dirty reads. Make sure, though, that it makes sense for your application to use this hint. SQL Server 2000 supported this hint only for queries that retrieve data. SQL Server 2005 supports it also with queries that modify data. |

SQL Server's default isolation level is read committed.

**Read Uncommitted**

When working with the read uncommitted isolation level, readers do not request shared locks. Thus, they're never in conflict with sessions that modify data. That is, they can read data that is locked exclusively, and they do not interfere with processes that modify data. Of course, at this level readers might get uncommitted changes. In other words, dirty reads can happen, as well as all other concurrency-related problems I described earlier. Read uncommitted is the worst isolation level in terms of consistency but the best in terms of concurrency.

To demonstrate a dirty read, first issue the following UPDATE statement to change the value of *T1.col2* to the string *'Version 1'* in the row where *keycol* is equal to 2:

UPDATE dbo.T1 SET col2 = 'Version 1' WHERE keycol = 2;

Open two connections, and call them connection 1 and connection 2. From connection 1, issue the following code, which updates *col2* to *'Version 2'* within a transaction and retrieves the modified column values, keeping the transaction open:

BEGIN TRAN

UPDATE dbo.T1 SET col2 = 'Version 2' WHERE keycol = 2;

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

You get *'Version 2'* as the output, showing you the new state of the value your own transaction changed.

From connection 2, set the session's isolation level to read uncommitted, and read the data:

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

Even though another transaction changed the data and had not committed yet, you are able to see the uncommitted change—you get the output *'Version 2'*. Note that the modifying transaction still maintains an exclusive lock on the data, but if a process that reads the data doesn't request a shared lock, it cannot get a refusal.

From connection 1, issue a rollback:

ROLLBACK TRAN

If at this point you read the *col2* value from the row where *keycol* is equal to *2*, you will get *'Version 1'* back. You realize that *'Version 2'* was never committed and that processes working with the read uncommitted isolation level might have relied on a state of the data that was never "approved."

**Read Committed**

Read committed is the default isolation level of SQL Server. In this isolation level, processes request a shared lock to read data and release it as soon as the data has been read—not when the transaction terminates. This means that dirty reads cannot happen; rather, the only changes you can read are those that have been committed. However, all other concurrency-related problems can happen with this isolation level.

To demonstrate the fact that a dirty read will not occur when working with the read committed isolation level, first run the following code in connection 1, changing the value of *col2* from *'Version 1'* to *'Version 2'*:

BEGIN TRAN

UPDATE dbo.T1 SET col2 = 'Version 2' WHERE keycol = 2;

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

You get the output *'Version 2'*, because you can read your own changes, of course.

Now try to read the data from connection 2, working in the read committed isolation level, and you will be blocked:

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

Commit the change in connection 1:

COMMIT TRAN

Connection 1 will release the exclusive lock, and connection two will get '*Version 2'* back, which is the committed state of the value after the change.

For cleanup, change the value back to *'Version 1'*:

UPDATE dbo.T1 SET col2 = 'Version 1' WHERE keycol = 2;

**Repeatable Read**

Processes working with the repeatable read isolation level also request a shared lock when reading data, meaning that dirty reads cannot occur at this level. But unlike with read committed, at the repeatable read level transactions keep shared locks until they are terminated. You are guaranteed to get repeatable reads (consistent analysis) because no other process will be able to obtain an exclusive lock in between your reads.

Lost updates cannot happen at this level as well. If two processes that read data preserve shared locks until the end of the transaction, an attempt to modify that data by both will cause a deadlock, because each will request an exclusive lock that will be blocked by the other. When SQL Server detects a deadlock, it chooses a victim—typically, the process that performed less work—and rolls the victim's transaction back. The victim process will get the infamous error 1205 and can reissue the transaction. Although lost updates cannot happen in repeatable read, phantoms are still possible.

To demonstrate a case where you get consistent analysis when working at the repeatable read level, run the following code from connection 1:

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRAN

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

You get the output *'Version 2'*, and the process keeps a shared lock on the data because the transaction is still open.

If you attempt to modify the data from connection 2, you will be blocked:

UPDATE dbo.T1 SET col2 = 'Version 3' WHERE keycol = 2;

Read the data again in connection 1, and then commit:

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

COMMIT TRAN

You still get *'Version 2'* back, meaning you got a repeatable read even though another process attempted to change the data in between your reads. Once the transaction committed, the shared lock was released, and connection 2 could obtain the exclusive lock it needed to update the data.

For cleanup, change the value back to *'Version 1'*:

UPDATE dbo.T1 SET col2 = 'Version 1' WHERE keycol = 2;

**Serializable**

The serializable isolation level is similar to repeatable read, with an additional facet—active transactions acquire key-range locks (placed on indexes) based on query filters. This applies not only to readers, but also to writers. Obtaining a key-range lock means that it's as if you logically lock all data that meets the query's filter. You not only lock whatever data was physically found when you accessed it, but you also lock data that does not exist yet that would happen to meet your query's filter. This level adds the prevention of phantoms to the list of problems that the repeatable read level's list doesn't handle.

To demonstrate the prevention of phantoms with the serializable isolation level, first create an index on *T1.col1*:

CREATE INDEX idx\_col1 ON dbo.T1(col1);

Then run the following code from connection 1:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRAN

UPDATE dbo.T1 SET col2 = 'Version 2'

WHERE col1 = 102;

Your transaction modifies the rows where *col1 = 102* (currently, there's only one such row in the table) and obtains an exclusive key-range lock in the index *idx\_col1* based on the filter.

Next, from connection 2, attempt to introduce a phantom row—a row that meets the filter of the modification submitted by connection 1:

INSERT INTO dbo.T1(keycol, col1, col2) VALUES(5, 102, 'D');

You will be blocked. If you make a similar attempt in any other isolation level, the insert will be accepted.

To terminate the open transaction, commit the change from connection 1:

COMMIT TRAN;

Connection 1 releases the key-range lock, and connection 2 can insert the new row.

Before you continue, issue the following code to drop the index *idx\_col1*, and to change the value of *col2* in the row where *keycol = 2* back to *'Version 1'*:

DROP INDEX dbo.T1.idx\_col1;

GO

UPDATE dbo.T1 SET col2 = 'Version 1' WHERE keycol = 2;

Also, set all connections to work under the default read committed isolation level:

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

**New Isolation Levels**

SQL Server 2005 introduces a new *row versioning* technology that allows it to maintain older images (versions) of rows that resulted from committed transactions by using linked lists in tempdb. A source row can point to a linked list in tempdb, potentially containing multiple consistent versions of the row that were available in previous points in time, from newest to oldest.

The row versioning technology supports different aspects of the product, including: two new isolation levels, which I'll describe here; constructing the *inserted* and *deleted* tables in triggers, as described in the [previous chapter](mk:@MSITStore:C:\Facultad\Taller%202009\DVDTallerOracleySQLServer\Taller%20SQL%20Server%202005\Libros%20Adicionales\Inside%20Microsoft%20SQL%20Server%202005%20-%20T-SQL%20Programming%20(2006).chm::/14325/BBL0063.html#683); online index operations; and MARS.

The two new isolation levels that rely on row versioning are *snapshot* and *read committed* *snapshot*. In both isolation levels, a process does not request shared locks when reading data and is never in conflict with other processes modifying data. When reading data, if a requested row is locked, SQL Server uses the row versioning store to return an older consistent state of the row. Both of the snapshot-related isolation levels provide an optimistic concurrency model.

The following sections describe the two new isolation levels.

**Snapshot**

When a process reads data during a transaction running at the snapshot isolation level, the process will get the latest consistent version of the data that was available when the *transaction* started. A transaction is technically considered to have started when the first statement within the transaction is issued. Whenever a transaction modifies a row while at least one other transaction is running at the snapshot isolation level, SQL Server needs to store a consistent version of the row before the modification, regardless of whether the modifying transaction is running at the snapshot isolation or not. While the transaction that modified the data is open, another process working under snapshot isolation might request that older consistent version of the row.

Working with snapshot isolation has a performance impact in terms of transactions that modify data even when they're not working at snapshot level. Therefore, SQL Server requires you to turn on a database option to allow working with snapshot isolation in the database:

ALTER DATABASE testdb SET ALLOW\_SNAPSHOT\_ISOLATION ON;

If this option is turned off, snapshot isolation will not be allowed in the database and row versions will not be recorded in tempdb for snapshot isolation purposes.

To demonstrate working with snapshot isolation, run the following code from connection 1 (making sure you first set the database option just shown):

SET NOCOUNT ON;

USE testdb;

GO

BEGIN TRAN

UPDATE dbo.T1 SET col2 = 'Version 2' WHERE keycol = 2;

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

The value of *col2* is changed from *'Version 1'* to *'Version 2'*. Notice that I did not request to change the session's isolation level to snapshot. Rather, the session works in the default read committed isolation level. However, because you turned on the database option that allows snapshot isolation, this transaction had to store the state of the row before the change (*col2 = 'Version 1'*) in tempdb.

You can examine the row versions that SQL Server currently maintains in the version store by querying the sys.dm\_tran\_version\_store view:

SELECT \* FROM sys.dm\_tran\_version\_store;

Currently, you will find one row in the version store.

Next run the following code in connection 2, which sets the session's isolation level to snapshot, opens a transaction, and reads the contents of T1:

SET NOCOUNT ON;

USE testdb;

GO

SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

BEGIN TRAN

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

You will get the output *'Version 1'*, which was the most recent consistent state of the data when the transaction started (that is, when the first statement in the transaction was issued). Of course, SQL Server acquired that version of the row from the version store.

Now commit the transaction in connection 1:

COMMIT TRAN

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

The current committed state of the value is now *'Version 2'*. However, the snapshot isolation level is still in effect for the open transaction in connection 2, and remember that when reading data you're supposed to get the latest consistent version when the transaction started. Therefore, the row cannot be removed yet from the version store.

Issue the following query in connection 2:

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

You will still get the value *'Version 1'*.

Note that if another process opens a transaction and modifies data, another version of the row (the one with the value *'Version 2'*) will be added to the linked list in tempdb in front of the existing one. When connection 1 reads the data again, it will end up traversing a longer linked list. This means that the longer transactions working under snapshot isolation remain open, the linked lists in tempdb grow longer, and readers end up traversing longer linked lists.

A cleanup process runs about every minute to remove unneeded row versions from the linked lists. However, it will only remove a contiguous section within the linked list starting at the tail of the list (that is, the oldest version). This means that long-running transactions might prevent the cleaning of intermediate versions following the oldest one even if they are no longer needed. This cleanup architecture is similar to the cleanup architecture of records in the transaction log.

Bearing the row versioning architecture in mind, you should try to minimize the length of your transaction when working with the snapshot isolation level. Also, you should understand that the snapshot isolation level is not suitable to all environments, but only to environments that mostly read and occasionally modify data.

At this point, commit the transaction in connection 2 and reread the data:

COMMIT TRAN

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

You will get the latest committed value *'Version 2'*.

**Conflict Detection** Snapshot isolation also provides update-conflict detection capabilities. Remember that snapshot isolation allows you to work in an optimistic concurrency model. When you read data, you don't acquire any locks. You might want to access data and perform calculations based on the data you accessed for a later update in the same transaction. If in between the time you first accessed the data and the time you attempted to modify it another process modified that data, SQL Server will detect the update conflict and abort your transaction. If appropriate, you can reissue the transaction, which will rely on the new state of the data.

As an example of update-conflict detection in action, issue the following code in connection 1:

-- Connection 1, Step 1

SET NOCOUNT ON;

USE testdb;

GO

SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

BEGIN TRAN

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

Under the snapshot isolation level, you opened a transaction and queried the data, getting back the value *'Version 2'*. Suppose that you now perform calculations based on the input and then want to modify the data. You issue the following code (still in connection 1), changing the value to *'Version 3'* and committing the transaction:

-- Connection 1, Step 2

UPDATE dbo.T1 SET col2 = 'Version 3' WHERE keycol = 2;

COMMIT

The code completed successfully because there was no update conflict—no other process modified the data in between the time you read it and updated it.

Now, still in connection 1, open a new transaction and query the data:

-- Connection 1, Step 3

BEGIN TRAN

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

You will get back the value *'Version 3'*. In connection 2, update this value, changing it to *'Version 4'*:

-- Connection 2, Step 1

SET NOCOUNT ON;

USE testdb;

GO

UPDATE dbo.T1 SET col2 = 'Version 4' WHERE keycol = 2;

Back in connection 1, imagine that you have used the *col2* value in a calculation and determined that you should update the value to *'Version 5'*:

-- Connection 1, Step 4

UPDATE dbo.T1 SET col2 = 'Version 5' WHERE keycol = 2;

SQL Server detects that someone modified the data in between your read and write, terminates your transaction, and produces the following error, which notifies you of the update conflict:

Msg 3960, Level 16, State 2, Line 1

Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot

isolation to access table 'dbo.T1' directly or indirectly in database 'testdb' to update,

delete, or insert the row that has been modified or deleted by another transaction. Retry

the transaction or change the isolation level for the update/delete statement.

You can write exception-handling code that will reissue the transaction in the event it fails after an update conflict. In the [next chapter](mk:@MSITStore:C:\Facultad\Taller%202009\DVDTallerOracleySQLServer\Taller%20SQL%20Server%202005\Libros%20Adicionales\Inside%20Microsoft%20SQL%20Server%202005%20-%20T-SQL%20Programming%20(2006).chm::/14325/BBL0076.html#832), you can find coverage of exception handling with examples of the treatment of update conflicts.

Note that snapshot isolation is not suitable for modification-intensive environments with the potential for many update conflicts. Rather, it is recommended for environments that mainly read data, with occasional modifications and infrequent update conflicts.

For cleanup, change the value of *T1.col2* in the row where *keycol = 2* back to *'Version 1'*:

UPDATE dbo.T1 SET col2 = 'Version 1' WHERE keycol = 2;

At this point, close all connections.

**Read Committed Snapshot**

"[Read committed snapshot](mk:@MSITStore:C:\Facultad\Taller%202009\DVDTallerOracleySQLServer\Taller%20SQL%20Server%202005\Libros%20Adicionales\Inside%20Microsoft%20SQL%20Server%202005%20-%20T-SQL%20Programming%20(2006).chm::/14325/BBL0072.html#ch09lev3sec2)" is a new implementation of read committed isolation that can be used on a per-database basis. The database option READ\_COMMITTED\_SNAPSHOT controls which version of read committed is used. As soon as you set this database option to ON, all sessions working at the read committed level (default) will actually be working at the read committed snapshot level. This is a database global behavioral change that you get just by setting the database option to ON.

The read committed snapshot isolation level differs from the snapshot isolation level in two ways. The first difference is that readers get the latest consistent version of data that was available when the *statement* started, as opposed to when the transaction started. Also, this isolation level does not detect update conflicts.

This isolation level is especially useful for applications that you migrate from platforms that support obtaining earlier consistent versions of data, such as when an application migrates from Oracle to SQL Server.

The version store maintained for this isolation level is typically smaller than for snapshot isolation, and the linked lists of row versions do not grow long so easily because row versions can be removed from the linked lists sooner.

To look at the read committed snapshot isolation level, first turn on the database option in the testdb database:

ALTER DATABASE testdb SET READ\_COMMITTED\_SNAPSHOT ON;

Open two new connections, and issue the following code from connection 1:

SET NOCOUNT ON;

USE testdb;

GO

BEGIN TRAN

UPDATE dbo.T1 SET col2 = 'Version 2' WHERE keycol = 2;

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

The default isolation level is read committed as always, but that level is now functioning in its new (snapshot) implementation. The code opened a new transaction, modified the value of *T1.col2* in the row where *keycol = 2* from *'Version 1'* to *'Version 2'*, and queried it. Before the value was modified, SQL Server stored the row with *'Version 1'* in case another session later requests it.

Query the data in connection 2:

SET NOCOUNT ON;

USE testdb;

GO

BEGIN TRAN

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

You will get back the value *'Version 1'*. This is the latest consistent state of the data that was available when the SELECT statement started.

Commit the transaction in connection 1:

COMMIT TRAN

At this point, the latest consistent version of the data is *'Version 2'*. In connection 2, issue the following query:

SELECT col2 FROM dbo.T1 WHERE keycol = 2;

COMMIT TRAN

You will get back the value *'Version 2'*. Had you worked with the snapshot isolation level, you would have gotten back *'Version 1'*.

|  |  |  |
| --- | --- | --- |
|  | Tip | If you want to request a shared lock while working with the read committed snapshot isolation level, you can do so by specifying the READCOMMITTEDLOCK table hint. Using this hint, a reader will be blocked when requesting a shared lock on a resource that is under modification (exclusively locked). This hint allows readers to work at a similar level to read committed while at read committed snapshot level. |

At this point, close all connections.

Restore the testdb database to its default settings:

ALTER DATABASE testdb SET ALLOW\_SNAPSHOT\_ISOLATION OFF;

ALTER DATABASE testdb SET READ\_COMMITTED\_SNAPSHOT OFF;

For your convenience, [Table 9-6](mk:@MSITStore:C:\Facultad\Taller%202009\DVDTallerOracleySQLServer\Taller%20SQL%20Server%202005\Libros%20Adicionales\Inside%20Microsoft%20SQL%20Server%202005%20-%20T-SQL%20Programming%20(2006).chm::/14325/BBL0072.html#ch09table06) provides a summary of all isolation levels, the problems that each allows or prevents, the supported concurrency model, and whether or not the isolation detects update conflicts for you.

| Table 9-6: Summary of Isolation Levels | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Isolation** | **Dirty Reads** | **Lost Updates** | **Nonrepeatable Reads** | **Phantoms** | **Concurrency Model** | **Update Conflict Detection** |
| Read Uncommitted | Yes | Yes | Yes | Yes | Pessimistic | No |
| Read Committed | No | Yes | Yes | Yes | Pessimistic | No |
| Repeatable Read | No | No | No | Yes | Pessimistic | No |
| Serializable | No | No | No | No | Pessimistic | No |
| Snapshot | No | No | No | No | Optimistic | Yes |
| Read Committed Snapshot | No | Yes | Yes | Yes | Optimistic | No |