Transazioni – Locks - Concorrenza

Cosa succede quando due transazioni competono per una risorsa?

Il presente documento costituisce un'introduzione al sistema con cui SQL Server gestisce i lock sulle risorse e quali sono le differenze tra i vari tipi di transazione.

Si darà per scontata la conoscenza di concetti come Record, Pagina, Tabella, Consistenza.

Per "risorsa", nel contesto di un Database, s'intende sostanzialmente un set di dati che può essere costituito da uno o più record in una tabella.
Un set di dati viene identificato da uno statement di Select che può coinvolgere un numero arbitrario e variabile di record (es. Select * From Tab Where Field > Value).
Un set di dati "cambia" se vengono aggiunti, eliminati o modificati i record che lo compongono.

L'accesso ad una risorsa può essere in lettura o in scrittura.
La fase più critica è ovviamente la scrittura perchè il server deve garantire la consistenza dei dati. A questo scopo, le operazioni di modifica avvengono sempre in due tempi:

- preparazione di un batch con le modifiche da effettuare
- salvataggio persistente dei dati (Commit).

Le operazioni di scrittura, a seconda della mole di dati coinvolti, può richiedere anche dei tempi relativamente lunghi (nel contesto di un db, 1 secondo è un tempo lungo).
Per evitare risultati impredicibili durante la modifica di un set di dati, il server gestisce un sistema di locking automatico all'interno di transazioni implicite.
Il comportamento basilare di default, può essere riassunto in:

- una sola connessione può accedere in scrittura al set di dati
- tutte le connessioni possono leggere contemporaneamente i dati committati.

Per garantire ad una connessione l'accesso esclusivo ad una risorsa, il server imposta una sorta di flag di "lock". I Lock possono essere in sola scrittura o in lettura/scrittura.
Possono essere impostati a livello di record, pagina, tabella o DB.
Il livello di lock viene selezionato dal server a seconda della mole di dati coinvolta: se si lavora su pochi record sarà a livello di record, ma oltre un certo numero verrà "scalato" a livello di pagina o di tabella. I lock a livello db vengono usati per operazioni di manutenzione massive.

E' importante rilevare che, in generale, l'esigenza di garantire la consistenza dei dati contrasta con l'esigenza di avere elevate prestazioni, perchè maggiori sono la frequenza, la durata e la portata dei lock imposti durante le operazioni di I/O, maggiori saranno i tempi di attesa necessari per portare a termine tali operazioni.
In pratica i lock riducono la capacità di parallelismo e quindi le prestazioni del server.

Il comportamento di default del server può essere modificato e la politica di locking va decisa a seconda del tipo di utilizzo "tipico" del DB e comunque è bene adottare sempre le soluzioni che minimizzano i lock.

Per variare il comportamento di default del Server è possibile aprire esplicitamente una transazione impostando il livello di isolamento e agire sui singoli statement con gli Hints. Nel primo caso si imposta il livello di "protezione" dei dati su cui si vuole operare, mentre con gli Hints si possono forzare lock aggiuntivi non previsti dal livello di isolamento o ignorare (in lettura) i lock imposti da altre transazioni.

I livelli di isolamento utilizzabili aprendo le transazioni sono i seguenti:

- READ UNCOMMITTED:
  - vede anche le modifiche non committate (che potrebbero essere annullate)
  - Consente ad altre transazioni di modificare i dati letti

- READ COMMITTED (Default):
  - vede solo i dati committati da altre transazioni
  - Consente ad altre transazioni di modificare i dati letti

- REPEATABLE READ:
  - vede solo i dati committati da altre transazioni
  - impedisce che altri modifichino i dati letti (imposta i lock in lettura)

- SERIALIZABLE
  - vede solo i dati committati da altre transazioni
  - impedisce che altre transazioni modifichino i dati letti (imposta i lock in lettura)
  - impedisce che altre transazioni inseriscano nuovi record nel set di dati letto

Le transazioni gestite implicitamente dal server sono limitate ad un singolo statement SQL. Per definizione ogni statement è sempre "atomico" cioè, dal punto di vista dell'utente, non è splittabile in un insieme di operazioni più semplici.
Dal punto di vista del server, invece, ogni statement comporta un certo numero di operazioni di lettura (e scrittura) perchè ad esempio è necessario recurere dati da più tabelle in join e in ragione di questo vengono aperte transazioni implicite.

Le transazioni esplicite vengono utilizzate per rendere "atomiche" operazioni che coinvolgono più statement, in modo che altre transazioni non possano interferire modificando in modo incontrollato i dati coinvolti.
A seconda della "criticità" dell'operazione, verrà selezionato un livello di isolamento adeguato. Nell'ottica di ridurre al minimo i lock, in una transazione, è possibile impostare globalmente un livello non molto restrittivo e impostare lock specifici aggiuntivi.

Non esiste una politica di impostazione dei lock corretta (o scorretta) a priori, ma occorre sempre decidere una corretta logica di gestione della polica che è stata adottata.

L'operazione più semplice che illustra in modo abbastanza completo la problematica legata alla gestione della concorrenza è la seguente:

- Leggere il valore attuale di un campo
- Calcolare un nuovo valore in base a quello attuale
- Scrivere il nuovo valore nello stesso campo

Quest'operazione, se eseguita da una sola transazione non comporta alcun problema, ma se le transazioni diventano due e vengono eseguite in parallelo i problemi legati alla concorrenza diventano evidenti.

Di seguito riporto alcuni esempi di ciò che può succedere durante l'esecuzione della suddetta operazione (sono presenti solo le combinazioni più interessanti).

Ogni esempio consiste in un elenco degli step eseguiti da due transazioni "interlacciate".

L' obbiettivo auspicabile è che ogni operazione eseguita in una transazione non venga influenzata dalle operazioni eseguite dalle altre transazioni.

E' interessante notare che, usando certe modalità perfettamente lecite di gestione delle transazioni, il risultato finale può cambiare in modo imprevedibile perchè non è prevedibile la sequenza temporale degli step eseguiti.

Use Case:

Tran1: Read Field F on Record R, Update Field F on Record R
Tran2: Read Field F on Record R, Update Field F on Record R

Tran1

```
Declare @TypeCD nvarchar(255)
Begin Tran
Select @TypeCD=TypeCD from MMTypes where TypePK = 0
Update MMTypes Set TypeCD = @TypeCD + N'-TR1' where TypePK = 0
Commit Tran -- or Rollback Tran
```

Tran2

```
Declare @TypeCD nvarchar(255)
Begin Tran
Select @TypeCD=TypeCD from MMTypes where TypePK = 0
Update MMTypes Set TypeCD = @TypeCD + N'-TR2' where TypePK = 0
Commit Tran -- or Rollback Tran
```

Si vorrebbe il seguente comportamento:

- TypeCD = "CD"
- Viene eseguita Tran1
- TypeCD = "CD-TR1"
- Viene eseguita Tran2
- TypeCD = "CD-TR1-TR2"

Note:

- il caso esaminato contempla 2 transazioni, ma potrebbero essere molte
- l'operazione considerata è composta da due statement ma potrebbero essere molti
- L'operazione lavora su un solo record, ma potrebbero essere molti
- ....

Transazioni con livello di isolamento [Read Committed]

	Tran1	***Tran2***
1	Begin Tran	
2		Begin Tran
3	Read TypeCD	
4		Read TypeCD
5	Update TypeCD (*Set a Lock*)	
6		Try to Update TypeCD
		Waiting on Tran1 lock
7	Commit Tran (*Remove the Lock*)	
		Set TypeCD
8		Commit Tran

Scorretto! – il valore di TypeCD è: prima "CD", dopo "CD-TR2".

	Tran1	***Tran2***
1	Begin Tran	
2		Begin Tran
3	Read TypeCD	
4	Update TypeCD (*Set a Lock*)	
5		Read TypeCD
		Waiting on Tran1 lock
6	Commit Tran (*Remove the Lock*)	
		Get TypeCD
7		Update TypeCD
8		Commit Tran

Corretto! – il valore di TypeCD è: prima "CD", dopo "CD-TR1-TR2".

Transazioni con livello di isolamento [Read Committed] e Hint With(NoLock)

L'Hint viene posizionato sulla Select:

```
Select @TypeCD=TypeCD from MMTypes with(nolock) where TypePK = 0
```

	Tran1	***Tran2***
0	Begin Tran	
1		Begin Tran
2	Read TypeCD	
3	Update TypeCD (*Set a Lock*)	
4		Read TypeCD (Ignore Lock on Tran1)
5		Update TypeCD
		Waiting on Tran1 lock
7	Rollback Tran (*Remove the Lock*)	
		Set TypeCD
8		Commit Tran

Scorretto! - TypeCD Value: before "CD", after "CD-TR1-TR2".
La Tran1 è andata in rollback, ma Tran2 aveva letto il valore non committato.

	Tran1	***Tran2***
0	Begin Tran	
1		Begin Tran
2	Read TypeCD	
3	Update TypeCD (*Set a Lock*)	
4		Read TypeCD (Ignore Lock on Tran1)
5		Update TypeCD
		Waiting on Tran1 lock
7	Commit Tran (*Remove the Lock*)	
		Set TypeCD
8		Commit Tran

Corretto! - TypeCD Value: before "CD", after "CD-TR1-TR2".

	Tran1	***Tran2***
0	Begin Tran	
1		Begin Tran
2	Read TypeCD	
3		Read TypeCD
4	Update TypeCD (*Set a Lock*)	
5		Update TypeCD
		Waiting on Tran1 lock
7	Commit Tran (*Remove the Lock*)	
		Set TypeCD
8		Commit Tran

Scorretto! - TypeCD Value: before "CD", after "CD-TR2".

Transazioni con livello di isolamento Repeatable Read (o Serializable)

	Tran1	***Tran2***
1	Begin Tran	
2		Begin Tran
3	Read TypeCD (*Set a Lock*)	
4		Read TypeCD (*Set a Lock*)
5	Update TypeCD	
	Waiting on Tran2 lock	
6		Update TypeCD
		Waiting on Tran1 lock -> DeadLocked
7	Commit Tran	
8		***Connection Lost***

Scorretto! – il valore di TypeCD è: prima "CD", dopo "CD-TR1".

Transazioni con livello di isolamento Repeatable Read e Hint With(NoLock)

L'Hint viene posizionato sulla Select:

```
Select @TypeCD=TypeCD from MMTypes with(nolock) where TypePK = 0
```

	Tran1	***Tran2***
1	Begin Tran	
2		Begin Tran
3	Read TypeCD (*Set a Lock*)	
4		Read TypeCD (*Ignore Lock on Tran1*)
5	Update TypeCD (*Set a Lock*)	
6		Update TypeCD
		Waiting on Tran1 lock
7	Commit Tran (*Remove the Lock*)	
8		Commit Tran

Scorretto! - TypeCD Value: before "CD", after "CD-TR2".

	Tran1	***Tran2***
1	Begin Tran	
2		Begin Tran
3	Read TypeCD (*Set a Lock*)	
4	Update TypeCD (*Set a Lock*)	
5		Read TypeCD (Ignore Lock on Tran1)
6		Update TypeCD
		Waiting on Tran1 lock
7	Commit Tran (*Remove the Lock*)	
8		Commit Tran (overriding Tran1 update)

Corretto! - TypeCD Value: before "CD", after "CD-TR1-TR2".

Dagli esempi riportati, si nota che in un contesto ad alta concorrenza la probabilità di ottenere un risultato scorretto, per operazioni di questo tipo, è piuttosto elevata.

Per fortuna le operazioni in cui la concorrenza diventa problematica, tipicamente, non sono molto frequenti ed essendo solitamente in numero finito possono essere gestite con approcci particolareggiati.

Esistono alcune tecniche sviluppate durante la pratica quotidiana che sono applicabili in molti casi e rendono "deterministico" il comportamento di SQL Server.
Di seguito riporto alcuni esempi validi per il caso esaminato, supponendo per semplicità che l'operazione venga sempre eseguita utilizzando la stessa Stored Procedure.

1) Ridurre l'operazione ad un unico statement atomico:

```
Update MMTypes Set TypeCD = TypeCD + N'-TR1' where TypePK = 0
```

Nel caso fosse anche necessario recuperare il valore precedente di TypeCD, possiamo usare:

```
Declare @TypeCD nvarchar(255)
Update MMTypes Set @TypeCD = TypeCD,
                   TypeCD = TypeCD + N'-TR1'
      Where TypePK = 0
```

2) Inserire il seguente statement subito dopo la "Begin Tran":

```
Update MMTypes Set TypeCD = TypeCD where TypePK = 0
```

Tale statement, da un punto di vista logico non ha alcun effetto, ma forza l'impostazione di un lock esclusivo sul record da modificare
La prima transazione che esegue la SP assume il controllo e un'altra che arrivi subito dopo, sarà costretta ad attendere il completamento della prima.

In un caso più complesso in cui sia necessario operare su un set di dati esteso magari non determinabile a priori, si potrebbe usare uno statement di Update simile ma eseguito su una tabella d'appoggio creata allo scopo di "semaforizzare" le operazioni critiche:

```
Update MMOpeLocker Set LockField = 0 where Operation = N'UpdateTypeCD'
```

Questa tecnica assicura una serializzazione deterministica delle transazioni che eseguono l'operazione, ma non richiede alcun tipo di "reset" del lock (quindi non si avranno mai lock che "restano appesi").

Livello di Isolamento Snapshot

Questo livello di isolamanto introdotto da SQL Server 2005, merita un discorso a parte perchè presuppone una progettazione del database e delle logiche di gestione dei dati apposite.

In estrema sintesi, quando il DB lavora in modalità SNAPSHOT, le operazioni di lettura non vengono bloccate dai lock imposti dalle operazioni di scrittura, e le operazioni di scrittura non vengono bloccate dalle operazioni di lettura.

In un certo senso, l'effetto è simile a quello che si ottiene utilizzando il livello READ UNCOMMITTED o gli hint With(NoLock), ma con il livello SNAPSHOT invece di "vedere" i dati non ancora committati, si vedono i dati originali (prima della modifica non committata).
Per ottenere questo risultato SQL Server è costretto a salvare una copia dei dati che verranno modificati nel TempDB, in modo da poterli restituire ad un eventuale lettore.
Al termine delle operazioni di modifica, gli snapshot creati in TempDB dovranno essere eliminati. Queste operazioni di lettura/scrittura su TempDB ovviamente hanno un costo sia in termini di performance che di spazio occupato.

In caso di concorrenza sulle stesse risorse, con il livello SNAPSHOT si avranno a seconda del modo in cui si "intersecano" le operazioni, dei comportamenti corretti o scorreti come per tutti gli altri livelli (vedi sopra).
Fortunatamente le tecniche per evitare i conflitti descritte nel paragrafo precedente, sono utilizzabili anche con il livello SNAPSHOT.

Quando si adotta la modalità SNAPSHOT come default per il DB, si dovrebbe implementare una logica basata su "Lock Ottimistico" per tutte le operazioni di modifica dei dati. Tale logica si può sintetizzare con una sequenza simile alla seguente:

1. Aprire la transazione
2. Leggere i Dati necessari
3. Apportare tutte le modifiche richieste
4. Verificare che i dati iniziali non siano stati modificati da un'altra transazione
5. Eseguire il Commit (o il Rollback se i dati iniziali sono stati modificati)

Di seguito riporto un documento che riassume le informazioni sul livello SNAPSHOT.
Le informazioni sono ricavate da http://www.informit.com/articles/article.aspx?p=357098

SQL Server 2005's Snapshot Isolation

In this article, I'll explain the new Snapshot Isolation feature. It may not have the most obvious payoff, but implementing Snapshot Isolation on a suitable database has the potential to eliminate data contention and reduce deadlocks, lock contentions, and session waits.

If you answer yes to one or more of the following questions, your database is a possible candidate for Snapshot Isolation:

- Do you have long-running batch jobs that run while your user queries are running?

- Do you run statistics queries fairly frequently?
  Hint: Typical statistic queries have Sum, Count, Average and similar keywords.
  They also touch a larger number of rows than do normal transactional queries.

- Do you have stored procedures or application logic that hold transactions open for longer periods while they are working? "Longer periods" is in database terms; it could be anywhere from a few seconds to a few minutes, maybe more.

I will not tax your patience with more questions.
The general characteristic of these database queries is that they need to hold onto locks for longer period of time. As such, they block other readers (select queries) and writers (insert, update, and delete queries).

Snapshot Isolation provides a mechanism that eliminates the blocking of other "readers." It is akin to optimistic locking, in which you make a copy of the data (typically at the front end) and make changes to your copy of the data.
When you are ready to save the changes back to the database, you (typically) check to see whether the original data has changed while you were working with it and decide whether you want to save.

Snapshot Isolation (also called *Row Versioning*) is optimistic locking, but it is completely transparent to your users and is handled by the database. The database keeps a copy of the original data while you are changing it, and serves up the original data to anybody who wants to read it in the interim.

NOTE
Snapshot Isolation is also called Row Versioning because SQL Server keeps "versions" of rows that are being changed; that is, the original version and the version being changed.

Following sections will explain the differences between SQL Server 2000 and SQL Server 2005.

SQL Server 2000 Behavior

To see what this feature accomplishes, let's first examine how we were affected by its lack in previous SQL Server versions. We use the omnipresent PUBS database and the authors table. The table has au_id as the primary key and an index on the au_lname, au_fname columns.

Let's now run some queries in SQL Server 2000.

Session 1
Open a Query Analyzer window and run these queries.

```
USE PUBS

BEGIN TRANSACTION

UPDATE authors
 SET phone = '111 111-1111'
 WHERE au_id = '172-32-1176'
```

Session 2
Open another Query Analyzer window and run these queries.

```
USE PUBS

SELECT * FROM authors
```

The SELECT statement cannot read the row updated in Session 1, so it blocks (that is, it waits until Session 1 commits or aborts).
Note that there are some variations to this behavior. Let's look at it right away; you may be puzzled if you did something slightly different and got a different result.

Run this query in the second session window:

```
SELECT *
 FROM authors
 WHERE au_id <> '172-32-1176'
```

This statement succeeds because there is an index on the au_id column. The execution path completely avoids the row locked by Session 1.

Run this query, also in the second session window:

```
SELECT *
 FROM authors
 WHERE phone = '111 111-1111'
```

This statement fails. The execution plan includes a table or clustered index scan because there is no index on the phone column. A table scan requires that every row be read, and the query fails.

Another scenario, in which even queries with indexes will fail, is when you update a large number of rows (more than 3,000 in SQL Server 2000), and the engine escalates the lock to a table level lock.

SQL Server 2005 Behavior

Now for the difference in SQL Server 2005. With Snapshot Isolation enabled in SQL Server 2005, all the previous Session 2 queries succeed because SQL Server 2005 maintains the old state (or version) of the changed row until the new row is committed or rolled back. All the select queries will read the older version until the updated data is committed or rolled back.

The old copies of the updated row or rows are stored in the tempdb database. If the updated row is committed, the row version in the tempdb is deleted. If the update is aborted, the old values for the row(s) are restored in the original table, and the copy in the tempdb is deleted.

When Session 1 updates the row, the old version of the row is copied to the tempdb:

Data in pubs

au_id	**phone**
724-08-9931	415 843-2991
172-32-176	**111 111-1111**

Data in tempdb

au_id	**phone**
172-32-176	**408 496-7223**

When Session 2 tries to read the row, the server will read the old version from tempdb:
Data returned from pubs (the highlighted data is read from tempdb)

au_id	**phone**
724-08-9931	415 843-2991
172-32-176	408 496-7223

Here are the updated queries for use with SQL Server 2005, so that the select queries will not block.

NOTE
You need to enable the Snapshot Isolation feature for the pubs database for this to work. See the section "Enabling Snapshot Isolation in SQL Server 2005," which discusses this in detail.
Run this query to enable Snapshot Isolation:

```
ALTER DATABASE pubs SET ALLOW_SNAPSHOT_ISOLATION ON
```

Optionally, to enable the Row Committed Isolation level, run this command:

```
ALTER DATABASE pubs SET READ_COMMITTED_SNAPSHOT ON
```

If you choose the ALLOW_SNAPSHOT_ISOLATION option, each transaction can decide whether it wants to enable the Snapshot Isolation feature.

On the other hand, if you choose the READ_COMMITTED_SNAPSHOT isolation mode, Snapshot Isolation is enabled for all transactions automatically.

Now, let's run our modified queries to see how the feature works:

Session 1

```
USE PUBS
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
BEGIN TRANSACTION
     UPDATE authors
        SET phone = '111 111-1111'
      WHERE au_id = '172-32-1176'
```

Session 2

```
USE PUBS
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
SELECT * FROM authors
```

The SET TRANSACTION ISOLATION LEVEL SNAPSHOT statement is optional if you enable the READ COMMITTED isolation. So your existing applications can automatically make use of this feature without any change to stored procedures or any embedded SQL.

Where Would This Feature Be Useful?

Because Snapshot Isolation is not turned on by default in SQL Server 2005, you need to evaluate and decide whether it is appropriate for you. If your application has any of these characteristics, you might consider enabling Snapshot Isolation.

Applications that Continuously Read the Recently Updated Data

Imagine that you are an e-business CEO for a company selling millions of dollars of goods over the Internet. The sales records are continuously inserted and updated into the database. As a CEO, you want a statistics page that displays various sales statistics, such as sales in each category, sales amount in dollars, and so on. Furthermore, these statistics should update every two seconds.

Without Snapshot Isolation, the statistic queries would create contention on the inserted/updated rows by trying to read them even as they are being committed into the database.

Long-Running Transactions

Consider a database where the transactions are long-running. This could be for various reasons, including the following:

An OLTP application that has to update the ERP system before a transaction is committed

A mission-critical application that needs to replicate copies of data to geologically distributed databases before the transaction is committed

Without Snapshot Isolation, your read queries would be blocked until the long transactions complete.

More Useful Qualities

In general, Snapshot Isolation is a nice feature if there were no costs involved in implementing it. The feature guarantees that readers do not block writers, and writers do not block readers. Who wouldn't want such a system? Unless two sessions want to update the same piece of data, there is no contention ever! It would even protect the database against a badly written application that keeps transactions open for a long time.

In Oracle, this feature (called UNDO) is enabled by default. In Oracle 9i and 10g, the UNDO data is used for some very nice side effects, such as looking at previous changes to a row at a later time. Now that SQL Server 2005 implements this feature, it joins Oracle—leaving behind other leading relational database servers (that is, DB2, Sybase, and Informix).

Oracle implements Row Versioning (again, called UNDO) as a standard feature, but you have to explicitly enable it in SQL Server 2005. I have seen articles mentioning the option to enable/disable Snapshot Isolation as a feature, wherein you enable it only on certain databases because there is an overhead in doing so. I have not heard of any concrete percentage by which this feature would adversely affect your database. But if your database has a 5–10 percent update and remaining read activity, you should not have any problem. Run some tests on your database with and without Snapshot Isolation turned on to see how much impact it has. Your database is your baby; make sure that it does not cry at night!

The benefits of Snapshot Isolation come at a price. The overhead involved in enabling Snapshot Isolation feature includes the following:

- More space for tempdb (you can test it by updating a very large table and see how tempdb grows)
- More processing while updating (work is involved in making copies of older versions in tempdb)
- More maintenance (the server has to keep track of data copies in tempdb and delete it when it is no longer needed)
- More processing while reading (the read process has to look for copies of data in tempdb)

Where can you afford to turn it on? This is something you have to decide on your own, on a case-by-case basis, after performance testing. However, the general rule of thumb is to use snapshotting in these types of applications:

- Applications with fewer updates and lot of reads
- Applications that need to collect frequent statistics on recent data
- Applications with hotspots—where updates and reads are concentrated in a few database pages

A note of caution, however. If you have an application that relies on being blocked when data is being updated, you should be careful. The application will no longer wait for the data to be committed; it will read the previously committed view of the data. But again, if the application really *does* need such behavior, you should re-examine it and use serializable or repeatable read transactions if necessary.

Also, Snapshot Isolation does not prevent two or more writers who are trying to update the same piece of data from blocking. It is only the *readers* who benefit from this feature. Multiple writers still have to wait until the writer(s) higher up in the queue finish their task. This is different from a front end-based "optimistic locking;" for example, using the ADO "batch optimistic" locking or an ADO.NET dataset and data adapter-based update.

Enabling Snapshot Isolation in SQL Server 2005

Enabling Snapshot Isolation is a two-step process. First, the DBA has to enable it per database. Then the queries have to execute in a session enabled for Snapshot Isolation.

To enable Snapshot Isolation on a database, use this:

```
ALTER DATABASE pubs SET ALLOW_SNAPSHOT_ISOLATION ON
```

or this:

```
ALTER DATABASE pubs SET READ_COMMITTED_SNAPSHOT ON
```

Here's what to type to enable the Snapshot Isolation in a session (this is needed only if you use the ALLOW_SNAPSHOT_ISOLATION option rather than READ_COMMITTED_SNAPSHOT):

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
```

Summary

The Snapshot Isolation feature is a cool new feature introduced in SQL Server 2005. Implementing it in your database might provide improvements in data availability. It reduces data contention and guarantees that writers do not block readers, and vice versa. You need to evaluate whether it is appropriate for your database by comparing the costs versus benefits.