7

Introduzione a T-SQL

Il linguaggio T-SQL, molto simile ad altri linguaggi di programmazione, ha un ruolo di primo piano nei database di SQL Server. Scopriamolo insieme.

Cos'è T-SQL

T-SQL è l'abbreviazione di **Transact-SQL**, un'implementazione molto potente di SQL, sviluppata da Microsoft.
L'acronimo **SQL** significa **Structured Query Language**, cioè linguaggio di interrogazione strutturato. In realtà, come vedremo in seguito, la versione di SQL che utilizziamo in SQL Server è molto più che un linguaggio di sola interrogazione, dato che permette di creare dei veri e propri programmi per la gestione di qualsiasi oggetto del database.
Con **SQL Server Management Studio** abbiamo la possibilità di gestire qualsiasi oggetto di database in modo visuale e interattivo. Sebbene questa sia la modalità di gestione più facile e piacevole, esistono molti casi in cui diventa necessario poter eseguire tali operazioni in modo automatico o anche ripetitivo, secondo una sequenza di operazioni programmata: ecco, quindi, dove entra in gioco T-SQL.
Molti puristi obiettano che T-SQL non è SQL standard, dato che ha delle estensioni che lo rendono molto più simile a un linguaggio di programmazione, seppure specializzato. Noi non entreremo in questa disputa, restando comunque dell'opinione che le estensioni di T-SQL rendano questo linguaggio molto flessibile e potente, in grado di fornirci degli strumenti che ci permettono il massimo controllo sulle attività che vogliamo svolgere.

Come si pronuncia SQL?

A molti di voi forse non interesserà, ma è una questione che più volte ha sollevato molte discussioni: come si pronuncia SQL?
Trattandosi di un acronimo, la dizione inglese tecnicamente corretta è "*ès-kiù-èl*", ma la dizione che è entrata nell'uso corrente è soprattutto "*siquel*", forse tenendo conto anche del fatto che SQL Server deriva da una prima implementazione che era stata chiamata SEQUEL.

La storia di SQL

SQL nasce nel 1974 nei laboratori IBM, grazie all'ingegno e all'intuito di Donald Chamberlin. Sono incredibili la quantità e la qualità delle innovazioni informatiche nate nei laboratori IBM!
Il prodotto di Chamberlin fu preso in seria considerazione da IBM, al contrario dello scarso interesse che fu riservato dalla stessa IBM a **Codd**, il padre della teoria relazionale.
Nel tempo, SQL ha avuto vari ampliamenti e miglioramenti fino ad arrivare alla versione SQL/86, la prima versione riconosciuta come standard da parte di ANSI e di ISO. Successivamente, sono state prodotte nuove versioni standard, denominate SQL/89, SQL/92 e infine SQL/2003.
Microsoft e Sybase, partendo dalla versione standard di SQL, hanno implementato una propria versione del linguaggio, con tutta una serie di estensioni che hanno dato origine a T-SQL.

La struttura generale di T-SQL

Il linguaggio T-SQL, come SQL, mette a disposizione varie istruzioni che possono essere classificate in tre precise tipologie:

- **DDL = Data Definition Language**: sono le istruzioni che permettono la definizione e la modifica dello schema del database;
- **DML = Data Manipulation Language**: sono le istruzioni che vengono utilizzate per gestire i dati (selezione, inserimento, modifica, cancellazione);
- **DCL = Data Control Language**: sono le istruzioni che permettono di definire i permessi agli utenti, affinché questi ultimi possano utilizzare le istruzioni DDL e DML, nonché le stesse istruzioni DCL.

Inoltre, T-SQL prevede molte altre istruzioni e strutture che possiamo classificare nelle seguenti categorie principali:

- commenti;
- identificatori;
- istruzioni per la definizione e l'assegnazione di valori letterali, variabili e costanti;
- istruzioni per il controllo del flusso del programma.

Istruzioni DDL

T-SQL fornisce delle istruzioni **DDL** (**Data Definition Language**) che hanno il compito di eseguire varie operazioni di gestione degli oggetti di database. Con il termine "gestione" intendiamo la possibilità di creare, modificare e cancellare un oggetto di database. Con un'istruzione di tipo DDL possiamo gestire i database, le tabelle, le viste, gli indici, i vincoli e così via.
Vedremo presto degli esempi concreti di istruzioni DDL.

Istruzioni DCL

Le istruzioni **DCL** (**Data Control Language**) permettono di gestire l'assegnazione e la revoca di permessi, per consentire agli utenti di utilizzare le istruzioni DDL e DML e le stesse istruzioni DCL.
Nei prossimi capitoli vedremo le principali istruzioni DCL disponibili, come la creazione, la modifica e la cancellazione di un utente, l'assegnazione, la modifica e la revoca di permessi.
L'analisi dettagliata di tutte queste istruzioni va oltre lo scopo di questo libro. Per le istruzioni più importanti, comunque, vediamo gli schemi sintattici da rispettare e analizziamo le clausole e gli argomenti più significativi e generalmente più utilizzati.
Per eventuali approfondimenti su queste e su altre istruzioni, sulle clausole e sugli argomenti non trattati qui, vi invitiamo a fare riferimento alla documentazione di SQL Server 2016.

Istruzioni DML

Veniamo ora all'ultimo gruppo di istruzioni: **DML** (**Data Manipulation Language**).
Questo insieme di istruzioni permette di gestire i dati veri e propri, eseguendo tutte le operazioni comunemente utilizzate:

- selezione di dati, con l'istruzione `SELECT`;
- inserimento di dati, con l'istruzione `INSERT`;
- modifica di dati, con l'istruzione `UPDATE`;
- cancellazione di dati, con l'istruzione `DELETE`.

Trattandosi di operazioni molto utilizzate in tutte le applicazioni che utilizzano insiemi di dati (quindi la maggior parte), le vedremo in modo abbastanza approfondito.

Commenti

Ogni buon linguaggio che si rispetti permette di inserire dei commenti all'interno del codice, per fornire informazioni proprio nei punti in cui queste sono necessarie.
Il corretto inserimento di un numero adeguato di commenti, chiari e completi, rende il codice autodocumentante e, spesso, sostituisce la documentazione esterna. In teoria, quest'ultima dovrebbe sempre esserci, ma in realtà sono pochi gli sviluppatori che ce l'hanno davvero.
Ecco, quindi, l'importanza di inserire dei commenti nei punti strategici del codice, per informare chi dovrà mettervi mano o per ricordare a noi stessi come funziona, quando dovremo modificarlo dopo che è trascorso un po' di tempo.
Possiamo definire un commento in due modi diversi.
Il simbolo di doppio trattino (`--`) è utile nei casi in cui vogliamo inserire un commento su un'unica riga. Per esempio:

```
-- Questo è un commento
```

Se invece dobbiamo inserire più righe di commento, possiamo utilizzare la coppia di simboli tanto familiare ai programmatori C++ o C#, e cioè `/*` e `*/`, rispettivamente all'inizio e alla fine di un blocco di commento. Per esempio:

```
/*
   Questo è un commento piuttosto lungo che non
   riesco a far stare su un'unica riga e quindi decido
   di utilizzare questi simboli
*/
```

Il primo modo fa parte dello standard ANSI SQL, mentre il secondo è un'estensione di T-SQL.
Durante l'esecuzione di una query, le righe di commento sono totalmente ignorate.

Identificatori

Gli identificatori vengono utilizzati per identificare univocamente degli oggetti come database, tabelle, indici, viste, colonne e così via.
Un identificatore è composto da una stringa di lunghezza variabile compresa tra 1 e 128 (o tra 1 e 116 per gli identificatori delle tabelle temporanee).
Un identificatore può essere regolare o delimitato.

Identificatori regolari

Un **identificatore regolare** è composto da cifre, lettere nella codifica Unicode e alcuni caratteri speciali, come il carattere di sottolineatura (_), la chiocciola (@), il cancelletto (#) e il simbolo del dollaro ($).
Per il primo carattere di un identificatore regolare vengono definiti alcuni significati speciali:

- come primo carattere non è ammessa una cifra numerica;
- il simbolo # nel primo carattere identifica una tabella temporanea;
- il simbolo @ nel primo carattere identifica una variabile;
- il simbolo @@ nei primi due caratteri identifica generalmente una funzione predefinita di T-SQL e, quindi, ne sconsigliamo l'uso nei vostri identificatori.

Un esempio di delimitatore valido è presente nell'istruzione di creazione tabella seguente:

```
CREATE TABLE TabellaXY
(Chiave INT PRIMARY KEY, Descrizione nvarchar(100))
```

In questo caso abbiamo diversi identificatori:

- `TabellaXY` è l'identificatore della tabella;
- `Chiave` e `Descrizione` sono gli identificatori dei due campi che fanno parte della tabella.

Identificatori delimitati

Un **identificatore delimitato** è un identificatore in cui la stringa viene iniziata e terminata con un carattere di doppio apice (") o con i caratteri corrispondenti alle parentesi quadre ([e]).
La stringa che compone l'identificatore delimitato può essere formata da qualsiasi combinazione di caratteri, cifre o simboli, escluso solamente il carattere di delimitazione stesso.
Alcuni esempi di identificatori delimitati sono i seguenti:

```
SELECT *
FROM [Mia Tabella]   -- contiene uno spazio
WHERE [SELECT] = 150 -- SELECT è una parola riservata di T-SQL
```

NOTA

Sconsigliamo l'uso di identificatori delimitati, specialmente quelli contenenti parole riservate di T-SQL, perché possono rendere le istruzioni meno leggibili.

Costanti e variabili

Costanti o valori letterali

Un **valore letterale** è un valore espresso in modo esplicito e invariabile (**costante**), costituito da una sequenza alfanumerica (stringa) o da un valore esadecimale o numerico. Una stringa contiene uno o più caratteri racchiusi tra due apici (`'`), detti anche virgolette singole, o tra due virgolette doppie (`"`). Dato che queste ultime sono utilizzate anche in altri contesti, come nel caso già visto degli identificatori delimitati, è preferibile utilizzare gli apici.
Se abbiamo la necessità di inserire un apice all'interno della stringa delimitata da apici, possiamo utilizzare due volte il carattere apice (`''`).
Un valore esadecimale permette di rappresentare caratteri non stampabili o altri dati binari. Inizia sempre con la sequenza `0x`, seguita da coppie di cifre esadecimali.
Di seguito riportiamo alcuni esempi di valori letterali validi e non validi:

```
/* VALORI VALIDI: */
'Blog di Mario De Ghetto'
'http://deghetto.wordpress.com'  -- caratteri speciali
'19/04/2003'  -- data in formato stringa
'L''Uomo'  -- ripetizione dell'apice all'interno
0xFF0044AB

/* VALORI NON VALIDI: */
'Errato uso dell'apostrofo'
"Virgolette di tipo diverso'
```

Eventuali valori incompleti di una stringa possono essere rappresentati con caratteri speciali detti **caratteri jolly** (**wildcard**).
Tra questi, i più conosciuti sono il simbolo "percento" (`%`) in sostituzione di qualsiasi sequenza di caratteri e il carattere di sottolineatura (_) in sostituzione di qualsiasi carattere (ogni carattere di sottolineatura costituisce un segnaposto per un carattere qualsiasi). Esempi:

```
'Mar%'    -- vanno bene Mario, Maria, Mariella, Martedì …
'Mar_o'   -- vanno bene Mario, Marco, Marmo …
```

Non tutti sanno, però, che T-SQL supporta altri caratteri speciali che permettono di indicare intervalli di caratteri:

- i caratteri `[` e `]` con, al loro interno, un insieme di caratteri ammessi o un intervallo di caratteri, per esempio `[a-f]` oppure `[aeiou]`. Le parentesi quadre sono utilizzate anche per indicare che il carattere segnaposto che si trova tra di esse è da considerare come carattere semplice;
- il carattere `^` viene utilizzato come negazione dei successivi caratteri o intervallo di caratteri.

Vediamo alcuni esempi concreti di questi casi.

```
-- parentesi quadre
'[DML]ario' -- vanno bene Dario, Mario e Lario
-- intervallo di caratteri
'[A-D]735'  -- vanno bene A735, B735, C735 e D735
-- simbolo %
'%[_]%'     -- va bene qualsiasi sequenza di caratteri
            -- prima e dopo di un carattere di sottolineatura
-- negazione, insieme di caratteri e simbolo %
'^[BCD]%'   -- va bene qualsiasi sequenza di caratteri che
            -- non inizi per B, C o D
-- negazione, intervallo di caratteri e %
'^[B-Y]%'   -- è equivalente alla sequenza '[AZ]%'
```

Variabili

In T-SQL, il termine "variabile" ha lo stesso significato che ha in qualsiasi altro linguaggio di programmazione tradizionale: è un'area di memoria, più o meno grande, che viene associata a un nome simbolico e a un tipo di dati (vedi Capitolo 6). Tale area di memoria può contenere un valore che viene assegnato con un'istruzione chiamata, appunto, "di assegnazione".
La variabile deve essere preventivamente dichiarata con l'istruzione `DECLARE`.
Per esempio:

```
-- dichiaro una variabile di nome "nome"
DECLARE @nome varchar(30)
-- assegno un valore di tipo stringa alla variabile
SET @nome = 'Giuseppe Verdi'
-- visualizzo il valore
PRINT @nome
```

NOTA

Per provare questo esempio, aprite SQL Server Management Studio, cliccate sul pulsante "**Nuova query**" e, nella finestra che apparirà, inserite le istruzioni dell'esempio. Per eseguirlo, cliccate sul pulsante "**Esegui**" (quello con un punto esclamativo di colore rosso). Vedrete così apparire, nel riquadro "**Messaggi**", la scritta "Giuseppe Verdi".

Ovviamente si può accedere a una variabile anche per leggere il suo contenuto. L'istruzione `PRINT @nome`, che abbiamo appena visto, è uno dei modi per leggere il valore (e stamparlo), ma è possibile anche assegnare il valore di una variabile a un'altra variabile:

```
-- dichiaro una variabile di nome "nome"
DECLARE @nome varchar(30)
-- assegno un valore di tipo stringa alla variabile
SET @nome = 'Giuseppe Verdi'
-- dichiaro una variabile di nome "nominativo"
DECLARE @nominativo varchar(30)
-- leggo il valore della variabile nome
-- e l'assegno alla variabile nominativo
SET @nominativo = @nome
```

Più dichiarazioni di variabili possono essere fatte con la stessa istruzione `DECLARE`, è sufficiente separare le dichiarazioni con una virgola:

```
DECLARE @nome varchar(30), @nominativo varchar(30)
```

Operatori aritmetici

Gli operatori che possono essere utilizzati in operazioni aritmetiche sono quelli classici: addizione (`+`), sottrazione (`-`), divisione (`/`) e moltiplicazione (`*`).
A questi si aggiunge l'operatore "modulo" (`%`), che restituisce il resto di una divisione non intera. Per esempio: `12 % 5 = 2`.
In T-SQL possiamo utilizzare anche la forma di operazione con assegnamento. I seguenti operatori hanno lo stesso significato degli operatori semplici, con la differenza che il risultato viene assegnato alla variabile utilizzata nei calcoli (in tutti i calcoli presumiamo che nella variabile "x" sia memorizzato il numero valore 12):

- addizione con assegnazione (`+=`). Esempio: `x += 5` (risultato x = 17);
- sottrazione con assegnazione (`-=`). Esempio: `x -= 2` (risultato x = 10);
- divisione con assegnazione (`/=`). Esempio: `x /= 3` (risultato x = 4);
- moltiplicazione con assegnazione (`*=`). Esempio: `x *= 4` (risultato x = 48);
- modulo con assegnazione (`%=`). Esempio: `x %= 10` (risultato x = 2).

Operatori bit a bit

Questi operatori restituiscono un risultato che viene formato confrontando le coppie di bit corrispondenti dei due operandi.
Gli operatori bit a bit (in inglese "**bitwise**") disponibili sono tre: `&`, `|` e `^`.

- `&` (`AND`): se entrambi i bit di ciascuna coppia sono impostati a 1, il bit risultante sarà uguale a 1; negli altri casi sarà uguale a 0. Ecco un esempio:

```
(A & B) con A=170 e B=75
0000 0000 1010 1010
0000 0000 0100 1011
-------------------
0000 0000 0000 1010 = 8+2 = 10 (base decimale)
```

- | (OR): se almeno un bit di ciascuna coppia è impostato a 1, il bit risultante sarà uguale a 1; negli altri casi sarà uguale a 0. Ecco l'esempio:

```
(A | B) con A=170 e B=75
0000 0000 1010 1010
0000 0000 0100 1011
-------------------
0000 0000 1110 1011 = 128+64+32+8+2+1 = 235 (base decimale)
```

- ^ (OR esclusivo): come per l'operatore OR bit a bit, ma escludendo le coppie di bit entrambe uguali a 1. Solo uno dei bit deve essere impostato a 1 per restituire un bit uguale a 1. Ecco l'esempio:

```
(A ^ B) con A=170 e B=75
0000 0000 1010 1010
0000 0000 0100 1011
-------------------
0000 0000 1110 0001 = 128+64+32+1 = 225 (base decimale)
```

In T-SQL abbiamo le versioni di questi stessi operatori con assegnamento, dove il risultato dell'operazione viene assegnato alla variabile posta a sinistra del simbolo di eguaglianza (&=, |= e ^=):

- &= (AND con assegnamento):

```
DECLARE @A int
SET @A = 170
DECLARE @B int
SET @B = 75
SET @A &= @B
PRINT @A       -- risultato: 10
```

- |= (OR con assegnamento):

```
DECLARE @A int
SET @A = 170
DECLARE @B int
SET @B = 75
SET @A |= @B
PRINT @A       -- risultato: 235
```

- ^= (OR esclusivo con assegnamento):

```
DECLARE @A int
SET @A = 170
DECLARE @B int
SET @B = 75
SET @A ^= @B
PRINT @A       -- risultato: 225
```

Infine, sempre in T-SQL, abbiamo un operatore bit a bit aggiuntivo, ma in questo caso non esiste la versione con assegnamento perché l'operatore agisce direttamente sul valore, invertendo tutti i bit da 0 a 1 e viceversa:

```
~ (NOT):

DECLARE @A int
SET @A = 1
PRINT ~ @A

(~A) con A=170
0000 0000 1010 1010 =  170 (base decimale)
-------------------
1111 1111 0101 0101 = -171 (base decimale)
```

NOTA

La cifra "1" più significativa (la prima a sinistra) indica il segno e pertanto, in questo caso, l'operazione restituisce un valore negativo.

Operatori di confronto

Gli operatori di confronto sono utilizzati per confrontare due valori, con lo scopo di decidere un diverso percorso del programma oppure di selezionare o escludere un record durante l'esecuzione di una query.
Come potete osservare nella Tabella 7.1, sono disponibili vari operatori di confronto.

Tabella 7.1 - Operatori di confronto.

Tipo	Dimensione
=	Eguaglianza
>	Maggiore di ...
<	Minore di ...
>=	Maggiore o uguale
<=	Minore o uguale
<>	Diverso da ...
!=	Diverso da ... (non uguale a ...)
!>	Non maggiore di ...
!<	Non minore di ...

NOTA

Il punto esclamativo posto prima degli operatori di confronto indica una negazione degli operatori stessi (equivale a NOT ...).

Operatori di concatenamento stringhe

Molto spesso è necessario poter concatenare stringhe per creare una stringa più grande contenente tutte le stringhe precedentemente trattate. Questa operazione è realizzata attraverso l'operatore di concatenamento rappresentato dal simbolo "+".
È possibile utilizzare anche la forma di concatenamento con assegnazione, mediante il simbolo "+=", esattamente nello stesso modo in cui viene effettuata un'operazione di somma di numeri con assegnazione. Per esempio:

```
DECLARE @str varchar(20), @str2 varchar(20), @str3 varchar(20)
SET @str = 'SQL Server'
SET @str2 = ' '
SET @str3 = '2016'
PRINT @str + @str2 + @str3
```

NOTA

Osservate come è stato aggiunto uno spazio tra le due stringhe.

L'esempio di concatenamento con assegnazione è il seguente:

```
DECLARE @str varchar(20), @str2 varchar(20), @str3 varchar(20)
SET @str = 'SQL Server'
SET @str2 = ' '
SET @str3 = '2016'
SET @str += @str2
SET @str += @str3
PRINT @str
```

Operatori unari

Gli operatori unari sono così chiamati perché agiscono su un unico operando, al contrario di molti altri operatori che ne richiedono almeno due.
Uno degli operatori unari l'abbiamo già incontrato in precedenza nel paragrafo dedicato agli operatori "bit a bit": si tratta del "bitwise NOT", rappresentato dal simbolo ~ (tilde).
Altri operatori unari sono semplicemente i simboli relativi al segno di un numero:

- "+": simbolo di numero positivo (per esempio: +125);
- "-": simbolo di numero negativo (per esempio: -18).

Operatori logici

AND

L'operatore AND restituisce True (vero) solo se tutti i valori utilizzati nell'espressione sono veri. Se anche uno solo dei valori è False (falso), sarà falso anche il risultato. Se l'espressione contiene solo valori True e NULL, anche il risultato sarà NULL.
La forma in cui viene utilizzato l'operatore AND è la seguente:

```
valore1 AND valore2
```

Nella tabella della verità dell'operatore **And** (Tabella 7.2) vediamo tutte le possibili combinazioni che possiamo comporre con i valori True (V), False (F) e NULL.

Tabella 7.2 - La tabella della verità - Operatore AND.

A	B	A AND B
V	V	V
V	F	F
F	V	F
F	F	F
V	Null	Null
Null	V	Null
F	Null	F
Null	F	F
Null	Null	Null

Il motivo di questo comportamento è molto semplice:

- se tutti i valori sono True, il risultato è True;
- se tutti i valori sono NULL, ovviamente il risultato è NULL;
- se almeno uno dei valori è falso, non ha importanza quale valore contengano gli altri elementi dell'espressione (vero o NULL), perché il risultato non può essere che falso. Il fatto che un altro valore sia NULL, cioè un valore sconosciuto, non incide sul significato logico;
- se uno o più valori sono True e uno o più valori sono NULL, il risultato è comunque NULL, perché non sappiamo se il valore reale dell'elemento sconosciuto sia vero o falso.

OR

L'operatore OR restituisce True se anche uno solo dei valori utilizzati nell'espressione è vero. Se tutti i valori sono falsi, anche il risultato è falso. Se l'espressione contiene solo valori False e NULL, anche il risultato sarà NULL.
La forma in cui viene utilizzato l'operatore OR è la seguente:

```
valore1 OR valore2
```

Nella tabella della verità dell'operatore Or (Tabella 7.3) vediamo tutte le possibili combinazioni che possiamo comporre con i valori True (V), False (F) e NULL.

Tabella 7.3 - La tabella della verità - Operatore OR.

A	B	A OR B
V	V	V
V	F	V
F	V	V
F	F	F
V	Null	V
Null	V	V
F	Null	Null
Null	F	Null
Null	Null	Null

Anche in questo caso possiamo spiegare facilmente il comportamento di queste espressioni:

- se tutti i valori valutati dall'espressione sono False, allora il risultato è ovviamente falso;
- se almeno uno dei valori valutati dall'espressione è True, allora anche il risultato è True;
- se tutti i valori sono NULL oppure almeno un valore è NULL e gli altri non sono veri, allora il risultato è NULL perché non sappiamo se uno o più valori NULL rappresentino dei valori veri che non sono a nostra conoscenza.

NOT

L'operatore NOT inverte semplicemente il valore da True a False e viceversa. Nel caso di un valore NULL non esiste un valore opposto: se un valore è sconosciuto rimane sconosciuto e quindi otteniamo ugualmente un NULL.

La forma in cui viene utilizzato l'operatore NOT è la seguente:

```
NOT valore
```

Nella tabella della verità dell'operatore Not (Tabella 7.4) vediamo tutte le possibili combinazioni che possiamo comporre con i valori True (V), False (F) e NULL.

Tabella 7.4 - La tabella della verità - Operatore NOT.

A	NOT A
V	F
F	V
Null	Null

ALL

L'operatore ALL confronta un valore scalare con i valori contenuti in un'intera colonna di un set di dati.
La forma in cui viene utilizzato l'operatore ALL è la seguente:

```
valore/espressione
   {
      = | <> | != | > | >= | !> | < | <= | !<
   }
ALL (subquery)
```

All'interno delle parentesi graffe sono elencati vari operatori di confronto. È quindi necessario scegliere uno di tali operatori. Nell'espressione non devono essere indicate le parentesi graffe.
Il simbolo all'interno delle parentesi tonde (subquery) rappresenta un'istruzione SQL di tipo SELECT semplice (senza ordinamento).
Ecco un breve esempio:

```
USE AdventureWorks2008R2 ;
GO
DECLARE @Giorni AS int
SET @Giorni = 2
DECLARE @OrderID AS int
SET @OrderID = 43659
IF
@Giorni >= ALL
   (
    SELECT DaysToManufacture
    FROM Sales.SalesOrderDetail
    JOIN Production.Product
    ON Sales.SalesOrderDetail.ProductID =
       Production.Product.ProductID
    WHERE SalesOrderID = @OrderID
   )
```

```
PRINT 'Tutti i prodotti di questo ordine possono essere ' +
      'fabbricati nel numero di giorni indicato o meno.'
ELSE
PRINT 'Alcuni prodotti di questo ordine non possono ' +
      'essere fabbricati nel numero di giorni indicato.' ;
GO
```

Se provate a eseguire il codice, otterrete la seguente risposta:

```
Alcuni prodotti di questo ordine non possono essere fabbricati nel numero di giorni
indicato.
```

Infatti, il numero di giorni di alcuni record è maggiore di 2 e quindi non è vero che TUTTI (`ALL`) i record sono inferiori o al massimo uguali a 2 (cioè il valore memorizzato nella variabile `Giorni`).

SOME / ANY

Gli operatori `SOME` e `ANY` sono equivalenti. Essi confrontano un valore scalare con i valori contenuti in un'intera colonna di un set di dati.
La forma in cui viene utilizzato l'operatore `ALL` è la seguente:

```
valore/espressione
   {
      = | < > | ! = | > | > = | ! > | < | < = | ! <
   }
{SOME | ANY} (subquery)
```

Sull'elenco degli operatori di confronto e sulla subquery valgono le considerazioni già fatte per l'operatore `ALL`.
Ecco un breve esempio:

```
USE AdventureWorks2008R2 ;
GO
DECLARE @Giorni AS int
SET @Giorni = 3
DECLARE @OrderID AS int
SET @OrderID = 43659
IF
@Giorni < SOME
   (
    SELECT DaysToManufacture
    FROM Sales.SalesOrderDetail
    JOIN Production.Product
    ON Sales.SalesOrderDetail.ProductID =
       Production.Product.ProductID
    WHERE SalesOrderID = @OrderID
   )
PRINT 'Alcuni prodotti di questo ordine non possono ' +
      'essere fabbricati nel numero di giorni indicato.' ;
```

```
ELSE
PRINT 'Tutti i prodotti di questo ordine possono essere ' +
      'fabbricati nel numero di giorni indicato o meno.'
GO
```

Se provate a eseguire il codice, otterrete la seguente risposta:

```
Alcuni prodotti di questo ordine non possono essere fabbricati nel numero di giorni
indicato.
```

Infatti, il numero di giorni di alcuni record è maggiore di 3 e quindi è vero che il valore memorizzato nella variabile `Giorni` è inferiore ad ALCUNI (`SOME`) record.

NOTA

Rispetto all'esempio del paragrafo precedente notate che, oltre ad aver cambiato la parola `ALL` con `SOME`, è stato cambiato anche il verso dell'operatore di confronto (da "`>=`" a "`<`").

BETWEEN

L'operatore `BETWEEN` permette di confrontare i valori contenuti in un'intera colonna di un set di dati con un intervallo di validità ("range").
La forma in cui viene utilizzato l'operatore `BETWEEN` è la seguente:

```
espressione [ NOT ] BETWEEN inizio_range AND fine_range
```

Ecco un breve esempio:

```
USE NorthWind;
GO
SELECT e.Dipendente, e.Vendite
FROM dbo.[Analisi vendite] e
WHERE e.Vendite BETWEEN 1000 AND 5000
ORDER BY E.Vendite DESC;
GO
```

Il risultato che otterremo è il seguente:

```
Dipendente           Vendite
-------------------- -----------
Mario Greco           4200,00
Giovanni Bianchi      3240,00
Maria Ferrari         2250,00
Luigi Bruno           1950,00
Anna Ferraro          1930,00
Luigi Bruno           1740,00
Esposito Antonio      1590,00
Francesca Leonetti    1560,00
```

```
Maria Ferrari           1400,00
Mario Greco             1392,00
Giovanni Bianchi        1275,00
Francesca Leonetti      1218,00
Anna Ferraro            1000,00
```

Se non potessimo utilizzare l'operatore `BETWEEN`, saremmo costretti a indicare la clausola `WHERE` in questo modo:

```
USE NorthWind;
GO
SELECT e.Dipendente, e.Vendite
FROM dbo.[Analisi vendite] e
WHERE e.Vendite >= 1000 AND e.Vendite <= 5000
ORDER BY E.Vendite DESC;
GO
```

Il risultato è lo stesso, ma al prezzo di dover utilizzare ben tre operatori (`>=`, `AND` e `<=`).

EXISTS

L'operatore `EXISTS` permette di confrontare ciascun valore contenuto in un'intera colonna di un set di dati con una lista di valori o con il risultato di una subquery. Il singolo record viene selezionato se la subquery restituisce nel risultato almeno una riga.
La forma in cui viene utilizzato l'operatore `EXISTS` è la seguente:

```
EXISTS subquery
```

Il seguente esempio permette di selezionare un elenco di clienti, con la condizione che ciascun cliente abbia registrato un ordine con il dipendente che ha l'ID "9":

```
USE NorthWind;
GO
SELECT a.Cognome, a.Nome, a.Città
FROM Clienti AS a
WHERE EXISTS
   (SELECT *
    FROM dbo.Ordini AS b
    WHERE a.ID = b.[ID cliente]
     AND b.[ID dipendente] = '9');
GO
```

Il risultato che otterremo è il seguente:

```
Cognome         Nome            Città
--------------- --------------- ----------------
Bonaldi         Raffaella       Reggio Emilia
Tanara          Marco           Messina
Scotti          Elisabetta      Ancona
Valverde        Eva             Bologna
Rossi           Giuseppe        Caserta
```

Notate che non sono state restituite righe duplicate: l'operatore EXISTS verifica che esistano delle righe corrispondenti nella sottoquery, ma, non trattandosi di un'operazione di join, le righe duplicate non vengono riportate, perché anche nella query principale non abbiamo righe duplicate.

IN

L'operatore IN permette di confrontare ciascun valore contenuto in un'intera colonna di un set di dati con una lista di valori e di selezionare i record che rispettano il criterio indicato.
La forma in cui viene utilizzato l'operatore IN è la seguente:

```
test_expression [ NOT ] IN
   ( subquery | expression [ ,...n ] )
```

Come nell'esempio che vi abbiamo mostrato per l'operatore EXISTS, nel seguente esempio selezioniamo un elenco di clienti, con la condizione che ciascuno di essi abbia registrato un ordine con il dipendente che ha l'ID "9".
La differenza è nella clausola WHERE della query principale, ma anche nella clausola WHERE della query secondaria:

```
USE NorthWind;
GO
SELECT a.Cognome, a.Nome, a.Città
FROM Clienti AS a
WHERE a.ID IN
(SELECT b.[ID cliente]
    FROM dbo.Ordini AS b
    WHERE b.[ID dipendente] = '9');
GO
```

Il risultato è esattamente uguale a quello che abbiamo ottenuto dall'esempio dell'operatore EXISTS:

```
Cognome         Nome            Città
--------------- --------------- ---------------
Bonaldi         Raffaella       Reggio Emilia
Tanara          Marco           Messina
Scotti          Elisabetta      Ancona
Valverde        Eva             Bologna
Rossi           Giuseppe        Caserta
```

Anche in questo caso non otteniamo righe duplicate, per lo stesso motivo illlustrato in precedenza.

LIKE

L'operatore `LIKE` permette di determinare se una determinata stringa di caratteri corrisponde a un modello specificato. Tale modello può contenere caratteri specifici e caratteri jolly.
La forma in cui viene utilizzato l'operatore `LIKE` è la seguente:

```
match_expression [ NOT ] LIKE pattern
   [ ESCAPE escape_character ]
```

In una ricerca di questo tipo i normali caratteri devono corrispondere esattamente a quelli specificati nella stringa di caratteri indicata nel modello. I caratteri jolly, invece, sostituiscono un carattere o un frammento della stringa di dimensione variabile (Tabella 7.5).

Tabella 7.5 - Caratteri jolly.

Carattere	Significato
%	Stringa composta da zero o più caratteri. Esempio: `WHERE nome LIKE '%Maria%'` permette di individuare tutti i nomi che contengono la parola "Maria" (Maria, Anna Maria, Maria Grazia ecc.)
_ (carattere di sotto-lineatura)	Carattere singolo. Esempio: `WHERE nome LIKE 'Mari_'` permette di individuare tutti i nomi che contengono la stringa "Mari" e che terminano con un carattere qualsiasi (Mario, Maria, Marie...)
[ ]	Carattere singolo compreso in un intervallo (`[a-f]`) o in un insieme di caratteri (`[abcdef]`) specificato. Esempio: `WHERE nome LIKE '[MD]ari[ao]'` permette di individuare tutti i nomi che iniziano con la lettera "M" o con la lettera "D", che contengono la stringa "ari" e che terminano con la lettera "a" o con la lettera "o" (Maria, Mario, Daria, Dario)
[^]	Il simbolo ^ indica una negazione: rappresenta un carattere singolo NON compreso in un intervallo (`[^a-f]`) o in un insieme di caratteri (`[^abcdef]`) specificato. Esempio: `WHERE codice LIKE 'A941[^5-9])'` permette di individuare tutti i codici che iniziano con la stringa "A941" e che terminano con un carattere o una cifra non inclusa nell'intervallo da "5" a "9".

L'operatore LIKE è molto più flessibile rispetto agli operatori di confronto tra stringhe (= e !=).
Se il tipo di dati degli argomenti non corrisponde a quello della stringa di caratteri, il motore di database di SQL Server è in grado di convertire automaticamente gli argomenti nel tipo di dati della stringa, se possibile.

Ordine di precedenza tra operatori

Gli operatori non hanno tutti lo stesso livello di priorità: in un'espressione complessa, con due o più operatori, non sempre si valutano le operazioni da sinistra a destra, perché esiste una precisa gerarchia di priorità degli operatori, come nelle espressioni matematiche. L'ordine di priorità è il seguente, dal primo all'ultimo:

- ~ (bitwise NOT)
- * (moltiplicazione), / (divisione), % (modulo, cioè il resto di una divisione intera)
- + (segno positivo), – (segno negativo), + (addizione), + (concatenamento), – (sottrazione), & (bitwise AND), ^ (bitwise OR esclusivo o XOR), | (bitwise OR)
- operatori di confronto: =, >, <, >=, <=, <>, !=, !>, !<
- NOT
- AND
- ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
- operatore di assegnazione: =

Valore nullo

Nel paragrafo precedente abbiamo introdotto il valore nullo (NULL) senza spiegarlo: vediamo in maggiore dettaglio che cos'è.
Un valore nullo (NULL) identifica un dato che è sconosciuto o non è applicabile nel caso in questione. NULL non corrisponde né allo zero dei valori numerici (zero è comunque un valore definito) né alla stringa di lunghezza zero per i valori di tipo stringa: un valore nullo è nullo e basta.
Supponiamo, per esempio, di dover registrare dei numeri di telefono di varie persone: un valore "stringa di lunghezza zero" potrebbe rappresentare il fatto che la persona non ha proprio nessun numero di telefono, mentre un valore NULL rappresenta il fatto che non sappiamo se una certa persona abbia un numero di telefono (potrebbe averlo e noi non lo sappiamo).
Una caratteristica interessante del valore NULL che bisogna conoscere, altrimenti potremmo avere delle sorprese spiacevoli, è il fatto che tale valore nullo si propaga al risultato di espressioni che lo contengono.

Se, per esempio, sommiamo due valori numerici e uno di questi contiene un valore NULL, otterremo un valore NULL anche come risultato. Questo comportamento vale con qualsiasi espressione numerica e anche con gli operatori di confronto.
Riportiamo, di seguito, alcuni esempi:

```
-- A contiene un numero qualsiasi e B un valore NULL
A + B     -- risultato = NULL
A - B     -- risultato = NULL
A / B     -- risultato = NULL
A * B     -- risultato = NULL
A > B     -- risultato = NULL
A <= B    -- risultato = NULL
```

Se, invece, lavoriamo con gli operatori booleani (AND, OR e NOT), il risultato sarà quello indicato nelle tabelle di verità che abbiamo visto nei paragrafi precedenti.

Istruzioni per il controllo del flusso del programma

Come abbiamo già affermato, T-SQL può essere considerato un vero e proprio linguaggio di programmazione. Lo dimostra il fatto che è dotato di un nutrito insieme di istruzioni di controllo del flusso del programma, cioè di quelle istruzioni che permettono di cambiare il corso del programma durante la sua esecuzione, in base soprattutto alla valutazione dello stato del programma stesso (valore delle variabili).

BEGIN ... END

L'istruzione BEGIN ... END definisce un insieme di istruzioni da eseguire come un'unità a sé stante.
La forma in cui viene utilizzata questa istruzione è la seguente:

```
BEGIN
     {
        sql_statement | statement_block
     }
END
```

Per esempio, possiamo vedere il seguente codice, molto semplice, nel quale verifichiamo il valore della variabile @@FLAG. Se il valore è uguale a zero, eseguiamo tutte le istruzioni all'interno del blocco BEGIN ... END, costituite dalla modifica del valore e dalla stampa di un messaggio:

```
USE NorthWind;
GO
IF @@FLAG = 0
BEGIN
    @@FLAG = 1;
    PRINT N'Attivato il flag.';
```

```
END;
PRINT N'Fine elaborazione.';
GO
```

Ovviamente, se `@@FLAG` è uguale a "1", nessuna delle istruzioni all'interno del blocco viene eseguita.

IF ... ELSE

L'istruzione `IF` definisce un blocco di istruzioni che viene eseguito solamente se la condizione è vera. Opzionalmente è possibile inserire un ulteriore blocco di istruzioni, preceduto dalla clausola `ELSE`, che viene eseguito in caso contrario, cioè se la condizione è falsa.
La forma in cui viene utilizzata questa istruzione è la seguente:

```
IF Boolean_expression
     { sql_statement | statement_block }
[ ELSE
     { sql_statement | statement_block } ]
```

Supponiamo di voler confrontare il volume di vendite di un rappresentante con la media delle vendite di tutti i rappresentanti:

```
DECLARE @vendite money, @media money
SET @vendite = 1000
SET @media = 500
IF @vendite < @media
    PRINT 'Hai venduto troppo poco.'
ELSE
    PRINT 'Complimenti!'
```

In questo caso le vendite sono superiori alla media e quindi verrà visualizzata la stringa "`Complimenti!`". Se la variabile `@vendite` avesse un valore inferiore a 500, invece, verrebbe visualizzata la stringa "`Hai venduto troppo poco.`".

NOTA

In questo caso abbiamo valorizzato le variabili direttamente nel codice, per semplicità, ma nulla vieta di valorizzarle in modo dinamico, utilizzando il risultato di una query.

GOTO

Permette di interrompere l'esecuzione sequenziale delle istruzioni e di saltare direttamente a un'etichetta, definita nel codice.
Un'etichetta può essere definita indicando un nome di etichetta all'inizio della riga, seguita da due punti (`:`).

Per esempio:

```
miaetichetta :
```

Per eseguire le istruzioni che seguono l'etichetta, è sufficiente inserire l'istruzione `GOTO` in questo modo:

```
GOTO miaetichetta
```

Un esempio pratico:

```
PRINT 'Vediamo come si usa GOTO'
GOTO saltaqui
PRINT 'Riga 2'
saltaqui :
PRINT 'Ho fatto il salto!'
```

Il risultato sarà il seguente:

```
Vediamo come si usa GOTO
Ho fatto il salto!
```

L'istruzione `PRINT 'Riga 2'` non sarà mai eseguita.
Per utilizzare l'istruzione in modo più utile, è possibile inserire `GOTO` all'interno di un'istruzione `IF ... ELSE`:

```
PRINT 'Vediamo come si usa GOTO'
IF @flag = 1
    GOTO saltaqui
ELSE
    PRINT 'Riga 2'
saltaqui :
PRINT 'Ho fatto il salto!'
```

In questo caso il risultato sarà diverso a seconda del valore della variabile `@flag`. Se il valore è uguale a uno, otterremo il seguente risultato:

```
Vediamo come si usa GOTO
Riga 2
Ho fatto il salto!
```

Se, invece, il valore è diverso, otterremo il risultato che abbiamo già visto nel primo semplice esempio.

RETURN

Permette di uscire immediatamente da una query o da una procedura.
Questa istruzione può essere utilizzata in qualsiasi punto di una procedura, batch o blocco di istruzioni. Ovviamente tutte le istruzioni successive all'istruzione `RETURN` non possono essere eseguite.

La sintassi è la seguente:

```
RETURN [espressione intera]
```

L'argomento `[espressione intera]` è opzionale e deve consistere in un valore intero da restituire. Salvo altre indicazioni documentate, tutte le stored procedure di sistema restituiscono un valore `0` (zero). Tale valore indica che la stored procedure è stata eseguita con successo, mentre qualsiasi altro valore intero indica che si è verificato un errore. Vediamo un paio di esempi sull'uso di `RETURN`.
Nel primo esempio vediamo semplicemente l'uscita da un blocco di istruzioni:

```
CREATE PROCEDURE TrovaProcessi @nome sysname = NULL
AS
IF @nome IS NULL
    BEGIN
        PRINT 'Devi fornire un nome di un utente.'
        PRINT '(prova con 'sa' o un altro amministratore)'
        RETURN
    END
ELSE
    BEGIN
        SELECT sysobj.name, sysobj.id, sysobj.uid
        FROM sysobjects sysobj
            INNER JOIN master..syslogins syslog
            ON sysobj.uid = syslog.sid
        WHERE syslog.name = @nome
    END;
```

Nel secondo esempio, vediamo come può essere restituito un valore intero:

```
USE NorthWind;
GO
CREATE PROCEDURE VerificaProvStato @param int
AS
IF (SELECT [Stato/Provincia]
       FROM Clienti
    WHERE ID = @param) = 'RM'
    RETURN 1
ELSE
    RETURN 2;
GO
```

NOTA

Non preoccupatevi se anticipiamo qualche esempio e concetto delle stored procedure, perché ci sarà modo di approfondire l'argomento nel corso del libro. In particolare, vedremo come si verifica e come si esegue la query che crea una stored procedure, nonché come si fa a eseguire la stored procedure creata per ottenere in restituzione un risultato.

USE

Nell'esempio che abbiamo appena visto, avrete notato che la prima istruzione è `USE NorthWind`.
L'istruzione `USE` specifica che tutti i comandi successivi dovranno essere eseguiti mantenendo come riferimento il database indicato.
In questo modo, anche se in un dato momento stiamo lavorando a un database diverso da quello che ci interessa, possiamo eseguire una query o una stored procedure che utilizza le informazioni memorizzate nel database indicato nell'istruzione `USE`, senza il rischio di sbagliare.

WAITFOR

L'istruzione `WAITFOR`, come si può dedurre dal nome, blocca l'esecuzione di un batch, di una stored procedure o di una transazione finché non è trascorso un preciso lasso di tempo, non è stata raggiunta l'ora attesa oppure un'istruzione specificata non ha modificato o restituito almeno una riga.
La sintassi di questa istruzione è la seguente:

```
WAITFOR
{
    DELAY 'tempo da trascorrere'
  | TIME 'tempo di esecuzione'
  | [(receive_statement) | (get_conversation_group_statement)]
    [, TIMEOUT timeout] }
```

Vediamo quale significato hanno le varie sezioni di questa istruzione:

- `DELAY 'tempo da trascorrere'`: indica per quanto tempo bisogna attendere, fino a un massimo di 24 ore, prima di proseguire con l'esecuzione;
- `TIME 'tempo di esecuzione'`: indica l'orario preciso in cui deve essere fatta partire l'esecuzione;
- `receive_statement`, `get_conversation_group_statement` e `TIMEOUT`: sono applicabili solo ai messaggi del **Service Broker**.

Gli argomenti data/ora da passare all'istruzione `WAITFOR` sono espresse con dati di tipo `datetime` e in uno dei formati validi.
Vediamo ora qualche esempio: nel primo proviamo a eseguire la stored procedure `sp_update_job` per modificare il nome di un job. La seconda istruzione `EXECUTE` verrà eseguita esattamente alle ore 15:36.

```
USE msdb;
EXECUTE sp_add_job @job_name = 'TrovaProcessi';
BEGIN
```

```
    WAITFOR TIME '15:36';
    EXECUTE sp_update_job @job_name = 'TrovaProcessi',
        @new_name = 'TrovaProcessiAggiornato';
END;
GO
```

Il seguente esempio, invece, mostra come eseguire una stored procedure dopo aver atteso per un certo lasso di tempo (nell'esempio è stato inserito l'argomento `'00:01'` che indica un minuto, cioè esattamente 60 secondi):

```
BEGIN
    WAITFOR DELAY '00:01';
    EXECUTE sp_helpdb;
END;
GO
```

WHILE

L'istruzione `WHILE` permette di eseguire ripetutamente un'istruzione o un blocco di istruzioni. Il numero di ripetizioni del ciclo `WHILE` è determinato dal valore booleano (`True`/`False`) che viene restituito da un'espressione che costituisce la condizione da verificare. A ogni ripetizione del ciclo viene nuovamente verificata la condizione e, se quest'ultima restituisce True, il ciclo si ripete. In caso contrario, il controllo dell'esecuzione prosegue con la prima istruzione successiva al blocco di istruzioni incluso nell'istruzione `WHILE`.
All'interno del ciclo possono essere inserite delle parole riservate che permettono di modificare il corso dell'esecuzione: `BREAK` e `CONTINUE` (vedremo a cosa servono nei prossimi due paragrafi).
L'istruzione è formata sintatticamente come segue:

```
WHILE Boolean_expression
{
     sql_statement
   | statement_block
   | BREAK
   | CONTINUE
}
```

Nell'esempio seguente eseguiamo un ciclo veramente molto semplice. Definiamo la variabile `@max` che costituisce il valore più alto da raggiungere e la variabile `@conteggio` che è la variabile che conta il numero di ripetizioni di ciclo eseguite. Successivamente definiamo l'istruzione `WHILE` con il test necessario (`@conteggio <= @max`) e, infine, scriviamo il blocco di istruzioni da eseguire:

```
DECLARE @max int;
SET @max = 10;
DECLARE @conteggio int;
SET @conteggio = 1;

WHILE @conteggio <= @max
    BEGIN
     PRINT @conteggio
     SET @conteggio = @conteggio + 1
    END;
```

Il risultato è costituito dai numeri da 1 a 10, uno dopo l'altro su righe successive.

BREAK

La parola riservata BREAK forza l'uscita immediata dal ciclo WHILE.
Se all'interno di un ciclo WHILE si incontra la parola BREAK, tutte le istruzioni successive fino alla parola END vengono ignorate.

CONTINUE

La parola riservata CONTINUE riavvia un ciclo WHILE. Qualsiasi istruzione successiva alla parola CONTINUE viene ignorata.
La parola CONTINUE è utilizzata frequentemente, anche se non sempre, all'interno di un blocco IF: in questo modo il ciclo si riavvia solamente se è verificata la condizione.

```
USE NorthWind;
GO
WHILE (SELECT AVG([Prezzo di listino]) FROM Prodotti) < 300
BEGIN
   UPDATE Prodotti
      SET [Prezzo di listino] = [Prezzo di listino] * 2
   SELECT MAX([Prezzo di listino]) FROM Prodotti
   IF (SELECT MAX([Prezzo di listino]) FROM Prodotti) > 500
      BREAK
   ELSE
      CONTINUE
END
PRINT 'Troppo da sopportare per il mercato';
```

Espressioni

Nel linguaggio T-SQL esistono anche tre importanti espressioni che ci permettono di controllare meglio il corso dell'esecuzione di un programma: CASE, COALESCE e NULLIF.

CASE

L'espressione CASE valuta una lista di condizioni e restituisce uno solo tra tutti i possibili risultati dell'espressione.

Esistono due formati diversi:

- l'espressione **CASE semplice**: confronta un'espressione con un insieme di espressioni semplici per determinare il risultato;
- l'espressione **CASE avanzata**: valuta un insieme di espressioni booleane per determinare il risultato.

Entrambi i formati supportano l'argomento `ELSE` opzionale.
`CASE` può essere utilizzato in qualsiasi istruzione o clausola che permette un'espressione valida. Per esempio, è possibile utilizzare questa espressione in istruzioni come `SELECT`, `UPDATE`, `DELETE` e `SET`, ma anche in clausole quali `IN`, `WHERE`, `ORDER BY` e `HAVING`.
La sintassi dell'espressione **CASE semplice** è la seguente:

```
CASE input_expression
     WHEN when_expression THEN result_expression [ ...n ]
     [ ELSE else_result_expression ]
END
```

La sintassi dell'espressione **CASE avanzata**, invece, è la seguente:

```
CASE
     WHEN Boolean_expression THEN result_expression [ ...n ]
     [ ELSE else_result_expression ]
END
```

Nell'esempio seguente sostituiamo la sigla della provincia di residenza dei dipendenti con il nome per esteso, utilizzando un'espressione **CASE semplice** basata sul controllo di eguaglianza. Non sono consentiti altri tipi di confronto (maggiore, minore ecc.):

```
USE NorthWind;
GO
SELECT   Cognome, Nome, Provincia =
      CASE [Stato/Provincia]
         WHEN 'RM' THEN 'Roma'
         WHEN 'LT' THEN 'Latina'
         ELSE 'Altra provincia'
      END
FROM Dipendenti
ORDER BY Cognome;
GO
```

Con un'espressione **CASE avanzata** possiamo invece utilizzare tutti gli operatori di confronto:

```
USE NorthWind;
GO
SELECT   [Codice Prodotto], [Nome prodotto],
   'Intervallo di prezzo' =
      CASE
```

```
         WHEN [Prezzo di listino] =  0
            THEN 'Non destinato alla vendita'
         WHEN [Prezzo di listino] < 50
            THEN 'Inferiore a 50 €'
         WHEN [Prezzo di listino] >= 50
            and [Prezzo di listino] < 250
            THEN 'Inferiore a 250 €'
         WHEN [Prezzo di listino] >= 250
            and [Prezzo di listino] < 1000
            THEN 'Inferiore a 1000 €'
         ELSE 'Oltre 1000'
      END
FROM Prodotti
ORDER BY [Codice prodotto] ;
GO
```

Vediamo ora un esempio di utilizzo dell'espressione `CASE` in una clausola `ORDER BY`. In questo esempio ordiniamo i record secondo il cognome del dipendente, ma con due criteri ben distinti:

- se la posizione è "`Venditore`", ordiniamo in senso crescente;
- se la posizione è diversa da "`Venditore`", ordiniamo in senso decrescente.

```
SELECT Cognome, Posizione
FROM Dipendenti
ORDER BY CASE Posizione WHEN 'Venditore' THEN Cognome END
    ,CASE WHEN Posizione <> 'Venditore' THEN Cognome END DESC;
GO
```

COALESCE

L'espressione `COALESCE` restituisce la prima espressione non `Null` tra gli argomenti relativi.
La sintassi da utilizzare è piuttosto semplice:

```
COALESCE ( expression [ ,...n ] )
```

Questa espressione equivale all'espressione `CASE` seguente:

```
CASE
    WHEN (expression1 IS NOT NULL) THEN expression1
    WHEN (expression2 IS NOT NULL) THEN expression2
    ...
ELSE expressionN
END
```

Per chiarire ancora meglio il concetto, vediamo un paio di esempi, uno semplice e uno un po' più complesso.

```
USE AdventureWorks2008R2;
GO
SELECT Name, Class, Color, ProductNumber,
```

```
COALESCE(Class, Color, ProductNumber) AS FirstNotNull
FROM Production.Product;
GO
```

In questo esempio, nella colonna `FirstNotNull` otteniamo il primo dei valori non-nulli, tra tutti i campi indicati come argomento della funzione `COALESCE` e nello stesso ordine indicato. Il comportamento della funzione, quindi, è il seguente:

- se tutti i campi sono diversi da `NULL`, sarà riportato il valore della colonna `Class`;
- se il primo campo (`Class`) è `NULL`, sarà riportato il valore di `Color`;
- se i primi due campi sono uguali a `NULL`, sarà riportato il valore della colonna `ProductNumber`;
- se tutti i campi valgono `NULL`, otterremo un `NULL`.

Come esempio un po' più complesso, vediamo il seguente:

```
SET NOCOUNT ON;
GO
USE tempdb;
IF OBJECT_ID('dbo.es_coalesce') IS NOT NULL
    DROP TABLE es_coalesce;
GO
CREATE TABLE dbo.es_coalesce
(
    id            tinyint   identity,
    tariffa_ora   decimal   NULL,
    salario       decimal   NULL,
    commissioni   decimal   NULL,
    num_vendite   tinyint   NULL
);
GO
INSERT dbo.es_coalesce (tariffa_ora, salario, commissioni, num_vendite)
VALUES
    (10.00, NULL, NULL, NULL),
    (20.00, NULL, NULL, NULL),
    (30.00, NULL, NULL, NULL),
    (40.00, NULL, NULL, NULL),
    (NULL, 10000.00, NULL, NULL),
    (NULL, 20000.00, NULL, NULL),
    (NULL, 30000.00, NULL, NULL),
    (NULL, 40000.00, NULL, NULL),
    (NULL, NULL, 15000, 3),
    (NULL, NULL, 25000, 2),
    (NULL, NULL, 20000, 6),
    (NULL, NULL, 14000, 4);
GO
SET NOCOUNT OFF;
GO
SELECT CAST(COALESCE(tariffa_ora * 40 * 52,
   salario,
   commissioni * num_vendite) AS money) AS 'Salario totale'
FROM dbo.es_coalesce
ORDER BY 'Salario totale';
GO
```

In questo esempio eseguiamo le seguenti operazioni:

- verifichiamo se esiste una tabella di nome `es_coalesce` nel database `tempdb` e, se esiste, la eliminiamo;
- creiamo una nuova tabella `es_coalesce`, definendone la struttura;
- inseriamo alcune righe di dati nella tabella;
- infine, selezioniamo i dati, utilizzando l'espressione `COALESCE`.

Il risultato sarà il seguente:

```
Salario totale
--------------------
10000,00
20000,00
20800,00
30000,00
40000,00
41600,00
45000,00
50000,00
56000,00
62400,00
83200,00
120000,00

(12 row(s) affected)
```

In base ai dati esistenti, viene eseguita la corrispondente operazione e viene fornito un risultato non nullo per tutte le righe.

NULLIF

L'espressione `NULLIF` restituisce un valore `NULL` se le due espressioni specificate sono uguali.
La sintassi di questa espressione è la seguente:

```
NULLIF (expression, expression)
```

Nell'esempio seguente creiamo una nuova tabella (`budget`), la popoliamo con qualche dato, tra cui anche alcuni `NULL`, e poi calcoliamo la media dei valori. Dove non esiste il dato per l'anno corrente, utilizziamo il dato dell'anno precedente, sfruttando ancora una volta l'espressione `COALESCE`.

```
USE NorthWind;
GO
IF OBJECT_ID ('dbo.budget','U') IS NOT NULL
   DROP TABLE budget;
GO
SET NOCOUNT ON;
```

```
CREATE TABLE dbo.budget
(
   dipartimento      tinyint   IDENTITY,
   anno_corrente     decimal   NULL,
   anno_precedente   decimal   NULL
);
INSERT budget VALUES(100000, 150000);
INSERT budget VALUES(NULL, 300000);
INSERT budget VALUES(0, 100000);
INSERT budget VALUES(NULL, 150000);
INSERT budget VALUES(300000, 250000);
GO
SET NOCOUNT OFF;
SELECT AVG(NULLIF(COALESCE(anno_corrente,
   anno_precedente), 0.00)) AS 'Budget Medio'
FROM budget;
GO
```

Il risultato che otterremo è il seguente:

```
Budget Medio
---------------------------------------
212500.000000
Warning: Null value is eliminated by an aggregate or other SET operation.

(1 row(s) affected)
```

Ecco, invece, un esempio che permette di confrontare l'espressione `NULLIF` con un costrutto `CASE ... END`.

```
USE AdventureWorks2008R2;
GO
SELECT ProductID, MakeFlag, FinishedGoodsFlag,
   NULLIF(MakeFlag,FinishedGoodsFlag)AS 'Null if Equal'
FROM Production.Product
WHERE ProductID < 10;
GO

SELECT ProductID, MakeFlag, FinishedGoodsFlag,'Null if Equal' =
   CASE
       WHEN MakeFlag = FinishedGoodsFlag THEN NULL
       ELSE MakeFlag
   END
FROM Production.Product
WHERE ProductID < 10;
GO
```

Come potete vedere, il risultato è lo stesso, ma ovviamente l'utilizzo di `NULLIF` semplifica e riduce la dimensione del codice da scrivere:

```
ProductID   MakeFlag FinishedGoodsFlag Null if Equal
----------- -------- ----------------- -------------
1           0        0                 NULL
2           0        0                 NULL
3           1        0                 1
4           0        0                 NULL

(4 row(s) affected)

ProductID   MakeFlag FinishedGoodsFlag Null if Equal
----------- -------- ----------------- -------------
1           0        0                 NULL
2           0        0                 NULL
3           1        0                 1
4           0        0                 NULL

(4 row(s) affected)
```

Conclusioni

Abbiamo visto varie istruzioni ed espressioni T-SQL, con le quali è possibile creare veri e propri programmi, in forma di **stored procedure**.

L'argomento è molto importante, perché permette di inserire parte della logica applicativa nel livello dati di un sistema, semplificando in questo modo lo sviluppo delle applicazioni che utilizzano i database di SQL Server. L'argomento non si esaurisce in questo capitolo, perché T-SQL è utilizzato anche per manipolare oggetti di database di ogni genere, come vedremo nei prossimi capitoli.