DA T-SQL ALL'INTEGRAZIONE CON IL FRAMEWORK.NET.
ECCO COME SFRUTTARE AL MASSIMO IL DATABASE DI MICROSOFT

SQL SERVER
ASPETTI AVANZATI

Vito Vessia

EDIZIONI MASTER

SQL SERVER
ASPETTI AVANZATI

di Vito Vessia

EDIZIONI
MASTER

introduzione

Scopo del volume 8
Prerequisiti per la corretta fruizione del volume 9

Programmazione con T-SQL

1.1 I blocchi condizionali 14
1.1.1 IF 14
1.1.2 WHILE 15
1.2 Gestione degli errori 17
1.3 Le stored procedure 20
1.4 Le funzioni 22
1.4.1 Le funzioni che ritornano valori scalari 23
1.4.2 Le funzioni che ritornano resultset 25
1.5 Cursori 27
1.6 Tabelle temporanee 29
1.7 Le funzioni e le stored procedure di sistema 31
1.8 I trigger 32

Programmare SQL Server in .NET

2.1 Perché scrivere codice SQL Server in .NET 38
2.2 Metodologia alternative per estendere SQL Server 42
2.2.1 Chiamate ad oggetti COM da T-SQL 46
2.2.2 Vantaggi dell'approccio .NET rispetto alle due precedenti tecnologie 54
2.3 Il supporto .NET 2.0 per SQL Server 2005 55
2.3.1 Preparazione del database di esempio 55
2.3.2 Il progetto in Visual Studio 2005 59
2.4 Funzioni in CLR 61
2.4.1 Registrazione dell'assembly e della funzione senza Visual Studio 2005 Professional 68
2.4.2 Test della funzione 68
2.4.3 Debug della funzione 69
2.5 Stored procedure in CLR 71

2.5.1 Il codice 72
2.5.2 Registrazione, test e debug della stored procedure . . .80
2.6 Trigger in CLR 81
2.6.1 Registrazione, test e debug del trigger 84
2.7 Aggregati personalizzati 85
2.7.1 Registrazione, test e debug dell'aggregato 91
2.8 Tipo definito dall'utente 92
2.8.1 Registrazione, test e debug del tipo utente 98

Catalogo e funzioni di sistema

3.1 Tabelle del Catalogo di database 104
3.1.1 sysobjects 106
3.1.2 syscolumns 107
3.1.3 sysindexes 108
3.1.4 sysusers 108
3.1.5 sysdatabases 109
3.1.6 sysdepends 111
3.1.7 sysconstraints 112
3.2 Viste di Catalogo 113
3.2.1 sys.objects 113
3.2.2 sys.columns 114
3.3.3 sys.database_principals 115
3.3 Stored procedure di sistema 116
3.3.1 sp_help 116
3.3.2 sp_depends 118
3.3.3 sp_helptext 119
3.4 Funzioni scalari predefinite 120
3.4.1 Funzioni numeriche 122
3.4.2 Funzioni di data 123
3.4.3 Funzioni di stringa 124
3.4.4 Funzioni di testo/immagine 126
3.4.5 Funzioni di sistema 126

Nuove funzionalità in SQL Server 2005

4.1 Colonne calcolate persistenti 129
4.2 M.A.R.S. (Multiple Active Result Set) 130
4.3 Supporto migliorato al formato XML 133
4.3.1 XPath e XQuery 137
4.3.2 Modificare i dati dei campi XML 140
4.4 Novità nella gestione della sicurezza 141
4.4.1 Ruoli di amministrazione e di database 146
4.4.2 Indicazione e ricerca Full Text 147
4.4.3 Esempio di interrogazioni Full Text 149
4.4.4 Attivazione delle funzionalità 149

INTRODUZIONE

Microsoft SQL Server è il sistema di database relazionale (RDBMS) prodotto e sviluppato da Microsoft. Affonda le sue origini nel prodotto Sybase SQL Server 3 per Unix, da cui però si è fortemente evoluto e differenziato a partire dalla versione 7, che di fatto rappresenta una vera e propria riscrittura del code base. L'obiettivo originario, agli inizi degli anni 90, era quello di sviluppare una versione del prodotto per il sistema operativo OS/2. Tuttavia questa decisione è stata quasi da subito disattesa, seguendo la più generale strategia di uscita adottata da Microsoft nei confronti di OS/2, a favore del suo Windows NT. Nel 1994 la collaborazione con Sybase non viene rinnovata e da allora SQL Server subisce un fork di codice notevole.

SQL Server 2005 è il nome commerciale dell'ultima versione del prodotto RDBMS di Microsoft e durante tutta la sua fase di sviluppo ha mantenuto il nome in codice Yukon (un'area geografica del Canada). Come tutti i prodotti Microsoft, SQL Server 2005 gira solo su piattaforma Windows e in particolare su Windows 2000 (Desktop e Server), Windows XP e Windows Server 2003. Questo consente al prodotto di trarre il massimo dalla piattaforma S.O. su cui si basa, permettendo una politica di fortissima integrazione (dal file system, alla gestione della sicurezza e all'uso ottimizzato delle API di sistema), proprio perché non ha velleità di prodotto multipiattaforma. Inoltre gli consente di mantenere il look & feel dei tipici prodotti per Windows oltre ad offrire una facilità di installazione che non ha pari nelle versioni per Windows di altri prodotti multipiattaforma. Ma probabilmente il maggior vantaggio rispetto ad altri prodotto concepiti su piattaforma Unix (praticamente tutti gli altri) è la scala-

bilità: ci sono versioni di SQL Server 2005, come la Express, che girano su portatili non particolarmente dotati e, dall'altra parte, ci sono versioni come la Enterprise che girano su potenti server in cluster. Con Windows come unico collante.

SCOPO DEL VOLUME

Questo breve volume fornisce una panoramica completa delle funzionalità e delle caratteristiche di SQL Server 2005 specifiche per gli sviluppatori e quindi rappresenta il secondo e conclusivo volume della serie sul prodotto.
Il primo capitolo affronta una trattazione ampia sulla programmazione attraverso il linguaggio T-SQL per scrivere stored procedure, funzioni, trigger, funzioni ricorsive e quant'altro questo potente linguaggio consente per realizzare complessi blocchi di logica applicativa da far girare direttamente nel database, scritti nella sua lingua nativa.
Il secondo capitolo propone una completa trattazione di una delle più rilevanti novità di SQL Server 2005: la possibilità di scrivere e far eseguire codice .NET all'interno di SQL Server, rendendolo così di fatto un potente application server che consente di superare il modello tradizionale client/server. Con SQL Server 2005, infatti, è possibile scrivere direttamente stored procedure, funzioni e trigger in C# o nei linguaggi .NET. Questo codice gira direttamente in hosting in SQL Server 2005, che dispone di una sua completa macchina virtuale CLR, e questi oggetti possono essere richiamati direttamente da T-SQL come se fossero funzioni native o di sistema.
Il terzo capitolo descrive la struttura dei metadati di SQL Server e cioè le metodologie per interrogare la struttura

stessa degli oggetti del database usando un approccio simile a quello che si usa per interrogare i dati di tabelle definite dall'utente, dato che SQL Server usa speciali tabelle relazionali, dette di sistema, per conservare anche la sua struttura interna. Il capitolo si chiude con una carrellata sulle principali funzioni scalari integrate nel motore e utilizzabili dall'utente e dallo sviluppatore. Infine, l'ultimo capitolo è una miscellanea di diversi aspetti avanzati di SQL Server 2005: dal supporto al formato XML, al supporto alla tecnolgia M.A.R.S. e altre funzionalità innovative dell'ultima versione del prodotto.

REQUISITI PER LA CORRETTA FRUIZIONE DEL VOLUME

Questa guida fa riferimento alla versione standard di SQL Server 2005, dotata di versione completa del tool SQL Management Studio. Però la gran parte delle informazioni è applicabile anche alla versione Express, distribuita gratuitamente da Microsoft insieme ad una versione leggera dello strumento di amministrazione, SQL Mangement Studio Express. Pertanto, coloro che si avvicinano per la prima volta a questo prodotto o per gli altri utilizzatori di SQL Server che non hanno ancora fatto la migrazione alla versione 2005, la versione gratuita Express offre un'ottima possibilità di studiare e scoprire il prodotto a costo zero. Peraltro, la licenza che accompagna la versione Express la rende di fatto un prodotto vero e non solo una versione vetrina-giocattolo perché con la Express è possibile sviluppare vendere applicazioni che si basano su questa versione. Senza costi per lo sviluppatore e per l'utente finale, garantendo così un

percorso di migrazione molto scalabile verso le versioni successive a pagamento e rappresentando una valida alternativa a prodotti come Microsoft Access o a soluzioni più avanzate open source come MySQL o Firebird.
SQL Server 2005 Express è scaricabile gratuitamente dal seguente indirizzzo:
http://www.microsoft.com/downloads/details.aspx?FamilyID=220549B5-0B07-4448-8848-DCC397514B41&displaylang=it
La versione leggera dello strumento di amministrazione, SQL Management Studio Express, è disponibile in download gratuito dal seguente indirizzo:
http://www.microsoft.com/downloads/details.aspx?familyid=C243A5AE-4BD1-4E3D-94B8-5A0F62BF7796&displaylang=it
Entrambi prodotti sono disponibili in lingua italiana.
Nel volume si è fatto riferimento a tre database di esempio: AdventureWorks, il vero database di riferimento della versione 2005, Northwind, il database che ha accompagnato migliaia di programmatori nello studio SQL Server 7.0 e SQL Server 2000 e l'antico e un po' superato anche se sempre valido pubs.
Il primo è fornito in dotazione con SQL Server 2005 nelle versioni a pagamento. Gli utilizzatori della versione Express possono scaricare lo script T-SQL di creazione di AdventureWorks da questo indirizzo:
http://www.microsoft.com/downloads/details.aspx?FamilyID=e719ecf7-9f46-4312-af89-6ad8702e4e6e&DisplayLang=en
Northwind e Pubs sono invece disponibili per tutti (infatti non vengono distribuiti più con SQL Server 2005, nemmeno nelle versioni commerciali), al seguente indirizzo:

http://www.microsoft.com/downloads/details.aspx?familyid=06616212-0356-46a0-8da2-eebc53a68034&displaylang=en

Se vi risulta troppo complicato riportare questi indirizzi, usata il buon vecchio Google usando le keyword "SQL Server 2005 Samples"... Una volta scaricati i setup ed installati, è sufficiente aprire i file .Sql da SQL Management Studio (o Express) ed eseguirli. Genereranno l'intero database compresi i dati di esempio preziosi per i nostri esempi e per il vostro studio.

PROGRAMMAZIONE CON T-SQL

T-SQL offre numerosi costrutti del linguaggio che consentono di realizzare veri e proprio programmi e di gestire i flusso di esecuzione. Sono necessari quando non ci si deve limitare ad effettuare query, a manipolare o modificare dati o a modificare la strutture degli oggetti del database, ma quando si implementa della vera e propria logica in T-SQL.
Una prima principale estensione è costituita dai blocchi di dichiarazione; infatti è possibile delimitare blocchi di codice con una sequenza del tipo:

```
BEGIN
--codice T-SQL
--altro codice
END
```

Inoltre, come in ogni linguaggio di programmazione che si rispetti, è possibile definire variabili locali da usare all'interno di query, di blocchi di logica e in generale da codice T-SQL. Vediamone un esempio sul solito Northwind:

```
--dichiarazione delle variabili locali
DECLARE  @MINBIRTHDATE AS DATETIME,
         @COUNTRY AS NVARCHAR(15)
--assegnazione di una variabile
SET @COUNTRY = 'UK'
--uso di una variabile nella clausola WHERE
--assegnazione di una variabile attraverso una query
SELECT @MINBIRTHDATE = MIN(BirthDate)
FROM Employees
WHERE Country = @COUNTRY
```

```
--uso del contenuto di una variabile
PRINT @MINBIRTHDATE
```

Le variabili vengono dichiarate nel blocco DECLARE, ciascuna variabile locale è preceduta da una @ e dopo il nome segue il tipo della variabile (secondo il sistema di tipi di SQL Server), preceduto dalla parola chiave AS. Se nella DECLARE vi sono più variabili queste vanno separate da virgole. Naturalmente è possibile avere diversi blocchi DECLARE nello stesso blocco di codice. Le variabili possono essere usate all'interno di query (in tutte le clausole), all'interno i istruzioni INSERT, UPDATE e DELETE oltre che in blocchi di codice T-SQL di ogni genere, come si può intuire dall'esempio.

1.1 I BLOCCHI CONDIZIONALI

T-SQL offre un set di costrutti condizionali che lo rendono un completo linguaggio di programmazione procedurale consentendo, così, di costruire codice di logica che vada al di là delle possibilità offerte dalle query e dai comandi DML.

1.1.1 IF

La parola chiave IF consente di introdurre blocchi condizionali, come mostrato nel seguente esempio commentato:

```
DECLARE      @Country AS NVARCHAR(15),
    @RefEmployeeId AS INT
SET @Country = 'Italy'
--se vi sono almeno due dipendenti provenienti dal paese contenuto
--nella variabile locale @Country
IF (SELECT COUNT(*) FROM Employees WHERE Country = @Country) > 2
BEGIN
    --allora viene recuperata la chiave (EmployeeId) del più
    SELECT TOP 1 @RefEmployeeId = EmployeeID
```

```
    FROM Employees
    WHERE Country = @Country
    ORDER BY HireDate DESC
END
ELSE
BEGIN
    --altrimenti se ne provvede ad inserire uno nuovo
    INSERT INTO Employees
    (LastName, FirstName, BirthDate, Country)
    VALUES
    ('Vessia', 'Vito', '19740830', @Country)
    --e a conservarne la chiave identity attribuita dal database attraverso
                                                    la variabile
    --globale di sistema @@identity che contiene sempre l'ultima identity
                                                    assegnata
    --ad una tabella del database contenente campi identity
    SELECT @RefEmployeeId = @@identity
END
--stampa del risultato finale
PRINT @RefEmployeeId
```


1.1.2 WHILE

Si può gestire, inoltre, il tipo costrutto condizionale WHILE che consente di ripetere un determinato blocco di codice fino a quando resta vera una certa condizione. Osserviamo il seguente esempio commentato:

```
DECLARE @Country AS NVARCHAR(15),
    @RefEmployeeId AS INT
SET @Country = 'Italy'
--viene ripetuto il blocco nella WHILE fino a che non vi sono almeno quat
                                                    tro
--dipendenti provenienti dal paese contenuto nella variabile locale
```

```
                                                    @Country
WHILE (SELECT COUNT(*) FROM Employees WHERE Country = @Country)
                                                    < 4
BEGIN
    --altrimenti se ne provvede ad inserire uno nuovo
    INSERT INTO Employees
    (LastName, FirstName, BirthDate, Country)
    VALUES
    ('Vessia', 'Vito', '19740830', @Country)
END
--allora viene recuperata la chiave (EmployeeId) più recente
SELECT TOP 1 @RefEmployeeId = EmployeeID
FROM Employees
WHERE Country = @Country
ORDER BY EmployeeID DESC
--stampa del risultato finale
PRINT @RefEmployeeId
```

Una WHILE può essere interrotta prima che la condizione di valutazione smetta di essere vera introducendo un'istruzione BREAK all'interno del codice stesso.
Infine SQL Server 2005 introduce una nuova e più potente gestione delle eccezioni. In passato era solo possibile sollevare un'eccezione con l'istruzione RAISERROR e testando la variabile globale @@ERROR. Osserviamo il seguente esempio su Northwind:

```
IF (SELECT COUNT(*) FROM Employees WHERE Country = 'Italy') < 10
BEGIN
    RAISERROR ('Non vi sono abbastanza dipendenti nella filiale',
                                                    --messaggio
            1,  --gravità
            5)  --stato
END
```

1.2 GESTIONE DEGLI ERRORI

Se nella tabella Employees ci sono meno di dieci impiegati di origine italiana viene sollevata un'eccezione con l'istruzione RAISERROR. Il primo parametro rappresenta la stringa del messaggio di errore, il secondo la gravità e il terzo lo stato. I tre parametri hanno un significativo esclusivamente applicativo e pertanto ciascuno può gestirlo a piacimento. SQL Server risponderebbe con un perentorio:

```
Non vi sono abbastanza dipendenti nella filiale
Msg 50000, Level 1, State 5
```

Esiste anche un'altra versione dell'istruzione che permette di non specificare ogni volta il testo del messagio di errore, ma di registrare nel sistema un codice numerico di errore personalizzato e di associare ad esso un messaggio, magari configurato per le varie lingue. Vediamo come fare con il seguente esempio:

```
EXEC sp_addmessage 50002, 1, 'Non vi sono abbastanza dipendenti nella
                                                              filiale'

IF (SELECT COUNT(*) FROM Employees WHERE Country = 'Italy') < 10
BEGIN
    RAISERROR (50002,  --messaggio
               1,  --gravità
               5)  --stato
END
```

In questo caso viene registrato nel sistema il codice di errore 50002 (i codici di errori applicativi devono necessariamente partire da 50000) con il testo corrispondente. Questa operazione va fatta una sola volta e il nuovo messaggio viene registrato definitivamente fino a successiva variazione o rimozione. A questo punto è possibile sollevare l'errore semplicemente l'errore attraverso il suo codice, come mo-

strato, per ottenere il seguente messaggio dal sistema:

```
Non vi sono abbastanza dipendenti nella filiale
Msg 50002, Level 1, State 5
```

La gestione e l'intercettazione di errori sollevati direttamente dal sistema era un'operazione complicata basata sul vecchio approccio alla BASIC e comunque non consentiva di annullare un errore ma solo di intercettarlo. Osserviamo il seguente esempio:

```
--tentativo di inserimento di un record duplicato su Customers di Northwind
--l'operazione solleva l'errore 2627 di sistema che è la violazione della
                                                        chiave primaria
INSERT INTO Customers
(CustomerId, CompanyName)
VALUES
('AROUT', 'Nuova Arout')
--viene riletto il valore di errore corrente dalla variabile globale @@ERROR
--è importante ricordare che essa viene azzerata ad ogni istruzione
                                                              e quindi
--il suo valore va riletto immediatamente dopo l'istruzione che può causare
                                                              l'errore
IF @@ERROR = 0
BEGIN
    --in caso di errore si salta alla label Fine
    GOTO Fine
END
--esempio di label applicativa che gestisce il blocco di errore
Errore:
    PRINT 'Si è verificato un errore di inserimento'
--esempio di label applicativa posta alla fine del blocco di codice
Fine:
    PRINT 'Fine operazione'
```

L'output di questo blocco di codice è il seguente:

```
Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK_Customers'. Cannot insert
                                duplicate key in object 'dbo.Customers'.
The statement has been terminated.
Si è verificato un errore di inserimento
Fine operazione
```

La gestione di errori rudimentali è comunque riuscita ad intercettare e a gestire l'errore, ma non ad annullarne l'effetto nel blocco di codice. Proviamo a riscrivere l'esempio usando la nuova e potente gestione degli errori di SQL Server 2005 basata sul costrutto TRY CATCH:

```
BEGIN TRY
    --blocco di codice che causa l'errore
    INSERT INTO Customers
    (CustomerId, CompanyName)
    VALUES
    ('AROUT', 'Nuova Arout')
END TRY
BEGIN CATCH
    --blocco di codice che gestisce l'errore
    PRINT 'Si è verificato un errore di inserimento'
END CATCH
PRINT 'Fine operazione'
```

Nel blocco TRY si inserisce il codice che si intende proteggere perché foriero di eventuali errori. Nel blocco CATCH, invece, va gestito l'errore eventuale e viene saltato se non si verifica. A parte la notevole pulizia e chiarezza del codice rispetto al codice precedente, proviamo ad osservare anche l'output:

```
Si è verificato un errore di inserimento
Fine operazione
```

I messaggi applicativi sono gli stessi dell'esempio nella vecchia maniera, ma l'errore è stato annullato e l'esecuzione del resto del codice è fatta salva perché si è gestita applicativamente la situazione d'errore.

1.3 LE STORED PROCEDURE

Le stored procedure sono uno dei metodi più efficaci per spostare la logica di programmazione sul server. Esse sono invocabili dal codice T-SQL come normali procedure in un qualsiasi linguaggio di programmazione. Al loro interno posso effettuare scritture o manipolazioni sui dati delle tabelle del database e/o eseguire interrogazioni. Eccone la sintassi in un esempio commentato che deriva estende l'esempio precedente basato visto per l'istruzione IF:

```
--creazione di una stored procedura con la parola chiave
                                        CREATE PROCEDURE <nome>
--la modifica di un'eventuale procedura già esistente si effettua con la
                                                  ALTER PROCEDURE
CREATE PROCEDURE [dbo].[GetLastestClerk]
--dichiarazione dei paramatri passati come argomento; i parametri di
                                              default vengono passati
--con la sintassti @parametro AS tipo = valore_default
    @Country AS NVARCHAR(15),
    @LastName  AS NVARCHAR(20),
    @FirstName  AS NVARCHAR(10),
    @BirthDate AS DATETIME = NULL
--fine degli argomenti passati
AS
--corpo della stored procedure; nell'esempio: se vi sono almeno due
```

```
                                                    dipendenti
--provenienti dal paese contenuto nella variabile locale @Country
IF (SELECT COUNT(*) FROM Employees WHERE Country = @Country) <= 2
BEGIN
    --altrimenti se ne provvede ad inserire uno nuovo
    INSERT INTO Employees
    (LastName, FirstName, BirthDate, Country)
    VALUES
    (@LastName, @FirstName, @BirthDate, @Country)
END
--allora viene recuperata la chiave (EmployeeId) del più
SELECT TOP 1 EmployeeId, FirstName, LastName, Country
FROM Employees
WHERE Country = @Country
ORDER BY EmployeeID DESC
```

In pratica la procedura accetta il parametro @Country, cioè il paese di riferimento su cui effettuare le ricerche sul numero di dipendenti per quel paese e i dati anagrafici del nuovo dipendente da inserire nella tabella Employees qualora i dipendenti per il paese indicato come parametro siano in numero inferiore a 2. Le stored procedure posso ritornare un resultset, in tal caso l'ultima operazine SQL da eseguire nel corpo della procedura deve ritornare un resultset oppure può eseguire DML. È altresì possibile passare parametri per referenza, che quindi potranno essere rivalorizzati all'interno della procedura. Ecco come dovrebbe la definizione di una stored procedure un parametro passato come argomento:

```
ALTER PROCEDURE [dbo].[GetLastestClerkWithOut]
    @Country AS NVARCHAR(15),
    @LastName  AS NVARCHAR(20),
    @FirstName  AS NVARCHAR(10),
    @BirthDate AS DATETIME = NULL,
```

```
    @IsNew AS BIT OUT
AS
--assegnazione del parametro di output
SET @IsNew = 0
```

Richiamare una stored procedura da codice SQL è molto semplice. Ecco come richiamare la procedura appena scritta (in Figura 1.1 è mostrato l'esempio in esecuzione):

```
EXEC GetLastestClerk 'Italy', 'Vessia', 'Vito'
```

Se invece volessimo chiamare la stored procedure con parametro di output, dovremmo:

```
DECLARE @isnew AS BIT
EXEC [GetLastestClerkWithOut] 'Italy1', 'Vessia', 'Vito', '19740830',
                                                          @isnew OUT
SELECT @isnew
```

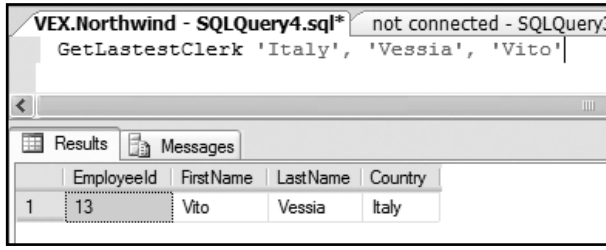


Figura 1.1: Esecuzione di una stored procedura che restituisce resultset

1.4 LE FUNZIONI

Le funzioni hanno in SQL lo stesso significato dell'analogo termine nella programmazione procedurale: si tratta di procedure a cui si

passano argomenti e che restituiscono risultati.

1.4.1 Le funzioni che ritornano valori scalari

La prima e più semplice forma è rappresentata dalle funzioni che restituiscono valori scalari. Osserviamo un esempio basato sulla stored procedure dell'esempio precedente che, questa a volta, a fronte del parametro @Country, si limita a restituire l'EmployeeId del dipendente assunto più di recente e proveniente dal paese indicato dall'argomento:

```
--creazione di una funzione con la parola chiave CREATE FUNCTION
                                                        <nome>
--la modifica di un'eventuale procedura già esistente si effettua con la
                                                    ALTER  FUNCTION
CREATE FUNCTION [dbo].[GetLatestClerkScalarFunction]
(
--dichiarazione dei paramatri passati come argomento; i parametri di
                                                default vengono passati
--con la sintassti @parametro AS tipo = valore_default
@Country AS NVARCHAR(15)
)
--tipo del valore di ritorno scalare della funzione (nell'esempio sarà
                                                        un intero)
RETURNS INT
AS
--corpo della funzione racchiuso in un blocco BEGIN END
BEGIN
DECLARE @RefEmployeeId AS INT

--allora viene recuperata la chiave (EmployeeId) del più
SELECT TOP 1 @RefEmployeeId = EmployeeID
FROM Employees
```

```
WHERE Country = @Country
ORDER BY HireDate DESC
--valore di ritorno della funzione: la RETURN deve essere sempre presente
                                                    nella funzione
--ma può ripetersi più a volte a seconda del flusso di esecuzione
                                                    della funzione e non deve
--necessariamente stare alla fine
RETURN @RefEmployeeId
--istruzione di chiusura del corpo della funzione
END
```

L'uso di una funzione è semplice come l'uso di una stored procedure; dato che restituisce un valore scalare, può essere adoperato direttamente con l'istruzione T-SQL PRINT che stampa un valore:

```
PRINT dbo.GetLatestClerkScalarFunction('Italy')
```

Oppure può essere utilizzato all'interno di una query e, più in generale, all'interno di qualsiasi blocco di codice T-SQL. Nel seguente esempio viene usato per restituire la colonna di una query e quindi è usato nella clausola SELECT, inoltre compare anche nella clausola WHERE come condizione:

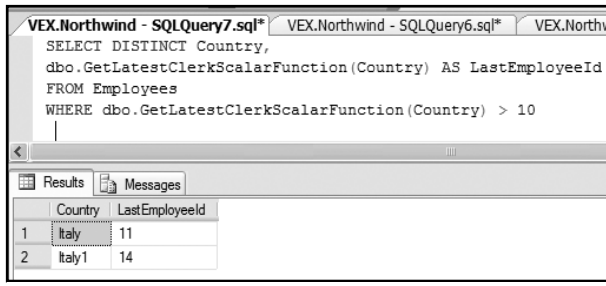


Figura 1.2: Una funzione scalare in azione

```
SELECT DISTINCT Country,
dbo.GetLatestClerkScalarFunction(Country) AS LastEmployeeId
FROM Employees
WHERE dbo.GetLatestClerkScalarFunction(Country) > 3
```

In Figura 1.2 ne possiamo osservare il risultato.

1.4.2 Le funzioni che ritornano resultset

Esiste un'interessante variante delle funzioni che restituisce un resultset (un oggetto TABLE) anziché un valore scalare, analogamente a quanto farebbe una query o una stored procedure. Osserviamone una definizione derivata dal precedente esempio che, questa volta, non si limita a restituire l'Id del dipendente assunto più di recente proveniente dal paese passato come argomento, ma un resultset corrispondente ai due

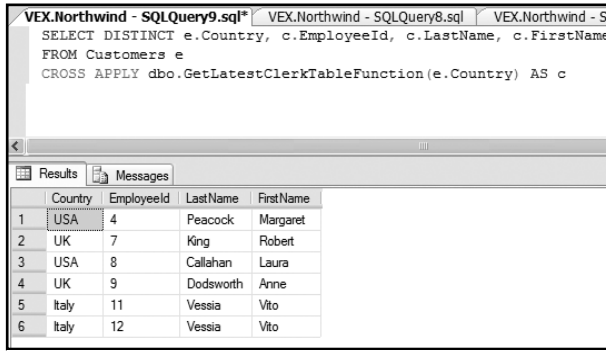
```
--creazione di una funzione con la parola chiave CREATE FUNCTION
                                                        <nome>
--la modifica di un'eventuale procedura già esistente si effettua con la
                                                    ALTER  FUNCTION
CREATE FUNCTION [dbo].[GetLatestClerkTableFunction]
(
--dichiarazione dei paramatri passati come argomento; i parametri di
                                                default vengono passati
--con la sintassti @parametro AS tipo = valore_default
      @Country AS NVARCHAR(15)
)
--questa volta la RETURNS riporta la parola chiave TABLE ad indicare la
                                                restituizione di un resultset
RETURNS TABLE
AS
RETURN
(
```

```
    SELECT TOP 2 *
    FROM Employees
    WHERE Country = @Country
    ORDER BY HireDate DESC
)
```

L'uso di queste funzioni è molto potente perché possono essere adoperate nelle query come normali oggetti tabella o viste, come nell'esempio che segue.

```
SELECT *
FROM dbo.GetLatestClerkTableFunction('Italy')
```

Fino a SQL Server 2000, però, questa modalità d'uso era fortemente limitata dall'impossibilità di correlare in una query l'argomento della funzione di tipo TABLE con i campi delle altre tabelle o viste in JOIN. Era infatti possibile semplicemente possibile valori scalari dichiarati esternamente alla query o valori costanti, ma non valori di correlazione. SQL Server 2005 introduce una potente novità: l'istruzione APPLY che supera questo limite. Osserviamo il seguente esempio:



	Country	EmployeeId	LastName	FirstName
1	USA	4	Peacock	Margaret
2	UK	7	King	Robert
3	USA	8	Callahan	Laura
4	UK	9	Dodsworth	Anne
5	Italy	11	Vessia	Vito
6	Italy	12	Vessia	Vito

Figura 1.3: Una funzione scalare in azione

```
SELECT DISTINCT e.Country, c.EmployeeId, c.LastName, c.FirstName
FROM Customers e
CROSS APPLY dbo.GetLatestClerkTableFunction(e.Country) AS c
```

Esso va interpretato come una INNER JOIN tra Customers e la funzione GetLatestClerkTableFunction in cui la correlazione è a livello di argomento passato alla funzione. In Figura 1.3 se ne può osservare il risultato.

1.5 CURSORI

SQL Server offre il supporto ai cursori, cioè alla possibilità di non ottenere il risultato di una query come un blocco di dati unico e continuo, ma come un flusso su cui effettuare l'enumerazione, analogamente ad una for each di un linguaggio moderno ad alto livello. Piuttosto che procedere con lunghe e noiose spiegazioni teoriche, procediamo ad analizzare e studiare un esempio completo

```
DECLARE @Country as NVARCHAR(15)
--dichiarazione di un cursore
DECLARE EmployeesCountriesCursor CURSOR FOR
--la query su cui agirà il cursore
SELECT Country
FROM Employees
GROUP BY Country
--apertura del cursore
OPEN EmployeesCountriesCursor
--recupero della prima riga del cursore
FETCH NEXT FROM EmployeesCountriesCursor
--il campo o i campi della query vanno a finire in altrettante variabili locali
                                                    dello stesso tipo;
--tali variabili vanno precedentemente dichiarate e nella INTO vanno
```

```
                                        indicate tutte queste variabili
--nella stessa posizione in cui vengono restituite dal loro corrispondente
                                                    campo nella query
INTO @Country

--il cursore viene testato con una WHILE fino non si giunge all'EOF
WHILE @@FETCH_STATUS = 0
BEGIN
    --corpo di operazioni eseguibili all'interno del ciclo del cursore
    --nell'esempio viene invocata una stored procedure a partire dai valori
                                                                    della
    --riga corrente del cursore
    EXEC GetLastestClerk @Country, 'Nuovo Dipendente', 'Nuovo'

    --viene recuperata la riga successiva del cursore, se c'è
    FETCH NEXT FROM EmployeesCountriesCursor
    INTO @Country
END
```

Figura 1.4: La APPLY in azione

```
--viene chiuso il canale dati col cursore
CLOSE EmployeesCountriesCursor
--viene deallocato il cursore per liberare risorse
DEALLOCATE EmployeesCountriesCursor
```

In Figura 1.4 è possibile osservare l'esempio in esecuzione.

1.6 TABELLE TEMPORANEE

Le tabelle temporanee sono delle normali tabelle di SQL Server, definite allo stesso modo delle tabella tradizionali, ma che hanno la particolarità di avere uno scope temporaneo, cioè nascono e muoiono nell'ambito della singola sessione di connessione e database, non lasciando alcuna traccia sul database dopo la loro scomparsa. Per la loro stessa natura transitoria non risiedono fisicamente sul database correntemente in uso ma sul database Temp di SQL Server che è un database di sistema adoperato dall'engine proprio per eseguire operazioni temporanee. Osserviamo la definizione di una tabella temporanea. L'unico elemento che la distingue dall'analoga creazione di una tabella fisica è l'obbligo del prefisso # davanti al nome della tabella. Questa è condizione necessaria e sufficiente perché la tabella venga considerata e gestita come temporanea.

```
CREATE TABLE #NewClerks
(
    EmployeeId INT NOT NULL,
    LastName NVARCHAR(20),
    FirstName NVARCHAR(10),
    Country NVARCHAR(20)
)
```

Osserviamo il seguente esempio d'uso della tabella temporanea ap-

pena creata. Si tratta di una versione leggermente modificata del codice di esempio del cursore introdotto nel paragrafo precedente, ma che questa volta non si limita a chiamare la stored procedure, ma a inserirne i valori del resultset di ritorno nella tabella temporanea d'esempio, adoperando una versione particolare dell'istruzione INSERT che consente di inserire direttamente i valori di una stored procedure in una tabella:

```
DECLARE @Country as NVARCHAR(15)

DECLARE EmployeesCountriesCursor CURSOR FOR
SELECT Country
FROM Employees
GROUP BY Country

OPEN EmployeesCountriesCursor

FETCH NEXT FROM EmployeesCountriesCursor
INTO @Country

WHILE @@FETCH_STATUS = 0
BEGIN
    --viene invocata la stored procedure  GetLastestClerk, come
                                                nell'esempio precedente,
    --essa restituisce i campi  EmployeeId,  LastName,  FirstName, Country
                                                che vengono
     --inseriti come nuova riga della tabella temporanea (questa modalità
                                                della INSERT
    --è adoperabile anche nelle tabelle normali)
    INSERT INTO #NewClerks
    EXEC GetLastestClerk @Country, 'Nuovo Dipendente', 'Nuovo'

    FETCH NEXT FROM EmployeesCountriesCursor
```

```
    INTO @Country
END
CLOSE EmployeesCountriesCursor
DEALLOCATE EmployeesCountriesCursor
```

Eseguiamo una interrogazione sulla tabella temporanea, analogamente a quanto faremmo con le tabelle fisiche. In Figura 1.5 possiamo osservarne il risultato.

```
SELECT *
FROM #NewClerks
```

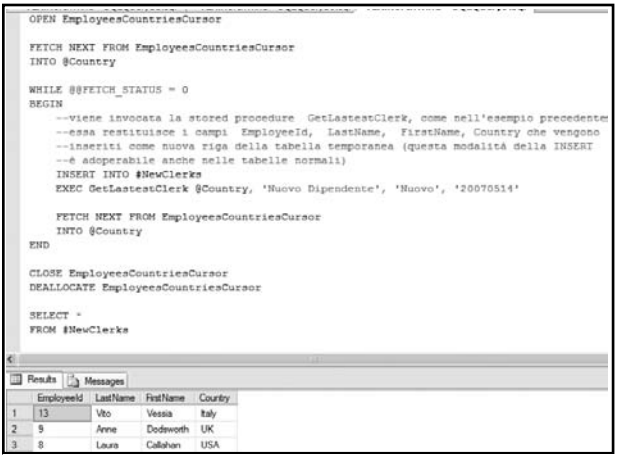


Figura 1.5: Le tabelle temporanee

1.7 LE FUNZIONI E LE STORED PROCEDURE DI SISTEMA

SQL Server offre una quantità impressionante di funzioni e stored procedure di sistema. Dalla funzione GetDate() per ottenere la data corrente di sistema, alle stored procedure per impostare parametri ed

impostazioni dei vari oggetti di sistema. Riportarli tutti sarebbe impossibile, ma anche riportarne una parte avrebbe poco senso vista l'eseguità dello spazio e dunque si rimanda alla documentazione ufficiale del produttore per consultare una lista esaustiva di queste informazioni. Esse verranno affrontate più diffusamente in uno dei prossimi capitoli.

1.8 I TRIGGER

I trigger sono l'equivalente degli eventi nella programmazione tradizionale. In pratica sono porzioni di codice T-SQL che possono essere fatti eseguire in automatico dal database engine quando su una tabella si verifica un'operazione di INSERT, UPDATE e/o DELETE. Osserviamone la sintassi:

```
CREATE TRIGGER [ schema.]nome_trigger
ON { nome_tabella | nome_vista }
{ FOR|AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS { sql_statement  [ ; ] [ ...n ] }
```

Il trigger può agire sul tabella o sulla vista il cui nome segue la clausola ON, può agire su operazioni di INSERT, UPDATE e/o DELETE (lo stesso trigger può servire anche più operazioni DML sulla stessa tabella o vista) e può essere di due tipi:

- " FOR, agisce prima che venga effettuata l'operazione di scrittura sulla tabella e pertanto può essere usato per impedirne il completamento in caso di inconsistenza del dato da scrivere;
- " AFTER, agisce solo dopo che l'operazione di scrittura sulla tabella è stata completata senza errori e non è applicabile sulle viste;
- " INSTEAD OF, agisce in sostituzione dell'operazione DML, signi-

fica che se, ad esempio, viene impostato un trigger INSTEAD OF DELETE su una determinata vista o tabella, la DELETE non verrà mai realmente eseguita ma al suo posto verrà eseguito il trigger.

Osserviamo il seguente esempio di trigger di tipo AFTER tratto dal AdventureWorks che agisce sulla tabella Department contenuta nello schema HumanResources:

```
--creazione del trigger con definizione della tabella di riferimento
CREATE TRIGGER [HumanResources].[uDepartment] ON
                                        [HumanResources].[Department]
--il trigger è di tipo AFTER UPDATE
AFTER UPDATE NOT FOR REPLICATION AS
BEGIN
   SET NOCOUNT ON;
   --dopo un'operazione di update di una riga di Department, il trigger
                                        modifica il campo ModifiedDate
   --della tabella stessa impostando la data corrente come suo valore
   UPDATE [HumanResources].[Department]
   SET [HumanResources].[Department].[ModifiedDate] = GETDATE()
   --la parola chiave "inserted" rappresenta la tabella coinvolta dal trigger,
                                        ne è in pratica il this
   FROM inserted
   WHERE inserted.[DepartmentID] =
                        [HumanResources].[Department].[DepartmentID];
END;
```

I trigger sono certamente molto comodi e potenti anche se a volte rendono meno comprensibile il flusso di modifica dei dati proprio perché sono asincroni e non vengono mai esplicitamente invocati dal chiamante. Inoltre rappresentano evidentemente un overhead per il sistema e quindi vanno adoperati con moderazione. Microsoft, però, da SQL Server 2000 introduce una variante di trigger di tipo INSTEAD

OF che quindi non segue l'operazione di modifica della tabella o della vista, ma la sostituisce del tutto. Il tipico esempio d'uso è costituito dalle applicazioni che non cancellano mai veramente i record, ma si limitano ad operare cancellazioni logiche in modo da salvaguardare lo storico e poter facilmente recuperare eventuali situazione di inconsistenza dei dati a seguito di cancellazioni maldestre. L'approccio tipico è di aggiungere un campo BIT Deleted nella tabella interessata dalla cancellazione logica, facendogli assumere il valore 0 quando il record è valido e 1 quando il record è cancellato. L'approccio tradizionale alle cancellazioni prevedeva l'uso di UPDATE (sul campo Deleted) per operare le cancellazioni logiche. Con il nuovo approccio basta introdurre un trigger INSTEAD OF DELETE che, a fronte di un'operazione di DELETE sulla riga, effettui l'UPDATE sul campo di scope Deleted.
Osserviamo, infine, un altro esempio di trigger INSTEAD OF DELETE tratto da AdventureWorks:

```
ALTER TRIGGER [HumanResources].[dEmployee] ON
                                        [HumanResources].[Employee]
INSTEAD OF DELETE NOT FOR REPLICATION AS
BEGIN
   SET NOCOUNT ON;

   DECLARE @DeleteCount int;

   SELECT @DeleteCount = COUNT(*) FROM deleted;
   IF @DeleteCount > 0
   BEGIN
      RAISERROR
         (N'Employees cannot be deleted. They can only be marked as not
                                                current.', -- Message
         10, -- Severity.
         1); -- State.
```

```
    -- Rollback any active or uncommittable transactions
    IF @@TRANCOUNT > 0
    BEGIN
        ROLLBACK TRANSACTION;
    END
  END;
END;
```

PROGRAMMARE SQL SERVER IN .NET

SQL Server 2005 introduce il supporto alla scrittura di stored procedure, funzioni, trigger e tipi definiti dall'utente in un qualsiasi linguaggio del CLR Common Language Runtime (C#, Visual Basic .NET ecc...). Questi oggetti verranno ospitati ed eseguiti all'interno di una istanza della macchina virtuale integrata in SQL Server.

Le stored procedure, le funzioni e i trigger vengono scritti come metodi statici di classi .NET, invece i tipi definiti dall'utente sono scritti come intere classi. Il codice così definito viene compilato in un assembly che viene caricato in SQL Server 2005 usando l'istruzione T-SQL CREATE ASSEMBLY. Ogni stored procedure, funzioni, trigger e tipo viene registrato nel database con la corrispondente CREATE PROCEDURE, CREATE FUNCTION, CREATE TRIGGER, CREATE TYPE e CREATE AGGREGATE. Da quel momento essere possono essere impostate ed utilizzate come se fossero oggetti nativi T-SQL e dunque richiamabili anche da T-SQL.

La versione di .NET supportata da SQL Server 2005 è la 2.0 e pertanto Visual Studio 2005 ne supporta lo sviluppo, il caricamento e il debugging nel database. È interessante rilevare che i progetti Visual Studio 2005 che producono codice .NET per SQL Server 2005 possono riferire assembly esterni, come ogni altra applicazione .NET, e che sia Visual Studio che SQL Server offrono il supporto a questa funzionalità.

Come vedremo nel proseguo del capitolo, tutte le operazioni di caricamento degli assembly e di registrazione degli oggetti .NET in SQL Server 2005 sono effettuabili attraverso comandi T-SQL predisposti ad hoc, pertanto è possibile a costo zero costruire applicazioni .NET (il runtime è scaricabile gratuitamente) con il notepad o con qualsiasi editor di testo o con IDE gratuiti di sviluppo per .NET quali SharpDevelop o lo stesso Visual Studio 2005 Express, anch'esso gratuito. L'applicazione può far uso di SQL Server 2005 Express, che come abbiamo visto è gratuito, e già in questa versione possiamo scrivere le nostre estensioni .NET per

SQL Server e farle girare nella versione Express di SQL Server 2005. Useremo, poi, i comandi T-SQL di registrazione di questi oggetti per caricarli e farli eseguire nel database della nostra applicazione. E, a costo di risultare ripetitivo, tutto ciò è gratuito e quindi non dovrete spendere nemmeno un euro per avere il tutto sul vostro PC e, poi, sul PC su cui dovrà girare l'applicazione. E questo anche per utilizzi commerciali: dunque cioè potrete vendere regolarmente le vostre applicazioni scritte in C# con Visual Studio C# Express e SQL Server 2005 Express.
Tuttavia, acquistando Visual Studio 2005, almeno nella versione Professional, viene offerto il supporto alla scrittura, al debugging e al deployment automatico ed integrato di oggetti .NET all'interno di SQL Server 2005 (qualsiasi versione, anche la versione gratuita Express). In tal caso non avrete quasi per nulla a che fare con il codice T-SQL di installazione e manutenzione dei vostri assembly e oggetti .NET in SQL Server, ma Visual Studio farà tutto il lavoro. E, soprattutto, vi offrirà il supporto al debugging di questo codice.

2.1 PERCHÉ SCRIVERE CODICE SQL SERVER IN .NET

L'approccio tradizionale allo sviluppo di applicazioni gestionali, meglio se di classe enterprise, prevedeva il classico modello client/server, con un'applicazione client scritta, in questo caso in C# ma in generale con qualsiasi linguaggio di programmazione di alto livello dotato di un buon IDE, e la parte server costituita dal database server. Il modello successivo che si è imposto negli ultimi anni è il multi-tier che prevede uno o più strati intermedi tra il client, che così si snellisce e smette di mantenere della logica applicativa che è tipicamente lato-server, e il database server. In mezzo c'è il middle-tier, tipicamente costituito da un application server ospitato da un object broker (COM+), da soluzioni personalizzate (ad esempio application server auto-costruiti ospitati su servizi o applicazioni host o sul web server e raggiungibili attraverso protocollo proprietari come il buon vecchio .NET Remoting, i web service SOAP o

l'attuale WCF) o da veri e proprio ambienti server potenti, robusti ed in grado di offrire il supporto all'hosting di applicazioni lato server. Quest'ultimo approccio è tipicamente usato dall'altra metà del cielo, cioè dalla piattaforma Java e dalla sua estensione server J2EE. (IBM WebSphere, BEA WebLogic e ogni altra implementazione, anche gratuita degli application server J2EE).

In Microsoft mancava un approccio di quest'ultimo tipo o comunque non era così sviluppato. In realtà IIS negli ultimi anni si è evoluto per diventare un vero e proprio host di applicazioni lato server, ma di certo i servizi offerti da questo alle applicazioni ospitate al suo interno non sono assimilabili alla ricchezza di supporto offerto da prodotti come WebSphere o Jboss. Qualcosa del genere è offerto da Sharepoint ed Exchange, due prodotti server di Microsoft, ma essi risolvono nicchie applicative specifiche e non sono prodotti generalisti.

Con SQL Server 2005 Microsoft va finalmente a colmare questa lacuna o quanto meno a gettare le basi per farlo. SQL Server 2005, ospitando pezzi di codice e di logica scritti in linguaggio non T-SQL, si pone come una prima robusto e significativa alternativa agli application server J2EE.

È fondamentale comprendere il corretto uso di questa estensione nello sviluppo delle proprie applicazioni. .NET non sostituisce T-SQL in SQL Server, ma va a coprire ambiti applicativi differenti. Infatti scrivere query SQL è e probabilmente resterà sempre il modo migliore per effettuare interrogazioni sul database perché è l'unico modo per sfruttare tutte le ottimizzazioni previste dall'engine nelle interrogazioni. Paradossalmente, molte delle ragioni che avrebbero fatto desiderare l'uso di codice .NET in SQL Server 2000, come l'assenza del supporto alle query ricorsive, l'accesso alle righe in una query per indice posizionale (in ADO.NET, se vogliamo accedere alla quarta riga di un result set dobbiamo semplicemente accedere all'indice myDataTable.Rows[3]) e alcune altre limitazioni, in SQL Server 2005 sono venute meno grazie alla significativa estensione del linguaggio T-SQL della nuova versione.

Tuttavia esistono degli ambiti applicativi in cui l'uso di codice .NET al-

l'interno del database risulta molto utile. Un primo esempio banale potrebbe essere la validazione del codice fiscale in una query su una tabella di anagrafiche. Ad esempio:

```
SELECT CodSoggetto, Cognome, Nome, CodiceFiscale,
                                    CodiceFiscaleOK(CodiceFiscale)
FROM Soggetti
```

In questo caso la funzione CodiceFiscaleOK accetta come argomento una stringa contenente un codice fiscale e restuisce un BIT (true o false):

```
CREATE FUNCTION [dbo].[CodiceFiscaleOK
(
    @codiceFiscale [varchar(16)
)
RETURNS [dbo].[BIT]
AS
BEGIN
--codice di validazione T-SQL
END
```

Scrive il codice di validazione del codice fiscale è un'operazione possibile in T-SQL ma certamente non proprio banale e comunque e più agevole se fatta, magari, in C#. SQL Server 2005 ci offre proprio quest'ultima possibilità. Ma esistono altri esempi in cui in T-SQL non è proprio possibile risolvere alcune classi di problemi. Si consideri, ad esempio, la possibilità in una query di filtrare le righe da restituire in base al livello di autorizzazione dell'utente che invoca la query, magari attraverso una maschera QBE dell'applicazione, e la verifica di autorizzazione riga per riga non può essere effettuata direttamente da SQL, magari scrivendo la query in JOIN con una tabella autorizzativa del database, ma debba essere un server esterno a validare riga per riga la tupla CodiceRiga, CodiceUtente. Vediamo un esempio semplice:

```
SELECT CodiceRiga, DescrizioneRiga, Quantita, CodiceGruppo
FROM DatiSecretati
INNER JOIN LogCronologico
    ON LogCronologico.CodiceCronologico = DatiSecretati.
                                          CodiceCronologico
        AND  LogCronologico.Data <= GetDate() - 10
WHERE VerificaAutorizzazione(CodiceRiga, @CodiceUtente) = TRUE
```

La query presenta due condizioni di filtro. Una di essa è il vincolo a non restituire le informazioni più vecchie di 10 giorni. Questo filtro è risolto direttamente da SQL Server attraveso una banale INNER JOIN.
Invece, l'altro filtro è gestito dalla funzione VerificaAutorizzazione che, se per il suo lavoro, avesse bisogno di accedere ad un webservice remoto o eseguire del codice .NET specifico, questa query non potrebbe essere scritta. In questo caso l'approccio tradizionale sarebbe quello di far ritornare tutte le righe disponibili, magari gestendo nella clausola WHERE lo scarto delle solo condizioni verificabili da database, e poi verificare applicative tutte le righe nel middle-tier o nel client scartando quelle non valide.
Questo approccio non è certamente sbagliato, ma presenta degli indubbi svantaggi:

- " la logica di autorizzazione e di filtro non è centralizzata ma è suddivisa tra database e codice applicativo;
- " la restituzione dal database di righe che poi verranno comunque scartate produce un inutile overhead prestazionale sia come occupazione di memoria per SQL Server che come occupazione di banda per la trasmissione del result set;
- " potenzialmente presenta problemi di sicurezza perché eventuali dati non disponibili viaggiano comunque sulla rete.

Dal punto di vista del disegno dell'applicazione, poi, ne fa perdere compattezza e rende a più fasi un'operazione come la validazione del dato

che sarebbe idealmente atomica. In generale, al di la dei casi come quello appena mostrato in cui l'uso di .NET è indispensabile a meno di non cambiare l'approccio al problema, si può dire che, nell'ambito delle operazioni eseguibili sia con T-SQL che con .NET, le operazioni di accesso intensivo ai dati sono da preferire se sviluppati direttamente in T-SQL, invece le operazioni di elaborazione intensiva sono da preferire se sviluppati in .NET che in questo ha un indubbio vantaggio prestazionale.

In effetti scrivere stored procedure in .NET che si limitano a far eseguire query SQL al database engine non ha molto senso perché il supporto .NET in SQL Server non è fornisce un modello alternativo di accesso ai dati ma usa T-SQL per recuperarli, analogamente a quanto accade quando di eseguire direttamente il relativo codice T-SQL. Ed è per questo che nell'accesso puro ai dati T-SQL è più veloce, perché è meno mediato.

È altresì evidente che se il codice T-SQL che viene eseguito da una singola stored procedure .NET è molto intensivo e richiede molto tempo, le differenze di prestazione rispetto alla chiamata diretta dello stesso codice T-SQL quasi si annullano perché il tempo di overhead derivante da .NET è minoritario rispetto al tempo complessivo di esecuzione. Tuttavia in questo caso l'approccio .NET resta sconsigliato perché è comunque inutile usare .NET come mero passante di query T-SQL.

2.2 METODOLOGIA ALTERNATIVE PER ESTENDERE SQL SERVER

Prima di addentrarci nei dettagli pratici dell'estensione di SQL Server 2005 con la piattaforma . NET, è opportuno fare una breve digressione nelle due metodologia alternative che permettevano di estendere il prodotto attraverso codice con T-SQL. La prima è supportata direttamente dal produttore e nasce per fare esattamente questo lavoro: stiamo parlando della possibilità di scrivere extended stored procedure in C++. Esisteva però anche un'altra possibilità, meno ortodossa, per fare lo

stesso lavoro: sfruttare la possibilità di invocare oggetti COM di automazione da T-SQL. In questo modo, costruendo un certo pattern basato su funzioni e stored procedure SQL che fungevano da proxy verso l'oggetto COM stesso, era possibilità invocare questi oggetti esterni da codice SQL come normali funzioni o stored procedure.

Le extended stored procedure in C++

Microsoft, già dalle precedenti versioni SQL Server 7 e 2000, ha fornito un meccanismo per estendere le stored procedure attraverso la scrittura di nuove stored procedure non in linguaggio T-SQL, ma bensì in forma di dll binarie scritte in C++. Questa tecnologia è detta Extended Stored Procedure ed era supportata da un comodo wizard fornito dapprima con Visual C++ 6.0 e in seguito con il Visual C++ di Visual Studio .NET 2002 e 2003, dal quale è possibile realizzare un esempio funzionante di extended stored procedure in pochi istanti (Figura 2.1).

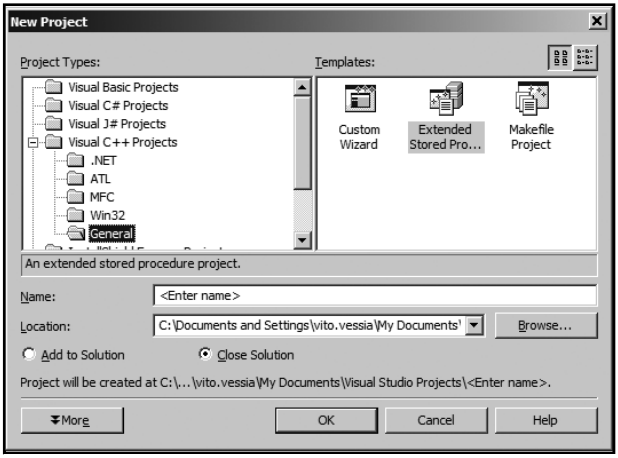


Figura 2.1: Wizard di Visual Studio .NET 2003 per la creazione di extended stored procedure in C++

Il wizard fa tutto il lavoro: prepara lo scheletro di una dll Win32, predi-

spone la funzione di esportazione richiesta dalle extended sp e prepara già un esempio completo del metodo che implementerà la logica della stored procedure. Osserviamone l'esempio basato su quello generato dal wizard e presente nei sorgenti allegati nella cartella \extended_sp_cpp:

```
RETCODE __declspec(dllexport) xp_proc(SRV_PROC *srvproc)
{
   DBSMALLINT i = 0;
   DBCHAR colname[MAXCOLNAME];
     DBCHAR spName[MAXNAME];
     DBCHAR spText[MAXTEXT];
     // Name of this procedure
     _snprintf(spName, MAXNAME, "xp_proc");

     //Set up the column names
   _snprintf(colname, MAXCOLNAME, "ID");
   srv_describe(srvproc, 1, colname, SRV_NULLTERM, SRVINT2,
                        sizeof(DBSMALLINT), SRVINT2, sizeof(DBSMALLINT), 0);
   _snprintf(colname, MAXCOLNAME, "spName");

   // Update field 2 "spName", same value for all rows
   srv_setcoldata(srvproc, 2, spName);
   srv_setcollen(srvproc, 2, static_cast<int>(strlen(spName)));

     // Send multiple rows of data
   for (i = 0; i < 3; i++) {

           // Update field 1 "ID"
           srv_setcoldata(srvproc, 1, &i);

           srv_setcoldata(srvproc, 3, spText);
      srv_setcollen(srvproc, 3, static_cast<int>(strlen(spText)));
```

```
            // Send the entire row
        srv_sendrow(srvproc);
    }
    // Now return the number of rows processed
    srv_senddone(srvproc, SRV_DONE_MORE | SRV_DONE_COUNT,
                                        (DBUSMALLINT)0, (DBINT)i);

    return XP_NOERROR ;
}
```

Certo, con pochi clic un esempio funzionante è già pronto ed è sufficiente copiare la dll generata nella directory \Binn di SQL Server ed eseguire la registrazione della nuova stored procedure in SQL Server con il comando:

```
sp_addextendedproc 'nome_storedprocedure', 'nomelibreria.DLL'
```

Si è così finalmente pronti ad utilizzare la nuova stored procedure, come mostrato in Figura 2.2.

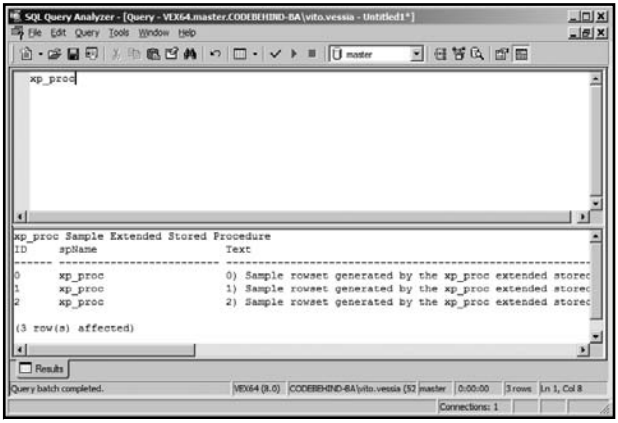


Figura 2.2: L'extended stored procedure in azione

D'altro canto, dalla produzione dell'esempio funzionante proposto in automatico dal template del wizard alla realizzazione di una propria stored procedure da zero in C++ il passo non è così breve. Se poi si considera che non tutti conoscono il C++ o vogliono avere a che fare con il livello di complessità di una soluzione del genere, questa soluzione presenta una complessità elevata.

2.2.1 Chiamate ad oggetti COM da T-SQL

Il database Microsoft SQL Server, già dalle versioni 7.0 e 2000 offre la possibilità di effettuare chiamate COM ad oggetti che supportano l'interfaccia di automazione. A queste funzioni COM, via T-SQL, è possibile passare valori scalari (numeri, stringhe, date e quant'altro ma non interi record) ed è possibile ottenere come risultato un valore scalare. È sostanzialmente quanto si può fare anche con le extended stored procedure, ma da queste è possibile ottenere al ritorno anche record come per le stored procedure tradizionali. Questa limitazione delle chiamate COM potrebbe sembrare rilevante, ma non è poi così grave. Infatti abbiamo osservato in precedenza in questo capitolo quali siano gli ambiti migliori di utilizzo delle estensioni non SQL.

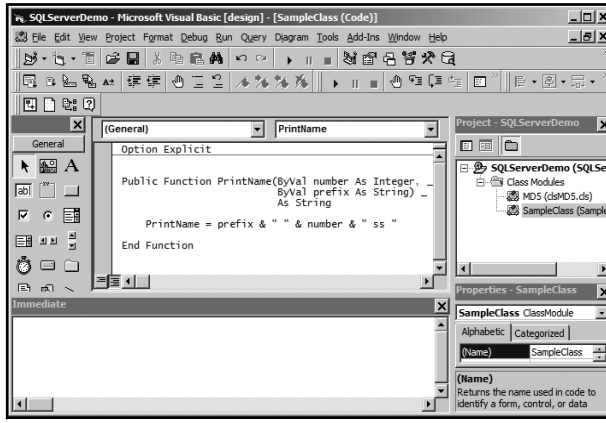


Figura 2.3: La nostra classe Visual Basic 6

Ma è arrivato il momento di procedere con un esempio reale. In T-SQL è certamente possibile effettuare chiamate COM ad oggetti già presenti nel sistema, ma è altresì vero che si può pensare di scrivere propri oggetti COM che verranno richiamati da T-SQL. Ed è quello che faremo scrivendo una semplice DLL COM in Visual Basic 6 (Figura 2.3). Si tratta della "complessissima" classe SampleClass definita nella libreria SQLServerDemo e che pertanto è definita dal ProgId SQLServerDemo.SampleClass. Essa espone il metodo PrintName che accetta in ingresso due parametri number (di tipo intero) e prefix di tipo stringa e restituisce una stringa così composta: prefix & " " & number & " ss ":

```
Public Function PrintName(ByVal number As Integer, _
                    ByVal prefix As String) As String

   PrintName = prefix & " " & number & " ss "

End Function
```

Gli esempi di codice possono essere più o meno intelligenti e questo lo è di sicuro… ma sicuramente è sufficiente al nostro scopo: mostrare l'interazione da T-SQL. La prima operazione da effettuare è la creazione dell'istanza dell'oggetto COM. Eccone la sintassi:

```
sp_OACreate progid, | clsid,    token_oggetto OUTPUT   [ , contesto
                                                ( 1= inprocess, 2=locale(.exe) ]
```

Osserviamo un estratto di codice T-SQL:

```
declare @ServerID INT     --identificativo dell'istanza
declare @OLEResult int    --Hresult della chiamata
declare @errorSource INT  --numero di errore
```

[4] www.ws-i.org [5] http://www.ws-i.org/Profiles/BasicProfile-1.1.html

```
declare @errorDescription varchar(220)  --descrizione dell'errore
EXEC @OLEResult = sp_OACreate 'SQLServerDemo.SampleClass',
                                                    @ServerID OUT
if @OLEResult <> 0
BEGIN
    EXEC sp_OAGetErrorInfo @ServerID, @errorSource OUTPUT,
                                          @errorDescription OUTPUT
END
```

In @ServerID finirà il riferimento all'istanza appena creata con la funzione di sistema sp_OACreate e verrà utilizzato per l'invocazione dei metodi dell'istanza. @OLEResult, invece, conterrà il valore HResult della chiamata, infatti, in caso di valore diverso da 0, e quindi di errore, sarà possibile chiamare la funzione sp_OAGetErrorInfo che restituisce proprio il numero e la descrizione precisa dell'errore:

```
sp_OAGetErrorInfo [ token_oggetto ]    [ , source OUTPUT ]    [ ,
                                                    description OUTPUT ]
  [ , helpfile OUTPUT ]    [ , helpid OUTPUT ]
```

Queste informazioni finiranno nelle variabili @erroreSource ed @errorDescription Naturalmente questa eccezione può anche essere prodotta applicativamente all'interno del nostro codice Visual Basic 6. Se l'istanziazione è andata a buon fine, possiamo effettuare la chiamata al metodo PrintName della nostra dll con la seguente sintassi:

```
sp_OAMethod token_oggetto,    nome_metodo    [ , valore_ritorno
                                                          OUTPUT ]
  [ , [ @nome_parametro = ] valore_parametro [ OUTPUT ]    [ ...n ] ]
Ed ecco come effettuare la chiamata al nostro metodo PrintName:
declare @prefixToPrint INT   --prefisso da passare alla funzione Visual Basic
declare @number int          --numero da passare alla funzione Visual Basic
declare @retString varchar(8000  --variabile che conterrà il valore
```

```
                                                di ritorno della funzione
set @prefixToPrint = 'prova'
set @number = 15

--esecuzione della chiamata al metodo PrintName della classe Visual Basic
                                                di esempio
EXEC @OLEResult = sp_OAMethod @ServerID, 'PrintName', @retString
                OUTPUT, @number = @numToPrint, @prefix = @prefixToPrint

print @retString  --il risultato atteso sarà:   prova 15 ss
```

Abbiamo dichiarato le due variabili che conterranno i valori da passare al metodo PrintName, essere veranno passate con la sintassi @nomeParametro = valore, dove @nomeParametro è proprio il nome del parametro così come definito nella classe Visual Basic 6. Per il valore di ritorno, invece, è sufficiente aggiungere il postfisso OUTPUT. Si noti che non è necessario passare i parametri nell'ordine previsto dalla funzione da chiamare perché la convenzione di passaggio è basata sui nomi e non sulla posizione. È l'equivante di usare la seguente sintassi cara ai programmatori Visual Basic:

```
Dim o As Object
Set o = CreateObject("SQLServerDemo.SampleClass")

Dim value as String
value = o.PrintName( prefix:= "ciro", number:= 544 )
```

Tornando alla nostra chiamata da T-SQL, se non si sono verificati errori, comunque tracciabili da una successiva chiamata a sp_OAGetErrorInfo, è possibile leggere o stampare il valore di ritorno della funzione VB6. Terminata la chiamata alla classe, non ci resta che rilasciare correttamente le risorse (l'istanza stessa della classe) al fine di evitare fastidiosi memory leak. Di questo si occupa un'altra stored procedure di

sistema, la sp_OADestroy:

```
sp_OADestroy objecttoken
```

Ed eccone l'uso nel nostro esempio reale:

```
EXEC @OLEResult = sp_OADestroy @ServerID
```

Per effetto della location transparency, possiamo senza nessun problema effettura il debug della classe Visual Basic 6 direttamente dalla chiamata T-SQL. Sarà suffciente aprire il sorgente del progetto della DLL dall'IDE di Visual Basic 6, mettere l'IDE in run con l'opzione Wait for components to be created, impostare un breakpoint all'inizio del metodo PrintName, effettuare la chiamata della funzione da T-SQL e attendere che Visual Basic 6 si attivi con l'istruzione col brealpoint i bella mostra...
Sebbene il meccanismo fin qui mostrato sia realmente molto potente e permetta di ottenere risultati straordinari con poco sforzo, potrebbe risultare un po' macchinoso data la complessità della sintassi prevista. Con l'approccio alla C++ la chiamata ad una extended stored procedure è indistinguibile dalla chiamata ad una stored procedure tradizionale. E allora perché non provare a semplificarsi la vita?
In SQL Server è prevista la possibilità di scrivere delle proprie funzioni richiamabili da query, stored procedure e, più in generale, da T-SQL. Ecco un esempio di user function che esegue la somma di due numeri:

```
CREATE FUNCTION dbo.MySum
(
    @num1 float,
    @num2 float
)
RETURNS float
AS
```

```
BEGIN
    return @num1 + @num2
END
```

Dovremo creare questa funzione all'interno di un database SQL Server, dopo di che ne sarà possibile la chiamata:

```
print dbo.MySum(2.3, 5.9)
--il risultato atteso è:  8.2
```

Potremo anche chiamare la funzione da una query:

```
select Campo1, dbo.MySum(Campo2, Campo3) as Somma from Tabella
```

E allora perché non creare una funzione che incapsuli la chiamata alla nostra PrintName? Detto fatto:

```
CREATE FUNCTION dbo.GetNumber
(
    @numToPrint int,
    @prefixToPrint varchar(255)
)
RETURNS varchar(255)
AS
BEGIN
    declare @OLEResult int,
    @ServerID INT,
    @retString varchar(8000),
    @errorSource INT,
    @errorDescription varchar(220)


    EXEC @OLEResult = sp_OACreate 'SQLServerDemo.SampleClass',
```

```
                                                    @ServerID OUT
    if @OLEResult<>0
    BEGIN
        EXEC sp_OAGetErrorInfo @ServerID, @errorSource OUTPUT,
                                                @errorDescription OUTPUT
        RETURN -1
    END

    EXEC @OLEResult = sp_OAMethod @ServerID, 'PrintName',
@retString OUTPUT, @number = @numToPrint, @prefix = @prefixToPrint
    IF (@OLEResult <> 0)
    BEGIN

        EXEC sp_OAGetErrorInfo @ServerID, @errorSource OUTPUT,
                                                @errorDescription OUTPUT

        RETURN @errorDescription
    END
    ELSE
    BEGIN
        RETURN @retString
    END

    EXEC @OLEResult = sp_OADestroy @ServerID
    IF @OLEResult <> 0
    BEGIN
        EXEC sp_OAGetErrorInfo @ServerID, @errorSource OUTPUT,
                                                @errorDescription OUTPUT

        RETURN @errorDescription
    END
RETURN @retString
END
```

Questa funzione così come tutti gli altri esempi fin qui mostrati, è stata creata all'interno di una versione modificata del database Northwind di SQL Server. Un backup di questa versione accompagna i sorgenti allegati. Pertanto tutte le prove verranno effettuate puntando a questo database. Ed ecco, infatti, un'esempio di chiamata alla nostra funzione in una query:

```
select OrderID, CustomerID, dbo.GetNumber(OrderID, 's') as CampoCOM
                                                        from orders
```

Beh, è tutt'altra cosa rispetto alla complessità vista in precedenza. Naturalmente la chiamata a questa funzione potrà essere incapsulata in un'altra funzione:

```
CREATE FUNCTION dbo.GetNumberProxy
(
    @numToPrint int,
    @prefixToPrint varchar(255)
)
RETURNS varchar(255)
AS
BEGIN
    RETURN dbo.GetNumber(@numToPrint, 'Proxy ' + @prefixToPrint)
END
```

O in una stored procedure, colmando finalmente il gap rispetto alle estendend sp:

```
CREATE PROCEDURE GetOrdersEx
    @MinimunFreight int = 0
AS
select orderid, shipname, freight, dbo.GetNumberProxy(orderid, shipname)
```

```
                                                    as CampoFunzione
from orders where freight >= @MinimunFreight
GO
```

2.2.2 Vantaggi dell'approccio .NET rispetto alle due precedenti tecnologie

Il primo e più importante vantaggio dell'approccio .NET rispetto alle due soluzioni che abbiamo appena visto, oltre alla maggiore semplicità, almeno rispetto all'approccio C++ e alla minore macchinosità rispetto alla soluzione Visual Basic 6, è certamente la sicurezza: il codice .NET è codice gestito e, come vedremo in seguito, quando si registrata la stored procedure o la funzione .NET in SQL Server è possibile autorizzarne il livello di sicurezza, che può esser SAFE, EXTERNAL_ACCESS, o UNSAFE. Impostando la massima restrizione (SAFE), sarà impossibile che il codice .NET chiamato da SQL acceda a risorse esterne (ad esempio il file system o le connessioni di rete) o che faccia altre operazioni extra database, se non siamo noi ad autorizzarle assegnando all'oggetto .NET un livello più alto. Né l'approccio C++, né l'approccio VB6 offrono questo indubbio vantaggio. Con EXTERNAL_ACCESS si può sostanzialmente accedere a qualsiasi risorse interna ed esterna al database, ma non è consentita l'esecuzione di codice unmanaged, opzione invece consentita al livello UNSAFE.

In C++ è poi necessario aprire una connessione esplicita al database per accedere ai dati, nonostante la procedura giri all'interno del database stesso e questo porta ad un evidente overhead prestazionale, oltre che ad una robustezza complessiva maggiore. Inoltre SQL Server è in grado di controllare il livello di utilizzo delle risorse (memoria e CPU) da parte di una stored procedure .NET invocata, ed eventualmente ridurne o bloccarne l'utilizzo, ma non può fare lo stesso con le XP C++ o con le chiamate COM.

Con .NET, poi, è possibile definire tipi di dati aggiuntivi che sono implementabili direttamente come classi . NET.

Anche il presunto vantaggio prestazionale del codice compilato C++,

o persino del compilato VB6, che produce un codice binario X86 Win32 di buon livello, rispetto al codice eseguito dalla macchina virtuale .NET, nei fatti si dimostra non vero, sia perché il Jitter (Just In Time Compiler) di .NET fa un ottimo lavoro e sia perché l'eventuale vantaggio residuale è azzerato completamente dall'overhead di accesso a SQL Server all'interno di codice XP C++ e COM rispetto alla modalità meno mediata possibile con .NET.

2.3 IL SUPPORTO .NET 2.0 PER SQL SERVER 2005

Visual Studio 2005 offre il supporto integrato alla scrittura, al debugging e al test di stored procedure, funzioni e trigger .NET in SQL Server 2005. In realtà è possibile anche scrivere componenti CLR per SQL Server 2005 munendosi di SQL Server 2005 Express, Microsoft .NET Framework SDK e, opzionalmente, di Microsoft Visual C# Express 2005, il tutto a costo zero, ma perdendo la possibilità di effettuare un deployment ultra semplificato e, soprattutto, la possibilità di eseguire il debugging del codice CLR in esecuzione su SQL Server.
Pertanto accenneremo soltanto a questa possibilità ma ci concentreremo sul comodo ambiente a pagamento di Visual Studio 2005 Professional.

2.3.1 Preparazione del database di esempio

Piuttosto che descrivere teoricamente il tutto, procederemo con la creazione di un piccolo database di anagrafiche soggetti in cui applicare il supporto CLR per effettuare la verifica e il calcolo del Codice Fiscale, che è sempre un cavallo di battaglia degli articolisti sviluppatori italiani.
Innanzitutto predisponiamo la creazione di un nuovo database "Libro" e di un paio di tabelle Soggetti e Localita in relazione tra loro (la località di nascita del soggetto è in foreign key con la tabella delle località).

Ecco dunque gli script di creazione commentati:

```
USE master
GO
--creazione del database di esempio
CREATE DATABASE [Libro]
GO
--creazione della tabelle delle località: conterrà una coppia di valori
                                        NomeLocalità, CodiceBelfiore
--necessaria al calcolo del Codice Fiscale
CREATE TABLE [dbo].[Localita](
    [Localita] [nvarchar](50) COLLATE Latin1_General_CI_AS NOT NULL,
    [CodiceBelfiore] [nchar](4) COLLATE Latin1_General_CI_AS NULL,
 CONSTRAINT [PK_Localita_1] PRIMARY KEY CLUSTERED
(
    [Localita] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
                    IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON,
                            ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
GO
--creazione della tabella dei soggetti: contiene le informazioni di base per il
                                        calcolo del codice fiscale e
--quindi Cognome, Nome, Sesso, Data di nascita, Località di nascita
CREATE TABLE [dbo].[Soggetti](
    [IdSoggetto] [int] IDENTITY(1,1) NOT NULL,
    [Cognome] [varchar](50) COLLATE Latin1_General_CI_AS NULL,
    [Nome] [varchar](50) COLLATE Latin1_General_CI_AS NULL,
    [LocalitaNascita] [nvarchar](50) COLLATE Latin1_General_CI_AS
                                                        NULL,
    [CodiceFiscale] [varchar](16) COLLATE Latin1_General_CI_AS NULL,
    [Sesso] [nchar](1) COLLATE Latin1_General_CI_AS NULL,
    [DataNascita] [datetime] NULL,
```

```
 CONSTRAINT [PK_Soggetti] PRIMARY KEY CLUSTERED
(
      [IdSoggetto] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
                        IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON,
                                    ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
GO
--la località di nascita è in foreign key con la tabella delle località
ALTER TABLE [dbo].[Soggetti]  WITH CHECK ADD  CONSTRAINT
                          [FK_Soggetti_Localita] FOREIGN KEY([LocalitaNascita])
REFERENCES [dbo].[Localita] ([Localita])
GO
ALTER TABLE [dbo].[Soggetti] CHECK CONSTRAINT [FK_Soggetti_Localita]
GO
--il campo Sesso, di tipo NCHAR(1), può contenere solo i valori 'M' o 'S' e
                                                                  pertanto viene
--introdotto un vincolo Check Constraint per accertarsi della consistenza dei
                                                                  dati inseriti
ALTER TABLE [dbo].[Soggetti]  WITH CHECK ADD  CONSTRAINT
                            [CK_Soggetti] CHECK  (([Sesso]='F' OR [Sesso]='M'))
GO
ALTER TABLE [dbo].[Soggetti] CHECK CONSTRAINT [CK_Soggetti]
GO
```

Al fine di garantirci la possibilità di effettuare alcune prove, inseriamo alcune località di esempio nella tabella Localita con il relativo Codice Belfiore:

```
INSERT INTO Localita
 VALUES (N'Bari', N'A662')
INSERT INTO Localita
 VALUES (N'Foligno', N'D653')
```

```
INSERT INTO Localita
 VALUES (N'Verona', N'A737')
GO
```

Facciamo altrettanto con i soggetti inserendone alcuni. Siccome lo scopo dell'esempio e mostrare un esempio d'uso del CLR .NET all'interno di SQL Server 2005 per verificare e calcolare il codice fiscale, operazione tipicamente effettuata nel codice di backend o di frontend dell'applicazione, ma difficilmente realizzabile direttamente in codice T-SQL, alcuni dei soggetti che inseriremo avranno il codice fiscale errato:

```
INSERT INTO Soggetti
 (Cognome, Nome, LocalitaNascita, CodiceFiscale, Sesso, DataNascita)
 VALUES ('Vessia', 'Vito', N'Bari', 'VSSVSI74M30A662W', N'M', '1974-08-
                                                    30 00:00:00.000')
INSERT INTO Soggetti
 (Cognome, Nome, LocalitaNascita, CodiceFiscale, Sesso, DataNascita)
 VALUES ('Rallo', 'Daniela', N'Verona', 'RLLDNL74D49L781Q', N'F', '1974-
                                                    04-09 00:00:00.000')
INSERT INTO Soggetti
 (Cognome, Nome, LocalitaNascita, CodiceFiscale, Sesso, DataNascita)
 VALUES ('Rossi', 'Mario', N'Bari', 'RSSMRA35A01A662T', N'M', '1935-01-
                                                    01 00:00:00.000')
INSERT INTO Soggetti
 (Cognome, Nome, LocalitaNascita, CodiceFiscale, Sesso, DataNascita)
 VALUES ('Esposito', 'Cira', N'Foligno', 'SPSCRI70E59D653J', N'F', '1970-
                                                    05-19 00:00:00.000')
INSERT INTO dbo.Soggetti
 (Cognome, Nome, LocalitaNascita, CodiceFiscale, Sesso, DataNascita)
 VALUES ('Vessia', 'Antonio', N'Bari', 'VSSNTN79S23A662X', N'M', '1979-
                                                    11-23 00:00:00.000')
INSERT INTO dbo.Soggetti
 (Cognome, Nome, LocalitaNascita, CodiceFiscale, Sesso, DataNascita)
 VALUES ('Armani', 'Federica', N'Verona', 'RMNFRC79S50L781F', N'F',
```

```
                                        '1979-11-10 00:00:00.000')
GO
```

L'ultima operazione da effettuare in preparazione all'esempio, è proprio l'attivazione del supporto al CLR. Di default i database SQL Server 2005 hanno questo supporto disattivato e quindi va attivato esplicitamente con la seguente sequenza di stored procedure di sistema:

```
sp_configure "CLR ENABLED", 1
GO
reconfigure
GO
```

L'eventuale disattivazione si effettua lanciando la stessa sequenza di comandi ma con il parametro "CLR ENABLED" impostato a 0.

2.3.2 Il progetto in Visual Studio 2005

Tutti i nostri esempi saranno contenuti in un unico progetto Visual Stu-

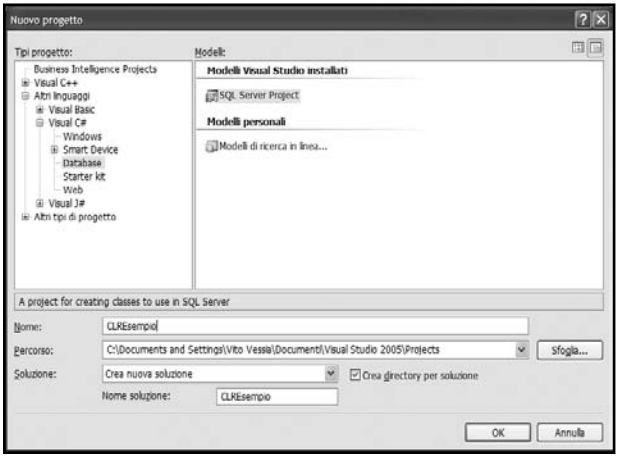


Figura 2.4: Il nuovo progetto CLR SQL Server 2005 da Visual Studio 2005

dio 2005 scritto in C#. Dunque procediamo alla sua creazione. Dopo aver aperto l'IDE, creiamo un nuovo progetto Visual C# –> Database –> SQL Server Project (Figura 2.4) che chiameremo CLREsempio.
A questo punto, se non è presente nessun'altra connessione al database tra quelle conservate nell'IDE, verrà proposta la creazione di una

Figura 2.5: La connessione a SQL Server 2005

nuova connessione a SQL Server 2005, come mostrato in Figura 2.5.

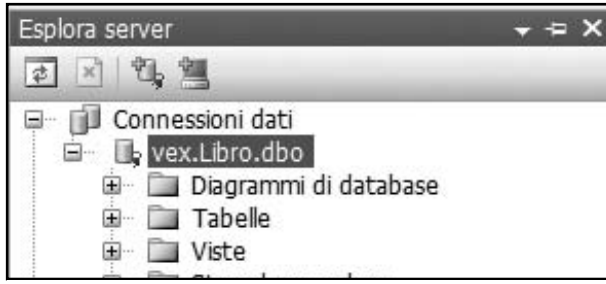


Figura 2.6: Gestione del database da Visual Studio 2005

Da questo momento, nel pannello Esplora server di Visual Studio 2005 (Figura 2.6) potremo visualizzare, gestire e modificare il nostro database di esempio quasi analogamente a quanto facciamo da SQL Server Management Studio.

Il nuovo progetto si presenterà vuoto e senza file di codice sorgente C#. Sarà presente la sola cartella Test Scripts, contenente un file .SQL di script T-SQL di esempio, che esamineremo nel proseguo. Sono già presenti tutti i riferimenti necessari al progetto e probabilmente vi stupirà sapere che si compongono dei soli assembly mscorelib.dll, System.dll e System.Data.Dll, cioè esattamente gli stessi che usate nella quasi totalità di applicazione .NET che scrivete. Il trucco consiste nel fatto che l'intero supporto alle estensioni CLR per SQL Server 2005 è contenuto nello stesso assembly di ADO.NET a cui è stato aggiunto un intero nuovo namespace Microsoft.SqlServer.Server, nella versione 2.0 del Microsoft .NET Framework. Dunque avete già tutto il supporto necessario senza dover nemmeno installare SQL Server 2005!

2.4 FUNZIONI IN CLR

Siamo pronti ad aggiungere la nostra prima estensione CLR nel database di esempio. Selezionando il progetto appena creato dal pannello

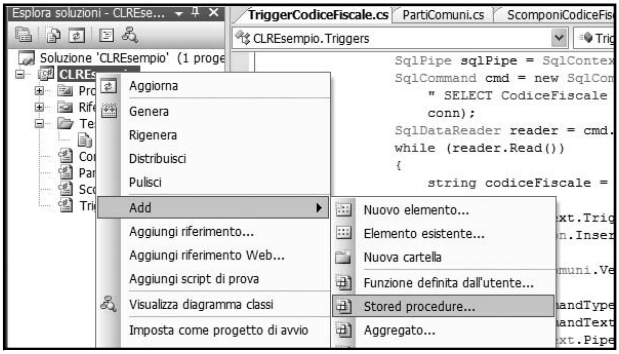


Figura 2.7: Aggiunta di un oggetto al progetto

Solution Explorer e attivando il menù contestuale con il tasto destro del mouse, sceglieremo la voce Add –> Funzione definita dall'utente. Visual Studio provvederà ad aggiungere un nuovo file .CS al progetto e a produrre lo scheletro di una funzione CLR richiamabile da SQL Server 2005 (Figura 2.7).
La nostra prima funzione si chiamerà ControlloCodiceFiscale, accetterà come argomento una stringa codiceFiscale e ritornerà un boolean per affermare la bontà o meno del codice fiscale; in pratica, per quei due o tre lettori che hanno vissuto finora in Canada e che non sanno quale algoritmo si cela dietro il calcolo del codice fiscale italiano, cerchiamo di riepilogarlo brevemente. Esso è costituito da una stringa alfanumerica di 16 caratteri, univoca per ciascun cittadino italiano (l'Agenzia delle Entrate gestisce le omonime generando codici fiscali fuori algoritmo ed è questa la ragione per cui l'unica vero codice fiscale è quello rilasciato da loro e con quello auto-calcolato non vi è la certezza dell'assoluta bontà). La descrizione dell'algoritmo verrà omessa perché, come potrete ben figurarvi, non è di certo argomento di questo volume. Per i più curiosi, però, esiste sempre il metodo C# CalcolaCodiceFiscale lo implementa e che dunque è un buon punto di partenza. La stringa si compone come:

```
CCC NNN AA M DD BBBB X
```

CCC contiene tre lettere del cognome, NNN contiene tre lettere del nome, AA è l'anno di nascita, M è la lettera corrisponde al mese di nascita, secondo una tabella di decodifica specifica dell'algoritmo, DD è il giorno di nascita (per le donne a DD si somma 40 e quindi DD ha la doppia valenza di indicare il giorno di nascita e il sesso), BBBB è il codice belfiore del comune di nascita, una codifica per tutti i comuni italiani e i principali stati mondiali per coloro che non sono nati in Italia, X rappresenta il codice di controllo e sta ad indicare la corretta consistenza sintattica del codice per segnalare eventuali errori di trascrizione del codice stesso. Ed è proprio su questo codice di controllo che si baserà la

nostra prima funzione SQL Server 2005 CLR scritta in C#. Osserviamone il sorgente:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.Text.RegularExpressions;
using System.Collections;
using Microsoft.SqlServer.Server;

namespace CLREsempio
{
   public partial class Controllo
   {

      [Microsoft.SqlServer.Server.SqlFunction]
      public static SqlBoolean ControlloCodiceFiscale(string codiceFiscale)
      {
         return new
               SqlBoolean(PartiComuni.VerificaCodiceFiscale(codiceFiscale));
      }
   }
}
```

La funzione è statica ed è inserita nella classe Controllo. Sulla funzione è definito l'attributo Microsoft.SqlServer.Server.SqlFunction che indica proprio che si comporterà come una funzione di SQL Server 2005. I tipi standard di .NET sono comunque utilizzabili nella definizione della firma del metodo (argomenti e valore di ritorno), ma è da preferire l'uso dei tipi specifici di SQL Server. Infatti il valore di ritorno boolean della funzione è espresso con un SqlBoolean. Il corpo della funzione è minimale perché contiene solo la chiamata ad un altro metodo statico (VerificaCodiceFiscale)

contenuto in una classe costituita da parti di logica comune alle varie estensioni CLR di esempio che introdurremo in questo capitolo. Dunque riportiamo il codice di questa funzione statica:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.Text.RegularExpressions;
using System.Collections;
using System.Collections.Generic;
using Microsoft.SqlServer.Server;

namespace CLREsempio
{
   class PartiComuni
   {
      private static readonly Hashtable pari = new Hashtable();
      private static readonly Hashtable dispari = new Hashtable();
      private static readonly Hashtable controllo = new Hashtable();
      private static readonly Regex reConsonanti =
         new Regex("b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z",
                                                RegexOptions.IgnoreCase);
      private static readonly Regex reVocali =
         new Regex("a|e|i|o|u", RegexOptions.IgnoreCase);
      private static readonly Regex reLettere =
         new Regex("a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z",
                                                RegexOptions.IgnoreCase);
      private static readonly Regex reCifre =
         new Regex("\\d", RegexOptions.IgnoreCase);
      private static readonly char[] mesi =
         { 'A', 'B', 'C', 'D', 'E', 'H', 'L', 'M', 'P', 'R', 'S', 'T' };
```

```
internal static void initializeValues()
{
    if (pari.Count > 0) return;

    pari.Add('0', 0);       pari.Add('1', 1);       pari.Add('2', 2);
                                                    pari.Add('3', 3);
    pari.Add('4', 4);       pari.Add('5', 5);       pari.Add('6', 6);
                                                    pari.Add('7', 7);
    pari.Add('8', 8);       pari.Add('9', 9);       pari.Add('A', 0);
                                                    pari.Add('B', 1);
    pari.Add('C', 2);       pari.Add('D', 3);       pari.Add('E', 4);
                                                    pari.Add('F', 5);
    pari.Add('G', 6);       pari.Add('H', 7);       pari.Add('I', 8);
                                                    pari.Add('J', 9);
    pari.Add('K', 10);      pari.Add('L', 11);      pari.Add('M', 12);
                                                    pari.Add('N', 13);
    pari.Add('O', 14);      pari.Add('P', 15);      pari.Add('Q', 16);
                                                    pari.Add('R', 17);
    pari.Add('S', 18);      pari.Add('T', 19);      pari.Add('U', 20);
                                                    pari.Add('V', 21)
  pari.Add('W', 22);        pari.Add('X', 23);      pari.Add('Y', 24);
                                                    pari.Add('Z', 25);

    dispari.Add('0', 1);    dispari.Add('1', 0);    dispari.Add('2',
                                                5); dispari.Add('3', 7);
  dispari.Add('4', 9);      dispari.Add('5', 13);   dispari.Add('6',
                                                15); dispari.Add('7', 17);
dispari.Add('8', 19);       dispari.Add('9', 21);   dispari.Add('A',
                                                1); dispari.Add('B', 0);
  dispari.Add('C', 5);      dispari.Add('D', 7);    dispari.Add('E',
                                                9); dispari.Add('F', 13);
dispari.Add('G', 15);       dispari.Add('H', 17);   dispari.Add('I',
                                                19); dispari.Add('J', 21);
```

```
        dispari.Add('K', 2);          dispari.Add('L', 4);          dispari.Add('M',
                                                    18);          dispari.Add('N', 20);
        dispari.Add('O', 11);         dispari.Add('P', 3);          dispari.Add('Q',
                                                    6);           dispari.Add('R', 8);
        dispari.Add('S', 12);         dispari.Add('T', 14);         dispari.Add('U',
                                                    16);          dispari.Add('V', 10);
        dispari.Add('W', 22);         dispari.Add('X', 25);         dispari.Add('Y',
                                                    24);          dispari.Add('Z', 23);

        controllo.Add(0, 'A');          controllo.Add(1, 'B');
                                controllo.Add(2, 'C');          controllo.Add(3, 'D');
        controllo.Add(4, 'E');          controllo.Add(5, 'F');
                                controllo.Add(6, 'G');          controllo.Add(7, 'H');
        controllo.Add(8, 'I');          controllo.Add(9, 'J');
                              controllo.Add(10, 'K');           controllo.Add(11, 'L');
        controllo.Add(12, 'M');           controllo.Add(13, 'N');
                              controllo.Add(14, 'O');           controllo.Add(15, 'P');
        controllo.Add(16, 'Q');           controllo.Add(17, 'R');
                               controllo.Add(18, 'S');          controllo.Add(19, 'T');
        controllo.Add(20, 'U');           controllo.Add(21, 'V');
                             controllo.Add(22, 'W');           controllo.Add(23, 'X');
        controllo.Add(24, 'Y');           controllo.Add(25, 'Z');
    }

    internal static bool VerificaCodiceFiscale(string codFiscale)
    {
        initializeValues();

        string codfisc = codFiscale.ToUpper();
        int codicecontrollo = 0;

        try
        {
```

```
            if (codfisc.Length == 16)
            {
                for (int i = 0; i < 15; i++)
                {
                    if ((i + 1) % 2 == 0)
                        codicecontrollo = codicecontrollo + (int)pari[codfisc[i]];
                    else
                     codicecontrollo = codicecontrollo + (int)dispari[codfisc[i]];
                }

                if ((char)controllo[(codicecontrollo % 26)] != codfisc[15])
                    return false;
                else
                    return true;
            }
            else
                return false;
        }
        catch (Exception ex)
        {
            throw new Exception(ex.Message);
        }
    }
  }
}
```

Siamo dunque pronti a compilare, fare l'upload dell'assembly su SQL Server, registrare la funzione e testare il codice della stessa direttamente da codice T-SQL. Semplicemente selezionando il progetto dal Solution Explorer, attivando il menù contestuale e scegliendo la voce Distribuisci, effettueremo le prime tre operazioni (compilazione, upload dell'assembly e registrazione della funzione). In realtà le effettuerà in automatico per noi Visual Studio 2005 che dispone di una connes-

sione diretta al nostro database impostata nelle fasi iniziali, come ricorderete.

2.4.1 Registrazione dell'assembly e della funzione senza Visual Studio 2005 Professional

Se, invece, non si disponesse di Visual Studio 2005 Professional, bisognerebbe compilare il progetto (a riga di comando direttamente con il compilatore CSC oppure, magari, con Visual C# Express 2005) e, supponendo che l'assembly si chiami CLREsempio.dll e che si trovi in C:\, lanceremo il nuovo comando CREATE ASSEMBLY:

```
CREATE ASSEMBLY CLR2005
FROM 'C:\CLREsempio.dll'
WITH PERMISSION_SET=SAFE
```

Riconoscerete il PERMISSION_SET che è possibile impostare a SAFE, EXTERNAL_ACCESS e UNSAFE, come descritto in precedenza. A questo punto possiamo registrare la nuova funzione in SQL Server definendo una firma esterna in codice T-SQL in modo che possa essere usata da codice T-SQL:

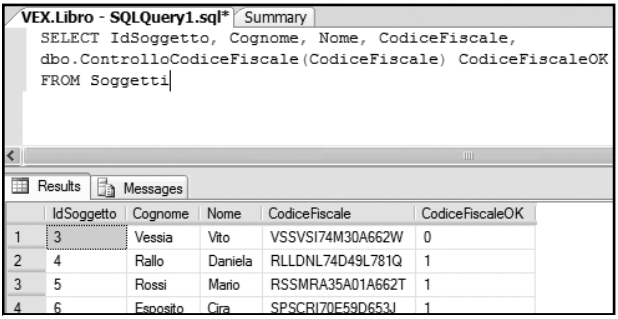
```
CREATE FUNCTION [dbo].[ControlloCodiceFiscale](@codiceFiscale
                                                [nvarchar](4000))
RETURNS [bit] WITH EXECUTE AS CALLER
AS
EXTERNAL NAME
        [CLREsempio].[CLREsempio.Controllo].[ControlloCodiceFiscale]
GO
```

2.4.2 Test della funzione

Possiamo utilizzare la funzione all'interno di normale codice T-SQL come quello della seguente query:

```
SELECT IdSoggetto, Cognome, Nome, CodiceFiscale,
dbo.ControlloCodiceFiscale(CodiceFiscale) CodiceFiscaleOK
FROM Soggetti
```

La funzione è usata per restituire un campo nella clausola SELECT, analogamente a quanto facciamo con le funzioni native. In Figura 2.8 è possibile osservare il resultset della query di esempio. Il campo Codice-



	IdSoggetto	Cognome	Nome	CodiceFiscale	CodiceFiscaleOK
1	3	Vessia	Vito	VSSVSI74M30A662W	0
2	4	Rallo	Daniela	RLLDNL74D49L781Q	1
3	5	Rossi	Mario	RSSMRA35A01A662T	1
4	6	Esposito	Cira	SPSCRI70E59D653J	1

Figura 2.8: La query che richiama la funzione CLR

FiscaleOK contiene i valori 0 (codice fiscale erratto) o 1 (codice fiscale corretto). In effetti, tra i record inseriti con le INSERT iniziali, vi è un soggetto con codice fiscale errato.

2.4.3 Debug della funzione

Noi programmatori moderni siamo sempre meno capaci, ahinoi, di scrivere codice senza il debugger. La cosa non è tanto grave in sé per ragioni etiche, deontologiche o estetiche, perché tutto sommato i debugger sono comodi e sono fatti a posta per semplificarci la vita, ma perché è l'indice dell'incapacità di figurarsi il flusso di esecuzione del codice senza prima eseguirlo step by step. Ma siccome le questioni filosofiche sono off topic esattamente come lo è l'algoritmo del codice fiscale, trala-

sciamo queste considerazioni per sottolineare, invece, come il supporto integrato offerto da Visual Studio 2005 Professional allo sviluppo di oggetti CLR per SQL Server 2005 contempla pure un potente debugger. In pratica SQL Server esegue il vostro codice CLR all'interno della sua macchina virtuale .NET e voi potete effettuarne il debugging dal vostro comodo IDE.

Come fare? Il file Test.sql contenuto nella cartella Test Scripts del progetto ci viene incontro. Infatti, riportando al suo interno il codice T-SQL in grado di richiamare la nostra nuova funzione (ad esempio la query precedente) e semplicemente impostando un breakpoint all'ingresso della funzione C#, con la semplice pressione del tasto F5, che avvia l'esecuzione dei progetti in Visual Studio, l'IDE eseguira il codice T-SQL in

Figura 2.9: A kind of managed magic

script.sql, che conterrà l'attivazione della funzione e che quindi potrà essere seguita in debug dall'IDE, analogamente a quanto avviene con le applicazioni desktop o web tradizionali. In Figura 2.9 si mostra la magia...

2.5 STORED PROCEDURE IN CLR

La prima funzione era volutamente semplice e non faceva uso di risorse del database o esterne perché tutto ciò che le serviva era contenuto nell'argomento ad essa passato. La scrittura di una stored procedure CLR è un'operazione altrettanto semplice e apparirebbe ripetitiva, pertanto facendolo ne approfitteremo per introdurre un maggiore livello di complessità. Selezionando il progetto appena creato dal pannello Solution Explorer e attivando il menù contestuale con il tasto destro del mouse, sceglieremo la voce Add –> Stored Proceure. La nostra stored procedure si chiama ScomponiCodiceFiscale, accetta un parametro IdSoggetto e restituisce un resultset contenente tutti i campi della tabella Soggetti, eventualmente filtrati secondo la condizione IdSoggetto = @IdSoggetto OR @IdSoggetto = -1, più due nuovi campi calcolati:

- " CodiceFiscaleCalcolato (il codice fiscale ricalcolato a partire dai campi Cognome, Nome, Sesso, DataNascita e LocalitaNascita)
- " Corretto (un campo boolean che indica se il valore ricalcolato corrisponde al codice fiscale memorizzato in tabella).

Dunque, nella stored procedure dovremo:

- " recuperare tutte le righe di Soggetti secondo la condizione di filtro indicata dall'argomento IdSoggetto;
- " per ciascun riga eseguire il calcolo del codice fiscale a partire dai campi Cognome, Nome, Sesso, DataNascita e LocalitaNascita;
- " verificare se il codice fiscale ricalcolato corrisponda a quello fisicamente salvato nella riga;
- " restituire l'intero resultset costituito dai campi della tabella più i due nuovi calcolati.

L'operazione sembrerebbe sostanzialmente semplice, ma è complicata dal fatto che dobbiamo eseguire internamente alla procedure un accesso al database sulla tabella Soggetti, operazione che non era neces-

saria nella funzione di esempio vista in precedenza e anche dal fatto che per il calcolo del codice fiscale non serve il nome della località di nascita ma bensì il corrispondente codice belfiore che, per effetto della Terza Forma Normale, è stato portato nella tabella Localita. Infatti Soggetti contiene solo il campo LocalitaNascita: il CodiceBelfiore va recuperato dall'altra tabella.
Dunque abbiamo tutti gli elementi per partire.

2.5.1 Il codice

Osserviamo il codice. Anche in questo caso il metodo che soggiace alla stored procedure è statico ed è incorniciato da un attributo Microsoft.SqlServer.Server.SqlProcedure. Esso è completamente commentato per una più facile comprensione, pertanto verrano evitati ulteriori commendi fuori dal codice:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

namespace CLREsempio
{
public partial class StoredProcedures
{
[Microsoft.SqlServer.Server.SqlProcedure]
public static void GetSoggettoConCodiceFiscale(int idSoggetto)
{
   //viene caricata la cache dei codici belfiore
   PartiComuni.CacheCodiciBelfiore();

   //si apre una connessione al database per eseguire la query
   //la connectionstring è semplicemente "context connection=true"
```

```
//che consente di agganciarsi alla connessione SQL Server
//corrente da cui è stata lanciata la stored procedure
using (SqlConnection conn =
    new SqlConnection("context connection=true"))
{
    //apertura dela connessione
    conn.Open();
    //recupero del Pipe, cioè del canale di comunicazione con SQL Server
    SqlPipe pipe = SqlContext.Pipe;

    //esecuzione della query che recupera i dati da Soggetti
    //filtrati secondo il paramatro passato come argomento alla sp
    SqlCommand cmd = new SqlCommand(
        " SELECT IdSoggetto, Cognome, Nome, LocalitaNascita, " +
        " CodiceFiscale, Sesso, DataNascita " +
        " FROM Soggetti " +
        " WHERE IdSoggetto = @IdSoggetto OR @IdSoggetto = -1",
        conn);
    //valorizzazione del parametro @IdSoggetto
    cmd.Parameters.AddWithValue("@IdSoggetto", idSoggetto);
    //apertura di una DataReader per il recupero del resultset
    SqlDataReader reader = cmd.ExecuteReader();

    //definizione della struttura del resultset
    //vengono indicati tutti i campi e il corrispondente tipo
    //la definzione è posizionale e pertanto i campi del resultset
    //si ritroveranno esattamente nello stesso ordine
    SqlMetaData[] metadata = new SqlMetaData[]
    {
        new SqlMetaData("IdSoggetto", SqlDbType.Int),
        new SqlMetaData("Cognome", SqlDbType.VarChar, 50),
        new SqlMetaData("Nome", SqlDbType.VarChar, 50),
        new SqlMetaData("LocalitaNascita", SqlDbType.NVarChar, 50),
```

```
            new SqlMetaData("CodiceFiscale", SqlDbType.VarChar, 16),
            new SqlMetaData("Sesso", SqlDbType.NChar, 1),
            new SqlMetaData("DataNascita", SqlDbType.DateTime),
            new SqlMetaData("CodiceFiscaleCalcolato", SqlDbType.VarChar,
                                                                    16),
            new SqlMetaData("Corretto", SqlDbType.Bit),

        };
        //creazione dell'oggetto record che incapsulerà il resultset
        SqlDataRecord record = new SqlDataRecord(metadata);
        //il pipe viene istruito sul fatto che verrà restituito un resultset
        pipe.SendResultsStart(record);
      //navigazione sulle righe del DataReader contenenti i dati ottenuti dalla
                                                                    query
        while (reader.Read())
        {
            //definzione e recupero dei valori dei campi della riga corrente del
                                                            datareader
            int idSogg;
            string cognome, nome, localita, codiceFiscale, sesso,
                                                    nuovoCodiceFiscale;
            DateTime dataNascita;

            idSogg = reader.GetSqlInt32(0).Value;
            cognome = reader.GetSqlString(1).Value;
            nome = reader.GetSqlString(2).Value;
            localita = reader.GetSqlString(3).Value;
            codiceFiscale = reader.GetSqlString(4).Value;
            sesso = reader.GetSqlString(5).Value;
            dataNascita = reader.GetDateTime(6).Date;
            //invocazione della funzione di ricalcolo del codice fiscale
            //ad essa vengono passati come argomento tutte le informazioni
                                                            necessarie al
```

```
            //ricalcolo
            nuovoCodiceFiscale =
                PartiComuni.CalcolaCodiceFiscale(cognome, nome, sesso,
                                                    localita, dataNascita);
            //valorizzazione dei campi della riga corrente; il primo parametro
            // delle funzioni Set indica proprio la posizione della riga
                                                                nel resultset
            record.SetInt32(0, idSogg);
            record.SetSqlString(1, cognome);
            record.SetSqlString(2, nome);
            record.SetSqlString(3, localita);
            record.SetSqlString(4, codiceFiscale);
            record.SetSqlString(5, sesso);
            record.SetSqlDateTime(6, dataNascita);
            record.SetSqlString(7, nuovoCodiceFiscale);
            record.SetSqlBoolean(8, (nuovoCodiceFiscale == codiceFiscale));
            //invio della riga corrente nel resultset
            pipe.SendResultsRow(record);
        }
        //chisura del datareader
        reader.Close();
        //il flusso del resultset è completo e il canale viene chiuso
        pipe.SendResultsEnd();
    }
}
}
}
```

Per completezza si riportano le due funzioni statiche CacheCodiciBelfiore e CalcolaCodiceFiscale, entrambe definite nella classe PartiComuni. Di questa classe abbiamo già riportato una porzione in precedenza che, dunque, ometteremo. Osserviamo la prima; essa fa uso di un'istanza di Dictionary di tipo generic che contiene tutte le coppie chiave – valore di

NomeLocalita – CodiceBelfiore. Questo oggetto viene popolato solo la prima volta viene invocato e successivamente funziona come cache per evitare ulteriori accessi alla base dati:

```
namespace CLREsempio
{
class PartiComuni
{
//cache dei valori dei codici belfiore: la chiave è la località e il valore
//è il codice belfiore
internal static readonly Dictionary<string, string> CodiciBelfiore =
    CodiciBelfiore = new Dictionary<string, string>();

//la funzione popola la cache
internal static void CacheCodiciBelfiore()
{
    //se la cache è già stato popolata non viene effettuata nessuna
                                                                operazione
    if (CodiciBelfiore.Count > 0) return;

    string codiceBelfiore, localita;

    //apertura della connessione a SQL Server
    using (SqlConnection conn =
        new SqlConnection("context connection=true"))
    {
        conn.Open();                    // open the connection
      //caricamento di tutti i dati presenti nella tabella Localita
        SqlCommand cmd = new SqlCommand(
            " SELECT Localita, CodiceBelfiore " +
            " FROM Localita " ,
            conn);
        SqlDataReader reader = cmd.ExecuteReader();
```

```
        while (reader.Read())
        {
          //riempimento della cache
           localita = reader.GetSqlString(0).Value;
           codiceBelfiore = reader.GetSqlString(1).Value;
           CodiciBelfiore[localita] = codiceBelfiore;
        }
        reader.Close();
    }
}
```

La successiva funzione, invece, effettua il ricalcolo del codice fiscale a partire da tutti i parametri che lo costituiscono. Verrà omessa ogni spiegazione perché non attinente allo scopo del volume:

```
internal static string CalcolaCodiceFiscale(
                string cognomeSoggetto,
                string nomeSoggetto,
                string sesso,
                string localitaNascita,
                DateTime dataNascita)
{
   initializeValues();

   try
   {
      int cnt = 0;
      string cognome = null;
      for (int i = 0; i < cognomeSoggetto.Length; i++)
      {
         if (reConsonanti.Match(cognomeSoggetto[i].ToString()).Success)
         {
            cognome = cognome + cognomeSoggetto[i].ToString();
```

```
            cnt++;
            if (cnt == 3)
                break;
        }
    }
    if (cnt < 3)
    {
        for (int i = 0; i < cognomeSoggetto.Length; i++)
        {
            if (reVocali.Match(cognomeSoggetto[i].ToString()).Success)
            {
                cognome = cognome + cognomeSoggetto[i].ToString();
                cnt++;
                if (cnt == 3)
                    break;
            }
        }
    }

    for (int i = cnt; i < 3; i++)
        cognome = cognome + "X";

    MatchCollection totCons = reConsonanti.Matches(nomeSoggetto);

    cnt = 0;
    string nome = null;
    int consContante = 0;
    for (int i = 0; i < nomeSoggetto.Length; i++)
    {
        if (reConsonanti.Match(nomeSoggetto[i].ToString()).Success)
        {
            consContante++;
            if (totCons.Count > 3 && consContante == 2)
```

```
                continue;
            else
            {
                cnt++;
                nome = nome + nomeSoggetto[i].ToString();
                if (cnt == 3)
                    break;
            }
        }
    }
    if (cnt < 3)
    {
        for (int i = 0; i < nomeSoggetto.Length; i++)
        {
            if (reVocali.Match(nomeSoggetto[i].ToString()).Success)
            {
                cnt++;
                nome = nome + nomeSoggetto[i].ToString();
                if (cnt == 3)
                    break;
            }
        }
    }
    for (int i = cnt; i < 3; i++)
        nome = nome + "X";

    string anno = dataNascita.ToString("yy");
    string mese = mesi[dataNascita.Month - 1].ToString();
    string giorno = sesso == "M" ? dataNascita.ToString("dd") :
                                    (dataNascita.Day + 40).ToString();

    string codicebelfiore = PartiComuni.CodiciBelfiore[localitaNascita];
    string codfisc = cognome + nome + anno + mese + giorno + codice
```

```
                                                                belfiore;
        codfisc = codfisc.ToUpper();
        int codicecontrollo = 0;
        for (int i = 0; i < 15; i++)
        {
            if ((i + 1) % 2 == 0)
                codicecontrollo = codicecontrollo + (int)pari[codfisc[i]];
            else
                codicecontrollo = codicecontrollo + (int)dispari[codfisc[i]];
        }
        codfisc = codfisc + controllo[(codicecontrollo % 26)];
        if ( VerificaCodiceFiscale(codfisc ) )
            return codfisc;
        else
            return null;
    }
    catch (Exception ex)
    {
        //
    }

    return null;
}
```

2.5.2 Registrazione, test e debug della stored procedure

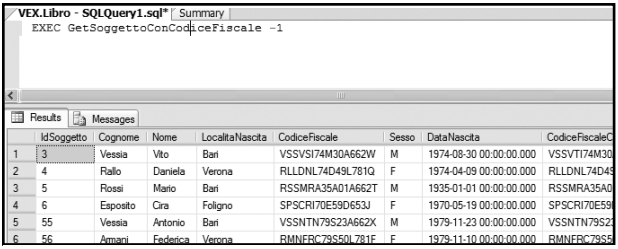
Visual Studio 2005 Professional come al solito si occupa della registrazione della procedure sul database. Ma se operassimo manualmente, dopo aver ricompilato e caricato l'assembly in SQL Server come abbiamo fatto nel precedente esempio, dovremmo registrare la stored procedure con il seguente comando T-SQL:

```
CREATE PROCEDURE [dbo].[GetSoggettoConCodiceFiscale]
```

```
    @idSoggetto [int]
WITH EXECUTE AS CALLER
AS
EXTERNAL NAME
 [CLREsempio].[CLREsempio.StoredProcedures].[GetSoggettoConCodiceFisc
                                                              ale]
GO
```

Per il test della stored procederemo semplicemente invocandola via T-SQL:

```
--per ottenere l'elenco di tutti i soggetti
EXEC GetSoggettoConCodiceFiscale -1
```



	IdSoggetto	Cognome	Nome	LocalitaNascita	CodiceFiscale	Sesso	DataNascita	CodiceFiscaleC
1	3	Vessia	Vito	Bari	VSSVSI74M30A662W	M	1974-08-30 00:00:00.000	VSSVTI74M30
2	4	Rallo	Daniela	Verona	RLLDNL74D49L781Q	F	1974-04-09 00:00:00.000	RLLDNL74D49
3	5	Rossi	Mario	Bari	RSSMRA35A01A662T	M	1935-01-01 00:00:00.000	RSSMRA35A0
4	6	Esposito	Cira	Foligno	SPSCRI70E59D653J	F	1970-05-19 00:00:00.000	SPSCRI70E59
5	55	Vessia	Antonio	Bari	VSSNTN79S23A662X	M	1979-11-23 00:00:00.000	VSSNTN79S23
6	56	Armani	Federica	Verona	RMNFRC79S50L781F	F	1979-11-10 00:00:00.000	RMNFRC79S5

Figura 2.10: La stored procedure in azione

```
--per ottenere solo il soggetto con id 2
EXEC GetSoggettoConCodiceFiscale 2
```

In Figura 2.10 possiamo osservare il resultset generato dalla sua chiamata.

2.6 TRIGGER IN CLR

Per dovere di completezza non possiamo concludere la lunga disanima

di questo argomento mostrando la creazione di un trigger in C#. SQL Server 2005 offre la possibilità di implementare l'intera gamma di trigger in .NET, ma per l'esempio ci concentreremo su uno molto semplice. Selezionando il progetto corrente dal pannello Solution Explorer di Visual Studio 2005 e attivando il menù contestuale con il tasto destro del mouse, sceglieremo la voce Add –> Trigger. Il nostro trigger di esempio si chiama TriggerCodiceFiscale ed è di tipo FOR UPDATE,INSERT sulla tabella Soggetti. Infatti sul solito metodo statico che lo implementa è presente l'attributo Microsoft.SqlServer.Server.SqlTrigger che, però, a differenza della stored procedure e della funzione, propone una serie di parametri aggiuntivi quali il nome (Name = "TriggerCodiceFiscale"), la tabella su cui agirà (Target = "Soggetti") e la tipologia di trigger (Event = "FOR INSERT,UPDATE").
Il trigger verificherà che il codice fiscale della riga in via di inserimento o modifica sia corretto. In caso affermativo non farò nulla, diversamente solleverà un'eccezione applicativa intercettabile e gestibile da codice T-SQL. Eccone il codice commentato:

```
using System;
using System.Data;
using System.Data.Sql;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

namespace CLREsempio
{
public partial class Triggers
{
[Microsoft.SqlServer.Server.SqlTrigger(Name = "TriggerCodiceFiscale",
                    Target = "Soggetti", Event = "FOR INSERT,UPDATE")]
public static void TriggerCodiceFiscale()
{
```

```
//apertura della connesione al database
using (SqlConnection conn =
      new SqlConnection("context connection=true"))
{
   conn.Open();
   //recupero del contesto del trigger
   SqlTriggerContext triggerContext = SqlContext.TriggerContext;
   //recupero del Pipe, cioè del canale di comunicazione con SQL Server
   SqlPipe sqlPipe = SqlContext.Pipe;
   //recupero dei dati in via di inserimento nella tabella e che
   //hanno scatenato il trigger
   SqlCommand cmd = new SqlCommand(
      " SELECT CodiceFiscale FROM INSERTED ",
      conn);
   SqlDataReader reader = cmd.ExecuteReader();
   while (reader.Read())
   {
      //lettura del parametro codice fiscale in via di inserimento
      string codiceFiscale = reader.GetSqlString(0).Value;
      reader.Close();
      if (triggerContext.TriggerAction ==
         TriggerAction.Insert)
      {
         //verifica della bontà del codice fiscale
         if (!PartiComuni.VerificaCodiceFiscale(codiceFiscale))
         {
            //se il codice fiscale non è valido viene sollevata un'eccezione
                                                          utente
            //con il buon vecchio comando RAISERROR
            cmd.CommandType = CommandType.Text;
            cmd.CommandText = "RAISERROR ('Codice Fiscale errato o
                                                   mancante', 1, 5)";
            SqlContext.Pipe.ExecuteAndSend(cmd);
```

```
            }
        }
        else
        {
            //gestisci UPDATE
        }
        break;
    }
    SqlContext.Pipe.Send("Trigger FIRED");
  }
}
}
}
```

2.6.1 Registrazione, test e debug del trigger

Ancora una volta Visual Studio 2005 Professional fa tutto il lavoro di deployment e registrazione del trigger per noi, ma operando manualmente, dopo l'inevitabile compilazione e upload dell'assembly già viste in precedenza, dovremmo effettuare la registrazione del trigger con il seguente comando:

```
CREATE TRIGGER [dbo].[TriggerCodiceFiscale] ON [dbo].[Soggetti]  AFTER
                                                    INSERT, UPDATE AS
EXTERNAL NAME
                [CLREsempio].[CLREsempio.Triggers].[TriggerCodiceFiscale]
GO
```

Il trigger non è direttamente invocabile come le stored procedure e le funzioni, ma viene automaticamente invocato dal database engine al verificarsi dell'evento per cui il trigger è registrato. Pertanto per il test e il debugging del nostro esempio dovremmo operare con un inserimento di una riga nella tabella Soggetti:

Figura 2.11: Il trigger puntiglioso

```
INSERT INTO Soggetti
 (Cognome, Nome, LocalitaNascita, CodiceFiscale, Sesso, DataNascita)
 VALUES ('Vessia', 'Vito', N'Bari', 'VSSVSI74M30A662W', N'M', '1974-08-
30
                                                    00:00:00.000')
```

Il codice fiscale inserito è volutamente errato. Osserviamo le reazioni di SQL Server all'esecuzione della INSERT in Figura 2.11.

2.7 AGGREGATI PERSONALIZZATI

SQL Server 2005 introduce una nuova possibilità nel suo supporto al CLR: gli aggregati personalizzati. Il linguaggio T-SQL è già ricco di aggregati, cioè di funzioni su base statistica (AVG, STDEV) o che in generale operano sulla massa di dati (delle righe restituite nel resultset) e non sul singolo campo (MIN, MAX, SUM, ecc...). Con SQL Server 2005 e .NET è possibile espandere all'infinito questa possibilità introducendo aggregati personalizzati che soddisfino qualsiasi esigenza. L'aggregato viene sviluppato in .NET, compilato, registrato e caricato su SQL Server, pronto per essere usato nelle query alla stregua degli aggregati convenzionali.

Per meglio comprenderne l'uso e le modalità di codifica, procediamo con il solito esempio che ha a che fare con il codice fiscale: immaginia-

mo un aggregato che agisca sul campo CodiceFiscale e che ritorni il codice fiscale corrispondente al Soggetto più giovane nell'ambito del resultset. Come sappiamo, la data di nascita, nell'ambito del codice fiscale, occupa una porzione centrale del codice stesso pertanto non si può utilizzare un normale criterio di ordinamento alfabetico per ordinare il risultato per data di nascita, ma bisogna procedere con l'interpretazione del codice fiscale stesso. Inoltre sappiamo anche che l'algoritmo, per distinguere le donne dagli uomini aggiunge 40 alla porzione relativa al giorno di nascita, aspetto che deve essere tenuto in considerazione quando si confrontano le date di nascita per determinare il più giovane. Una volta messo appunto il nostro aggregato Younger saremo in grado di usarlo nel seguente modo, come mostrato in Figura 2.12:

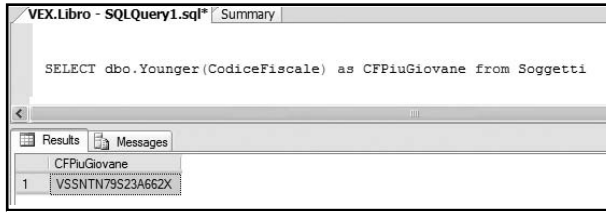


Figura 2.12: Il nostro nuovo e giovane aggregato

```
SELECT dbo.Younger(CodiceFiscale) from Soggetti
```

Ma procediamo con la creazione dell'aggregato: selezionando il progetto corrente dal pannello Solution Explorer di Visual Studio 2005 e attivando il menù contestuale con il tasto destro del mouse, sceglieremo la voce Add –> Aggregato. Verrà proposto un scheletro di codice di un tipo strutturato (una struct). Sulla struct è presente l'attributo Microsoft.SqlServer.Server.SqlUserDefinedAggregate dotato di un costruttore a cui passeremo il parametro Format. Questo è un enumerativo che può contenere i valori Native, UserDefined e Unknown. Il suo scopo è descrivere la modalità di persistenza dei valori all'interno dell'aggregato. Per sua stessa natura,

infatti, un aggregato ha memoria, cioè deve conservare i valori. Si pensi alla MIN che deve conservare il minimo valore nell'ambito del resultset, magari confrontando quel valore con quelli delle righe resultset corrente per determinare se altre righe hanno valori che possano costituire il nuovo minimo e questo fino al completo scorrimento di tutti i valori. Con la stessa finalità è presente anche l'attributo MaxByteSize che esprime la dimensione in byte del valore che dovrà essere conservato e restituito dall'aggregato. Il codice fiscale occupi 16 caratteri e così riserviamo 32 byte per il formato Unicode. Il codice dell'aggregato è riportato di seguito ed è completamente commentato pertanto si rimanda ai commenti per una migliore comprensione.

```
using System;
using System.IO;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

namespace CLREsempio
{
//l'aggregato viene marcato col doppio attributo di Serializable perché il suo
                                                    valore temporaneo verrà
//serializzato e conservato all'interno di SQL Server e con l'attributo
                                                    SqlUserDefinedAggregate
//che indica la sua natura di aggregato SQL Server 2005 CLR.
//l'argomento MaxByteSize indica la dimensione in byte dell'informazione
                                                    conservata dall'aggregato
//nel caso specifico si tratta di un codice fiscale che sappiamo occupare 16
                                                    caratteri e quindi 32 byte
//in formato Unicode
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedAggregate(Format.Unknown,
```

```
                                                    MaxByteSize = 32)]
public struct Younger : IBinarySerialize
{
    //la variabile privata che conterrà il codice fiscele del soggetto
                                                            più giovane
    private SqlString _cfGiovane;

    //metodo di inizializzazione dell'aggregato che viene invocata all'inizio
                                                        della query in cui
    //compare l'aggregato custom; nel caso specifico viene messo a null la
                                                        variabile che contiene
    //il codice fiscale del più giovane in modo che il primo elmento restituito
                                                        dalla query venga
    //reso automaticamente il più giovane a meno di confronti con i CF dei
                                                        record successivi
    public void Init()
        _cfGiovane = null;
    }

    //è il metodo fondamentale dell'aggregato e viene invocato ad ogni
                                                iterazione passando l'argomento
    //passato all'aggregato nella query di esempio.
    //     Es.          SELECT dbo.Younger(CodiceFiscale) FROM Soggetti
    public void Accumulate(SqlString Value)
    {
        //se la variabile membro è nulla (condizione possibile solo sul primo
                                                        record visto che
        //il campo codice fiscale non è nullable), viene considerato il primo
                                                        soggetto come il
        //più giovane
        if (_cfGiovane.IsNull )
        {
            _cfGiovane = Value;
```

```
    }
    else
    {
      //la data di nascita del codice fiscale è espressa nella forma YYMDD
       // YY = ultime due cifre dell'anno di nascita
       // M = lettera corrispondente al mese di nascita nella sequenza
       //          { 'A', 'B', 'C', 'D', 'E', 'H', 'L', 'M', 'P', 'R', 'S', 'T' }
       // DD = giorno di nascita per i maschi, giorno di nascita + 40 per le
                                                                    donne
       //dunque per confrontare veramente due date di nascita è
                                                        necessario riportare
      //le due date da confrontare al maschile in modo che l'offset 40 per
                                                                le donne
       //non infici il confronto

      //codice fiscale del soggetto attualmente considerato il più giovane
       string dataMaschile = _cfGiovane.ToString();
       //codice fiscale del soggetto corrispondete all'attuale record
                                                          da confrontare
       string dataMaschileValue = Value.ToString();

       //se il codice fiscale del soggetto attualmente più giovane
                                                            appartiene
       //ad una donna avrà "DD" > 40
       if (int.Parse(dataMaschile.Substring(9, 2)) > 40)
       {
          //la data viene ricondatta al maschile per facilitare il confronto
          dataMaschile = dataMaschile.Substring(0, 9) +
                    (int.Parse(dataMaschile.Substring(9, 2)) -
                                                    40).ToString("00") +
                    dataMaschile.Substring(11, 5);
       }
       //stessa considerazione da fare per il codice fiscale del soggetto
```

```
                                                    attualmente oggetto
        //del confronto
        if (int.Parse(dataMaschileValue.Substring(9, 2)) > 40)
        {
            dataMaschileValue = dataMaschileValue.Substring(0, 9) +
                                (int.Parse(dataMaschileValue.Substring(9, 2)) -
                                                    40).ToString("00") +
                                dataMaschileValue.Substring(11, 5);
        }
        //una volta riportate le porzioni di codice fiscale relative alle date
                                                    di nascita
        //rapportate al maschile, il confronto tra due date si riconduce ad
                                                    un banale confronto tra
        //stringhe visto che la parte più significa è l'anno che è già
                                                    espressa in forma numerica
        //facilmente confrontabile, il carattere del mese è ordinato in modo
                                                    crescente al crescere del
        //mese stesso e la parte del giorno è un numero di due cifre non più
                                                    soggetto al problema
        //dell'addizionamento di 40 per le donne a seguito
                                                    della conversione al maschile
        if (dataMaschile.Substring(6,
5).ToUpper().CompareTo(dataMaschileValue.Substring(6, 5).ToUpper()) <
                                                    0)
        {
            //se la data di nascita del soggetto corrente è inferiore a quella
                                                    dell'attuale soggetto
            //più giovane, il primo diventa il nuovo soggetto più giovane
            _cfGiovane = Value.ToString();
        }
    }
}
```

```
    //pattern standard per la funzione di Merge dei valori
    public void Merge(Younger Group)
    {
        Accumulate(Group.Terminate());
    }

    //il metodo di chiusura dell'aggregato, invocato al termine della query
    public SqlString Terminate()
    {
        return _cfGiovane;
    }

    //serializzazione del valore del codice fiscale dell'attuale soggetto più giovane
    //viene usato un BinaryWriter per salvare il valore
    public void Write(BinaryWriter w)
    {
        w.Write(_cfGiovane.ToString());
    }

    //deserializzazione del valore del codice fiscale dell'attuale soggetto più giovane
    //viene recuperato il valore da BinaryWriter
    public void Read(BinaryReader r)
    {
        _cfGiovane = r.ReadString();
    }
}
}
```

2.7.1 Registrazione, test e debug dell'aggregato

Ancora una volta Visual Studio 2005 Professional fa tutto il la-

voro di deployment e registrazione dell'aggregato per noi, ma operando manualmente, dopo l'inevitabile compilazione e upload dell'assembly già viste in precedenza, dovremmo effettuare la registrazione del trigger con il seguente comando:

```
CREATE AGGREGATE [dbo].[Younger]
(@Value [nvarchar](4000))
RETURNS[nvarchar](4000)
EXTERNAL NAME [CLREsempio].[CLREsempio.Younger]
GO
```

2.8 TIPO DEFINITO DALL'UTENTE

Per completare questa ricca disquisizione sul supporto al CLR fornito da SQL Server 2005 è necessario introdurre un ultimo ed importante elemento: la possibilità di creare tipi di dati personalizzati da usare in tabelle, viste e query T-SQL e che siano in grado da fornire al tipo di dato una logica consistente e strutturata che solo il codice di programmazione può dare. L'esempio è naturalmente d'obbligo e, a questo punto della trattazione, sarete sicuramente giunti alla conclusione che il mondo dei database, e forse anche l'intero universo conosciuto, ruotano intorno al codice fiscale e dunque quale esempio di tipo utente personalizzato migliore da mostrare se non proprio il tipo CodiceFiscale? L'idea è piuttosto semplice: il tipo utente CodiceFiscale sarà un campo in grado naturalmente di ospitare codici fiscali, ma avrà anche la logica, una volta inserito in un campo di una tabella, di rifiutare la INSERT di un nuovo record qualora il codice fiscale sia errato, analogamente a quanto visto in precedenza col trigger, ma senza aggiungere altra logica alla tabella perché essa sarà gestita dal campo stesso, analogamente a quanto avviene per le Check Constraint.

Inoltre questo campo renderà inutile l'aggiunta dei campi Località di Nascita, Data di Nascita e Sesso nella tabella anagrafica perché queste informazioni saranno ricavabili dal codice fiscale stesso. Paradossalmen-

te, infatti, si potrebbe quasi affermare che l'uso di un campo Codice Fiscale viola la Prima Forma Normale proprio perché aggrega più informazioni nello stesso campo, ma naturalmente prendete questa affermazione come una provocazione e continuate a tranquillamente a definire i campi codice fiscale nelle vostre tabelle... Osserviamo in Figura 2.13 l'uso del nuovo tipo come campo di una tabella.

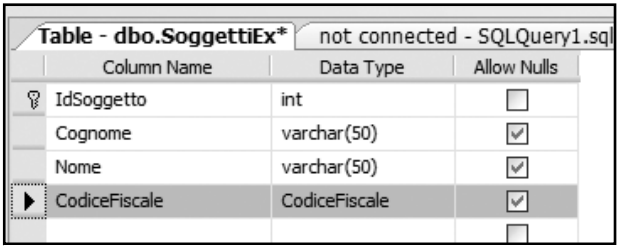


Figura 2.13: Uso del tipo di dato personalizzato in una tabella

Selezionando il progetto corrente dal pannello Solution Explorer di Visual Studio 2005 e attivando il menù contestuale con il tasto destro del mouse, sceglieremo la voce Add –> Tipo definito dall'utente. Il wizard produrrà un nuovo file di sorgente nel progetto che contiene una struct. Su di essa verrà posto l'attributo Microsoft.SqlServer.Server.SqlUserDefinedType a cui viene passato l'argomento già visto in precedenza Format e i parametri MaxByteSize (anch'esso già incontrato) e IsByteOrdered che esprime la modalità di ordinamento dei campi. Infatti si immagini una query su una tabella che dispone di un campo di tipo CodiceFiscale: l'attributo esprime come si dovrà comportare la eventuale clausola ORDER BY CodiceFiscale.
Nell'esempio specifico, poi, il tipo implementa anche l'interfaccia IBinarySerializer già incontrata in precedenza nell'aggregato e l'interfaccia INullable che permette di gestire campi con valori di tipo null per il tipo personalizzato appena introdotto. Ma si osservi il codice commentato per una maggiore comprensione:

```
using System;
using System.Runtime.InteropServices;
using System.Reflection;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

namespace CLREsempio
{
//Un UDT è implementato come una normale struct di .NET
//va corredato con l'attributo SqlUserDefinedType a cui si passa il tipo
                                                    di formato di serializzazione
//come per gli aggregati, la dimensione richiesta dal tipo espressa in byte,
                                                    il tipo di ordinamento
//e altre informazioni
[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.UserDefined,
                                    IsByteOrdered=true, MaxByteSize=32)]
//l'UDT può implementare le interfacce per la gestione del NULL e per la
                                                    serializzazione custom
public struct CodiceFiscale : INullable, IBinarySerialize
{
   //variabile privata membro che contiene il codice fiscale vero e proprio
   //il suo valore sarà l'unico salvato e recuperato effettivamente
                                                    dal database, tutti gli
   //altri valori verranno semplicemente estratti e ricalcolati in base
                                                    a questo
   //in questo modo si riduce l'occupazione di spazio sul database
                                                    portandola alla stessa
   //occupazione di un normale campo NVARCHAR(16) in grado
                                                    di conservare un codice fiscale
   private SqlString m_CF;
```

```
//campo data di nascita, estratto dinamicamente dal codice fiscale
public SqlDateTime DataNascita;
//campo sesso, può assumere i valori "M" o "F", estratto
                                        dinamicamente dal codice fiscale
public SqlString Sesso;
//campo Codice Belfiore della località di nascita, estratto
                                        automaticamente dal codice fiscale
public SqlString CodiceBelfiore;
//serve per testare la condizione di null
private bool m_Null;
//il metodo ToString() implementa il classico metodo ToString() di CLR,
                                                utile in questi casi
//quando si intende ottenere una rappresentazione in stringa del valore
public override string ToString()
{
    return m_CF.ToString();
}

//l'UDT implementa l'interfaccia INullable e così gestisce la possibilità di
                                                valori null nel campo
public bool IsNull
{
    get
    {
        return m_Null;
    }
}

//metodo statico dell'UDT in grado di restituire un'istanza null del tipo
public static CodiceFiscale Null
{
    get
    {
```

```
            CodiceFiscale h = new CodiceFiscale();
            h.m_Null = true;
            return h;
        }
    }

    //è il metodo fondamentale degli UDT CLR: consente l'analisi e il parsing
                                                        del valore stringa
    //che soggiace al tipo personalizzato: nell'esempio specifico verrà
                                                    passato proprio il codice
    //fiscale da gestire e a questo punto verrà validato e ne verranno
                                                    estratte le componenti
    //fondamentali
    public static CodiceFiscale Parse(SqlString s)
    {
        //verifica dell condizione di null
        if (s.IsNull)
            return Null;

        //viene verificata la bontà del codice fiscale attraverso l'algoritmo di
                                                                controllo
        //del carattere di controllo finale
        if (!PartiComuni.VerificaCodiceFiscale(u.m_CF.ToString()))
            throw new ArgumentException("Codice Fiscale Errato!");

        //se il codice fiscale è valido, vengono estratte le sue parti salienti
        CodiceFiscale u = new CodiceFiscale();
        u.m_CF = s.ToString().ToUpper();
        u.CodiceBelfiore = s.ToString().Substring(11, 4).ToUpper();
        u.Sesso = int.Parse(u.ToString().Substring(9, 2)) > 40 ? "F" : "M";
        u.DataNascita = PartiComuni.CalcolaDataNascita(s.ToString());

        return u;
```

```
    }

    //gestione del recupero dell'informazione dallo stream di persistenza del
                                                    valore nel database
    //l'implementazione di questo metodo è necessaria perché si è scelto il
                                                    formato di seralizzaizone
    //Format.UserDefined in abbinamento all'implementazione
                                                    dell'interfaccia IBinarySerialize
    //in realtà, nell'esempio specifico, l'unica informazione da recuperare è
                                                    proprio la stringa di
    //16 caratteri del codice fiscale perché tutti i campi che costituiscono il
                                                    codice fiscale vengono
    //calcolati ed estratti dinamicamente a partire da questa
    public void Read(System.IO.BinaryReader r)
    {
        m_CF = r.ReadString();
        CodiceBelfiore = m_CF.ToString().Substring(11, 4).ToUpper();
        Sesso = int.Parse(m_CF.ToString().Substring(9, 2)) > 40 ? "F" : "M";
        DataNascita = PartiComuni.CalcolaDataNascita(m_CF.ToString());
    }

    //gestione della serializzazione dell'informazione in uno stream binario
    public void Write(System.IO.BinaryWriter w)
    {
        w.Write(m_CF.ToString());
    }
}
}
```

Le caratteristiche salienti del tipo utente CodiceFiscale sono la possibilità di gestire una stringa di 16 caratteri corrispondente ad un codice fiscale e di questa validarne la bontà (metodo Parse), di permetterne la serializzazione e la deserializzazione nella base dati (i metodi Read e

Write), di gestirne i valori Null (il metodo IsNull e il metodo statico Null che restituisce proprio un'istanza a valore null del tipo utente) e di restituirne il codice fiscale come stringa (il metodo convenzionale ToString derivante da System.Object).
Ma probabilmente gli elementi che rendono davvero uniche e rilevanti le caratteristiche di questo tipo utente e che, più in generale, mostrano la potenza e la versatilità del meccanismo dei tipo utente CLR in SQL Server 2005, è la possibilità di estrarre ed esporre del codice fiscali informazioni quali il Codice Belfiore (proprietà CodiceBelfiore), la data di nascita (proprietà DataNascita) e il sesso (proprietà Sesso).
Esso sono fruibili con la normale sintassi <nome_campo>.<proprietà>, pertanto, se volessimo conoscere il sesso di un soggetto, dovremmo usare la sintassi CodiceFiscale.Sesso.

2.8.1 Registrazione, test e debug del tipo utente

Ancora una volta Visual Studio 2005 Professional fa tutto il lavoro di deployment e registrazione del tipo utente per noi, ma operando manualmente, dopo l'inevitabile compilazione e upload dell'assembly già viste in precedenza, dovremmo effettuare la registrazione del trigger con il seguente comando:

```
CREATE TYPE [dbo].[CodiceFiscale]
EXTERNAL NAME [CLREsempio].[CLREsempio.CodiceFiscale]
GO
```

A questo punto siamo pronti ad introdurre una tabella di esempio che usi questo tipo. Si tratta di una versione semplificata di Soggetti usata nel corso del capitolo: essa, infatti, presenta i soli campi Cognome, Nome e CodiceFiscale e omette i campi DataNascita, LocalitaNascita e Sesso che sono tutte ricavabili direttamente del codice fiscale e che sono esposte dal tipo utente CodiceFiscale come proprietà richiamabili direttamente da query T-SQL,

come vedremo in seguito. Ecco lo script T-SQL di creazione della tabella:

```
CREATE TABLE [dbo].[SoggettiEx](
    [IdSoggetto] [int] IDENTITY(1,1) NOT NULL,
    [Cognome] [varchar](50) COLLATE Latin1_General_CI_AS NULL,
    [Nome] [varchar](50) COLLATE Latin1_General_CI_AS NULL,
    [CodiceFiscale] [dbo].[CodiceFiscale] NULL,
 CONSTRAINT [PK_SoggettiEx] PRIMARY KEY CLUSTERED
(
    [IdSoggetto] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
                    IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON,
                                ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
```

L'inserimento o la modifica di righe nella tabella è molto semplice perché il valore del tipo CodiceFiscale viene trattato come un normale NVARCHAR(16), come si evince dal metodo Merge implemenato nel codice del tipo utente di esempio. Ed ecco una INSERT esemplificativa:

```
INSERT INTO dbo.SoggettiEx
 (Cognome, Nome, CodiceFiscale)
  VALUES ('Vessia', 'Vito', 'VSSVSI74M30A662W')
```

Nella Merge, inoltre, abbiamo osservato come venga contestualmente fatta una verifica della validità del codice fiscale appena inserito sollevando un'eccezione in caso di codice fiscale non valido, condizione possibile grazie al ricalcolo del codice/carattere di controllo finale e del confronto con quello presente nel codice fiscale. In caso di INSERT o di UPDATE di codici fiscali non validi, verrà sollevata la seguente eccezione e l'operazione di scrittura non verrà eseguita:

```
Msg 6522, Level 16, State 2, Line 1
A .NET Framework error occurred during execution of user-defined routine
                                        or aggregate "CodiceFiscale":
System.ArgumentException: Codice Fiscale Errato!
System.ArgumentException:
   at CLREsempio.CodiceFiscale.Parse(SqlString s)
.
The statement has been terminated.
```

In Figura 2.14 possiamo osservare gli effetti di una INSERT in SoggettiEx con CodiceFiscale errato.

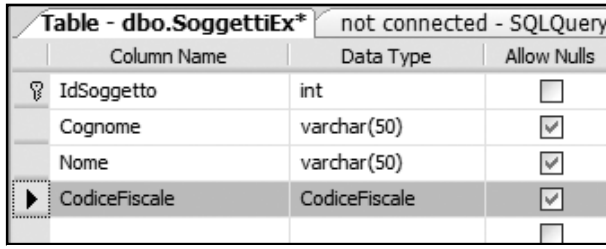


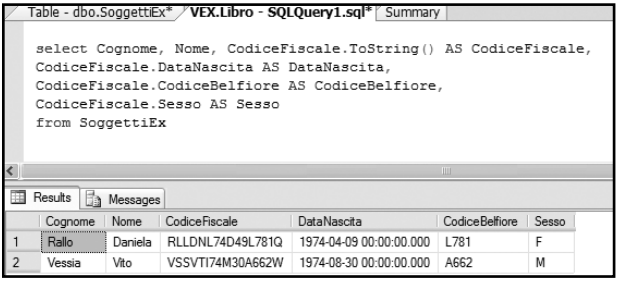
Figura 2.14: Inserimento di un codice fiscale errato nel tipo utente CodiceFiscale

La query che segue mostra tutta la potenza e la versatilità del nostro tipo utente personalizzato: dalla tabella SoggettiEx, oltre ai campi di tipo convenzionale Nome e Cognome, vengono estratte tutta una serie di informazioni dal campo CodiceFiscale. Per cominciare il codice fiscale stesso, invocando il metodo ToString() dell'oggetto che, nell'implementazione che abbiamo fornito per il tipo CodiceFiscale restituisce proprio il codice fiscale intero. Inoltre, invocando rispettivamente le proprietà DataNascita, CodiceBelfiore e Sesso del tipo, si possono estrarre direttamente le omonime informazioni rendendo così non necessaria la presenza esplicita di questi campi nella tabella che contiene il tipo utente

CodiceFiscale.

```
SELECT Cognome, Nome, CodiceFiscale.ToString() AS CodiceFiscale,
CodiceFiscale.DataNascita AS DataNascita,
CodiceFiscale.CodiceBelfiore AS CodiceBelfiore,
CodiceFiscale.Sesso AS Sesso
FROM SoggettiEx
```

La Figura 2.15 ci mostra proprio il risultato di questa query di esempio.



	Cognome	Nome	CodiceFiscale	DataNascita	CodiceBelfiore	Sesso
1	Rallo	Daniela	RLLDNL74D49L781Q	1974-04-09 00:00:00.000	L781	F
2	Vessia	Vito	VSSVTI74M30A662W	1974-08-30 00:00:00.000	A662	M

Figura 2.15: Uso intensivo delle caratteristiche del tipo utente CodiceFiscale

CATALOGO E FUNZIONI DI SISTEMA

La struttura di ciascun database e cioè l'insieme delle tabelle, delle viste, delle stored procedure, delle funzioni, dei trigger e di ogni altro elemento che lo costituisce è conservata e gestita attraverso il Catalogo di Sistema. In SQL Server sono presenti quattro database di sistema, cioè necessari a SQL Server per il suo funzionamento corrente e che vengono creati al momenti dell'installazione dell'istanza del motore:

- " master;
- " model;
- " tempdb;
- " msdb.

Master è il database principale, master, che conserva le informazioni e le impostazioni di base del motore e che poi vengono riapplicate a livello di ciascun database a meno che queste non vengano esplicitamente ridefinite. Comprende tutte le tabelle di sistema necessarie per lavorare con il sistema di database. Pertanto esso contiene tutte le informazioni su tutti gli altri database gestiti da SQL Server, sulle connessioni di sistema ai client e sulle autorizzazioni degli utenti.
TempDB, invece, fornisce lo spazio di memorizzazione necessario alle tabelle temporanee e a tutti gli altri oggetti temporanei definiti nelle interrogazioni e negli accessi a tutti gli altri database. Infatti, ogni volta che si crea un tabella temporanea in qualsiasi database, in realtà viene creata una tabella fisica reale in questo database e poi viene rimossa al termine della sessione.
Model viene usato come modello quando vengono creati i database definiti dall'utente e contiene un sottoinsieme di tutte le tabelle di sistema di Master necessarie alla definizione dei database utenti.
Msdb, infine, è il database alla base delle operazioni definibili per lo schedulatore integrato di SQL Server e cioè SQL Server Agent. In prati-

ca in esso vengono memorizzate sia le operazioni da compiere per ciascuna schedulazione programmata (dagli script T-SQL da eseguire, ai comandi agli script) che la loro programmazione temporale di esecuzione. In Figura 3.1 vengono mostrate queste tabelle da SQL Server Management Studio.

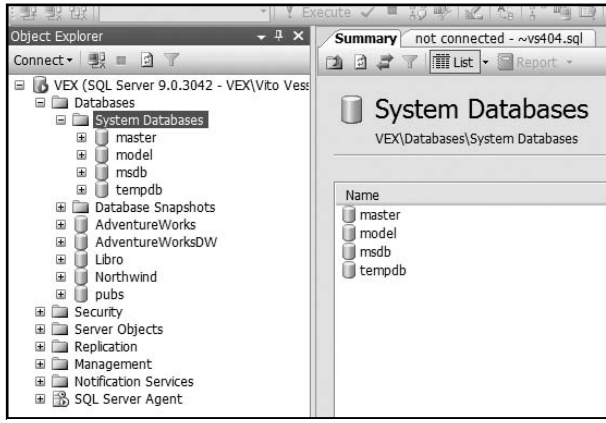


Figura 3.1: Database di sistema

3.1 TABELLE DEL CATALOGO DI DATABASE

Le tabelle di sistema del database master sono distinte dalle tabelle di sistema dei database creati dagli utenti. Le seconde costituiscono il Catalago di Database. Queste tabelle hanno la stessa struttura logica delle tabelle utente e pertanto si usa il medesimo approccio: per eseguirne l'inserimento, la modifica e la cancellazione di righe si procede attraverso il Data Manipulation Language, per effettuarne le interrogazioni, si fauso di query T-SQL. In realtà, nonostante l'inserimento, la modifica e la cancellazione di righe in queste tabelle sia possibile, può risultare altamente pericoloso e pertanto questo approccio viene scoraggiato dal produttore in favore dell'uso di stored procedure di sistema speci-

fiche che fanno le stesse operazioni ma con maggior cognizione di causa e tenendo conto di tutti gli aspetti di interdipendenza.
Di seguito viene riportato l'elenco delle principali tabelle di sistema. È da notare che ciascun oggetto del database è identificato da un nome univoco definito dall'utente al momento della creazione, qualora si tratti di un oggetto utente, e di un identificativo numerico univoco assegnato in maniera progressiva dal sistema al momento della creazione dell'oggetto, con un normale meccanismo di Identity come accede nelle nostre tabelle utente, a riprova del fatto che i dati nelle tabelle di sistema sono trattati alla stregua dei dati nelle tabelle utente.

COLONNA	DESCRIZIONE
id	L'identificativo univoco dell'oggetto all'interno del database
name	Il nome dell'oggetto del database
uid	L'identificativo del possesso dell'oggetto
type	Il tipo di oggetto del database. Può assumere i seguenti valori: C = vincolo check; D = predefinito; F = chiave esterna; K = chiave primaria o vincolo di univocità R = regola RF = stored procedure di replicazione S = tabella di sistema TR = trigger U = tabella ordinaria definita dall'utente V = vista X = stored procedure estesa
crdate	Data di creazione dell'oggetto nel database
parent_obj	Identificativo dell'oggetto padre. Non tutti gli oggetti hanno un padre, ma taluni si, ad esempio una chiave primaria (tipo K) ha per padre la tabella di cui rappresenta il vincolo. In questo modo è possibile risalire alla gerarchia di tutti gli oggetti del database.

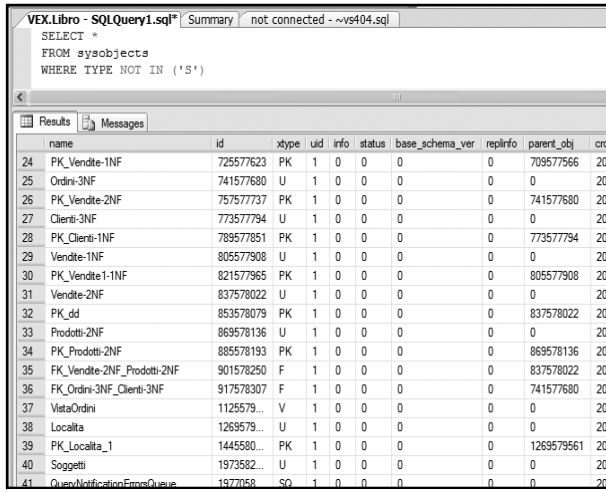
3.1.1 sysobjects

Si tratta della tabella di sistema principale di SQL Server. È presente sia nel database master che in tutti i database creati dall'utente. Contiene una riga per ciascun oggetto presente nel database e quindi ogni tabella, ogni stored procedure. Ogni vista ed ogni altro oggetto del database sono censiti come righe diquesta tabella. Di seguito se ne riportano i campi

Come si accennava in precedenza, tutte le tabella di sistema sono gestibili come normali tabelle create dall'utente, pertanto sarà possibile effettuare la seguente query di esempio:

```
SELECT *
FROM sysobjects
WHERE TYPE NOT IN ('S')
```

In Figura 3.2 è possibile osservare il risultato della query di esempio sul database Libro usato nella presente esposizione.


VEX.Libro - SQLQuery1.sql* | Summary | not connected - ~vs404.sql

```
SELECT *
FROM sysobjects
WHERE TYPE NOT IN ('S')
```

Results | Messages

	name	id	xtype	uid	info	status	base_schema_ver	replinfo	parent_obj	crdat
24	PK_Vendite-1NF	725577623	PK	1	0	0	0	0	709577566	2007
25	Ordini-3NF	741577680	U	1	0	0	0	0	0	2007
26	PK_Vendite-2NF	757577737	PK	1	0	0	0	0	741577680	2007
27	Clienti-3NF	773577794	U	1	0	0	0	0	0	2007
28	PK_Clienti-1NF	789577851	PK	1	0	0	0	0	773577794	2007
29	Vendite-1NF	805577908	U	1	0	0	0	0	0	2007
30	PK_Vendite1-1NF	821577965	PK	1	0	0	0	0	805577908	2007
31	Vendite-2NF	837578022	U	1	0	0	0	0	0	2007
32	PK_dd	853578079	PK	1	0	0	0	0	837578022	2007
33	Prodotti-2NF	869578136	U	1	0	0	0	0	0	2007
34	PK_Prodotti-2NF	885578193	PK	1	0	0	0	0	869578136	2007
35	FK_Vendite-2NF_Prodotti-2NF	901578250	F	1	0	0	0	0	837578022	2007
36	FK_Ordini-3NF_Clienti-3NF	917578307	F	1	0	0	0	0	741577680	2007
37	VistaOrdini	1125579...	V	1	0	0	0	0	0	2007
38	Localita	1269579...	U	1	0	0	0	0	0	2007
39	PK_Localita_1	1445580...	PK	1	0	0	0	0	1269579561	2007
40	Soggetti	1973582...	U	1	0	0	0	0	0	2007
41	QueryNotificationErrorsQueue	1977058...	SQ	1	0	0	0	0	0	2005

Figura 3.2: Interrogazione su sysobjects

3.1.2 syscolumns

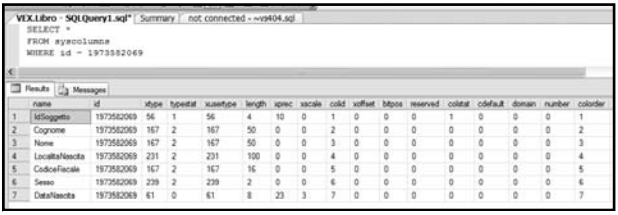
È presente sia nel database master che in tutti i database creati dall'utente. Contiene una riga per ciascuna colonne delle tabelle e delle viste presenti nel database e per ciascun parametro di stored procedure del database. Di seguito se ne riportano i campi più importanti:

COLONNA	DESCRIZIONE
id	Rappresenta l'identificativo univoco dell'oggetto nel quale compare la colonna
colid	Rappresenta l'identificativo univoco di colonna dell'oggetto rappresentato dal campo id
name	È il nome della colonna vera e propria

Si osservi al seguente query che estrae l'elenco delle colonne della tabella Soggetti definita nel database di esempio Libro:

```
SELECT *
FROM syscolumns
WHERE id = 1973582069 'id della tabella Soggetti
```

La query è filtrata per un id specifico che, nell'esempio, è proprio l'id della tabella Soggetti. In Figura 3.3 se ne può avere un saggio del risultato.



	name	id	xtype	typestat	xusertype	length	xprec	xscale	colid	xoffset	bitpos	reserved	colstat	cdefault	domain	number	colorder
1	IdSoggetto	1973582069	56	1	56	4	10	0	1	0	0	0	1	0	0	0	1
2	Cognome	1973582069	167	2	167	50	0	0	2	0	0	0	0	0	0	0	2
3	Nome	1973582069	167	2	167	50	0	0	3	0	0	0	0	0	0	0	3
4	LocalitaNascita	1973582069	231	2	231	100	0	0	4	0	0	0	0	0	0	0	4
5	CodiceFiscale	1973582069	167	2	167	16	0	0	5	0	0	0	0	0	0	0	5
6	Sesso	1973582069	239	2	239	2	0	0	6	0	0	0	0	0	0	0	6
7	DataNascita	1973582069	61	0	61	8	23	3	7	0	0	0	0	0	0	0	7

Figura 3.3: Interrogazione su syscolumns

3.1.3 sysindexes

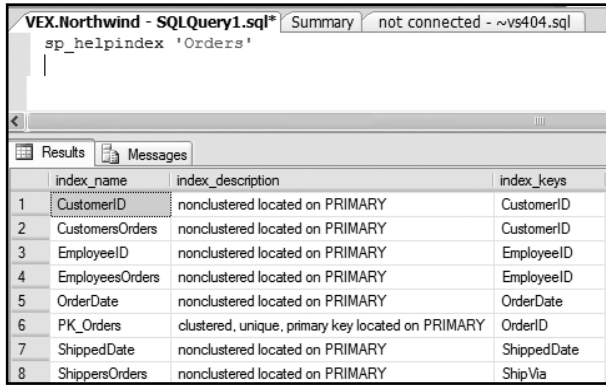
È presente sia nel database master che in tutti i database creati dall'utente. Contiene una riga per ciascun indice. La sua interpretazione e il suo uso sono però troppo complessi, pertanto è preferibile l'uso della stored procedure di sistema sp_helpindex che permette di riportare in formato decisamente più comprensibile lo stato degli indici delle tabelle del database. La sua sintassi è:

sp_helpindex @nome_oggetto

Ecco un'interrogazione di esempio degli indici della tabella Orders di Northwind:

```
sp_helpindex 'Orders'
```

In Figura 3.4 è possibile osservare il risultato dell'invocazione.



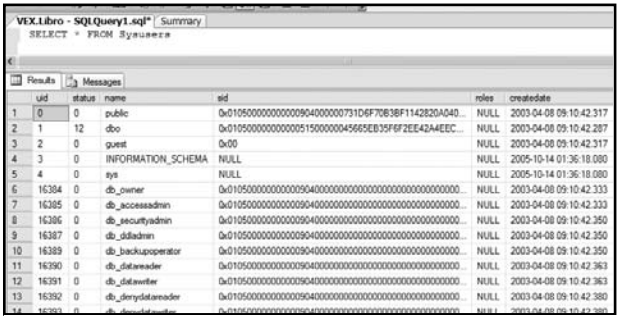
	index_name	index_description	index_keys
1	CustomerID	nonclustered located on PRIMARY	CustomerID
2	CustomersOrders	nonclustered located on PRIMARY	CustomerID
3	EmployeeID	nonclustered located on PRIMARY	EmployeeID
4	EmployeesOrders	nonclustered located on PRIMARY	EmployeeID
5	OrderDate	nonclustered located on PRIMARY	OrderDate
6	PK_Orders	clustered, unique, primary key located on PRIMARY	OrderID
7	ShippedDate	nonclustered located on PRIMARY	ShippedDate
8	ShippersOrders	nonclustered located on PRIMARY	ShipVia

Figura 3.4: Gli indici della tabella Orders di Northwind via sp_helpindex

3.1.4 sysusers

È presente sia nel database master che in tutti i database creati dall'utente. Contiene una riga per ciascun account di Windows, gruppo di Windows, login di SQL Server o ruolo di SQL Server. Di seguito se ne riportano i principali campi: In Figura 3.5 possiamo osservare una tipica interrogazione su questa tabella.

COLONNA	DESCRIZIONE
uid	L'identificativo numerico dell'utente, univoco nel database
sid	L'identificativo di sistema dell'utente creatore del database
name	Il nome dell'utente, anche quest'ultimo deve essere univoco nell'ambito del database



	uid	status	name	sid	roles	createdate
1	0	0	public	0x0105000000000009040000007 31D6F70B3BF1142820A040...	NULL	2003-04-08 09:10:42.317
2	1	12	dbo	0x0105000000000005150000004 5665EB35F6F2EE42A4EEC...	NULL	2003-04-08 09:10:42.287
3	2	0	guest	0x00	NULL	2003-04-08 09:10:42.317
4	3	0	INFORMATION_SCHEMA	NULL	NULL	2005-10-14 01:36:18.080
5	4	0	sys	NULL	NULL	2005-10-14 01:36:18.080
6	16384	0	db_owner	0x0105000000000009040000000000000000000000000000000...	NULL	2003-04-08 09:10:42.333
7	16385	0	db_accessadmin	0x0105000000000009040000000000000000000000000000000...	NULL	2003-04-08 09:10:42.333
8	16386	0	db_securityadmin	0x0105000000000009040000000000000000000000000000000...	NULL	2003-04-08 09:10:42.350
9	16387	0	db_ddladmin	0x0105000000000009040000000000000000000000000000000...	NULL	2003-04-08 09:10:42.350
10	16389	0	db_backupoperator	0x0105000000000009040000000000000000000000000000000...	NULL	2003-04-08 09:10:42.350
11	16390	0	db_datareader	0x0105000000000009040000000000000000000000000000000...	NULL	2003-04-08 09:10:42.363
12	16391	0	db_datawriter	0x0105000000000009040000000000000000000000000000000...	NULL	2003-04-08 09:10:42.363
13	16392	0	db_denydatareader	0x0105000000000009040000000000000000000000000000000...	NULL	2003-04-08 09:10:42.380
14	16393	0	db_denydatawriter	0x0105000000000009040000000000000000000000000000000...	NULL	2003-04-08 09:10:42.380

Figura 3.5: La tabella di sistema degli utenti del database

3.1.5 sysdatabases

Questa tabella di sistema appare nel solo database master e contiene una riga per ciascuno dei database presenti nell'istanza di SQL Server.

COLONNA	DESCRIZIONE
dbid	Identificativo numerico univoco del database nell'ambito dell'istanza
name	Nome simbolo univoco del database nell'ambito dell'istanza di SQL Server
sid	Identificativo di sistema del creatore del database
crdate	Data di creazione del database
filaname	Nome del file fisico che soggiace al database
status	Stato del database. Sono possibili diversi stati contemporaneamente sullo stesso database e pertanto viene usata la seguente codifica bitwise. Eccone alcuni: 1 = autoclose (ALTER DATABASE)

COLONNA	DESCRIZIONE
status	4 = select into/bulkcopy (ALTER DATABASE con l'opzione SET RECOVERY) 8 = trunc. log on chkpt (ALTER DATABASE con l'opzione SET RECOVERY) 16 = torn page detection (ALTER DATABASE) 32 = loading 64 = pre recovery 128 = recovering 256 = not recovered 512 = offline (ALTER DATABASE) 1024 = read only (ALTER DATABASE) 2048 = dbo use only (ALTER DATABASE con l'opzione SET RESTRICTED_USER) 4096 = single user (ALTER DATABASE) 16384 = ANSI null default (ALTER DATABASE) 32768 = emergency mode 65536 = concat null yields null (ALTER DATABASE) 4194304 = autoshrink (ALTER DATABASE) 1073741824 = cleanly shutdown

Ancora una volta, con una normale query su questa tabella, è possibile ottenere l'elenco dei database presenti nell'istanza, magari filtrati da una clausola di WHERE sullo stato usando l'operatore & di bitwise, come mostrato in Figura 3.6, per ottenere l'elenco dei soli database che dispongono dell'opzione concat null yields null.

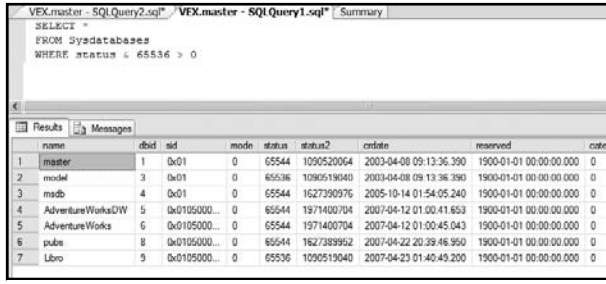


Figura 3.6: La tabella di sistema dei database

A volte può rendersi necessario conoscere il solo dbid di un certo database, ad esempio per impostare un filtro di profilazione nel SQL Server Profiler o per verificare lo stato dei lock in un certo database. A tal proposito esiste la comoda funzione di sistema db_id() che, a fronte del nome simbolico del database, ne restituisce proprio il dbid come mostrato di seguito nell'esempio:

```
print db_id('Libro')
'risposta --> 9
```

3.1.6 sysdepends

È presente sia nel database master che in tutti i database creati dall'utente. Contiene una riga per ciascuna relazione di dipendenza tra tabelle, viste e stored procedure. Di seguito se ne riportano i campi più importanti:

COLONNA	DESCRIZIONE
id	Identificativo numerico univoco della relazione di dipendenza nell'ambito dell'istanza
number	Il numero della stored procedure
depid	Il numero di identificazione dell'oggetto in relazione di dipendenza con l'oggetto identificato da id
depnumber	Il numero della stored procedure in relazione di dipendenza

Procediamo con il solito esempio selettivo di interrogazione sulla tabella di sistema in questione, come mostrato in Figura 3.7.

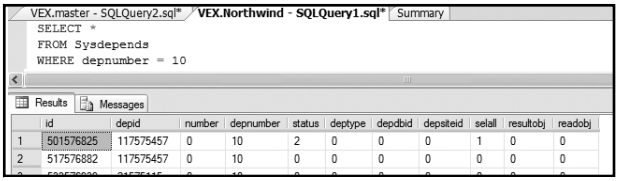


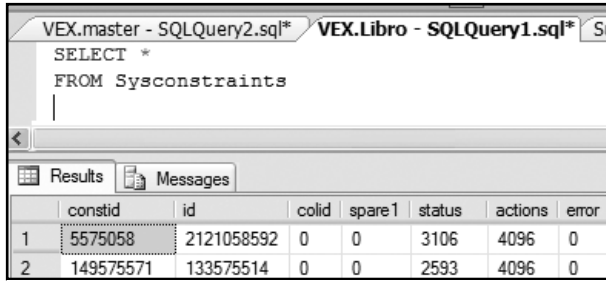
Figura 3.7: La tabella di sistema delle relazioni di dipendenza del database

3.1.7 sysconstraints

È presente sia nel database master che in tutti i database creati dall'utente. Contiene una riga per ciascun vincolo di integrità definito per un oggetto del database definito con una CREATE TABLE o con una ALTER TABLE. Di seguito se ne riportano i campi più importanti:

COLONNA	DESCRIZIONE
constid	Identificativo numerico univoco del vincolo nell'ambito dell'istanza
id	Identificativo univoco della tabella su cui è applicato il vincolo di integrita, come definito nella tabella sysobjects
colid	Identificativo univoco della colonna su cui è applicato il vincolo di integrita, come definito nella tabella syscolumns
status	Tipologia di vincolo di integrità, come elencato di seguito: 1 = vincolo PRIMARY KEY 2 = vincolo UNIQUE KEY 3 = vincolo FOREIGN KEY 4 = vincolo CHECK 5 = vincolo DEFAULT 6 = vincolo di colonna 7 = vincolo di tabella

In Figura 3.8 è possibile osservare un esempio di interrogazione su questa tabella di sistema.



	constid	id	colid	spare1	status	actions	error
1	5575058	2121058592	0	0	3106	4096	0
2	149575571	133575514	0	0	2593	4096	0

Figura 3.7: La tabella di sistema delle constraint del database

3.2 VISTE DI CATALOGO

Le Viste di Catalogo sono una novità di SQL Server 2005 e si affiancano alla normale interrogazione delle tabelle di sistema per conoscere la struttura e i metadati di ciascun database. Sono un metodo da preferirsi al primo che è mantenuto solo per ragioni di compatibilità. In pratica esiste una vista di catalogo per ciascuna delle principali tabelle di sistema. Osserviamo il seguente elenco di mappatura:

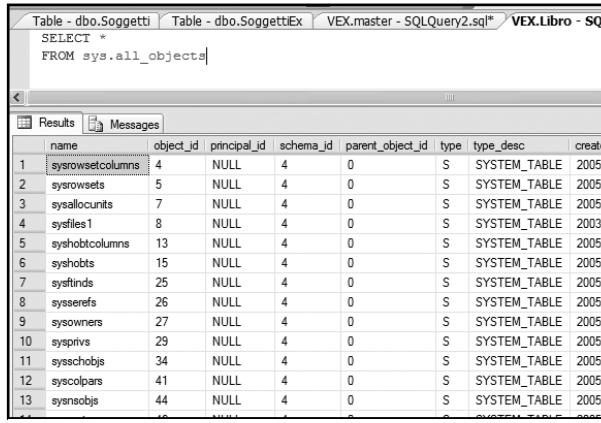
TABELLA DI SISTEMA	VISTA DI CATALOGO
sysdatabases	sys.databases
sysobjects	sys.objects (mostra i soli oggetti utente della tabella sysobjects) sys.system_objects (mostra i soli oggetti di sistema della tabella sysobjects) sys.all_objects (corrisponde esattamente alla tabella di sistema sysobjects)
syscolumns	sys.columns
sysusers	sys.users

3.2.1 sys.objects

Contiene una riga per ciascun oggetto presente nel database e quindi ogni tabella, ogni stored procedure, ogni vista ed ogni altro oggetto del database sono censiti come righe di questa tabella. Come già mostrato

COLONNA	DESCRIZIONE
name	Nome simbolico univoco dell'oggetto nell'ambito del database
object_id	Identificativo numerico univoco nell'ambito del database
schema_id	Identificativo dello schema a cui appartiene l'oggetto
type	Tipo dell'oggetto, rimappato direttamente sul campo type di sysobjects di cui, evidentemente, le condivide il dizionario di possibili valori.

nella tabella precedente, esistono in realtà tre viste di questo tipo: sys.objects, sys.system_objects e sys.all_objects. Tutte hanno la stessa struttura di campi. Di seguito se ne riportano i campi più importanti:
In Figura 3.9 è possibile osservare un esempio di interrogazione su questa tabella di sistema, in particolare su sys.all_objects.



	name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_d
1	sysrowsetcolumns	4	NULL	4	0	S	SYSTEM_TABLE	2005-10
2	sysrowsets	5	NULL	4	0	S	SYSTEM_TABLE	2005-10
3	sysallocunits	7	NULL	4	0	S	SYSTEM_TABLE	2005-10
4	sysfiles1	8	NULL	4	0	S	SYSTEM_TABLE	2003-04
5	syshobtcolumns	13	NULL	4	0	S	SYSTEM_TABLE	2005-10
6	syshobts	15	NULL	4	0	S	SYSTEM_TABLE	2005-10
7	sysftinds	25	NULL	4	0	S	SYSTEM_TABLE	2005-10
8	sysserefs	26	NULL	4	0	S	SYSTEM_TABLE	2005-10
9	sysowners	27	NULL	4	0	S	SYSTEM_TABLE	2005-10
10	sysprivs	29	NULL	4	0	S	SYSTEM_TABLE	2005-10
11	sysschobjs	34	NULL	4	0	S	SYSTEM_TABLE	2005-10
12	syscolpars	41	NULL	4	0	S	SYSTEM_TABLE	2005-10
13	sysnsobjs	44	NULL	4	0	S	SYSTEM_TABLE	2005-10

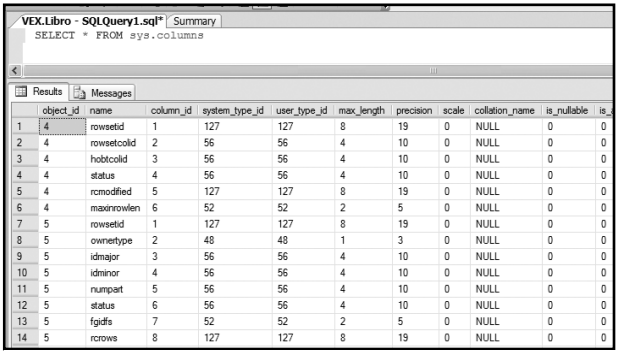
Figura 3.9: Interrogazione della Vista di Catalogo sys.all_objects

3.2.2 sys.columns

È mappata direttamente sulla tabella di sistema syscolumns e pertanto contiene una riga per ciascuna colonne delle tabelle e delle viste presenti nel database e per ciascun parametro di stored procedure del database. Di seguito se ne riportano i campi più importanti:

COLONNA	DESCRIZIONE
object_id	Rappresenta l'identificativo univoco dell'oggetto nel quale compare la colonna
column_id	Rappresenta l'identificativo univoco di colonna dell'oggetto rappresentato dal campo id
name	È il nome della colonna vera e propria

In Figura 3.10 possiamo osservare una tipica interrogazione su questa vista.



VEX.Libro - SQLQuery1.sql* | Summary

```
SELECT * FROM sys.columns
```

Results | Messages

	object_id	name	column_id	system_type_id	user_type_id	max_length	precision	scale	collation_name	is_nullable	is_
1	4	rowsetid	1	127	127	8	19	0	NULL	0	0
2	4	rowsetcolid	2	56	56	4	10	0	NULL	0	0
3	4	hobtcolid	3	56	56	4	10	0	NULL	0	0
4	4	status	4	56	56	4	10	0	NULL	0	0
5	4	rcmodified	5	127	127	8	19	0	NULL	0	0
6	4	maxinrowlen	6	52	52	2	5	0	NULL	0	0
7	5	rowsetid	1	127	127	8	19	0	NULL	0	0
8	5	ownertype	2	48	48	1	3	0	NULL	0	0
9	5	idmajor	3	56	56	4	10	0	NULL	0	0
10	5	idminor	4	56	56	4	10	0	NULL	0	0
11	5	numpart	5	56	56	4	10	0	NULL	0	0
12	5	status	6	56	56	4	10	0	NULL	0	0
13	5	fgidfs	7	52	52	2	5	0	NULL	0	0
14	5	rcrows	8	127	127	8	19	0	NULL	0	0

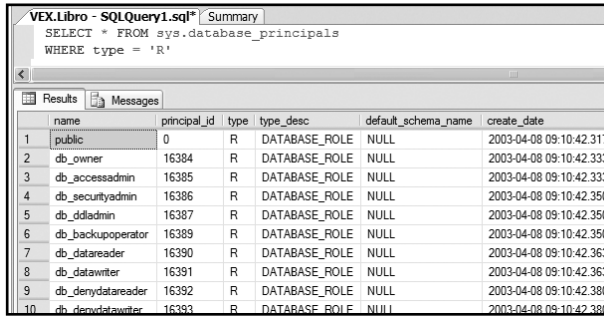
Figura 3.10: La Vista di Catalogo delle colonne

3.2.3 sys.database_principals

Contiene una riga per ciascuno degli oggetti di sicurezza presenti nel database e quindi per ciascun utente, gruppo e ruolo del database. Di seguito se ne riportano i campi più importanti:

COLONNA	DESCRIZIONE
name	È il nome univoco dell'oggetto di sicurezza (principal)
principal_id	Identificativo univoco dell'oggetto di sicurezza nell'ambito del database
type	Tipologia di principal. Può assumere i seguenti valori: S = SQL user U = Windows user G = Windows group A = Application role R = Database role C = User mapped to a certificate K = User mapped to an asymmetric key

Osserviamo, in Figura 3.11, un esempio di interrogazione di questa Vista di Catalogo per i soli tipi R.



	name	principal_id	type	type_desc	default_schema_name	create_date
1	public	0	R	DATABASE_ROLE	NULL	2003-04-08 09:10:42.317
2	db_owner	16384	R	DATABASE_ROLE	NULL	2003-04-08 09:10:42.333
3	db_accessadmin	16385	R	DATABASE_ROLE	NULL	2003-04-08 09:10:42.333
4	db_securityadmin	16386	R	DATABASE_ROLE	NULL	2003-04-08 09:10:42.350
5	db_ddladmin	16387	R	DATABASE_ROLE	NULL	2003-04-08 09:10:42.350
6	db_backupoperator	16389	R	DATABASE_ROLE	NULL	2003-04-08 09:10:42.350
7	db_datareader	16390	R	DATABASE_ROLE	NULL	2003-04-08 09:10:42.363
8	db_datawriter	16391	R	DATABASE_ROLE	NULL	2003-04-08 09:10:42.363
9	db_denydatareader	16392	R	DATABASE_ROLE	NULL	2003-04-08 09:10:42.380
10	db_denydatawriter	16393	R	DATABASE_ROLE	NULL	2003-04-08 09:10:42.380

Figura 3.11: La Vista di Catalogo dei principal

3.3 STORED PROCEDURE DI SISTEMA

Un'altra metodologia di accesso alle meta informazioni di sistema è costituita dalla chiamata alle stored procedure di sistema. Inoltre esse possono essere utilizzate non solo per la mera consultazione ma anche per operazioni di cambio del nome, di modifica o di cancellazione di tali oggetti. Questo approccio è da preferire all'accesso diretto in modifica alle tabelle di sistema o alle viste di catalogo perché garantisce maggiore affidabilità e consistenza nell'operazione. Alcune di queste stored procedure sono state già incontrate ed esaminate in precedenza e pertanto ci si limiterà a presentarne soltanto alcune altre rilevanti.

3.3.1 sp_help

Questa stored procedure visualizza le informazioni di uno o più oggetti del database. Se invocata senza parametri si limita ad elencare tutti gli oggetti di sistema, diversamente, se parametrizzata con il nome simbolico dell'oggetto da interrogare, restituirà un vero e proprio report completo di ogni informazione relativa all'oggetto in questione.

Si supponga di voler interrogare la tabella Soggetti del database di esempio Libro. Si procederà come di seguito:

```
USE Libro
GO
sp_help 'Soggetti'
GO
```

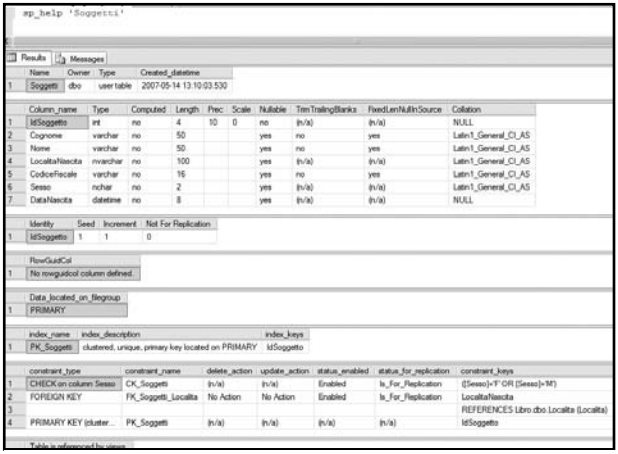


Figura 3.12: La potente sp_help in azione

Si osservi la Figura 3.12: essa ci mostra il risultato dell'interrogazione della tabella Soggetti attraverso la stored procedure in questione. Restituisce numerosi resultset:

- " un resultset relativo all'oggetto Soggetti stesso, corrispondente grosso modo alla riga di sysobjects relativa a questo oggetto (infatti sono visibili il nome, l'owner, la tipologia di tabella e la data di creazione);
- " un resultset relativo alle colonne della tabella, corrispondente grosso modo alle righe della tabella syscolumns filtrate per l'id del-

la tabella Soggetti;

- " un resultset relativo alla Identity impostata sul campo IdSoggetto della tabella, sempre che sia presente un identity nell'oggetto;
- " un resultset relativo alla eventuale presenza di tipi rowGuid nella tabella (nell'esempio specifico non vi sono colonne di questo tipo);
- " un resultset relativo al partizionamento fisico dei dati della tabella, utile qualora la tabella risulti partizionata;
- " un resultset relativo agli indici definiti su questa tabella, corrispondente alle informazioni accessibili attraverso la tabella di sistema sysindexes o alla stored procedure specifica di sistema sp_helpindex;
- " un resultset realtivo alle constraint presenti nella tabella (in effetti, nel caso specifico, è possibile osservare una constraint di tipo check sul campo Sesso, che può assumere isoli valori 'S' o 'N', una constraint di tipo chiave primaria di tipo CLUSTERED sul campo IdSoggetto e una constraint relativa ad una foreign key del campo LocalitaNascita sul campo chiave Localita della tabella Localita dello stesso database Libro);
- " infine, un resultset che indica se la tabella è usate in una o più viste e le informazioni di dettaglio relative (nessuna informazione di questo tipo viene restituita nell'esempio perché la tabella non viene adoperata in nessuna vista).

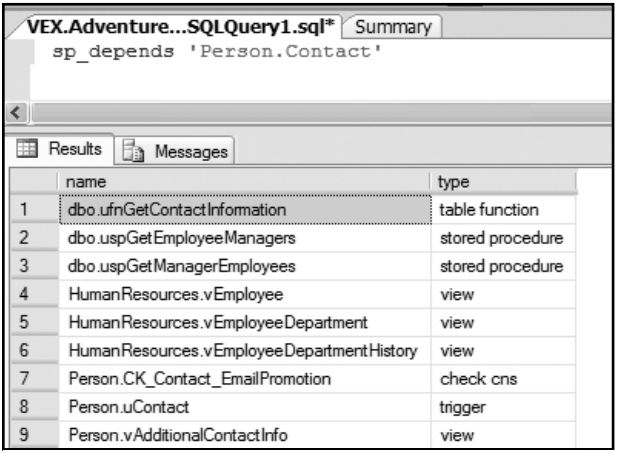
3.3.2 sp_depends

Questa stored procedure è molto utile perché permette di verificare le dipendenze tra tabelle, viste, trigger e stored procedure del database e pertanto risulta molto utile quando si deve realizzare un'analisi d'impatto di una modifica di un qualsiasi oggetto del database.
Proviamo ad esaminare le dipendenze della tabella Person.Contact del database di esempio AdventureWorks:

```
USE AdventureWorks
GO
```

```
sp_depends 'Person.Contact'
GO
```

SQL Server ci mosterà l'elenco di tutti gli oggetti del database dipendenti da quello passato come argomento, precisandone, per ciascuno di essi, il nome e la tipologia (tabella, stored procedure, funzione, ecc...), come mostrato in Figura 3.13.



	name	type
1	dbo.ufnGetContactInformation	table function
2	dbo.uspGetEmployeeManagers	stored procedure
3	dbo.uspGetManagerEmployees	stored procedure
4	HumanResources.vEmployee	view
5	HumanResources.vEmployeeDepartment	view
6	HumanResources.vEmployeeDepartmentHistory	view
7	Person.CK_Contact_EmailPromotion	check cns
8	Person.uContact	trigger
9	Person.vAdditionalContactInfo	view

Figura 3.13: Analisi delle dipendenze tra oggetti con sp_depends

3.3.3 sp_helptext

Questa comoda stored procedure si limita a resituire un'informazione molto utile: il corpo testuale degli oggetti definiti interamente in T-SQL e cioè stored procedure, viste, trigger e funzioni. Si passa il nome dell'oggetto da esaminare come argomento e la stored procedure restituisce un resultset in cui ciascuna riga contiene una linea del corpo del T-SQL dell'oggetto, suddiviso per line break.
Si osservi il seguente esempio:

```
USE AdventureWorks
GO
sp_helptext 'usp_LookupConversionRate'
GO
```

Nel caso specifico sarà richiesto di esaminare il corpo della stored procedure usp_LookupConversionRate. Il risultato sarà molto utile e persino decorativo, come mostrato in Figura 3.14.

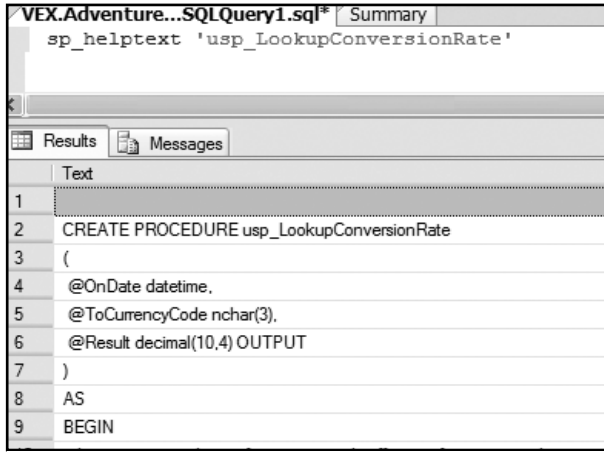


Figura 3.14: Come ottenere un output formattato del corpo degli oggetti definiti in T-SQL puro

3.4 FUNZIONI SCALARI PREDEFINITE

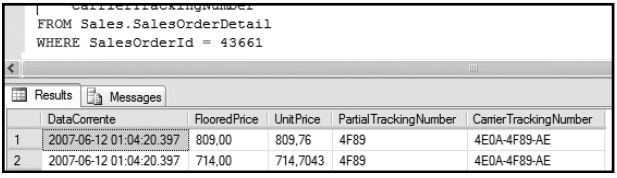
SQL Server fornisce numerose funzioni scalari che possono essere utilizzate all'interno di query, di stored procedure e di funzioni utente per costruire e restituire valori scalari. Esistono centinaia di queste funzioni built-in in SQL Server 2005, ma fondamentale possiamo suddividerle nei seguenti gruppi logici:

- " funzioni numeriche;
- " funzioni di data;
- " funzioni di stringa;
- " funzioni di testo/immagine;
- " funzioni di sistema.

Le funzioni scalari di sistema vanno usate esattamente come le funzioni scalari definite dall'utente. Si osservi il seguente uso all'interno di una banale query sulla tabella Sales.SalesOrderDetail di Northwind:

```
SELECT
    GetDate() AS DataCorrente,
    FLOOR(UnitPrice) AS FlooredPrice, UnitPrice,
    SUBSTRING(CarrierTrackingNumber, 6, 4) AS PartialTrackingNumber,
    CarrierTrackingNumber
FROM Sales.SalesOrderDetail
WHERE SalesOrderId = 43661
```


Si può osservare l'uso di diverse funzioni di sistema, con o senza parametri da passare come argomento delle stesse. In particolare sono state adoperate la funzione GetDate per ottenere la data corrente dal sistema, la funzione FLOOR per ottenere l'arrotondamento per eccesso di un numero decimale e la funzione SUBSTRING che restituisce una sottostringa di una stringa più ampia. In Figura 3.15 si può osservare il resultset generato dalla query di esempio.



	DataCorrente	FlooredPrice	UnitPrice	PartialTrackingNumber	CarrierTrackingNumber
1	2007-06-12 01:04:20.397	809,00	809,76	4F89	4E0A-4F89-AE
2	2007-06-12 01:04:20.397	714,00	714,7043	4F89	4E0A-4F89-AE

Figura 3.15: Esempio di uso di funzioni scalari predefinite in una query

3.4.1 Funzioni numeriche

Le funzioni numeriche sono funzioni matematiche finalizzate al calcolo e alla modifica di valori numerici. Di seguito se ne riportano le più rilevanti:

FUNZIONE	DESCRIZIONE
ABS(n)	Restituisce il valore assoluto (positivo) dell'espressione numerica specificata.
ACOS(n)	Restituisce l'angolo, espresso in radianti, il cui coseno corrisponde all'espressione di tipo float specificata (denominato anche arcocoseno).
ASIN(n)	Restituisce l'angolo, espresso in radianti, il cui seno corrisponde all'espressione float specificata. Il valore restituito viene definito anche arcoseno.
ATAN(n)	Restituisce l'angolo, espresso in radianti, la cui tangente corrisponde all'espressione di tipo float specificata (definito anche arcotangente).
ATN2(n, m)	Restituisce l'angolo, in radianti, tra l'asse X positivo e il raggio dall'origine al punto (x, y), dove x e y sono i valori delle due espressioni float specificate.
CEILING(n)	Restituisce il più piccolo valore integer maggiore o uguale all'espressione numerica specificata.
COS(n)	Restituisce il coseno dell'angolo specificato, espresso in radianti, nell'espressione specificata.
COT(n)	Restituisce la cotangente dell'angolo specificato, espressa in radianti, nell'espressione float specificata.
DEGREES(n)	Restituisce l'angolo in gradi corrispondente all'angolo specificato in radianti.
EXP(n)	Restituisce il valore esponenziale dell'espressione float specificata.
FLOOR(n)	Restituisce il valore integer maggiore che risulta minore o uguale all'espressione numerica specificata.
LOG(n)	Restituisce il logaritmo naturale dell'espressione float specificata.

FUNZIONE	DESCRIZIONE
LOG10(n)	Restituisce il logaritmo in base 10 dell'espressione float specificata.
PI()	Restituisce il valore della costante pi greco.
POWER(n, y)	Restituisce il valore dell'espressione specificata elevato alla potenza indicata y.
RADIANS(n)	Restituisce l'equivalente in radianti dell'espressione numerica specificata espressa in gradi.
RAND(n)	Restituisce un valore float casuale compreso tra 0 e 1.
ROUND(n, len, tipo)	Restituisce un valore numerico arrotondato alla lunghezza o alla precisione specificata. Il tipo indica il formato di conversione e può assumere i valori tinyint, smallint o int.
SIGN(n)	Restituisce il segno positivo (+1), zero (0) o il segno negativo (-1) dell'espressione specificata.
SIN(n)	Restituisce il seno dell'angolo specificato, espresso in radianti, in un'espressione numerica approssimata di tipo float.
SQRT(n)	Restituisce la radice quadrata del valore float specificato.
SQUARE(n)	Restituisce il quadrato del valore float specificato.
TAN(n)	Restituisce la tangente dell'espressione di input.

3.4.2 Funzioni di data

Le funzioni di data, come evidentemente si evince dal loro nome, agiscono sulle date e sulle ore calcolandone parti o interi e restituendo a loro volta date o intervalli temporali. Esse agiscono su intervalli temporali prestabiliti e che sono riassunti di seguito:

yy (anno);
qq (trimestre);
mm (mese);
dy (giorno dell'anno, da 1 a 366);
dd (giorno);
dw (giorno della settimana);
wk (settimana);

hh (ora);
mi (minuti);
ss (secondi;
ms (millisecondi).
Di seguito se ne riportano le più rilevanti:

FUNZIONE	DESCRIZIONE
GETDATE()	Restituisce la data e l'ora di sistema correnti nel formato interno standard di SQL Server 2005 per i valori datetime.
DATEPART (datepart, data)	Restituisce un valore integer che rappresenta la parte specificata della data indicata. L'argomento datepart può essere del tipo descritto all'inizio del paragrafo.
DATEANAME (datepart, data)	Restituisce una stringa di caratteri che rappresenta la parte specificata della data indicata. È applicabile solo a quei datepart che hanno un nome simbolico come ad esempio i nomi dei mesi e i giorni della settimana.
DATEDIFF (datepart, data_inizio, data_fine)	Restituisce il numero di limiti di data e ora che si sovrappongono tra due date specificate.
DATEADD (datepart, incremento, data)	Restituisce un nuovo valore datetime basato sull'aggiunta di un incremento alla data specificata.

Funzioni di stringa

Le funzioni di stringa manipolano stringhe effettuandone modifiche, concatenazioni e trasformazioni e restituendo le stringe risultati. Di seguito se ne riportano le più rilevanti:

FUNZIONE	DESCRIZIONE
ASCII(carattere)	Restituisce il codice ASCII corrispondente al primo carattere a sinistra in una stringa di caratteri.

FUNZIONE	DESCRIZIONE
CHAR(codice_numerico)	Converte un codice ASCII di tipo int in un carattere.
CHARINDEX(expr1, expr2[, loc_partenza])	Restituisce il punto iniziale dell'espressione specificata in una stringa di caratteri. expr1 include la sequenza di caratteri che si desidera trovare. L'espressione expr2, in genere una colonna, in cui viene eseguita la ricerca della sequenza specificata. Loc_partenza indica la posizione da cui iniziare la ricerca di expression1 in expression2. Se loc_partenza viene omesso, è un numero negativo oppure è uguale a zero.
DIFFERENCE (expr1, expr2)	Restituisce un valore integer che indica la differenza tra i valori SOUNDEX di due espressioni di caratteri.
LOWER (expr)	Restituisce un'espressione di caratteri dopo aver convertito i caratteri maiuscoli in caratteri minuscoli.
LTRIM(expr)	Restituisce una espressione di caratteri dopo aver rimosso gli spazi vuoti iniziali.
NEWID()	Crea un valore univoco di tipo uniqueidentifier.
REPLICATE(expr, numrip)	Ripete un valore stringa il numero di volte specificato.
REVERSE(expr)	Restituisce un'espressione di caratteri nell'ordine inverso.
RIGHT(expr)	Restituisce la parte finale di una stringa di caratteri, di lunghezza pari al numero di caratteri specificato.
RTRIM(expr)	Restituisce una espressione di caratteri dopo aver rimosso gli spazi vuoti finali.
SOUNDEX(expr)	Restituisce un codice di quattro caratteri (SOUNDEX) per valutare la similarità di due stringhe.
SPACE(numero_spazi)	Restituisce una stringa di spazi ripetuti.
STR(numero, lunghezza [,decimali])	Restituisce dati di tipo carattere convertiti da dati di tipo numerico. I parametri lunghezza e decimali indicano rispettivamente il numero di cifre per rappresentare la parte intera e il numero di cifre decimali.

FUNZIONE	DESCRIZIONE
STUFF(expr, inizio, fine, lunghezza, expr2)	Elimina un determinato numero di caratteri nella stringa expr e inserisce un altro set di caratteri a partire dal punto specificato nella stringa expr2.
SUBSTRING(expr, inizio, fine)	Restituisce parte di un'espressione di tipo carattere, binario, testo o immagine.
UPPER	Restituisce un'espressione di caratteri dopo aver convertito i caratteri maiuscoli in caratteri maiscoli.

3.4.3 Funzioni di testo/immagine

Queste colonne agiscono su colonne di tipo testo/immagine. Di seguito se ne riportano le più rilevanti:

FUNZIONE	DESCRIZIONE
PATINDEX ('%pattern%', espressione)	Restituisce la posizione di inizio della prima occorrenza di un criterio di ricerca in un'espressione specificata, oppure zero se il criterio di ricerca non viene trovato, in tutti i dati di tipo carattere e text validi.
TEXTPTR (colonna)	Restituisce il valore del puntatore di testo corrispondente a una colonna di tipo text, ntext o image in formato varbinary. È possibile utilizzare il valore del puntatore di testo recuperato nelle istruzioni READTEXT, WRITETEXT e UPDATETEXT.

3.4.4 Funzioni di sistema

Queste funzioni sono da ritenersi un'estensione dell'argomento precedente relativo agli oggetti e al catalogo di sistema. Ciascun oggetto di sistema è identificato da un codice numero univoco nell'ambito del database; le funzioni di sistema utilizzano proprio questi id per identificare ed effettuare ricerche su singoli oggetti. Di seguito se ne riportano le più rilevanti:

FUNZIONE	DESCRIZIONE
COLAESCE(expr, n)	Restituisce la prima espressione non Null tra i relativi argomenti.
COL_LENGTH ('tabella', 'colonna')	Restituisce la lunghezza definita di una colonna, espressa in byte.
COL_NAME (idtabella, idcolonna)	Restituisce il nome di una colonna corrispondente al numero di identificazione di tabella e al numero di identificazione di colonna specificati.
DATALENGTH(expr)	Restituisce il numero di byte utilizzati per rappresentare un'espressione.
GETANSINULL (['database'])	Restituisce l'impostazione predefinita relativa al supporto dei valori Null del database per la sessione corrente.
NULLIF(expr1, expr2)	Restituisce un valore Null se le due espressioni specificate sono uguali.

NUOVE FUNZIONALITÀ IN SQL SERVER 2005

Quest'ultimo capitolo non presenta una coerenza e continuità di argomenti, ma si pone quasi come una sorta di raccolta di appendici che richiamano alcune delle più interessanti novità di SQL Server 2005 che non è stato possibile affrontare direttamente nel corso del volume. Pertanto è possibile leggere il capitolo anche non in modo lineare, ma concentrandosi solo sui paragrafi di interesse e nell'ordine che si preferisce.

4.1 COLONNE CALCOLATE PERSISTENTI

Le query SQL sono piene di campi calcolati, che sono una funzionalità molto comoda ma spesso piuttosto onerosa per il database engine. Soprattutto perché a volte è quasi inevitabile riproporre sempre gli stessi campi ad ogni interrogazione sulla stessa tabella. Un esempio può essere il campo calcolato TotaleRiga, di una riga d'ordine, che per effetto delle regole di normalizzazione non può essere inserito come campo fisico, ma deve essere sempre calcolato come PrezzoUnitario * Quantita. SQL Server, però, offre la possibilità di definire i campi calcolati nella creazione di una tabella, in modo che possa essere utilizzata nelle interrogazioni come un normale campo fisico, ma il suo valore viene calcolato al volo ogni volta. Osserviamo un esempio:

```
CREATE TABLE RigheOrdine
(
    CodRiga INT,
    CodOrdine INT,
    CodProdotto INT,
```

```
    PrezzoUnitario FLOAT,
    Quantita INT,
    TotaleRiga AS PrezzoUnitario * Quantita
)
```

Questo era già possibile con SQL Server 2000, ma l'approccio ha un grosso limite: le colonne così definite, per la loro natura volatile non sono indicizzabili, costringendo così ad un table scan ogni volta che se ne interroga il valore in una WHERE. SQL Server 2005 introduce le colonne calcolate persistenti che consentono di salvare fisicamente nel database il risultato del calcolo, alla stregua di un normale campo fisico non calcolato, ma con il vantaggio di venire aggiornato in automatico dal sistema ogni volta che cambia il suo valore per effetto della variazione dei valori di uno o più campi che compongono la sua espressione di calcolo. Questi campi sono indicizzabili e dunque non offrono svantaggi prestazionali come in passato. Osserviamo la sintassi DML di definizione di un campo calcolato persistente:

```
CREATE TABLE RigheOrdine
(
    CodRiga INT,
    CodOrdine INT,
    CodProdotto INT,
    PrezzoUnitario FLOAT,
    Quantita INT,
    TotaleRiga AS PrezzoUnitario * Quantita PERSISTED
)
```

4.2 M.A.R.S. (MULTIPLE ACTIVE RESULT SET)

La fruizione dei dati di un database è possibile in ADO.NET abbina-

to a SQL Server in diversi modi. Tra le tante novità che questa nuova versione di SQL Server introduce, ce n'è una estremamente comoda chiamata MARS: Multiple Active Result Set. Essa funziona in abbinamento alla piattaforma .NET 2.0 e in particolare alla libreria di accesso ai dati ADO.NET 2.0.

Quando abbiamo bisogno di accedere a dei dati in modo veloce e leggero non possiamo che optare per il DataReader che consente di leggere un flusso di dati ricevuto da un database in modalità read-only, forward-only. Esso impone che la connessione al database venga mantenuta aperta finché tutti i dati non sono stati letti. Fino alla versione ADO.NET 1.1 e Microsoft SQL Server 2000 questo model lo di programmazione introduceva una limitazione: l'impossibilità di eseguire altre operazioni sul database finché non si è liberata la connessione, pena il sollevamento dell'eccezione che indicava la preesistenza di un DataReader già aperto associato alla connessione corrente. Non era possibile aprire altri DataReader nella stessa connessione prima che questo non venisse chiuso. Osserviamo il seguente esempio: facendo riferimento al solito database di esempio Northwind, consideriamo la tabella Orders (ordini) e le sue righe Order Details. Scorrendo la lista di tutti gli ordini, per ciascun ordine vogliamo determinare il valore dell'ordinato e cioè la somma del valore unitario di tutte le righe moltiplicato per la quantità ordinata per ciascuna riga.

L'algoritmo più intuitivo prevede che:

- si leggano le testate dalla tabella Orders;
- per ciascun ordine, usando la chiave primaria OrderId, si esegua una query che sommi il valore delle righe di ciascun ordine (quantità ordinata x valore unitario).

Per compiere questa operazione dobbiamo necessariamente eseguire due comandi T-SQL distinti sul database, ma sempre sulla stessa connessione. Cosa che, come abbiamo visto, non potremmo fare

con il DataReader.
Il Multiple Active Resultset Set ci viene in aiuto perché ci permette di eseguire comandi distinti attraverso la stessa connessione. Osserviamo il seguente esempio in C# (.NET 2.0):

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace MARS {
   class Program {
      static void Main( string[] args ) {
         SqlConnection myConnection = new SqlConnection();
         //Va impostato il parametro MultipleActiveResultSets sulla stringa
                                                        di connessione
        myConnection.ConnectionString = "Data Source=(local);Integrated
                              Security=SSPI;Persist Security Info=False;Initial
                        Catalog=Northwind;MultipleActiveResultSets=True";
         SqlCommand myCommand = myConnection.CreateCommand();
         myCommand.CommandType = CommandType.Text;
         myCommand.CommandText = "Select OrderId from Orders";
         myConnection.Open();
         SqlDataReader myDataReader = myCommand.ExecuteReader();
         while ( myDataReader.Read() ) {
            SqlCommand mySecondCommand =
                                             myConnection.CreateCommand();
            mySecondCommand.CommandType = CommandType.Text;
           mySecondCommand.CommandText = "Select cast(Sum(UnitPrice
    * Quantity) as float) as Venduto from [Order Details] where OrderId =
                                                        @OrderId";
            SqlParameter parameter =
```

```
                                        mySecondCommand.CreateParameter();
            parameter.ParameterName = "@OrderId";
            parameter.Value = (int)myDataReader["OrderId"];
            mySecondCommand.Parameters.Add( parameter );
            //Eseguo il secondo comando sulla stessa connessione
           double Ordinato = (double)mySecondCommand.ExecuteScalar();
            Console.WriteLine( "Order Id: {0}\r\nTotale Ordinato: {1}\r\n-----
          \r\n", myDataReader["OrderId"].ToString(), Ordinato.ToString() );
        }
        myConnection.Close();
    }
  }
}
```

Per sfruttare MARS la connectionstring deve essere predisposta impostando il parametro MultipleActiveResultSets = true nella stringa di connessione. La cosa va fatta solo per le connessioni che realmente necessitano di questa funzionalità, pena un decadimento delle prestazioni.

A questo punto viene prima eseguita una selezione su tutti i record della tabella Orders il cui risultato è un DataReader. Eseguendo un ciclo su tutti i record, viene recuperato l'OrderId che viene passato alla seconda query che effettua il conteggio dei quantitativi venduti. Impostare a true il parametro MultipleActiveResultSets nella stringa di connessione anche quando non serve, potrebbe avere impatti negativi sul fronte delle prestazione, dunque il suo uso va attentamente ponderato. Tuttavia la sua introduzione, ove serva, fornisce un'estrema semplificazione del codice di programmazione.

4.3 SUPPORTO MIGLIORATO AL FORMATO XML

Il supporto al formato XML in SQL Server è stato introdotto già dal-

la precedente versione 2000 grazie alla clausola FOR XML che consentiva (e consente) di restituire un resultset in formato XML oppure utilizzando OPENXML. SQL Server 2005 estende grandemente il suo supporto a questo formato introducendo l'implementazione del nuovo tipo di dati XML. Vi erano in passato però una serie di limitazioni: ad esempio, per memorizzare interamente un file XML nel database, era necessario inserire il contenuto in campi text o nei campi BLOB perdendo le potenzialità del formato XML. Con SQL Server 2005, invece, adesso abbiamo un tipo di dati nativo XML ovvero possiamo utilizzarlo come campo di una tabella o come variabile di input/output per stored procedure e funzioni. Dunque, all'interno del campo XML, si potranno inserire direttamente interi documenti o anche solo porzioni well-formed. Tale condizione sarà direttamente verificata dal parser xml interno di SQL Server 2005 che ne segnalerà l'errore analogamente a quanto viene quanto avviene con gli invalid cast. Quindi se proviamo ad inserire nel database un file non well-formed sarà direttamente SQL Server che ci avviserà che l'operazione non è possibile e quindi la INSERT o un UPDATE che vadano a modificare non avrà esecuzione. Al fine di meglio comprendere questa funzionalità, procediamo con un semplice esempio. Costruiamo una semplice tabella che contiene un campo di tipo XML:
CREATE TABLE TabellaEsempio

```
(
    IdRec int IDENTITY(1,1) NOT NULL,
    ValoreXML XML NULL
)
```

La tabella, a parte la solita chiave primaria identity, contiene un campo ValoreXML del nuovo data type XML. Siamo dunque pronti ad inserire un record in questa tabella:

```
INSERT INTO TabellaEsempio
```

```
(ValoreXML)
VALUES('<?xml version="1.0" encoding="utf-8" ?>
<Indice>
    <Capitolo>
        <Titolo>Cenni sul database relazionale</Titolo>
        <Sommario>Introduzione e storia del modello
                                        relazionale</Sommario>
        <Pagine>25</Pagine>
    </Capitolo>
    <Capitolo>
        <Titolo>Installazione e amministrazione di SQL Server
                                                2005</Titolo>
        <Sommario>La parte piu noiosa da scrivere di tutto i
                                        volume</Sommario>
    <Pagine>50</Pagine>
    </Capitolo>
</Indice>')
```

Il documento XML di esempio rappresenta l'estratto dell'indice del volume che state leggendo. Esso contiene un nodo principale Indice e diversi sottonodi Capitolo. Ciascun Capitolo è costituito da un nodo Titolo, da un nodo Sommario ed da un nodo Pagine (il numero di pagine del capitolo). Il documento XML è corretto nel senso che è well formed (è sintatticamente corretto, ad esempio tutti i nodi di aprono e si chudono correttamente, non ci sono caratteri non consentiti, esiste un solo nodo radice, ecc...). Infatti SQL Server non segnala alcun errore e consente l'inserimento del record. Proviamo ad effettuare una piccola modifica all'xml da inserire, magari usando un carattere non consentito. Si osservi il seguente documento XML errato:

```
<?xml version="1.0" encoding="utf-8" ?>
<Indice>
```

```
<Capitolo>
<Titolo>Funzionalità nuove a avanzate di SQL Server 2005</Titolo>
<Sommario>Il capitolo corrente</Sommario>
<Pagine>25</Pagine>
</Capitolo>
</Indice>
```

La lettera "à" di Funzionalità è accentata e quindi non fa parte dell'alfabeto UTF-8, come descritto nel preambolo del documento. SQL Server ci segnalerà un inequivocabile:
Msg 9420, Level 16, State 1, Line 9
XML parsing: line 10, character 22, illegal xml character
Per eseguire un'interrogazione su tabelle contenenti campi di tipo XML è possibile con la vecchia e rassicurante SELECT:
SELECT IdRec, ValoreXML FROM TabellaEsempio
Oppure utilizzare la vecchia clausola FOR XML per ritornare in forma XML l'intera riga, compreso il valore scalare tradizionale IdRec:
SELECT IdRec, ValoreXML FROM TabellaEsempio FOR XML AUTO
La Figura 4.1 ci mostra propria la resa nella griglia di SQL Server Management Studio delle due query. Cliccando sul contenuto del campo ValoreXML, Management Studio apre un nuovo documento, questa volta di tipo XML, in cui mostra il contenuto XML del campo con un layout simile a quello usato dai browser per rappresentare i

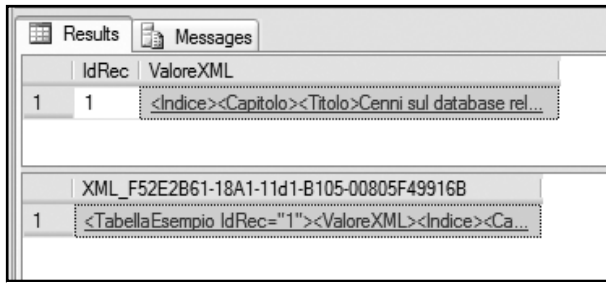


Figura 4.1: Un resultset XML nella griglia.

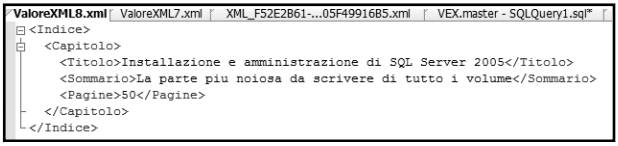


Figura 4.2: Il contenuto di un campo di tipo XML

documenti di questo formato, come si evince dalla figura 4.2.

4.3.1 XPath e XQuery

Il linguaggio SQL è nato per trattare nativamente i dati relazionali, ma forse è meno adeguato per formati strutturati e gerarchici come XML. Per interrogare questo formato esistono da anni diversi linguaggi. Il più diffuso è XPath che è alla base di quasi tutti i DOM XML attualmente in circolazione.

Attualmente, però, si sta imponendo un nuovo e più efficace standard: XQuery. Lo standard non è ancora definitivo, ma Microsoft, che fa parte del gruppo tecnico di definizione dello standard in seno al W3C, ne ha già implementato una versione draft in SQL Server 2005. XQuery utilizza una serie di istruzioni che possono essere raccolte ed identificate con la sigla FLOWR (For Let Order_By Where Return) dove ogni lettera corrisponde alla lettera iniziale di ogni istruzione.

Proviamo a realizzare alcuni esempi di query, ma prima di fare questo dobbiamo preparare la nostra precedente tabella di esempio per ospitare un numero più consistente di record sui quali effettuare l'interrogazione. Eseguiamo tre INSERT:

```
INSERT INTO TabellaEsempio
(ValoreXML)
VALUES('<?xml version="1.0" encoding="utf-8" ?>
<Indice>
<Capitolo>
<Titolo>Cenni sul database relazionale</Titolo>
```

```
<Sommario>Introduzione e storia del modello relazionale</Sommario>
<Pagine>40</Pagine>
</Capitolo>
</Indice>')

INSERT INTO TabellaEsempio
(ValoreXML)
VALUES('<?xml version="1.0" encoding="utf-8" ?>
<Indice>
<Capitolo>
<Titolo>Il linguaggio Transact-SQL</Titolo>
<Sommario>SQL per tutti</Sommario>
<Pagine>70</Pagine>
</Capitolo>
</Indice>')

INSERT INTO TabellaEsempio
(ValoreXML)
VALUES('<?xml version="1.0" encoding="utf-8" ?>
<Indice>
<Capitolo>
<Titolo>Installazione e amministrazione di SQL Server 2005</Titolo>
<Sommario>La parte piu noiosa da scrivere di tutto i volume</Sommario>
<Pagine>50</Pagine>
</Capitolo>
</Indice>')
```

Abbiamo introdotto tre righe con altrettanti documenti XML (Capitoli, nel nostro esempio) nella tabella. Con la seguente query XPath vogliamo estrarre i soli capitoli che hanno più di 40 pagine e cioè che il contenuto del sottonodo Pagine sia maggiore di 40:

```
SELECT IdRec, ValoreXML.query('/Indice/Capitolo[Pagine>40]')
```

```
FROM TabellaEsempio
```

La query è contenuta nella proprietà .query del campo ValoreXML che è di tipo XML. In Figura 3.3 è possibile osservare il risultato della query nel Query Editor.

Figura 4.3: Una query XPath

Riscriviamo l'analoga query nella nuova sintassi XQuery, più versatile e potente:

```
SELECT IdRec, ValoreXML.query(
'for $a in /Indice/Capitolo
where $a/Pagine>40
order by $a
return $a')
FROM TabellaEsempio
```

Il risultato è esattamente lo stesso ottenuto con la query XPath. Anche l'interpretazione della query è la stessa: la clausola for dice di cercare in tutti i sottonodi della gerarchia Indice/Capitolo, la where dice di cercare tutti i sottonodi Pagine che hanno valore maggiore di 40, la clausola where di restituire per intero quanto trovato (è possibile infatti restituirne anche dei sottonodi). L'unica apparente differenza è nella clausola group by che consente di restituire il risultato ordinato. In realtà Xquery è estramente più potente e complesso di quanto appaia nell'e-

sempio, ma lo studio della sua sintassi non è argomento di questo volume. Lo scopo, invece, era solo mostrare il livello di sofisticazione che ha raggiunto il supporto al formato XML in SQL Server 2005, tanto da far stare stretta a questo prodotto la definizione di mero RDMS.

4.3.2 Modificare i dati dei campi XML

XQuery consente anche di effetturare modifiche al contenuto dei campi XML. Immaginiamo di voler introdurre un attributo Completato a ciascun nodo Capitolo, preimpostando il valore a "NO". Osserviamo la seguente istruzione di UPDATE:

```
UPDATE TabellaEsempio
SET ValoreXML.modify('insert attribute Completato {"NO"}
                                                    into(/Indice[1])')
```

Questa istruzione consente di aggiungere un attributo Completato nel nodo radice Indice. Otterremo così, ad esempio:

```
<Indice Completato="NO">
 <Capitolo>

  <Titolo>Il linguaggio Transact-SQL</Titolo>

  <Sommario>SQL per tutti</Sommario>
  <Pagine>70</Pagine>

 </Capitolo>
</Indice>
```

XQuery offre anche la possibile di modificare i valori oltre che inserirne di nuovi, ma si rimanda alla documentazione specifica sull'argomento per approfondimenti.

4.4 NOVITÀ NELLA GESTIONE DELLA SICUREZZA

SQL Server 2005 introduce un modello di gestione della sicurezza completamente rivisitato. Osserviamone gli aspetti salienti. Un nuovo approccio per lo sviluppo del software è stato avviato con l'iniziativa Trustworthy Computing varata nei primi mesi del 2002 da parte di Microsoft. Essa si è resa necessaria a causa del crescente numero di exploit diretto al sistema operativo Microsoft e alle applicazioni, che ha portato il produttore a progettare e rilasciare prodotti sicuri per default, e cioè già inattaccabili con le sole impostazioni predefinite e senza che si dovesse procedere a complesse o esoteriche configurazione. Infatti, l'argomento principale usato dal produttore a propria difesa in questo ambito, è stato sempre quello che i suoi prodotti non fossero poco sicuri in sé, ma che nella maggior parte dei casi si trattasse di problemi di non corretta installazione e tuning del sistema.

In questo paragrafo discuteremo le caratteristiche relative alla autenticazione (il processo di identificazione che collega i dati di accesso

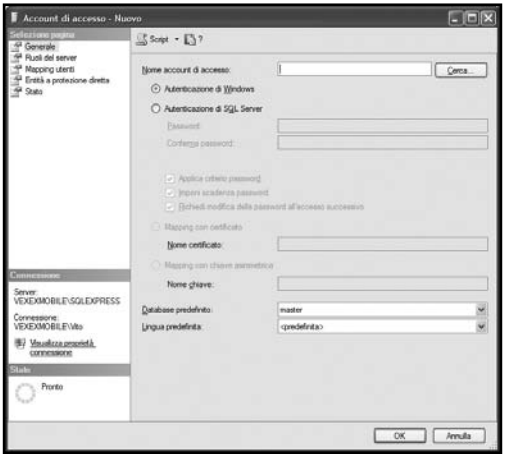


Figura 4.4: Creazione di una nuova login

a SQL Server e utenti che accedono a basi di dati), e proseguirà con l'autorizzazione (determinare il livello delle autorizzazioni concesse una volta che la connessione iniziale è stabilita).
Il primo dato interessante è che l'installazione tipica del prodotto, quella cioè eseguita senza specificare alcuna indicazione aggiuntiva rispetto a quelle proposte di default, evita l'installazione o la non attivazione di componenti essenziali e di impostazioni che possono esporre il server e dei suoi dati a potenziali attacchi. Tale impostazione è applicata ai principali componenti del server applicativo e cioè SQL Server Agent, l'indicizzazione Full Text Search, e Data Transformation Services, Analysis, Reporting, e Notification Services, SQL Browser, Service Broker di connettività di rete, Database Mirroring, SQLMail, o SQL Debugging.

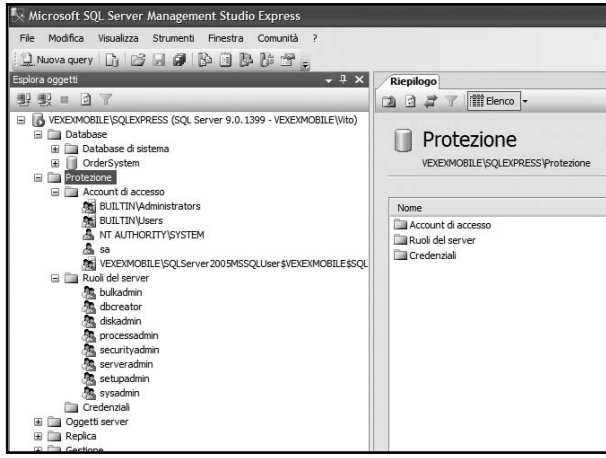


Figura 4.5: Elenco delle login esistenti

Proprio come i suoi predecessori, SQL Server 2005 supporta sia la modalità di autenticazione mista che quella Windows. In Windows Authentication Mode, è garantito l'accesso basato su un token di

protezione assegnato durante l'autenticazione effettuata correttamente al dominio o al server locale da parte di un account Windows. In Active Directory è disponibile un ulteriore livello di protezione fornito dal protocollo Kerberos. Sebbene l'autenticazione basata su Windows sia intrinsecamente più sicura di quella offerta dalla modalità mista, la sicurezza di SQL Server basato login è migliorata attraverso la sua cifratura con certificati.

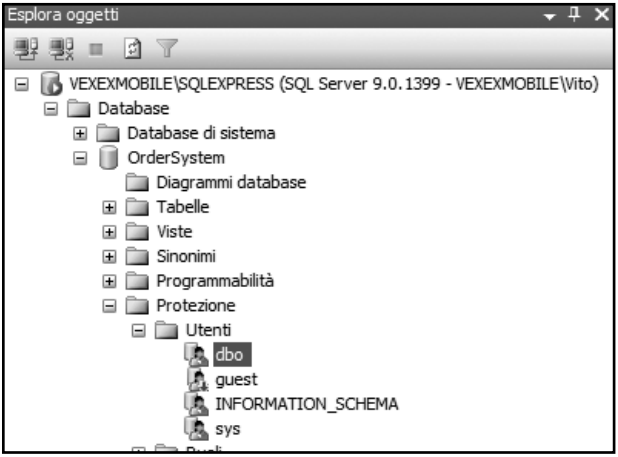


Figura 4.6: Elenco degli utenti

Tra le altre novità vi è la capacità di impostare e gestire alcune regole per la corretta definizione degli account e delle password per le login di SQL Server in modalità nativa. Questo permette di far rispettare tali limiti sulla password quali la complessità, la scadenza della password, e il blocco degli account. Le regole di complessità delle password possono essere riassunte come di seguito:

- la lunghezza della password deve essere di almeno 6 caratteri (in generale, le password di SQL Server può avere tra 1 e 128 caratteri);

- la password deve contenere almeno tre dei quattro tipi diversi di caratteri quali lettere maiuscole, lettere minuscole, numeri e caratteri non alfanumerici;
- la password deve differire dalle seguenti stringhe: admin, administrator, password, sa, sysadmin, il nome dell'host di hosting calcolare l'installazione di SQL Server, e in tutto o in parte e l'impossibilità che la password ricordi da vicino la login dell'utente correntemente connesso a Windows.

Si osservi che, indipendentemente dalla modalità di autenticazione e dalle policy, l'installazione guidata non consente password non vuote sull'utente SA. Inoltre vi sono delle forti restrizioni anche nella definizione della password. Pertanto si può affermare che le impostazioni di sicurezza possibili al momento dell'installazione sono persino più restrittive di quanto SQL Server già consenta di fare. Lo scopo è chiaro: tutti coloro che installano l'applicazione senza modificare sostanzialmente le impostazioni di default offerte dal setup, si ritroveranno immediatamente un sistema piuttosto sicuro e meno esposto a problemi di sicurezza.

È possibile utilizzare le clausole CHECK_EXPIRATION e CHECK_POLICY per la creazione di nuove login con la CREATE LOGIN per impostarne o disattivarne la conformità con le policy (con i parametri ON o OFF) e CHECK_EXPIRATION per le regole di scadenza della password, mentre CHECK_POLICY determina il livello di complessità della password.

```
CREATE LOGIN drevil
WITH
    PASSWORD = 'Ch4ngeMe' MUST_CHANGE,
    CHECK_EXPIRATION = ON,
    CHECK_POLICY = ON
```

La vista di sistema Sys.sql_logins indica se la policy della password e la scadenza della password sono state applicate per i dati di acces-

so di SQL esistenti. Le stesse informazioni sono disponibili per i singoli account attraverso l'interfaccia grafica di SQL Server Management Studio.
Lo statement ALTER LOGIN supporta la clausola UNLOCK, che ha la funzione di sbloccare le login di accesso di SQL Server, che sono stati bloccate da ripetuti errori di immissione della password, a seguito della policy che prevede il blocco dell'account dopo un certo nu
Osserviamo un altro tipico script di creazione della login prodotto

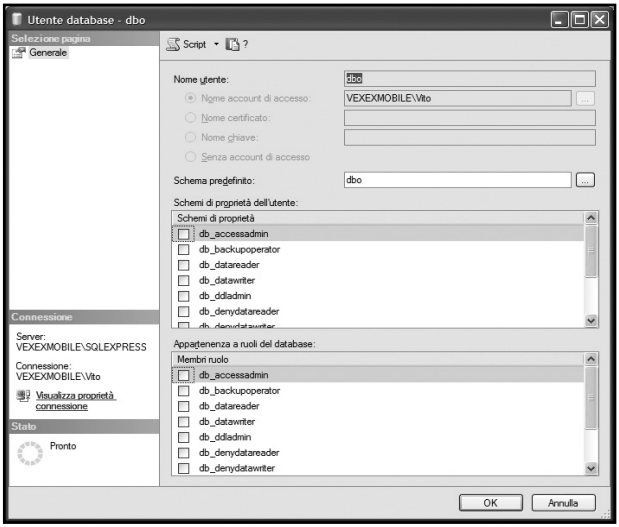


Figura 4.7: Informazioni dell'utente

direttamente dalla funzione di produzione degli script sugli oggetti di database via SQL Server Management Studio:

```
/****** Oggetto:  Login [sa]    Data script: 10/15/2007 23:56:29 ******/
/* For security reasons the login is created disabled and with a random
                                                              password. */
```

```
/****** Oggetto:  Login [sa]    Data script: 10/15/2007 23:56:29 ******/
CREATE LOGIN [sa] WITH PASSWORD=N'@_U±\;r_>q±ã)d_L]_
_ÎM7;0í|öØ2_', DEFAULT_DATABASE=[master],
                DEFAULT_LANGUAGE=[Italiano], CHECK_EXPIRATION=OFF,
                                                CHECK_POLICY=ON
GO
EXEC sys.sp_addsrvrolemember @loginame = N'sa', @rolename =
                                                        N'sysadmin'
GO
ALTER LOGIN [sa] DISABLE
```

L'utente riportato è SA e possiamo osservare come la sua password sia criptata in modo che, anche venendo non sia facilmente reversibile.

4.4.1 Ruoli di amministrazione e di database

A ciascun utente sono assegnati i ruoli che consentono di definire le operazione che l'utente può effettuare sui dati e sulla struttura del database. Ecco i principali ruoli amministrativi:

- **bulkadmin** Sono autorizzate operazioni di inserimento multiplo di massa
- **diskadmin** Gestiscono i file sul disco
- **dbcreator** Creano e modificano database
- **processadmin** Gestiscono i processi e i job
- **serveradmin** Configurano il server
- **setupadmin** Gestiscono la replica
- **sysadmin** Qualsiasi attività amministrativa sul database

Ai ruoli amministrativi, che sono trasfersabili perché riferiti al motore di accesso ai dati più che ai singoli database, si aggiungono quelli che, invece, si legano a questi ultimi. In realtà è possibile ap-

plicare questi ruoli con granularità molto elevata perché impostabili sui singoli oggetti del database (tabelle, campi, stored procedure, viste, ecc...).
Eccone i principali:

- **db_accessadmin** Gestiscono gli utenti del database
- **db_backupoperator** Possono eseguire le funzioni di backup
- **db_datareader** Possono esclusivamente visualizzare i dati
- **db_datawriter** Possono eseguire INSERT e UPDATE
- **db_ddladmin** Possono modificare gli oggetti o vincolarli
- **db_denydatareader** Non possono leggere i dati db_denydatawriter Non possono eseguire modifiche ai dati

Grazie a queste caratteristiche, oltre alla creazione personalizzata di ruoli, possiamo gestire al meglio la sicurezza.

4.4.2 Indicizzazione e ricerca Full Text

SQL Server 7 introduce il supporto nativo alla funzionalità di indicizzazione full text SQL Full-Text Search (SQL FTS). Grazie a tale funzionalità è possibile effettuare interrogazioni veloci ed efficienti su indicizzazioni di testi, ovvero query che si basano su termini chiave, parti di testo o semplici parole.
La ricerca full-text consente l'indicizzazione per query basate su parole chiave dei dati di testo archiviati in un database di Microsoft SQL Server. A differenza del predicato LIKE, che supporta solo i modelli di caratteri, le query full-text eseguono ricerche linguistiche su tali dati, operando su parole e frasi in base alle regole di una lingua specifica.
È possibile creare indici full-text su colonne che contengono dati char, varchar e nvarchar e su colonne che contengono dati binari formattati, ad esempio documenti di Microsoft Word, archiviati in una colonna varbinary(max) o image. Non è possibile utilizzare il predicato LIKE per eseguire query su dati binari formattati.

Per creare un indice full-text su una tabella, quest'ultima deve contenere una colonna singola, univoca e non Null. Si consideri ad esempio un indice full-text per la tabella Document in Adventure Works, in cui DocumentID è la colonna chiave primaria. L'indice full-text indica che la parola "instructions" è la 24° e la 44° nella colonna DocumentSummary della riga associata al valore DocumentID di 3. Questa struttura di indice supporta una ricerca efficiente di tutti gli elementi che includono parole indicizzate e operazioni di ricerca avanzate, ad esempio ricerche di frasi e di prossimità.
Durante l'elaborazione di una query full-text, il motore di ricerca restituisce a SQL Server i valori delle chiavi delle righe che soddisfano i criteri di ricerca. Se si desidera utilizzare una query full-text per individuare documenti che contengono la parola "instructions", dall'indice full-text si ottengono i valori DocumentID 3, 4, 6, 7 e 8. SQL Server utilizza quindi tali chiavi per restituire le righe corrispondenti. SQL Server consente l'installazione del servizio Full Text che quindi dovrà essere avviato (possiamo verificare lo stato del servizio tramite l'applicazione SQL Server Configuration Manager presente al-

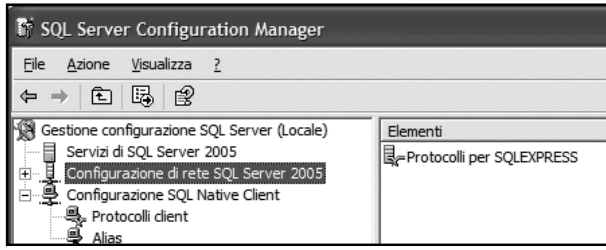


Figura 4.8: SQL Server Configuration Manager

l'interno dei "Configuration Tools"), come mostrato in Figura 4.5.
La funzionalità è offerta dal motore di indicizzazione, che si connette alla nostra istanza tramite un provider OLE-DB che si occupa del popolamento e della gestione degli indici e da un filter daemon co-

stituito da componenti responsabili dell'accesso ai dati delle tabelle e della relativa applicazione di filtri e un word breaking, meccanismo in grado di suddividere testo in token basandosi su regole lessicali.

4.4.3 Esempio di interrogazioni full text

Possiamo visualizzare tutti i linguaggi supportati da SQL Server 2005 interrogando la vista di sistema sys.fulltext_languages come:

```
select * from sys.fulltext_languages
```

Internamente il motore di indicizzazione costruisce uno o più indici, attraverso un processo chiamato popolamento in grado di associare le parole ed i termini con le loro posizioni all'interno dei dati, ed è in grado di comprendere ed estrarre il testo dalle colonne indicizzate.

Per conoscere i tipi di file supportati e disponibili per operazioni di indicizzazione possiamo interrogare la vista di sistema sys.fulltext_document_types, ovvero tramite:

```
select document_type, path from sys.fulltext_document_types
```

Con questa interrogazione siamo in grado di conoscere l'estensione del file ed il percorso della libreria che contiene il componente iFilter appropriato.

4.4.4 Attivazione delle funzionalità

Di seguito si riporta la sequenza di operazioni da eseguire per attivare ed utilizzare il motore di ricerca full text:

- abilitare la ricerca per il database (sp_fulltext_database)
- creare il catalogo (sp_fulltext_catalog)
- marcare la tabella per l'indicizzazione fulltext (sp_fulltext_table)
- aggiungere le colonne che si vogliono indicizzare (sp_fulltext_column)
- attivare l'indice (sp_fulltext_table)
- popolare l'indice (sp_fulltext_catalog)

NOTE

NOTE

NOTE

NOTE

NOTE

NOTE

NOTE

NOTE

NOTE

SQL SERVER
ASPETTI AVANZATI

Autore: Vito Vessia

EDITORE
Edizioni Master S.p.A.
Sede di Milano: Via Ariberto, 24 - 20123 Milano
Sede di Rende: C.da Lecco, zona ind. - 87036 Rende (CS)

Realizzazione grafica:
Cromatika Srl
C.da Lecco, zona ind. - 87036 Rende (CS)

Art Director: Paolo Cristiano
Responsabile grafico di progetto: Salvatore Vuono
Coordinatore tecnico: Giancarlo Sicilia
Illustrazioni: Tonino Intieri
Impaginazione elettronica: Francesco Cospite

Servizio Clienti
☎ **Tel. 02 831212 - Fax 02 83121206**
@ **e-mail: customercare@edmaster.it**

Stampa: Grafica Editoriale Printing - Bologna

Finito di stampare nel mese di Novembre 2007