

2008

Garantire l'univocità di una colonna ammettendo valori NULL multipli

Le novità di SQL Server 2008

Chi sviluppa in ambito professionale o per puro diletto prima o poi si troverà ad affrontare tematiche nuove ed avvincenti. In questo articolo cercherò di illustrare tre tecniche con i relativi pregi e difetti per permettere con SQL Server di garantire l'univocità di una colonna senza alcuna limitazione sui valori NULL



Sommario

Sommario	2
Problema	3
After trigger	5
Indexed views	7
Filtered Indexes	10
Conclusioni	12
Autore	12

© 2008 .net hell .it

Problema

Oggi vorrei affrontare un argomento che ogni tanto torna alla ribalta sui newsgroup cercando di analizzare gli strumenti offerti da SQL Server per risolvere un problema a prima vista banale, ma che in realtà nasconde alcune insidie.

La domanda suona più o meno così: *“Ho la necessità di definire un vincolo su una colonna che mi garantisca l'univocità dei valori ma al tempo stesso mi permetta di inserire più valori NULL. Come posso fare?”*

Iniziamo preparare un piccolo esempio che utilizzerò come base di partenza per descrivere le varie tecniche.

Come prima cosa lanciamo SQL Server Management Studio, spostiamoci nel *tempdb*, definiamo e popoliamo la tabella *dbo.Students*:

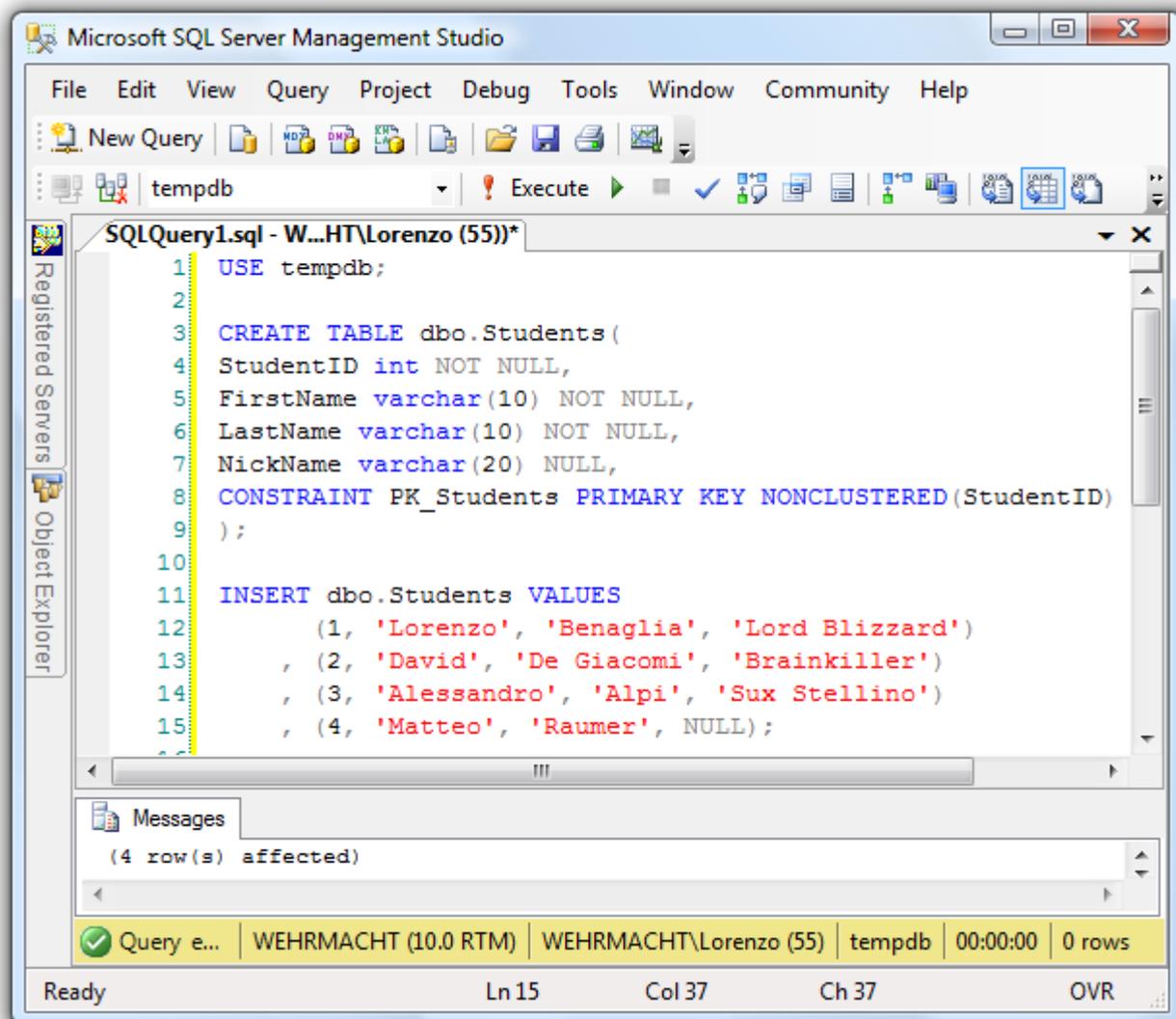


Figura 1

Supponiamo di voler creare un vincolo UNIQUE sulla colonna *NickName* in modo da evitare che due o più studenti abbiano lo stesso soprannome:

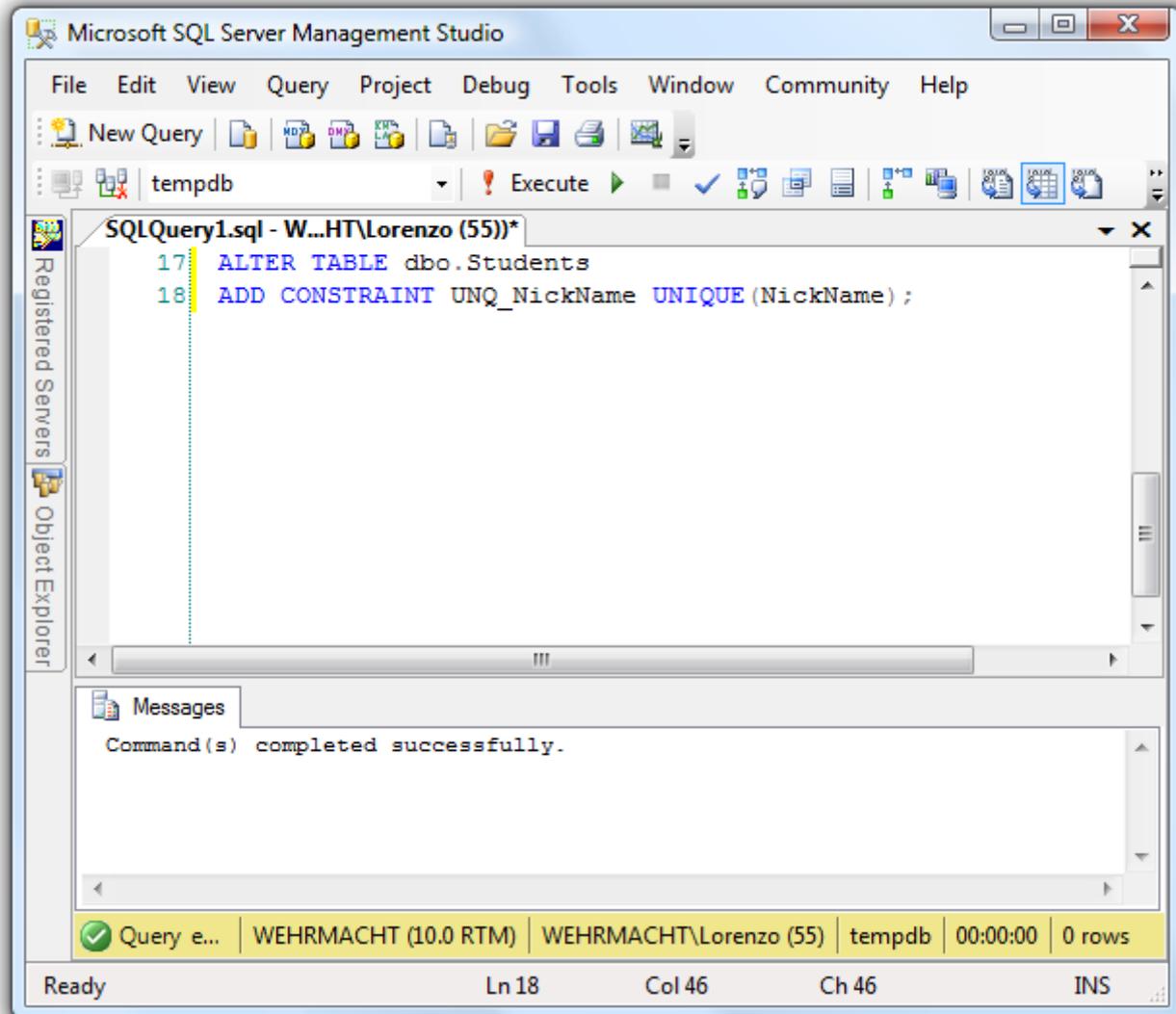


Figura 2

Che cosa succede se inserisco un nuovo studente privo di soprannome? In prima battuta mi aspetterei che l'inserimento vada a buon fine ma visto che esiste già un'altra riga con soprannome nullo, otterrò una violazione del vincolo con il conseguente messaggio d'errore:

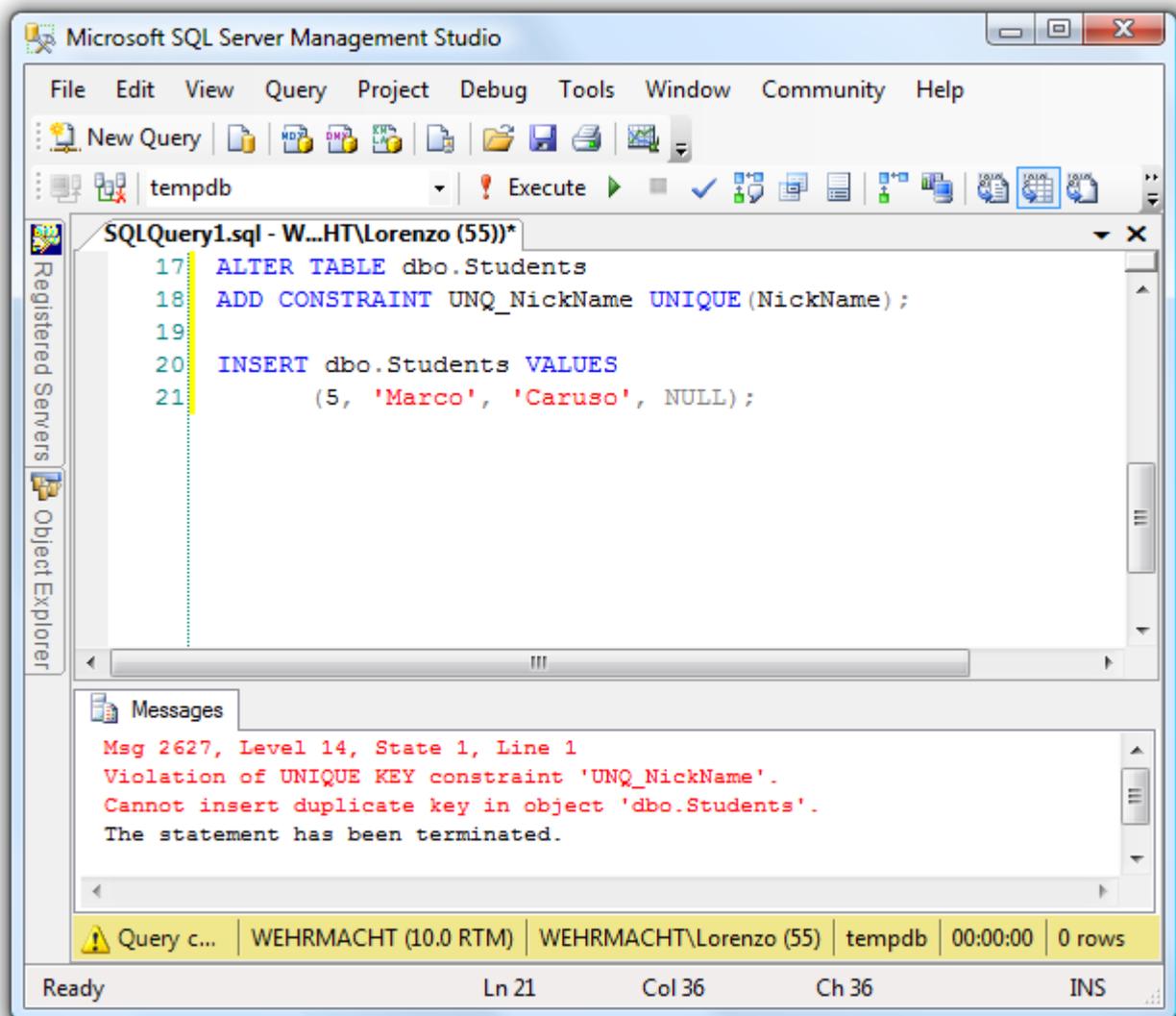


Figura 3

In alternativa alla definizione di un vincolo UNIQUE si potrebbe ricorrere a un indice UNIQUE, ma a differenza di altri DBMS come [Microsoft Access](#), SQL Server non permette di ignorare i valori NULL.

Proviamo a vedere quali possibilità ci rimangono.

After trigger

Probabilmente la prima idea che ci frulla per la testa è di ricorrere a un after trigger che vada a verificare se tra le righe che ci accingiamo a inserire o aggiornare ne esiste qualcuna con un soprannome già presente in tabella: nel caso la ricerca abbia esito positivo, il trigger provvederà a segnalare l'anomalia all'utente annullando la transazione in corso.

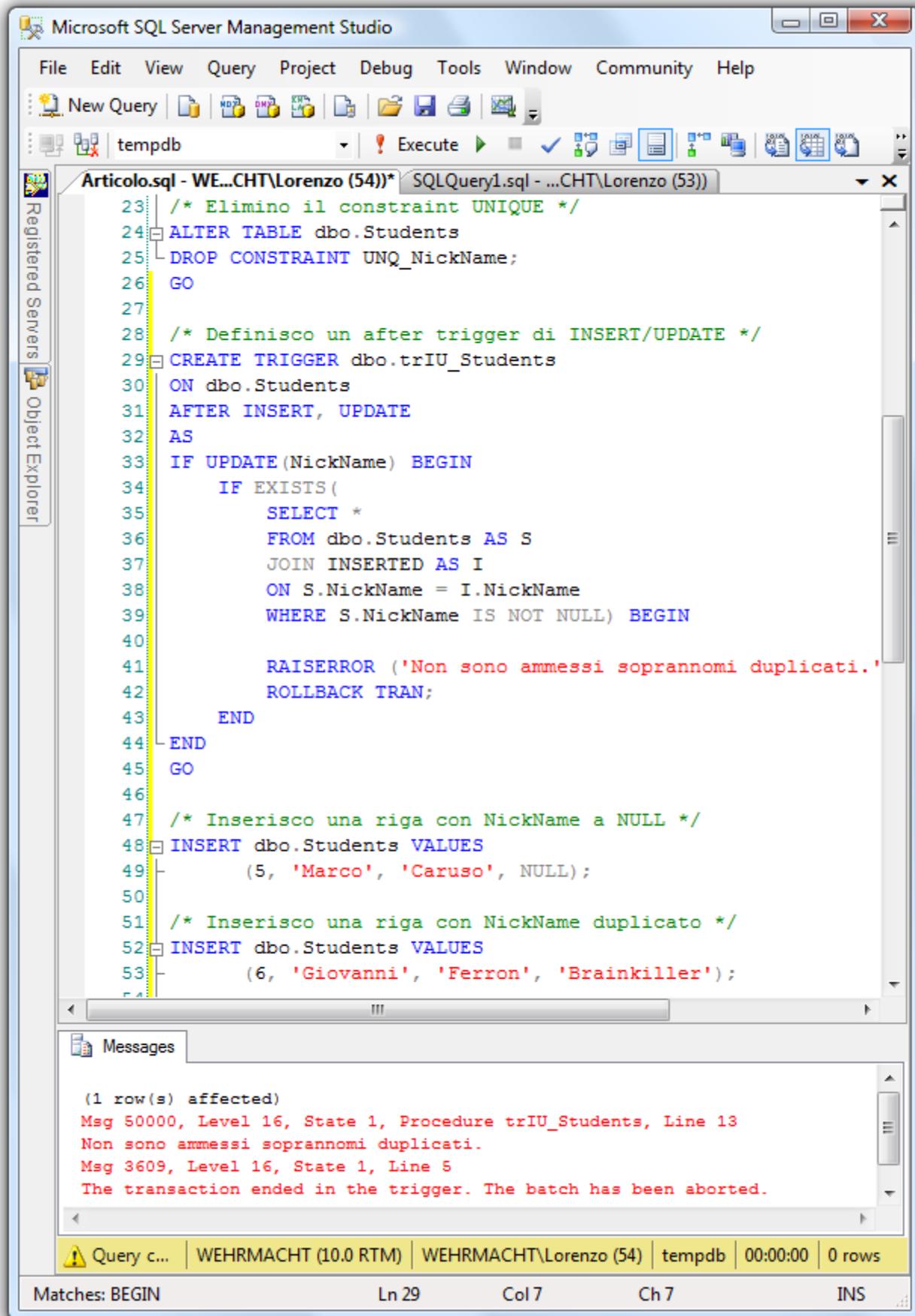


Figura 4

Osservando l'esempio si nota che la riga numero 5 è stata inserita con successo anche se avevamo già uno studente con *NickName* valorizzato a NULL, mentre il tentativo di inserire un soprannome già presente in tabella è fallito miseramente. Lo stesso errore lo avremmo avuto utilizzando un comando di UPDATE che avesse perseguito lo stesso obiettivo.

L'utilizzo di un after trigger ci ha permesso di raggiungere lo scopo, a discapito però delle performance dato che la transazione che l'ha scatenato verrà conclusa solo al termine dell'esecuzione di tutto il codice contenuto nel corpo del trigger stesso.

Proviamo a vedere se esistono soluzioni più eleganti ed efficienti.

Indexed views

Anni fa [Marcello](#) propose un'interessante soluzione basata su una vista indicizzata.

SQL Server con la versione 2000 ha introdotto le viste indicizzate (chiamate anche viste materializzate) nelle quali i dati sono memorizzati in modo permanente nel db alla stessa stregua di una normale tabella mediante la creazione di un indice clustered.

Generalmente le viste indicizzate sono utilizzate laddove siano richiesti calcoli complessi su un elevato numero di righe, come aggregazioni, join o colonne calcolate che causerebbero dei seri problemi di performance nel caso dovessero essere eseguiti ad ogni esecuzione.

[Marcello](#) però propose un modo ingegnoso per risolvere il problema discusso nell'articolo: si limitò a definire una vista che restituisse tutte le occorrenze non NULL della colonna e definendo su di essa un indice clustered. In questo modo saremo in grado di inserire un numero indefinito di righe prive di *NickName*, garantendo al tempo stesso l'univocità del suo valore:

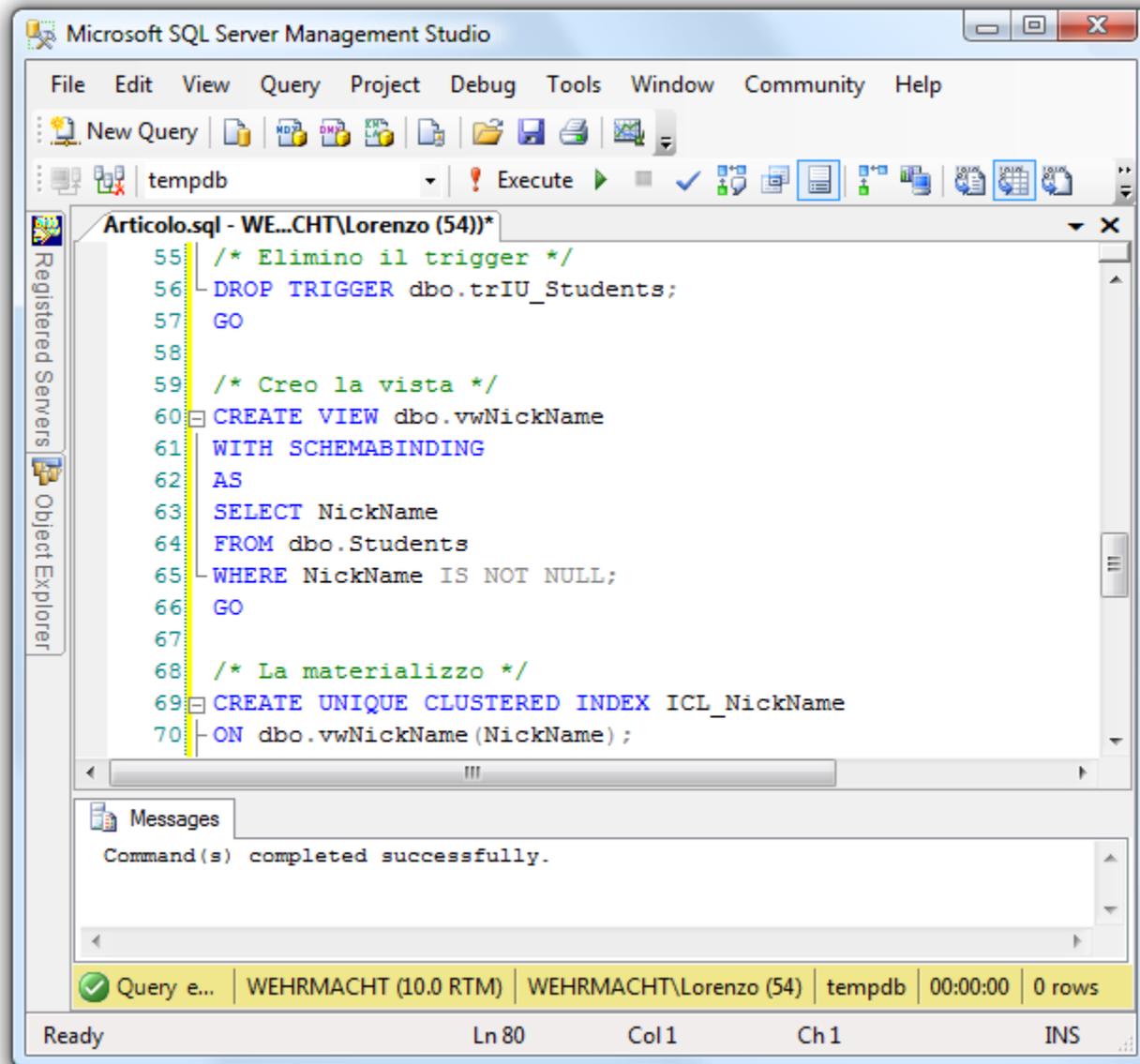


Figura 5

Verifichiamone il funzionamento inserendo un paio di righe:

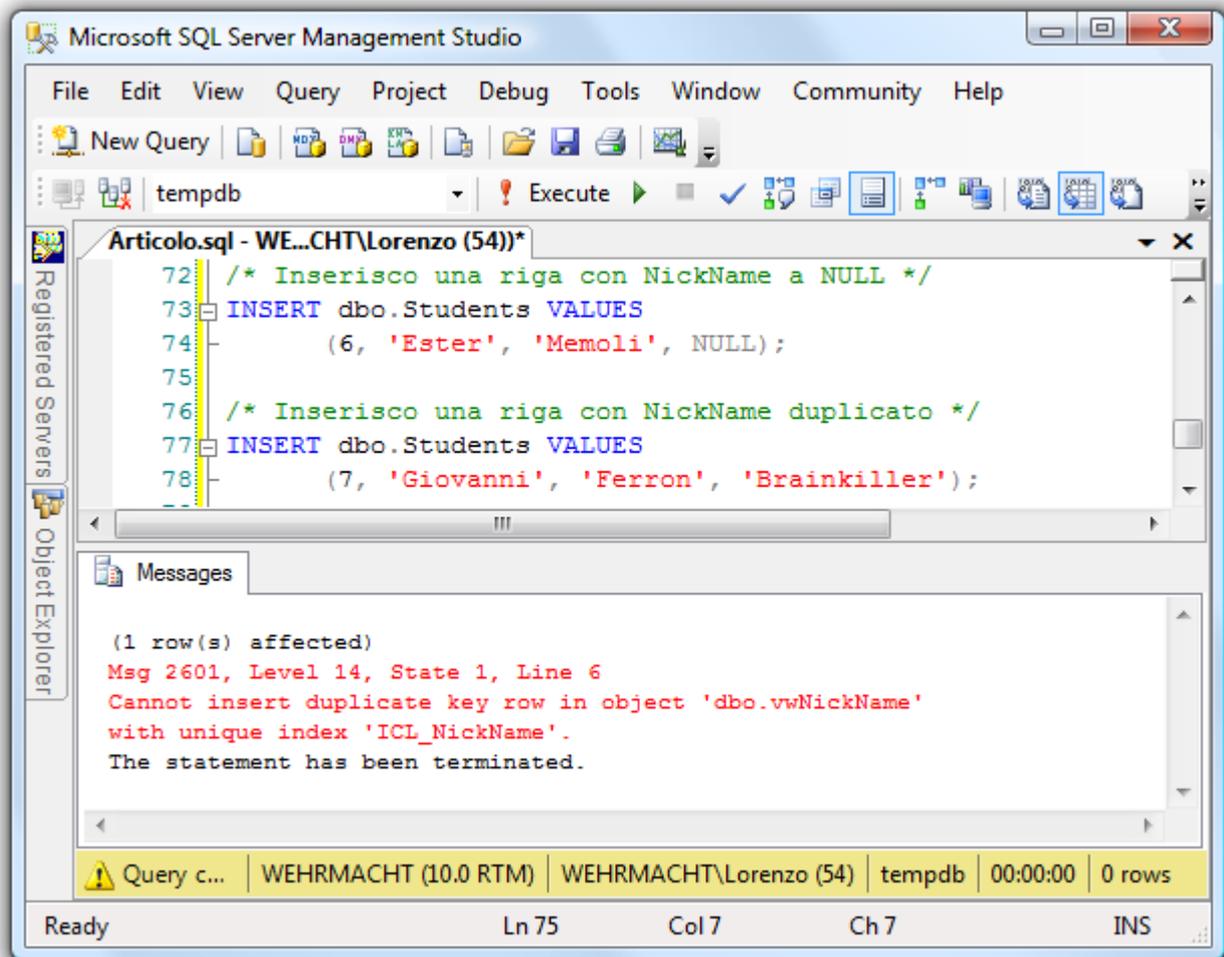


Figura 6

La riga numero 6 è stata inserita con successo anche se avevamo già due studenti con *NickName* a NULL, mentre la riga numero 7 ha causato una violazione dell'univocità dell'indice clustered, impedendone di fatto l'inserimento.

La soluzione di [Marcello](#) ha funzionato correttamente, ma volendo essere critici, possiamo individuare alcuni aspetti negativi:

- Abbiamo dovuto definire una nuova vista al nostro db il cui unico scopo è di garantire l'univocità del *NickName*;
- La materializzazione della vista mediante la definizione di un indice clustered comporta uno spreco di spazio su disco, ma soprattutto genera un certo overhead durante le operazioni DML (INSERT, UPDATE e DELETE) che coinvolgono la tabella e la colonna *NickName* per l'aggiornamento dell'indice;
- Ci giochiamo la possibilità di utilizzare l'indice clustered per un utilizzo più "nobile", ovvero quello di garantirci un efficiente piano di esecuzione definendolo sulle colonne utilizzate nelle aggregazioni, nella clausole BETWEEN e ORDER BY.

Filtered Indexes

SQL Server 2008 introduce un'interessantissima novità: i filtered indexes e le relative filtered statistics. Questi strumenti ci permettono di incrementare le prestazioni delle nostre query senza sprecare spazio prezioso su disco; offrono una più dettagliata e accurata distribuzione delle statistiche a tutto vantaggio di una migliore selettività e stima da parte del query optimizer e forniscono un'alternativa al problema che stiamo affrontando.

Prima di illustrare il loro utilizzo vediamo insieme di cosa si tratta.

In SQL Server 2008 possiamo definire un indice non clustered su un sottoinsieme di righe piuttosto che crearlo sull'intero set della tabella. OK, ma come? Semplicemente specificando una clausola WHERE nel comando CREATE INDEX.

L'implementazione in SQL Server 2008 ci permette di utilizzare un predicato piuttosto semplice basato sugli operatori logici IN, AND e sugli operatori di comparazione IS, IS NOT, =, <>, !=, >, >=, !>, <, <=, !<.

La possibilità di definire degli indici filtrati comporta indiscussi benefici ogni qualvolta sorga la necessità di interrogare frequentemente un sottoinsieme di righe. I vantaggi non si limitano al solo risparmio di spazio su disco per la memorizzazione del B-tree. Per esempio, le sole modifiche al subset di righe della tabella sottostante comporteranno l'aggiornamento delle relative index rows. La manutenzione dell'indice come la ricostruzione o la riorganizzazione sarà più veloce e meno "pesante" a livello di risorse.

Inoltre non dimentichiamoci che il numero massimo di passi negli istogrammi non è infinito, e poiché i filtered indexes sono definiti su un sottoinsieme di righe della tabella, ogni singolo passo nell'istogramma rappresenterà un numero inferiore di righe se confrontato con un indice non filtrato. Il risultato saranno statistiche più accurate.

Bene, dopo questa breve introduzione vediamo come gli indici filtrati possano tornarci utili per garantire l'univocità dei nostri soprannomi, senza però impedire la duplicazione di righe aventi la colonna *NickName* valorizzata a NULL.

Dai, non è difficile, sarà sufficiente definire un indice filtrato non clustered e univoco su tutte le righe con *NickName* NOT NULL:

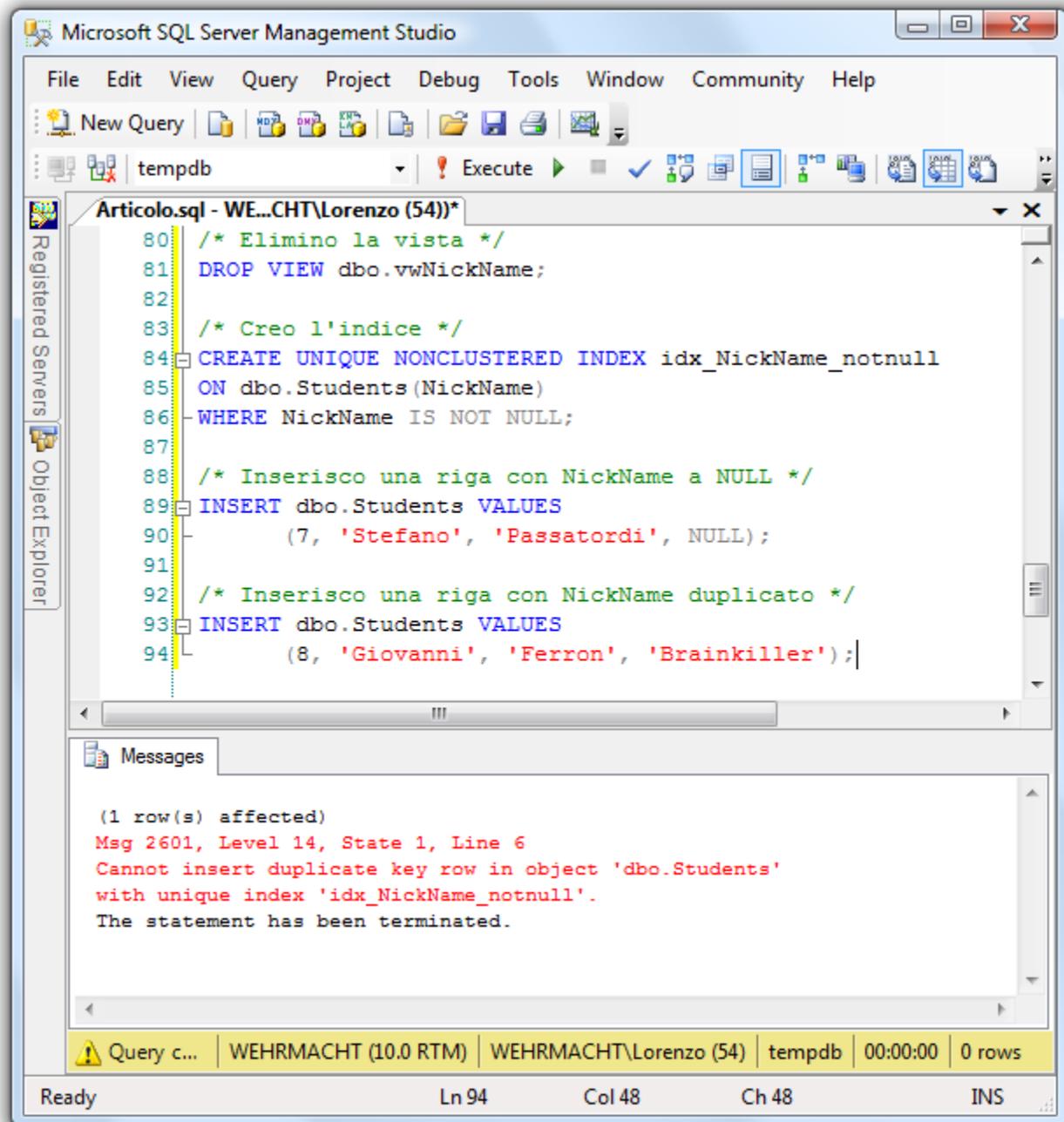


Figura 7

Anche in questo caso siamo giunti allo stesso risultato ottenuto nei due esempi precedenti, ma con indubbi vantaggi:

- Abbiamo evitato la creazione di nuovi oggetti come triggers e viste;
- Non ci siamo giocati la possibilità di definire l'unico indice clustered disponibile con colonne impegnate in aggregazioni, ordinamenti o scansioni per range;
- L'indice non clustered definito sulla tabella utilizzerà uno spazio su disco esiguo se confrontato con un normale indice, comporterà un minore utilizzo di risorse e avrà una manutenzione certamente meno onerosa.

Conclusioni

In questo articolo abbiamo affrontato un argomento tutt'altro che inconsueto sui forum o gruppi di discussione da un punto di vista diverso grazie ad una delle tante novità introdotte con SQL Server 2008.

Al [seguente link](#) potrete scaricare lo script SQL utilizzato negli esempi.

Vorrei ringraziare il mio amico [Marcello](#) che ci delizia spesso con i suoi esercizi di logica e con soluzioni inconsuete ma ricchissime di fascino ed eleganza ☺.

Autore

Lorenzo Benaglia lavora nel settore informatico come sviluppatore, trainer e DBA dal 1995 anche se la passione è iniziata una decina di anni prima con l'acquisto di un home computer MSX. Dall'anno 2000 svolge le mansioni di DBA e sviluppatore presso Il Sole 24 Ore S.p.A.

Dal 2004 è membro del [Team di DotNetHell](#) dove si occupa del [forum dedicato ai database](#).

Ha conseguito le certificazioni MCP, MCAD, MCSD (VS 6.0 e .NET), MCTS in SQL Server 2005 e 2008, MCITP Database Administrator, MCITP Database Developer, Oracle Database 10g Administrator Certified Associate, Oracle Database 10g Administrator Certified Professional e l'MVP Award per SQL Server (2003-2008).

© 2008 .netHell.it