ISTITUTO TECNICO INDUSTRIALE
G. M. ANGIOY
SASSARI

PROGETTAZIONE DI DATABASE

Microsft Transact SQL

Lezione 21

Versione del: 07/05/2015
Revisione numero: 4

Prof. Andrea Zoccheddu
Dipartimento di Informatica

Immagine di copertina da: http://www.iamsterdam.com/en-GB/living/education/Dutch-Education-System

DIPARTIMENTO
INFORMATICA E TELECOMUNICAZIONI

IL DIALETTO TRANSACT SQL DI MICROSOFT

MICROSOFT SQL SERVER

IL PRODOTTO

MICROSOFT SQL SERVER

Le dispense SQL descrivono un generico SQL utilizzato nella maggior parte dei database, noto come standard SQL92; tuttavia esistono delle varianti a questo linguaggio, generalmente indicate come «dialetti».

Microsoft Sql Server è un ambiente di gestione di basi di dati (RDBMS) che ha proposto diverse versioni ed utilizza uno specifico dialetto non come T-SQL, ovvero Transact-SQL. In questa dispensa analizzeremo la sintassi del linguaggio T-SQL per creare, gestire e analizzare i database.

Per saperne di più sul RDBMS e sul T-SQL è possibile consultare i seguenti link:

technet.microsoft.com	http://technet.microsoft.com/it-it/library/bb500469.aspx
Guida di riferimento a Transact-SQL	http://technet.microsoft.com/it-it/library/bb510741(v=sql.110).aspx
Versioni e distribuzioni di SQL Server	http://it.wikipedia.org/wiki/Microsoft_SQL_Server
Installazione per SQL Server 2012	http://technet.microsoft.com/it-it/library/bb500469.aspx

Le versioni di SQL Server sono state molte; traggo le principali da Wikipedia:

anno	*nome*	*nome in codice*	*versione gratuita*
1993	SQL Server 4.2	-	-
1995	SQL Server 6.0	SQL95	No
1996	SQL Server 6.5	Hydra	No
1998	SQL Server 7.0	Sphinx	No
1999	SQL Server 7.0 OLAP	Plato	No
2000	SQL Server 2000 32-bit	Shiloh	No
2003	SQL Server 2000 64-bit	Liberty	No
2005	SQL Server 2005 (sia 32-bit che 64-bit)	Yukon	Si
2008	SQL Server 2008 (sia 32-bit che 64-bit)	Katmai	Express
2010	SQL Server 2008 R2 (sia 32-bit che 64-bit)	-	Express
2012	SQL Server 2012 (sia 32-bit che 64-bit) in fase beta	Denali	Express

Il linguaggio TSQL che è affrontato in questa dispensa:

- Si riferisce alla versione SQL Server 2008 (sia 32-bit che 64-bit);
- È trattato in modo parziale e sintetico, tralasciando alcuni elementi troppo tecnici.

DATA DESCRIPTION LANGUAGE

Il DDL di TSQL permette di utilizzare i consueti comandi di gestione delle strutture del database (creazione, modifica, eliminazione) tra cui analizzeremo i seguenti:

COMANDO		DESCRIZIONE
CREATE	DATABASE	Serve per creare un nuovo database
CREATE OR ALTER	**TYPE**	Serve per definire un nuovo tipo di dato
CREATE OR ALTER	TABLE	Serve per definire una tabella dentro il database
CREATE OR ALTER	INDEX	Serve per definire un indice di ricerca sui dati
CREATE OR ALTER	VIEW	Serve per definire una vista (query con diritti)
CREATE OR ALTER	TRIGGER	Serve per definire un meccanismo automatico sui dati
CREATE OR ALTER	PROCEDURE	Serve per definire una procedura strutturata e articolata
CREATE OR ALTER	**RULE**	Serve per definire un vincolo globale sul database
CREATE OR ALTER	GROUP	Serve per definire un gruppo di utenti
CREATE OR ALTER	USER	Serve per definire un utente
CREATE OR ALTER	GRANT	Serve per definire i diritti di accesso ai dati

DATA MANIPULATION LANGUAGE

Il DML di TSQL permette di utilizzare i consueti comandi di gestione dei dati contenuti nelle strutture del database, tra cui analizzeremo i seguenti:

SELECT	Serve per interrogare il database e ottenere informazioni calcolate
INSERT INTO	Serve per inserire record nelle tabelle
DELETE	Serve per cancellare record dalle tabelle
UPDATE	Serve per modificare alcuni campi di record esistenti nelle tabelle

TSQL PER PROCEDURE E TRIGGER

Oltre ai precedenti comandi, il DML di TSQL permette di scrivere il corpo delle procedure create con il consueto comando e prevede una sintassi specifica per elaborare i dati; analizzeremo la sintassi relativa a:

VARIABILI	Dichiarazione di variabili e assegnazione di valori, con operazioni
IF	Serve per decidere l'esecuzione di blocchi in funzione di un controllo
WHILE	Serve per iterare l'esecuzione di blocchi in funzione di un controllo
PARAMETRI	Servono per far comunicare la procedura con il chiamante

TSQL PER GESTIRE STRINGHE E DATE

Infine, oltre alle consuete funzioni previste dal SQL92, il DML di TSQL permette di usare funzioni per gestire dati che spesso sfuggono allo standard; analizzeremo la sintassi relativa a:

STRINGHE	Funzioni per manipolare stringhe, compiere ricerche e produrre nuove stringhe
DATE	Funzioni per ottenere date e orari dal sistema operativo, per gestire i formati di date/orari

CREAZIONE DEL DATABASE

All'inizio c'era il nulla … e il database non esisteva ancora. Il verbo della sintassi standard è evocativo della realizzazione di un database dal nulla:

T-SQL

Sintassi generale del comando **CREATE DATABASE**:

```
CREATE DATABASE nome_database
    [ ON
        { [ PRIMARY ] [ <specificofile1> [ ,... specificofileK ]
        [ , <gruppofile1> [ ,... gruppofileH ] ]
    [ LOG ON { <specificofile1> [ ,... specificofileN ] } ] }
    ]
    [ COLLATE nome_raccolta ]
    [ WITH <opzioni_accesso_esterno> ]
];
```

Questo comando crea un nuovo database con un suo nome.

Quando viene creato, il database è vuoto ma, in realtà, il motore del database server costruisce un insieme di tabelle di sistema (non visibili ad utenti generici) per predisporre un ambiente agevole: ad esempio alcuni preparano le tabelle per i contatori (i campi auto-incrementanti) o per gli utenti e i diritti …; ai nostri fini è trascurabile sia l'analisi di queste tabelle, sia tutte le opzioni della sintassi TSQL: possiamo perciò limitarci alla sintassi più elementare (corretta) seguente:

Esempio

Esempio 1) creazione di database:

```
CREATE DATABASE Scuola;
```

sono tralasciate tutte le opzioni accessorie.

COMANDI CREATE

Per avere l'elenco completo dei comandi CREATE di T-SQL è possibile consultare qui: http://technet.microsoft.com/it-it/library/cc879262.aspx

COMANDI DROP

Per ogni comando Create di T-SQL esiste anche il suo antagonista DROP che serve per eliminare una struttura dal database. Per esempio DROP TABLE Biblioteca.

TIPI DI DATO PREESISTENTI

LINK

Fonte: http://msdn.microsoft.com/it-it/library/ms187752.aspx

Come tutti i linguaggi, anche TSQL dispone di tipi di dato pronti. Questi tipi servono per sapere cosa poter archiviare dentro le tabelle (numeri, testo, ecc…). I tipi sono:

Tipo di dato	Categoria	Ampiezza	Descrizione	Esempio
Bigint	Numerici esatti	8 byte	± 9 mld di mld circa	*accessi al sito web* ♥
Numeric	Numerici esatti			
Bit	Numerici esatti		usare per vero/falso, on/off	*si/no* ♥
Smallint	Numerici esatti	2 byte	± 32.000 circa	*pagine di un libro* ♥
Decimal	Numerici esatti			*temperatura febbre* ♥
Smallmoney	Numerici esatti			*prezzo di un oggetto* ♥
Int	Numerici esatti	4 byte	± 2 mld circa	♥
Tinyint	Numerici esatti	1 byte	da 0 a 255	*età umana* ♥
Money	Numerici esatti			*bilancio di una società* ♥
Float	Numerici approssimati	K byte		*programma scientifico*
Real	Numerici approssimati	4 byte		*programma scientifico*
Date	Tempo	da 01-01-0001 a 31-12-9999		*giorno storico* ♥
datetimeoffset	Tempo	Da 1 gennaio 1 D.C. al 31 dicembre 9999 D.C., 23:59:59.9999999		
datetime2	Tempo	Data come **Date**, Orari come **Time**		
smalldatetime	Tempo	da 1 gennaio 1900 a 6 giugno 2079,		*scadenza prestito* ♥
Datetime	Tempo	da 1/1/1753 a 31/12/ 9999, orari da 00:00:00.000 a 23:59:59.997		*momento ingresso docente* ♥
Time	Tempo	da 00.00.00.0000000 a 23.59.59.9999999 (decimilionesimo sec)		*orario ingresso* ♥
char (n)	Stringhe di caratteri	n byte	**n ≤ 1800**	*codice fiscale* ♥
varchar (n)	Stringhe di caratteri	**max** n byte	**n ≤ 1800**	*nome di persona* ♥
Text	Stringhe di caratteri	2.147.483.647 byte	Non unicode	*descrizione di un film* ♥
nchar (n)	Stringhe Unicode	n byte	**n ≤ 1800**	*codice fiscale*
nvarchar (n)	Stringhe Unicode	**max** n byte	**n ≤ 1800**	*nome di persona*
Ntext	Stringhe Unicode	2.147.483.647 byte	Non unicode	*descrizione di un film*
Binary	Stringhe binarie			*Video*
Varbinary	Stringhe binarie			*video* ♥
Image	Stringhe binarie			*foto* ♥
Cursor	Altri tipi di dati			*speciale*
Timestamp	Altri tipi di dati			*momento molto preciso*
Hierarchyid	Altri tipi di dati			
uniqueidentifier	Altri tipi di dati			
sql_variant	Altri tipi di dati			
Xml	Altri tipi di dati			*file xml*
Table	Altri tipi di dati			

Con il simbolo ♥ si mettono in evidenza i tipi usati più di frequente.

NUOVI TIPI DI DATO

In un sistema per database compatibile con lo standard è possibile definire nuovi tipi di dato idonei per informazioni particolari. Nel modello logico relazionale le tabelle si chiamano Relazioni e i valori ammissibili nei campi sono detti Domini.

CREATE TYPE

Per definire un nuovo tipo di dato occorre specificare il suo Nome, il tipo di partenza da cui ottenerlo e una condizione per verificare il requisito a cui il tipo deve rispondere. La sintassi è la seguente:

T-SQL

Sintassi del comando `CREATE TYPE`:

```
CREATE TYPE [ schema_name. ] type_name
FROM base_type
    [ ( precision [ , scale ] ) ]
    [ NULL | NOT NULL ]
  | EXTERNAL NAME assembly_name [ .class_name ]
  | AS TABLE ( { <column_definition> | <computed_column_definition> }
        [ <table_constraint> ] [ ,...n ] ) ;
```

Una volta che si è eseguito il comando è possibile usare il nuovo tipo come se fosse un tipo predefinito.

Esempio

Esempio 2) creazione di un nuovo tipo

```
CREATE TYPE Scuola.Piano
FROM CHAR (1) NOT NULL
AS TABLE (CONSTRAINT CHECK (Value = "T" OR Value BETWEEN "1" AND "5" ) ) ;
```

il tipo Piano è una stringa di un carattere obbligatorio che coincide con la lettera T oppure con il carattere che rappresenta una cifra compresa tra 1 e 5.

Esempio

Esempio 3) creazione di un nuovo tipo

```
CREATE TYPE Scuola.Indirizzo
FROM CHAR (11) NOT NULL
AS TABLE (CONSTRAINT CHECK (Value IN ("Biennio", "Chimica", "Elettronica",
                                      "Informatica", "Meccanica") ) ) ;
```

il tipo Indirizzo è una stringa di 11 caratteri obbligatori che sia incluso tra quelli specificati.

Esempio

Esempio 4) creazione di un nuovo tipo

```
CREATE TYPE Scuola.Sesso
FROM CHAR(1)
AS TABLE (CONSTRAINT CHECK (Value = "F" OR Value = "M") ) ;
```

il tipo Sesso è una stringa di 1 carattere facoltativo, che sia F oppure M (oppure NULL).

Esempio

Esempio 5) creazione di un nuovo tipo

```
CREATE TYPE Scuola.TipoPrezzo
FROM smallmoney
AS TABLE (CONSTRAINT CHECK (Value > 0) ) ;
```

il tipo TipoPrezzo è una moneta i cui elementi devono soddisfare la condizione di essere maggiori di zero (zero escluso, in questo caso).

DEFINIZIONE DI TABELLE

CREATE TABLE

La sintassi generale del comando è:

T-SQL

Sintassi generale della `CREATE TABLE`:

```
CREATE TABLE
    [ database_name . [ schema_name ] . | schema_name . ] table_name
    [ AS FileTable ]
    ( { <column_definition> | <computed_column_definition>
        | <column_set_definition> | [ <table_constraint> ] [ ,...n ] } )
    [ ON { partition_scheme_name ( partition_column_name ) | filegroup
        | "default" } ]
    [ { TEXTIMAGE_ON { filegroup | "default" } ]
    [ FILESTREAM_ON { partition_scheme_name | filegroup
        | "default" } ]
    [ WITH ( <table_option> [ ,...n ] ) ]
[ ; ]
```

La creazione delle tabelle è fondamentale per un database. Solitamente si devono costruire specificando anche chiavi primarie, esterne, vincoli interni ed esterni; ma è anche possibile modificare la struttura della tabella in seguito, variando campi e vincoli.

L'opzione **CONSTRAINT** serve per specificare i vincoli sul campo oppure sull'intera tabella. Se la parola segue un campo significa che i vincoli sono riferiti al campo; se la parola è posta dopo la dichiarazione dei campi significa che i vincoli sono riferiti alla tabella.

CHIAVI PRIMARIE E AUTO-INCREMENTO

Il vincolo **PRIMARY KEY** è ammesso una sola volta in una tabella, ma può essere riferito a più attributi insieme (si sconsiglia comunque di usare PK su molti attributi, conviene dichiarare un contatore).

Esempio

Esempio 6) Il caso più semplice di tabella impone di indicare il suo nome e almeno un campo; non è obbligatorio indicare una chiave primaria (ovvero è sintatticamente corretto) ma è fortemente sconsigliato avere tabelle prive.

```
CREATE TABLE [dbo].[Aule](
	[ID_Aula] [int] NOT NULL,
	[Corpo] [nchar](1) NOT NULL,
	[Piano] [nchar](1) NOT NULL,
	[Numero] [nchar](2) NOT NULL,
	[Mq] [numeric](4, 2) NULL,
   CONSTRAINT [PK_Aule] PRIMARY KEY CLUSTERED
	(
		[ID_Aula] ASC
	)
);
```

L'esempio definisce una tabella per le aule della scuola, il vincolo NOT NULL impone l'obbligatorietà di un valore. La parola chiave `CONSTRAINT` definisce un vincolo denominato `PK_AULE` il quale impone che il campo `ID_AULA` sia una chiave primaria ordinata in modo crescente.

Esempio

Esempio 7) La chiave primaria può essere di qualsiasi tipo ma, spesso, si preferisce usare il tipo intero col vincolo **IDENTITY** che indica un contatore (auto incremento) che può avere anche due argomenti: il primo è il valore di partenza; il secondo è il passo di incremento. L'esempio seguente impone una chiave primaria numerica contatore, che parte da 1 ed avanza di +1 ad ogni nuovo inserimento.

```
CREATE TABLE [dbo].[Aule](
	[ID_Aula] [int] IDENTITY(1,1) NOT NULL,
	[Corpo] [nchar](1) NOT NULL,
	[Piano] [nchar](1) NOT NULL,
	[Numero] [nchar](1) NOT NULL,
	[MetriQuadri] [int] NULL,
   CONSTRAINT [PK_Aule] PRIMARY KEY CLUSTERED
	(
		[ID_Aula] ASC
	)
)
```

la chiave primaria di Aule è costituita dal solo campo ID_Aula

CHIAVI PRIMARIE SU PIÙ CAMPI

Il vincolo `CONSTRAINT` posto dopo la dichiarazione dei campi; in tal caso è possibile esprimere vincoli su più campi insieme e, per esempio, imporre una chiave primaria su più campi oppure una chiave esterna su più campi. Per esempio:

Esempio

Esempio 8) definizione di una primary key su più campi:

```
CREATE TABLE [dbo].[Aule](
	[ID_Aula] [int] NOT NULL,
	[Corpo] [nchar](1) NOT NULL,
	[Piano] [nchar](1) NOT NULL,
	[Numero] [nchar](2) NOT NULL,
	[MetriQuadri] [numeric](4, 2) NULL,
   CONSTRAINT [PK_Aule_prova_1] PRIMARY KEY CLUSTERED
	(
		[Corpo] ASC,
		[Piano] ASC,
		[Numero] ASC
	)
)
```

la chiave primaria di Aule è costituita dalla tripla (Corpo, Piano, Numero)

INDICE UNIVOCO

Se occorre imporre che altri campi siano unici (e magari anche obbligatori diventando così delle chiavi alternative) è possibile usare il vincolo UNIQUE. Il vincolo si può usare su un solo campo o su un gruppo di campi.

Esempio

Esempio 9) imposizione di vincoli `UNIQUE` su più campi:

```
CREATE TABLE [dbo].[Aule](
	[ID_Aula] [int] IDENTITY(1,1) NOT NULL,
	[Corpo] [char](1) NOT NULL,
	[Piano] [char](1) NOT NULL,
	[Numero] [char](2) NOT NULL,
	[Mq] [decimal](4, 2) NULL,
	PRIMARY KEY CLUSTERED
	(	[ID_Aula] ASC		)
	UNIQUE NONCLUSTERED
	(	[Numero] ASC		)
	UNIQUE NONCLUSTERED
	(	[Piano] ASC		)
	UNIQUE NONCLUSTERED
	(	[Corpo] ASC		)
)
```

I vincoli `UNIQUE` stabiliti sulla tabella impongono che non possano esistere due aule con corpi identici (es. corpo H), né due aule con piani identici (es. piano 1), né due aule con numeri identici (es. numeri 17). Questo vincolo non funzionerebbe bene, poiché questo impedisce di avere più aule nello stesso corpo, impedisce di avere più aule con lo stesso piano, e più aule con stesso numero.

Esempio

Esempio 10) altra modalità di imposizione di vincoli `PK` e `UNIQUE`

```
CREATE TABLE [dbo].[Aule](
	[ID_Aula] [int] IDENTITY(1,1) NOT NULL,
	[Corpo] [char](1) NOT NULL,
	[Piano] [char](1) NOT NULL,
	[Numero] [char](2) NOT NULL,
	[Mq] [decimal](4, 2) NULL,
	CONSTRAINT [Nome_vincolo_PK] PRIMARY KEY CLUSTERED
	(	[ID_Aula] ASC		),
	CONSTRAINT [Nome_vincolo_UQ_Numero] UNIQUE NONCLUSTERED
	(	[Numero] ASC		),
	CONSTRAINT [Nome_vincolo_UQ_Piano] UNIQUE NONCLUSTERED
	(	[Piano] ASC		),
	CONSTRAINT [Nome_vincolo_UQ_Corpo] UNIQUE NONCLUSTERED
	(	[Corpo] ASC		)
)
```

In questo caso si sono costruiti i vincoli con rispettivi nomi (si potranno eliminare in seguito).

Sebbene l'esempio proposto non sia funzionale, è opportuno osservare che spesso l'indice viene creato separatamente con comandi separati come nel seguente esempio:

Esempio

Esempio 11) creazione della tabella:

```
CREATE TABLE [dbo].[Aule](
	[ID_Aula] [int] NOT NULL,
	[Corpo] [nchar](1) NOT NULL,
	[Piano] [nchar](1) NOT NULL,
	[Numero] [nchar](2) NOT NULL,
	[Mq] [numeric](4, 2) NULL,
  CONSTRAINT [PK_Aule] PRIMARY KEY CLUSTERED
  (
	[ID_Aula] ASC
  )
GO
```

Creazione del primo indice univoco:

```
CREATE UNIQUE NONCLUSTERED INDEX [NDX_Corpo] ON [dbo].[Aule]
(	[Corpo] ASC  )
GO
```

Creazione del primo indice univoco:

```
CREATE UNIQUE NONCLUSTERED INDEX [NDX_Piano] ON [dbo].[Aule]
(	[Piano] ASC  )
GO
```

Creazione del primo indice univoco:

```
CREATE UNIQUE NONCLUSTERED INDEX [NDX_Numero] ON [dbo].[Aule]
(	[Numero] ASC )
GO
```

Questa sequenza costruisce la tabella con i vincoli di unicità sui tre campi.

Abbiamo detto che i vincoli `UNIQUE` stabiliti sulla tabella non funzionerebbe bene, poiché questo impedisce di avere più aule nello stesso corpo, impedisce di avere più aule con lo stesso piano, e più aule con stesso numero.

La soluzione migliore invece sarebbe creare non tre indici univoci distinti ma un indice sulla tripla che coinvolge contemporaneamente i tre campi da vincolare.

Esempio

Esempio 12) imposizione di vincoli `UNIQUE` su più campi:

```
CREATE UNIQUE NONCLUSTERED INDEX [Unica_Tripla] ON [dbo].[Aule]
(
	[Corpo] ASC,
	[Piano] ASC,
	[Numero] ASC
)
```

Questo esempio impedisce di duplicare la tripla (Corpo, Piano, Numero) e quindi impedisce che esistano due aule nello stesso corpo, allo stesso piano e con lo stesso numero; permette comunque di avere due aule allo stesso corpo e piano ma con diverso numero, o stesso piano e numero, ma diverso corpo e così via.

CHIAVI ESTERNE

Il vincolo **FOREIGN KEY** è riferito ad un campo, ma può essere riferito a più attributi insieme (si sconsiglia) ed è sempre seguito dal riferimento alla tabella ed al campo correlato (solitamente una chiave primaria). Il vincolo **FOREIGN KEY** serve per imporre su un campo (o su un gruppo di campi) il vincolo di integrità referenziale ovvero la verifica che il valore sia presente in un campo di un'altra tabella correlata.

Esempio

Esempio 13) creazione di una tabella con una chiave esterna (attenzione alle virgole):

```
CREATE TABLE [dbo].[Classi](
	[ID_Classe] [int] IDENTITY(1,1) NOT NULL,
	[Anno] [char](1) NOT NULL,
	[Sezione] [char](1) NOT NULL,
	[Indirizzo] [char](1) NOT NULL,
	[ID_Aula] [int] NULL,
	PRIMARY KEY CLUSTERED
	(
		[ID_Classe] ASC
	),
	CONSTRAINT [UQ_Classi_Diverse] UNIQUE NONCLUSTERED
	(
		[Anno] ASC,
		[Sezione] ASC,
		[Indirizzo] ASC
	),
	CONSTRAINT [FK_Classi_Aule] FOREIGN KEY
	(
		[ID_Aula]
	) REFERENCES [dbo].[Aule]
)
GO
```

La tabella include un vincolo di chiave primaria, un vincolo di unicità a tre campi e un vincolo di chiave esterna.

CHIAVI ESTERNE SU PIÙ CAMPI

È opportuno ricordare che le chiavi esterne su più campi vincolano la n-pla a coincidere con una n-pla equivalente nella tabella correlata; questo vincolo non è consigliato, sebbene in alcuni casi sia utile per assicurare che i valori siano presenti nella tabella correlata. Nella pratica si impone una chiave primaria identity e poi si costruisce la FK su quel solo campo.

Il seguente esempio propone un vincolo di chiave esterna su più campi:

Esempio

Esempio 14) La chiave primaria seguente è definita su tre campi diversi:

```
CREATE TABLE [dbo].[Studenti](
	[ID_Studente] [int] IDENTITY(1,1) NOT NULL,
	[Cognome] [nchar](50) NOT NULL,
	[Nome] [nchar](50) NOT NULL,
	[ID_Anno] [char](1) NULL,
	[ID_Sezione] [char](1) NULL,
	[ID_Indirizzo] [char](1) NULL,
	CONSTRAINT [PK_Studenti] PRIMARY KEY CLUSTERED
	(	[ID_Studente] ASC	),
	CONSTRAINT [FK_Studenti_Classi] FOREIGN KEY
	(	[ID_Anno],
		[ID_Sezione],
		[ID_Indirizzo]
	) REFERENCES [dbo].Classi
		(	[Anno],
			[Sezione],
			[Indirizzo]	)
)
GO
```

la chiave esterna è costituita dalla tripla ID_Anno, ID_Sezione, ID_Indirizzo.

TABELLE ED INTEGRITÀ REFERENZIALE

GESTIONE DEI TENTATIVI DI VIOLAZIONE DI INTEGRITÀ REFERENZIALE

Chi ha studiato l'integrità referenziale ricorderà che essa obbliga i campi di una tabella (detta secondaria) ai campi di un'altra tabella (detta primaria) e spesso alle sue chiavi primarie. Questo vincolo può essere violato quando si cancellano dei record nella tabella primaria o si modificano i valori dei campi a cui fanno riferimento altre tabelle. Nella dichiarazione di tabella è possibile allora esplicitare le contromisure da adottare in questi casi. Vediamo le possibili sintassi:

SQL

Sintassi generale di creazione tabella con chiave esterna e gestione dei tentativi di violazione:

```
CREATE TABLE Tabella (
        . . . . . ,
        Campo_FK Tipo Vincolo,
        CONSTRAINT [Nome_Vincolo]
        FOREIGN KEY (Campo_FK) REFERENCES AltraTabella (AltroCampo)
                ON DELETE NO ACTION | CASCADE | SET NULL | SET DEFAULT
                ON UPDATE NO ACTION | CASCADE | SET NULL | SET DEFAULT
        . . . . . .
);
```

dopo aver specificato la chiave esterna è possibile indicare uno o due clausole di reazione:

- `ON DELETE`, che viene attivata nel caso sia cancellata una riga dalla tabella primaria
- `ON UPDATE`, che viene attivata nel caso sia modificato il valore della chiave primaria in una riga della tabella primaria

Per ciascuna delle due clausole è possibile scegliere uno tra tre possibili eventi:

- `NO ACTION`, significa che il comando è vietato e quindi la cancellazione o la modifica nella tabella primaria non deve avere effetti. È l'evento di default.
- `CASCADE`, significa che le righe della tabella secondaria subiscono la stessa sorte di quelle della tabella primaria (ovvero sono a loro volta cancellate o modificate)
- `SET NULL`, significa che nel campo chiave esterna delle righe correlate si impone il valore nullo. Questa opzione è ammissibile solo se la chiave esterna non sia obbligatoria (not null), altrimenti equivale a Non ACTION.
- `SET DEFAULT`, significa che nel campo chiave esterna delle righe correlate si impone il valore di base, indicato dalla CREATE TABLE.

Esempio

Esempio 15) Per ogni campo è possibile imporre un vincolo di chiave esterna e su ogni vincolo è possibile imporre la gestione delle violazioni:

```
CREATE TABLE VERIFICHE (
        ID_Studente INT NOT NULL,
        ID_Materia  INT DEFAULT 0,
        Data DATE NOT NULL,
        CONSTRAINT [FK_Studente]
        FOREIGN KEY (ID_Studente) REFERENCES Studenti (ID_Studente)
                ON DELETE NO ACTION
                ON UPDATE CASCADE
          ,
        CONSTRAINT [FK_Materia]
        FOREIGN KEY (ID_Materia) REFERENCES Discipline (ID_Materia)
                ON DELETE SET DEFAULT
                ON UPDATE SET NULL
);
```

Questo esempio impone che:

- nel caso si tenti di **cancellare** uno studente ma siano presenti delle verifiche in questa tabella, allora viene impedita la cancellazione dello studente;
- nel caso si tenti di **modificare** la PK di uno studente (tabella studenti) e siano presenti delle verifiche, allora queste ultime modificano la chiave esterna ed assumono la nuova indicata per lo studente;
- nel caso si tenti di **cancellare** una materia ma siano presenti delle rispettive verifiche, allora si sostituisce la materia della verifica con quella di codice 0 (ipotizzando che la materia di codice 0 esista, es. 'Cinese');
- nel caso si tenti di **modificare** il codice della materia e siano presenti delle verifiche, allora queste ultime modificano la chiave esterna col valore nullo.

TABELLE E CONTROLLI SUI CAMPI

CONTROLLI (CHECK)

Il vincolo **CHECK** serve per compiere un controllo come la verifica se il valore è uguale ad un certo valore, oppure è compreso in un intervallo o in un elenco.

SQL

Sintassi generale di creazione tabella con controllo (**CHECK**):

```
CREATE TABLE NomeTabella (
       . . . . . . . . . . ,
       [Campo1]  Tipo Vincolo,
       [Campo2]  Tipo Vincolo,
       [Campo3]  Tipo Vincolo,
       CONSTRAINT [Nome_Vincolo_1]
              CHECK ( condizione)
)
```

La condizione è una espressione booleana che può coinvolgere uno o più campi della tabella.

Esempio

Esempio 16) Nella seguente tabella si impongono dei vincoli sui campi detti **CHECK** (controlli) anche se per brevità si sono omessi i vincoli di chiave primaria, e univocità che possono essere affiancati.

```
CREATE TABLE [dbo].[Aule](
       [ID_Aula] [int] IDENTITY(1,1) NOT NULL,
       [Corpo]  [char](1) NOT NULL,
       [Piano]  [char](1) NOT NULL,
       [Numero] [char](2) NOT NULL,
       [Mq]  [decimal](4, 2) NULL,
       CONSTRAINT [CK_Corpo]
              CHECK (([Corpo]='A' OR [Corpo]='B' OR [Corpo]='C' OR [Corpo]='D')),
       CONSTRAINT [CK_Piano]
              CHECK (([Piano]='T' OR [Piano]='1' OR [Piano]='2' OR [Piano]='3')),
       CONSTRAINT [CK_Numero]
              CHECK (([Piano]>0 AND [Piano]<100))
)
GO
```

Esempio

Esempio 17) I vincoli **CHECK** possono anche essere imposti dopo la creazione della tabella. Il seguente codice di esempio costruisce un vincolo con un suo identificatore e poi lo impone sulla tabella:

```
ALTER TABLE [dbo].[Aule]
  WITH CHECK ADD  CONSTRAINT [CK_Corpo]
  CHECK  (([Corpo]='A' OR [Corpo]='B' OR [Corpo]='C' OR [Corpo]='D'))
GO

ALTER TABLE [dbo].[Aule]
  CHECK CONSTRAINT [CK_Corpo]
GO
```

CREATE INDEX

Un indice è un supporto per compiere ricerche più veloci su una tabella ed eventualmente effettuare controlli associati. La sintassi generale è la seguente:

SQL

Sintassi generale di creazione di indice (**INDEX**):

```
CREATE UNIQUE INDEX Nome-Indice
ON Nome-Tabella (campo1 ASC/DESC , campo2 ASC/DESC , ...)
```

L'indice è associato a una tabella e coinvolge uno o più campi della tabella.

Con le seguenti regole:

- La parola **UNIQUE** è opzionale. Se è indicata allora i valori dei campi non possono essere duplicati, altrimenti (se omessa) è possibile duplicarli. Si noti che in caso di molti campi allora la duplicazione si intende per la coppia o terna o n-pla.
- Il nome dell'indice è necessario perché serve per identificare l'indice nel database. Si deve scegliere un nome distinto da altri oggetti del database.

- L'indice è associato ad una sola tabella. Alla stessa tabella è possibile associare molti indici diversi. I vincoli CONSTRAINT della creazione di tabella creano automaticamente degli indici.
- L'indice può essere creato su un solo nome di campo oppure su molti nomi di campo. In un caso si valuta il singolo valore, nell'altro si crea un indice su coppie, triple, ennuple. Ogni campo dell'indice è ordinato in modo crescente (ASC) o decrescente (DESC).

Esempio

Esempio 18) Indice su un campo Il seguente indice è utilizzato per definire una cosiddetta chiave secondaria, un campo o un sottoinsieme di campi su cui effettuare frequenti ricerche:

```
CREATE INDEX NDX_Materie
ON Materie (Materia ASC)
GO
```

- Questo indice permette di effettuare ricerche più veloci sul campo Materia della tabella Materie

Esempio

Esempio 19) Indice su più campi Anche il seguente indice è utilizzato per effettuare frequenti ricerche:

```
CREATE INDEX Studenti_NDX
ON Studenti (Cognome ASC , Nome ASC)
GO
```

- Questo indice permette di effettuare ricerche più veloci sui campi cognome e nome degli alunni, come in effetti accade spesso

Esempio

Esempio 20) Indice su più campi Anche il seguente indice è utilizzato per effettuare frequenti ricerche:

```
CREATE UNIQUE INDEX Verifiche_NDX
ON Interrogazioni (ID_Alunno, ID_Materia, ID_Data)
GO
```

- Questo indice permette di evitare che un alunno sia interrogato due volte nella stessa materia nello stesso giorno. Poiché non è specificato l'ordinamento è sottinteso che sia crescente (ASCendente).

OPZIONI SQL

In alcuni dialetti SQL è possibile aggiungere all'indice alcuni vincoli per mezzo della clausola WITH, secondo la seguente sintassi:

SQL

Sintassi generale di creazione di indice (**INDEX**):

```
CREATE UNIQUE INDEX Nome-Indice
       ON Nome-Tabella (campo1 ASC/DESC , campo2 campo1 ASC/DESC , ...)
       WITH
       PRIMARY KEY || DISALLOW NULL || IGNORE NULL
```

- La parola chiave **WITH** può essere omessa con tutto quello che segue. Se specificata serve per impostare un vincolo sull'indice. I vincoli possibili sono tre ma se ne deve scegliere solo uno.
- **PRIMARY KEY** vieta l'esistenza di valori nulli nei campi della tabella e sottintende l'opzione **UNIQUE**
- **DISALLOW NULL** vieta i valori nulli (può essere usato senza **UNIQUE**) e rende il campo della tabella obbligatorio
- **IGNORE NULL** non vieta i valori nulli, e permette di costruire un indice generico

Vediamo alcuni esempi:

Esempio

Esempio 21) Indice di chiave primaria Il seguente indice crea una chiave primaria:

```
CREATE INDEX Studenti_Chiave_Primaria
       ON Studenti (ID_Studente)
       WITH
       PRIMARY KEY
GO
```

- Questo indice impone che il campo **ID_Studente** della tabella **Studenti** diventi una chiave primaria. Solleva errore se la tabella ha già una chiave primaria.

Esempio

Esempio 22) Indice di chiave alternativa Il seguente indice crea una chiave alternativa:

```
CREATE UNIQUE INDEX Studenti_Chiave_Alternativa
       ON Studenti (Matricola)
       WITH
       DISALLOW NULL
GO
```

- Questo indice impone che il campo Matricola della tabella Studenti diventi una un campo obbligatorio e univoco; in effetti è come una chiave primaria, anche se non verrà usato per i vincoli FK. Tuttavia poiché l'indice impone che ogni studente debba avere una Matricola e che sia univoca per ciascuno di fatto permette di individuare con certezza qualsiasi studente della tabella.

Esempio

Esempio 23) Indice generico Il seguente indice è utilizzato per effettuare frequenti ricerche:

```
CREATE INDEX NDX_Ricerche_Studenti
       ON Studenti (Città ASC , Indirizzo ASC)
       WITH
       IGNORE NULL
GO
```

- Questo indice ammette valori nulli per i campi Città e Indirizzo, quindi ammette Studenti senza Città e/o senza un indirizzo. Tuttavia permette di ricercare più velocemente i Studenti mediante Città e Indirizzo.

ELIMINARE UN INDICE

In SQL è possibile eliminare l'indice dal database quando non serve più. Il comando ha seguente sintassi:

SQL

Sintassi generale di eliminazione di indice (**INDEX**):

```
DROP INDEX Nome-Indice
```

- L'indice con identificatore Nome-Indice viene eliminato dal database. Eventuali vincoli imposti dall'indice saranno rimossi e rilasciati.

Vediamo alcuni esempi:

Esempio

Esempio 24) Indice generico Il seguente indice è utilizzato per effettuare frequenti ricerche:

```
DROP INDEX NDX_Ricerche_Studenti
GO
```

- Elimina l'indice NDX_Ricerche_Studenti dal database

CREATE VIEW

Una Vista è una tabella dinamica analoga ad una query ma che costituisce un elemento autonomo con propri diritti, protezioni e vantaggi.

In effetti una vista è calcolata attraverso l'esecuzione di una query ed il risultato costituisce però una vera e propria tabella.

La sintassi generale della vista è la seguente:

SQL

Sintassi generale di creazione di una vista (**VIEW**):

```
CREATE VIEW NomeVista (col1 , col2, ...) AS
(
       SELECT ...
);
```

- dove il numero di colonne della vista dovrebbe coincidere col numero di colonne della query.

Vediamo alcuni esempi:

Esempio

Esempio 25) Vista generica - Il seguente indice è utilizzato per effettuare frequenti ricerche:

```
CREATE VIEW NomeVista (Cognome, Nome, Età, Sesso) AS
(
       SELECT Cognome, Nome, Età, Sesso
       FROM Studenti
       WHERE Età >= 18;
); GO
```

- crea una vista che mostra gli studenti maggiorenni

PERCHÉ USARE UNA VISTA?

Il primo motivo per creare una vista è per mettere a disposizione di determinati utenti solo alcuni dati e non una intera tabella di dati. Per esempio un docente potrebbe avere accesso ai dati dei soli studenti delle classi in cui insegna ma non a quelli dell'intera scuola. In effetti alla vista possono essere associati dei diritti che permettono o impediscono l'accesso a determinati utenti o gruppi di utenti. In tali dati inoltre è possibile effettuare anche operazioni (inserimenti, modifiche e cancellazioni) purché in particolari condizioni e disponendo dei necessari diritti. Per esempio le query di Access sono in effetti delle viste. Altro motivo è che la vista può essere impiegata in una clausola FROM di una ulteriore query, come se fosse una tabella a tutti gli effetti. Questo è utile se il linguaggio non ammette subquery nella clausola FROM.

LINGUAGGIO T-SQL E DICHIARAZIONI DI VARIABILI

T-SQL VARIABILI

LINK

Fonte (informazione tratta dal seguente sito) :
http://msdn.microsoft.com/it-it/library/ms188927.aspx

Il linguaggio T-SQL dispone di variabili locali (che si possono usare sia in modo batch o all'interno di stored procedure in cui vengono dichiarate). Le variabili hanno tutte obbligatoriamente un tipo associato e devono essere dichiarate all'inizio del batch o della procedura in cui sono impiegate. Le variabili vengono dichiarate nel corpo di un batch o di una routine tramite l'istruzione DECLARE e i relativi valori vengono assegnati tramite un'istruzione SET o SELECT. È possibile dichiarare variabili di cursore con questa istruzione e utilizzarle insieme ad altre istruzioni correlate ai cursori. Dopo la dichiarazione, tutte le variabili vengono inizializzate con valore NULL, a meno che non venga fornito un valore nella dichiarazione.

DICHIARAZIONE DI VARIABILE

SQL

Sintassi della dichiarazione di una variabile T-SQL:

```
DECLARE  @nomevar  tipo
      [, @nomevar2    tipo2 [, …]]
```

Il comando DECLARE permette di dichiarare molte variabili insieme, separandole da virgole.

Ogni variabile deve essere dichiarata col proprio tipo.

Vediamo un esempio:

Esempio

Esempio di dichiarazione di variabile:

```
DECLARE       @voto  int ,
```

con cui si dichiarano tre variabili di tre diversi tipi.

Esempio

Esempio di dichiarazione di variabile:

```
DECLARE       @media  decimal(5,2) ,
              @conta  integer ,
              @nome   char(50)
```

con cui si dichiarano tre variabili di tre diversi tipi.

ASSEGNAZIONE DI VARIABILE

L'assegnazione di valori alle variabili può avvenire in due modi diversi.

Un primo modo consente una assegnazione di espressioni semplici, come nei classici comandi imperativi.

SQL

Sintassi generale di assegnazione di una espressione ad una variabile:

```
SET   @nomevar = valore
```

La variabile ottiene il valore specificato nella espressione di destra

Un comando SET può assegnare una sola variabile. Ciascun comando termina con un «a capo» senza punti e virgola né altri segni di fine comando.

Vediamo un esempio:

Esempio

Esempio di assegnazione di variabile:

```
SET      @media  = 3.14
```

con cui si assegna il valore 3,14 alla variabile @media

Esempio

Esempio di assegnazione di variabile:

```
SET      @conta  = 17
```

con cui si assegna il valore 17 alla variabile @conta

Esempio

Esempio di assegnazione di variabile:

```
SET      @nome   = 'Vico Lando'
```

con cui si assegna il valore « Vico Lando » alla variabile @nome

Un secondo modo consente una assegnazione del risultato di una query SQL.

SQL

Sintassi generale di assegnazione di una query ad una variabile:

```
SELECT @nomevar = valore [, ….]
FROM   Tabelle
-- altre clausole della select
```

La variabile ottiene il valore calcolato dalla query in una delle proiezioni della SELECT.

Una singola SELECT può assegnare molti valori ad altrettante variabili, separando con virgole le assegnazioni come avviene per la separazione dei campi di una ordinaria SELECT.

Esempio

Esempio di assegnazione di variabile:

```
SELECT   @conta  = COUNT(*), @media = AVG(Età)
FROM     Studenti
WHERE    Cognome = Nome ;
```

con cui si assegnano due valori a altrettante variabili dichiarate in precedenza

Esempio

Esempio di assegnazione di variabile:

```
SELECT   @media  = AVG(salario), @contaimpiegati  = COUNT(*)
FROM Impiegati ;
```

con cui si assegnano due valori a altrettante variabili dichiarate in precedenza

Esempio

Esempio di assegnazione di variabile:

```
SELECT   @nome   = (Cognome + " " + Nome)
FROM Studenti
WHERE Matricola = "110893" ;
```

con cui si assegna un valore alla variabile dichiarata in precedenza

L'assegnazione tramite `SELECT` deve essere opportunamente costruita in modo da garantire che qualsiasi esecuzione della `SELECT` produca esattamente un valore per ciascuna variabile. Se una `SELECT` produce più valori, la variabile riceve l'**ultimo** valore elaborato.

Esempio

Esempio di assegnazione di variabile:

```
SELECT  @nome  = (Cognome + " " + Nome)
FROM Studenti
WHERE Età > 17 ;
```

È una query di assegnazione pericolosa poiché non è possibile stabilire in anticipo quale valore sarà assegnato alla variabile @nome. La variabile comunque riceve l'ultimo valore dell'elenco.

Per arginare i difetti di una SELECT che rende molti valori potrebbe essere utile utilizzare un ordinamento e la scelta del solo primo elemento con l'operazione TOP 1.

Esempio

Esempio di assegnazione di variabile:

```
SELECT  @nome  = (Cognome + " " + Nome) TOP 1
FROM Studenti
WHERE Età > 17
ORDER BY Cognome, Nome;
```

Corregge la precedente query ordinando i valori e scegliendo il primo dell'elenco.

Se una SELECT non produce alcun valore, allora la variabile mantiene il valore assunto in precedenza.

Esempio

Esempio di assegnazione di variabile:

```
SELECT  @nome  = "Apollo"
FROM    Studenti
WHERE   1 = 0;
```

È una query di assegnazione inutile, poiché la condizione è sempre falsa e la variabile @nome non riceverà il valore "Apollo". La variabile quindi conserva il valore che aveva in precedenza.

Anche all'interno di un comando UPDATE è possibile assegnare valori a variabili:

SQL

Sintassi generale di assegnazione ad una variabile con un comando **UPDATE**:

```
UPDATE  Tabella
SET    @variabile = ( Campo = @variabile espressione)
```

La variabile ottiene il valore calcolato dal comando **UPDATE**.

Vediamo un esempio:

Esempio

Esempio di assegnazione di variabile:

```
DECLARE @conta int
SET     @conta = 1

UPDATE  Tabella
SET     @conta = Campo = @conta * 2
```

La variabile inizia da 1. La **UPDATE** esplora i record della Tabella e ad ogni record assegna il doppio del contatore; questo valore viene anche assegnato alla variabile che raddoppia ad ogni giro. In pratica assegna al Campo i valori 2, 4, 8, 16, 32, ecc …

T-SQL VARIABILI GLOBALI

Fonte (informazione tratta dal seguente sito) :

http://msdn.microsoft.com/it-it/library/ms187786.aspx

Il linguaggio T-SQL non dispone di variabili globali ma possiede comunque delle funzioni di sistema connotate da una doppia @@ che precede il loro nome. Le funzioni di sistema riportate di seguito eseguono operazioni e restituiscono informazioni su valori, oggetti e impostazioni in SQL Server. Tra le funzioni di sistema possiamo ricordare: **@@ERROR**, **@@IDENTITY**, **@@PACK_RECEIVED**, **@@ROWCOUNT**, **@@TRANCOUNT**.

T-SQL FUNZIONI PREDEFINITE

LINK

Fonte: http://msdn.microsoft.com/it-it/library/ms174318.aspx

SQL Server include numerose funzioni predefinite e consente inoltre di creare funzioni definite dall'utente. In questa pagina sono elencate le categorie delle funzioni predefinite.

Funzione	Descrizione
Funzioni per i set di righe	*Restituiscono un oggetto utilizzabile come i riferimenti a tabelle in un'istruzione SQL.*
Funzioni di aggregazione	*Vengono applicate su una raccolta di valori e restituiscono un singolo valore di riepilogo.*
Funzioni di rango	*Restituiscono un valore di rango per ogni riga di una partizione.*
Funzioni scalari	*Vengono applicate a un singolo valore e restituiscono un singolo valore. È possibile utilizzare le funzioni scalari in tutte le posizioni in cui sono consentite espressioni.*

FUNZIONI SCALARI

Categoria di funzioni	Descrizione
Funzioni di configurazione	*Restituiscono informazioni sulla configurazione corrente.*
Funzioni di conversione	*Supportano l'esecuzione del cast e la conversione del tipo di dati.*
Funzioni per i cursori	*Restituiscono informazioni sui cursori.*
Funzioni e tipi di dati di data e ora	*Eseguono operazioni su valori di input di data e ora e restituiscono valori stringa, numerici o di data e ora.*
Funzioni logiche	*Eseguono operazioni logiche.*
Funzioni matematiche	*Eseguono calcoli in base ai valori di input specificati come parametri per le funzioni e restituiscono valori numerici.*
Funzioni per i metadati	*Restituiscono informazioni sul database e sugli oggetti di database.*
Funzioni di sicurezza	*Restituiscono informazioni sugli utenti e sui ruoli.*
Funzioni per i valori stringa	*Eseguono operazioni su valori di input di tipo stringa (char o varchar) e restituiscono un valore stringa o numerico.*
Funzioni di sistema	*Eseguono operazioni e restituiscono informazioni su valori, oggetti e impostazioni in un'istanza di SQL Server.*
Funzioni statistiche di sistema	*Restituiscono informazioni statistiche sul sistema.*
Funzioni per i valori text e image	*Eseguono operazioni su valori di input o colonne di testo o immagini e restituiscono informazioni sul valore.*

Le funzioni predefinite scalari sono molto utili per gestire particolari tipi di dato.

In particolare possiamo soffermarci sulle funzioni per i tipi data e ora, utili per confronti, differenze e orari di sistema. Tutti i valori di data e ora di sistema derivano dal sistema operativo del computer in cui è in esecuzione l'istanza di SQL Server.

Le funzioni che ottengono valori della data e ora di sistema sono le seguenti:

Funzione	Sintassi	Valore restituito	Tipo reso
SYSDATETIME	SYSDATETIME ()	Restituisce il valore **datetime2**(7) che contiene la data e l'ora del computer in cui è in esecuzione l'istanza di SQL Server. La differenza di fuso orario non è inclusa.	datetime2(7)
SYSDATETIMEOFFSET	SYSDATETIMEOFFSET ( )	Restituisce il valore **datetimeoffset**(7) che contiene la data e l'ora del computer in cui è in esecuzione l'istanza di SQL Server. La differenza di fuso orario è inclusa.	datetimeoffset (7)
SYSUTCDATETIME	SYSUTCDATETIME ( )	Restituisce il valore **datetime2**(7) che contiene la data e l'ora del computer in cui è in esecuzione l'istanza di SQL Server. La data e l'ora vengono restituite in formato ora UTC (Coordinated Universal Time).	datetime2(7)
CURRENT_TIMESTAMP	CURRENT_TIMESTAMP	Restituisce il valore **datetime** che contiene la data e l'ora del computer in cui è in esecuzione l'istanza di SQL Server. La differenza di fuso orario non è inclusa.	datetime
GETDATE	GETDATE ( )	Restituisce il valore **datetime** che contiene la data e l'ora del computer in cui è in esecuzione l'istanza di SQL Server. La differenza di fuso orario non è inclusa.	datetime
GETUTCDATE	GETUTCDATE ( )	Restituisce il valore **datetime** che contiene la data e l'ora del computer in cui è in esecuzione l'istanza di SQL Server. La data e l'ora vengono restituite in formato ora UTC (Coordinated Universal Time).	datetime
DATENAME	DATENAME (*datepart* ,*date*)	Restituisce una stringa di caratteri che rappresenta la *datepart* della data specificata.	nvarchar
DATEPART	DATEPART (*datepart* ,*date*)	Restituisce un valore **intero** che rappresenta il valore *datepart* dell'argomento*date* specificato.	int
DAY	DAY (*date*)	Restituisce un valore **intero** che rappresenta la parte del giorno della *date*specificata.	int
MONTH	MONTH (*date*)	Restituisce un valore **intero** che rappresenta la parte mese di un valore *date*specificato.	int
YEAR	YEAR (*date*)	Restituisce un valore **intero** che rappresenta la parte dell'anno di *date*specificata.	int
DATEFROMPARTS	DATEFROMPARTS (*year*, *month*, *day*)	Restituisce un valore di tipo **date** per l'anno, il mese e il giorno specificati.	date
DATETIME2FROMPARTS	DATETIME2FROMPARTS (*year*, *month*, *day*,*hour*, *minute*, *seconds*, *fractions*, *precision*)	Restituisce un valore **datetime2** per la data e l'ora specificate e con la precisione indicata.	datetime2 (*precision*)
DATETIMEFROMPARTS	DATETIMEFROMPARTS (*year*, *month*, *day*, *hour*,*minute*, *seconds*, *milliseconds*)	Restituisce un valore di tipo **datetime**per la data e l'ora specificate.	datetime
DATETIMEOFFSETFROMPARTS	DATETIMEOFFSETFROMPARTS (*year*, *month*,*day*, *hour*, *minute*, *seconds*, *fractions*,*hour_offset*, *minute_offset*, *precision*)	Restituisce un valore **datetimeoffset**per la data e l'ora specificata e con gli offset e la precisione indicati.	datetime (*precision*)
SMALLDATETIMEFROMPARTS	SMALLDATETIMEFROMPARTS (*year*, *month*,*day*, *hour*, *minute*)	Restituisce un valore di tipo**smalldatetime** per la data e l'ora specificate.	smalldatetime
TIMEFROMPARTS	TIMEFROMPARTS (*hour*, *minute*, *seconds* ,*fractions*, *precision*)	Restituisce un valore **time** per l'ora specificate e con la precisione indicata.	time (*precision*)
DATEDIFF	DATEDIFF (*datepart* ,*startdate* , *enddate*)	Restituisce il numero di limiti di *datepart* della data e ora che si sovrappongono tra due date specificate.	int

Non è intenzione di questa dispensa analizzare in dettaglio l'intero panorama delle funzioni predefinite disponibili in SQL Server, per cui si rimanda alle pagine della guida in linea.

COMANDI IMPERATIVI IN LINGUAGGIO T-SQL

LINK

Fonte: http://msdn.microsoft.com/it-it/library/ms174290.aspx

Il linguaggio T-SQL dispone di comandi imperativi che permettono di dirigere il flusso di elaborazione per costruire programmi batch, funzioni o stored procedure. I comandi consentono di combinare i comandi tipici di SQL con quelli imperativi consueti ottenendo una notevole potenza elaborativa.

COMANDO SEQUENZA (BLOCCO DI ISTRUZIONI)

Il comando sequenza permette di racchiudere insieme gruppi di comandi per costruire un blocco unitario, come il `BEGIN … END` del linguaggio Pascal e il blocco `{ … }` dei linguaggi C-like (come C++, Java e C#).

SQL

Sintassi generale del comando sequenza:

```
BEGIN
 . . .
Elenco comandi
 . . .
END
```

Vediamo un esempio:

Esempio

```
BEGIN
        DECLARE @conta int

        SET @conta = 1

        UPDATE Tabella
             SET @conta = Campo = @conta * 2
END
```

COMANDO DECISIONALE (ESECUZIONE CONDIZIONALE)

Il comando decisionale permette di decidere se eseguire o meno un comando, come lo IF THEN / ELSE del linguaggio Pascal e lo if () dei linguaggi C-like (come C++, Java e C#).

SQL

Sintassi generale del comando:

```
IF ( condizione )
  Elenco comandi
ELSE
  Elenco comandi
END
```

La clausola `ELSE` è opzionale; il comando permette di far eseguire solo comandi sotto condizione.

Vediamo un esempio:

Esempio

Il seguente esempio valuta se inserire un record nuovo o modificare un record esistente:

```
DECLARE @quanti int
SET     @quanti = 0
SELECT  @quanti  = COUNT(*)
        FROM Versamenti
        WHERE IDsocio = @IDS AND IDprodotto = @IDP AND Data = "01/04/2014";
IF ( @quanti > 0 )
        UPDATE Versamenti
             SET Quantità = Quantità + @quantità
             WHERE IDsocio = @IDS AND IDprodotto = @IDP AND Data = "01/04/2014";
ELSE
        INSERT INTO Versamenti
             VALUES ("01/04/2014", @IDS, @IDP, @quantità)
```

Se il versamento esiste lo modifica, altrimenti ne inserisce uno nuovo.

COMANDO ITERATIVO (ESECUZIONE CICLICA)

LINK

Fonte: http://msdn.microsoft.com/it-it/library/ms178642.aspx

Il comando iterativo permette di ripetere l'esecuzione di comandi in un ciclo, come il **WHILE** del linguaggio Pascal e lo **while ()** dei linguaggi C-like (come C++, Java e C#).

SQL

Sintassi generale del comando:

```
WHILE (Condizione)
      { Comando | Sequenza | BREAK | CONTINUE }
```

- **CONDIZIONE** è l'espressione che restituisce TRUE o FALSE. Se l'espressione booleana include un'istruzione SELECT, tale istruzione deve essere racchiusa tra parentesi.
- **COMANDO** o Sequenza è una qualsiasi istruzione o un blocco di istruzioni Transact-SQL valide definite con un blocco di istruzioni. Per definire un blocco di istruzioni, utilizzare le parole chiave per il controllo di flusso BEGIN ed END.
- La parola chiave **BREAK** consente di uscire dal ciclo WHILE più interno. Vengono eseguite le istruzioni che si trovano dopo la parola chiave END, che segna la fine del ciclo.
- La parola chiave **CONTINUE** consente di ricominciare immediatamente il riavvio del ciclo WHILE (dalla condizione), ignorando tutte le istruzioni che seguono la parola chiave CONTINUE.

Vediamo un esempio:

Esempio

Il seguente esempio valuta se inserire un record nuovo o modificare un record esistente:

```
WHILE (SELECT AVG(ListPrice) FROM Production.Product) < $300
BEGIN
   UPDATE Production.Product
      SET ListPrice = ListPrice * 2
   SELECT MAX(ListPrice) FROM Production.Product
   IF (SELECT MAX(ListPrice) FROM Production.Product) > $500
      BREAK
   ELSE
      CONTINUE
END
PRINT 'Per il negozio è troppo costoso per rincarare ancora';
```

Nell'esempio precedente, se il prezzo medio di listino di un prodotto è minore di $300, il ciclo WHILE raddoppia i prezzi e quindi seleziona il prezzo massimo. Se il prezzo massimo è minore o uguale a $500, il ciclo WHILE viene riavviato e il prezzo viene nuovamente raddoppiato. Questo ciclo continua a raddoppiare i prezzi fino a quando il prezzo massimo non supera $500, quindi il ciclo WHILE viene terminato e viene stampato un messaggio.

STORED PROCEDURE IN T-SQL

LINK

Fonte: http://msdn.microsoft.com/it-it/library/ms187926.aspx

SINTASSI GENERALE

Il comando **CREATE PROCEDURE** consente di creare una **STORED PROCEDURE** Transact-SQL o CLR (Common Language Runtime) in SQL Server. Le **STORED PROCEDURE** sono simili alle procedure di altri linguaggi di programmazione in quanto sono in grado di:

- Accettare parametri di input e restituire più valori sotto forma di parametri di output alla procedura o al batch che esegue la chiamata.
- Includere istruzioni di programmazione che eseguono le operazioni nel database, tra cui la chiamata di altre procedure.
- Restituire un valore di stato a una procedura o a un batch che esegue la chiamata per indicare l'esito positivo o negativo (e il motivo dell'esito negativo).

Questa istruzione consente di creare una stored procedure permanente nel database corrente o una stored procedure temporanea nel database tempdb.

La sintassi della **CREATE PROCEDURE** è piuttosto ampia e complessa e lo studio di alcune sue parti esula dallo scopo di questa dispensa; pertanto procederemo ad un'analisi sommaria e parziale della sua struttura.

SQL

Sintassi generale del comando **CREATE PROCEDURE**:

```
--SQL Server Stored Procedure Syntax
CREATE        { PROC | PROCEDURE } [schema_name.] procedure_name
       [ { @parameter [ type_schema_name. ] data_type }
         [ VARYING ] [ = default ] [ OUT | OUTPUT | [READONLY]
    ] [ ,...n ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS { [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }
[;]
```

La prosecuzione della sintassi non è sviluppata.

La sintassi della **CREATE PROCEDURE** prevede le seguenti parti:

- L'intestazione inizia con **CREATE PROCEDURE** oppure con **CREATE PROC** seguito dal nome della procedura; il nome della procedura può essere preceduto dal nome dello schema (e da un punto);
- L'intestazione è seguita dall'elenco dei parametri, i cui nomi devono iniziare con la @ (l'elenco può essere racchiuso tra parentesi tonde, ma non è obbligatorio)
- Le opzioni **WITH** e **FOR REPLICATION** sono facoltative e per il momento le possiamo trascurare
- La parola chiave **AS** determina l'inizio del corpo della procedura; sebbene non obbligatorio è preferibile racchiudere il corpo con un blocco **BEGIN END**

Vediamo alcuni esempi:

Esempio

Esempio copiato dal sito Microsoft:

```
CREATE PROCEDURE usp_add_kitchen
      @dept_id int, @kitchen_count int NOT NULL
WITH EXECUTE AS OWNER, SCHEMABINDING, NATIVE_COMPILATION
AS BEGIN
  ATOMIC WITH
      (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'us_english')
  UPDATE dbo.Departments
  SET kitchen_count = ISNULL(kitchen_count, 0) + @kitchen_count
  WHERE id = @dept_id
END;
GO
```

Questa procedura è tratta direttamente dal sito Microsoft riportato sopra.

Esempio

Esempio riferito al database Biblioteca scolastica:

```
CREATE PROCEDURE Inserimento_Prestito
      @IDsocio int NOT NULL, @IDtesto int NOT NULL
AS BEGIN
  DECLARE @oggi
  DECLARE @scadenza
  @oggi = GETDATE ( )
  @scadenza = 90 + GETDATE ( )
  IF (@IDtesto NOT IN SELECT @IDtesto FROM Prestiti_Correnti)
    INSERT INTO Prestiti_Correnti
    VALUES (@IDsocio, @IDtesto, @oggi, @scadenza)
END;
GO
```

Questa procedura è finalizzata all'inserimento di un prestito di un libro. I due parametri servono per ricevere in chiamata le chiavi primarie rispettivamente dell'iscritto alla biblioteca e del testo richiesto. Le due variabili locali sono utilizzate per impostare la data del prestito e la scadenza.

PARAMETRI

I parametri di una STORED PROCEDURE sono analoghi ai parametri di una funzione di un linguaggio imperativo. I nomi dei parametri devono iniziare con @ e sono seguiti dal tipo e dalle opzioni che sono dei vincoli. Il nome del parametro deve essere conforme alle regole per gli identificatori. Poiché i parametri sono locali rispetto alla procedura, è possibile utilizzare gli stessi nomi di parametro in altre procedure.
È possibile dichiarare fino a 2.100 parametri per una procedura. Il valore di ogni parametro dichiarato deve essere specificato dall'utente quando viene chiamata la procedura, a meno che non venga indicato un valore predefinito per il parametro oppure il valore venga impostato in modo da corrispondere a quello di un altro parametro.
Se una procedura contiene parametri con valori di tabella e nella chiamata il parametro non è presente, viene passata una tabella vuota. I parametri possono rappresentare solo espressioni costanti, non nomi di tabella, nomi di colonna o nomi di altri oggetti di database.
Tutti i tipi di dati Transact-SQL possono essere utilizzati come parametri.
Per creare parametri con valori di tabella è possibile utilizzare il tipo di tabella definito dall'utente. I parametri con valori di tabella possono essere solo parametri di input e devono essere associati al vincolo READONLY.
I parametri possono avere alcuni vincoli o opzioni riportate di seguito:

OPZIONE	SIGNIFICATO
DEFAULT	Valore predefinito per un parametro. Se per un parametro viene definito un valore predefinito, la procedura può essere eseguita senza specificare un valore per tale parametro. Il valore predefinito deve essere una costante oppure NULL. Il formato del valore della costante può essere un carattere jolly; in questo modo sarà possibile utilizzare la parola chiave LIKE quando si passa il parametro nella procedura.
OUT | OUTPUT	Indica che si tratta di un parametro di output. Utilizzare i parametri OUTPUT per restituire i valori al chiamante della procedura. I parametri text, ntext e image non posso essere utilizzati come parametri OUTPUT, a meno che non si tratti di una procedura CLR. Un tipo di dati con valori di tabella non può essere specificato come parametro di output di una procedura.
READONLY	Indica che il parametro non può essere aggiornato o modificato all'interno del corpo della procedura. Se si tratta di un tipo di parametro con valori di tabella, è necessario specificare la parola chiave READONLY.
NULL | NOT NULL	Determina se i valori Null sono supportati in un parametro. Il valore predefinito è NULL.

Esempi

Esempio

Esempio generico:

```
CREATE PROCEDURE Prova
      @data DATE DEFAULT '01/01/2014' , --se non si indica vale 01/01/2014
      @rende INT OUT , --indica un parametro per risultati (in uscita)
      @fisso INT READONLY , --indica un parametro che non può essere modificato
      @obbligo INT NOT NULL , --indica un parametro obbligatorio
      @vuoto INT NULL , --indica un parametro che ammette il valore NULL
AS BEGIN
 . . .
END;
GO
```

Commento:

Per approfondimenti si rimanda al manuale di uso di SQL Server Management Studio.

INVOCAZIONE

L'invocazione di una **STORED PROCEDURE** può avvenire dall'interno mediante il comando **EXECUTE** (contratto anche in **EXEC**) oppure dall'esterno impiegando una chiamata da un linguaggio che possa interagire col DBMS.

Esempio

```
EXECUTE Bibliox.Inserimento_Prestito @IDsocio = 123, @IDtesto = 904;
EXECUTE Kitchen.usp_add_kitchen  @dept_id = 502, @kitchen_count = 706;
EXECUTE Scuola.Prova @data = NULL, @fisso = 13, @obbligo = 17, @vuoto = NULL
```

L'invocazione può avvenire specificando i parametri oppure affidandosi alla loro posizione; le seguenti invocazioni sono tutte equivalenti:

Esempio

```
EXECUTE Scuola.Inserimento_Studenti @Cognome = N'Carri', @Nome = N'Guido';
EXECUTE Scuola.Inserimento_Studenti N'Carri', N'Guido';
EXEC Scuola.Inserimento_Studenti N'Carri', N'Guido';;
```

FUNCTION IN T-SQL

SINTASSI GENERALE

LINK

Fonte: http://msdn.microsoft.com/it-it/library/ms186755.aspx

TSQL permette di creare una funzione definita dall'utente in SQL Server. Una funzione definita dall'utente è una routine Transact-SQL (o CLR che tralasciamo) tramite cui vengono accettati parametri, viene effettuata un'azione, ad esempio un calcolo complesso, e viene restituito il risultato di tale azione sotto forma di valore. Il valore restituito può essere un valore scalare (singolo) o una tabella. Utilizzare questa istruzione per creare una routine riutilizzabile che può essere utilizzata in queste modalità:

- All'interno di istruzioni Transact-SQL, come nella clausola **FROM** della **SELECT**.
- Per **PARAMETRIZZARE UNA VISTA** o per migliorare le funzionalità di una vista indicizzata.
- Per sostituire una **STORED PROCEDURE**.
- Nelle applicazioni che chiamano la funzione.
- Nella definizione di un'altra funzione definita dall'utente.
- Per definire una colonna di una tabella.
- Per definire un vincolo **CHECK** su una colonna.

Gran parte della sintassi della funzione è analoga alla **STORED PROCEDURE**, per cui si rimanda alla trattazione sopra esposta e agli approfondimenti sul sito Microsoft. A differenza della procedura tuttavia la funzione rende qualcosa; la funzione può rendere un valore scalare (un qualsiasi tipo predefinito di SQL oppure un tipo definito dall'utente nello schema) oppure una tabella.

FUNZIONI SCALARI

Le funzioni scalari restituiscono un singolo valore e la relativa sintassi è la seguente:

SQL

Sintassi generale delle funzioni scalari:

```
--Transact-SQL Scalar Function Syntax
CREATE FUNCTION [ schema. ] NOME_FUNZIONE
                    ( elenco dei parametri )
RETURNS tipo_reso
AS BEGIN
      --corpo della funzione
      RETURN espressione_scalare
END
```

- Le parole chiave **CREATE FUNCTION** sono obbligatorie
- Il nome dello **SCHEMA** è opzionale
- Il **NOME** della funzione è obbligatorio e identifica univocamente l'oggetto funzione del database
- Il nome della funzione permette in seguito alla creazione sia di invocarla sia di eliminarla
- La parola chiave **RETURNS** è obbligatoria e deve essere seguito dal tipo elementare restituito
- Il corpo della funzione stesse regole delle procedure ma prevede l'uso della parola chiave **RETURN**
- Il comando return specifica il valore che la funzione restituisce, conforme al tipo previsto

Esempio

```
CREATE FUNCTION Numero_Studenti_Di_Anno
      (
            @anno char(1) = '0',
      )
RETURNS int
AS
BEGIN
      SET NOCOUNT ON;
      DECLARE @ris int;
      SELECT @ris = COUNT(*)
            FROM Studenti , Classi
            WHERE Studenti.Classe = Classi.IDClasse
                  AND = Classi.Anno = @anno;
      RETURN @ris;
END
```

FUNZIONI TABELLARI

Le funzioni tabellari restituiscono una tabella e la relativa sintassi è la seguente:

SQL

Sintassi generale delle funzioni scalari:

```
--Transact-SQL Scalar Function Syntax
CREATE FUNCTION [ schema. ] NOME_FUNZIONE
                    ( elenco dei parametri )
RETURNS TABLE
AS BEGIN
       --corpo della funzione
       RETURN comando_SELECT
END
```

- Le parole chiave **CREATE FUNCTION** sono obbligatorie
- Il nome dello **SCHEMA** è opzionale
- Il **NOME** della funzione è obbligatorio e identifica univocamente l'oggetto funzione del database
- Il nome della funzione permette in seguito alla creazione sia di invocarla sia di eliminarla
- La parola chiave **RETURNS** è obbligatoria e deve essere seguito dalla parola chiave **TABLE**
- Il corpo della funzione stesse regole delle procedure ma prevede l'uso della parola chiave **RETURN**
- Il comando **RETURN** è seguita dalla **QUERY** che la funzione restituisce

Vediamo un esempio:

Esempio

```
CREATE FUNCTION Elenco_Studenti_Raggruppati (
              @anno char(1) = '0',
       )
RETURNS TABLE
AS
BEGIN
SET NOCOUNT ON;
RETURN SELECT Classi.Sezione, Classi.Indirizzo, Studenti.Età, MAX(Età), COUNT(*) AS Num
        FROM   Studenti , Classi
        WHERE  Studenti.ID_Classe = Classi.ID_Classe
          AND  Classi.Anno = @anno
        GROUP BY Classi.Sezione, Classi.Indirizzo, Studenti.Età
        ORDER BY Classi.Sezione, Classi.Indirizzo, Studenti.Età;
END
```

TRIGGER IN T-SQL

CREATE TRIGGER

Il Trigger (letteralmente grilletto, innesco) serve per definire un meccanismo automatico sui dati. Quando una determinata operazione viene effettuata sui dati il sistema verifica se esiste un trigger associato e lo esegue.

Questo meccanismo serve per garantire che le operazioni sui dati siano eseguite correttamente, magari correggendo o completando l'operazione richiesta. Purtroppo anche questo comando non è nello standard SQL92, sebbene sia ampiamente implementato in numerosi motori per database.

La sintassi del comando è la seguente:

SQL

Sintassi generale di creazione di un trigger (**TRIGGER**):

```
CREATE TRIGGER NomeTrigger [ BEFORE | AFTER ]
       ON Tabella
       FOR DELETE | INSERT | UPDATE
       EXECUTE
       Comandi //vedi procedure
       END;
```

- le due opzioni **BEFORE** e **AFTER** sono alternative (se ne usa una e solo una) :
- **AFTER**, si usa nel caso di voler effettuare delle azioni prima che i controlli sui vincoli della tabella siano stati controllati. È utile per impostare valori di chiave primaria calcolati o per riempire dei campi lasciati vuoti prima che siano effettuati i normali controlli;
- **BEFORE**, si usa per effettuare azioni dopo che i controlli sono stati superati (e nel caso che l'operazione abbia avuto successo), magari per modificare altre tabelle collegate o per modificare alcuni valori di alcuni campi.
- mentre le opzioni **DELETE** | **INSERT** | **UPDATE** sono facoltative (se ne usa una o due o tre) ed indicano il tipo di operazione che invoca l'esecuzione dei comandi.

Vediamo alcuni esempi; supponiamo di avere il seguente schema relazionale:

Esempio

```
PERSONE (ID_Persona, CF, Cognome, Nome, Età, Sesso);
       PK ID_Persona;
       UNIQUE (CF);
CANI (ID_Cane, Nome, Razza, Età, Sesso, CodiceChip, ID_Padrone);
       PK ID_Cane;
       FK ID_Padrone REF Persone (ID_Persona);
```

Esempio

Esempio 26) Trigger generico Il seguente trigger è utilizzato per correggere situazioni erronee:

```
CREATE TRIGGER Cani_Randagi BEFORE
       ON Cani
       FOR INSERT
       EXECUTE IF (NOT EXISTS NEW.ID_Padrone)
                     NEW.ID_Padrone = NULL;
       END;
GO
```

- crea un trigger che nel caso di inserimento di un cane con padrone inesistente, lo inserisce come senza padrone (campo chiave esterna nullo) ovvero randagio!

Vediamo alcuni esempi:

Esempio

Esempio 27) Trigger generico Il seguente trigger è utilizzato per correggere situazioni erronee:

```
CREATE TRIGGER Percentuali AFTER
       ON Composizioni
       FOR INSERT EXECUTE
              IF 100 < (  SELECT SUM(Composizione.Percentuale)
                          FROM Composizione
                          WHERE NEW.ID_Prodotto = Composizione.ID_Prodotto
                          GROUP BY Composizione.ID_Prodotto
                          HAVING SUM(Composizione.Percentuale) )
                     ROLLBACK;
       END;
GO
```

- crea un trigger che dopo ogni inserimento controlla se ho superato il 100% di ingredienti

ASSERZIONI IN T-SQL

CREATE RULE

L'asserzione è un vincolo articolato che può coinvolgere una, due o più tabelle e serve per garantire che le operazioni sui dati rispettino dei vincoli complessi.

La sintassi del comando è la seguente:

SQL

Sintassi generale di creazione di un trigger (**TRIGGER**):

```
CREATE RULE Nome_Regola
CHECK (espressione);
```

- le due opzioni **BEFORE** e **AFTER** sono alternative (se ne usa una e solo una) :

dove l'espressione deve essere booleana (vera o falsa) ed assume una forma analoga alle query annidate ovvero simili alle seguenti:

- EXISTS (SELECT . . . . .)
- Valore IN (SELECT . . . . .)
- (SELECT . . .) OP (SELECT . . .)

Vediamo alcuni esempi:

Esempio

Esempio 28) Regola generica La seguente regola:

```
CREATE RULE CentoPerCento
CHECK
(NOT EXISTS (SELECT SUM(Percentuale)
                    FROM Composizione
                    GROUP BY Prodotto
                    HAVING SUM(Percentuale) > 100    )
);
GO
```

crea una regola per garantire il fatto che non esiste alcun prodotto la cui composizione è formata da oltre il 100 % di ingredienti.

ESERCIZI

ESERCIZI SULLE TABELLE

ESERCIZIO 1.

Si implementi in T-SQL il seguente schema relazionale:

```
CLASSI (ID_Classe, Anno, Sezione, Indirizzo);
        PK ID_Classe;
        UNIQUE (Anno, Sezione, Indirizzo);
STUDENTI (ID_Studente, Cognome, Nome, Età, Sesso, ID_Classe);
        PK ID_Studente;
        FK ID_Classe REF Classi (ID_Classe);
```

ESERCIZIO 2.

Si implementi in T-SQL il seguente schema relazionale:

```
PERSONE (ID_Persona, CF, Cognome, Nome, Età, Sesso);
        PK ID_Persona;
        UNIQUE (CF);
CANI (ID_Cane, Nome, Razza, Età, Sesso, CodiceChip, ID_Padrone);
        PK ID_Cane;
        FK ID_Padrone REF Persone (ID_Persona);
```

ESERCIZIO 3.

Si implementi in T-SQL il seguente schema relazionale:

```
PIZZA (ID_Pizza, Nome, Prezzo);
        PK ID_Pizza;
INGREDIENTI (ID_Ingrediente, Nome, Costo, Scorte);
        PK ID_ Ingrediente;
COMPOSIZIONI (ID_Composizione, ID_Pizza, ID_Ingrediente, Quantità);
        PK ID_Composizione;
        FK ID_Pizza REF Pizze (ID_Pizza);
        FK ID_Ingrediente REF Ingredienti (ID_Ingrediente);
```

ESERCIZIO 4.

Si implementi in T-SQL il seguente schema relazionale:

```
PIZZA (ID_Pizza, Nome, Prezzo);
        PK ID_Pizza;
        UNIQUE (Nome);
        CHECK (Prezzo > 0);
INGREDIENTI (ID_Ingrediente, Nome, Costo, Scorte);
        PK ID_ Ingrediente;
COMPOSIZIONI (ID_Composizione, ID_Pizza, ID_Ingrediente, Quantità);
        PK ID_Composizione;
        UNIQUE (ID_Pizza, ID_Ingrediente);
        CHECK (Quantità > 0);
        FK ID_Pizza REF Pizze (ID_Pizza);
        FK ID_Ingrediente REF Ingredienti (ID_Ingrediente);
```

ESERCIZIO 5.

Si implementi in T-SQL il seguente schema relazionale:

```
CLASSI (ID_Classe, Anno, Sezione, Indirizzo, ID_Rappresentante_1, ID_Rappresentante_2);
        PK ID_Classe;
        UNIQUE (Anno, Sezione, Indirizzo);
        FK ID_Rappresentante_1 REF Studenti (ID_Studente);
        FK ID_Rappresentante_2 REF Studenti (ID_Studente);
STUDENTI (ID_Studente, Cognome, Nome, Età, Sesso, ID_Classe);
        PK ID_Studente;
        FK ID_Classe REF Classi (ID_Classe);
```

ESERCIZIO 6.

Si implementi in T-SQL il seguente schema relazionale:

```
STUDENTI (ID_Studente, Cognome, Nome, Età, Sesso);
        PK ID_Studente;
DOCENTI  (ID_Docente, Cognome, Nome, Età, Sesso);
        PK ID_Docente;
MATERIE  (ID_Materia, Nome, Descrizione);
        PK ID_Materia;
        UNIQUE (Nome);
        NOT NULL (Nome);
VERIFICHE (ID_Verifica, Data, Tipo, ID_Materia, ID_Docente, ID_Studente, Voto);
        PK ID_Verifica;
        UNIQUE (Data, ID_Materia, ID_Docente, ID_Studente);
        FK ID_Materia REF Materie (ID_Materia);
        FK ID_Docente REF Docenti (ID_Docente);
        FK ID_Studente REF Studenti (ID_Studente);
        NOT NULL (Data, Tipo, ID_Materia, ID_Docente, ID_Studente, Voto);
```

ESERCIZIO 7.

Si implementi in T-SQL il seguente schema relazionale:

```
PROGETTI (ID_Progetto, Nome, Budget, ID_Capo, ID_Portavoce);
        PK ID_Classe;
        UNIQUE (Nome);
        FK ID_Capo REF Studenti (ID_Studente);
        FK ID_Portavoce REF Studenti (ID_Studente);
STUDENTI (ID_Studente, Cognome, Nome, Età, Sesso);
        PK ID_Studente;
```

ESERCIZIO 8.

Si implementi in T-SQL il seguente schema relazionale:

```
PERSONE (ID_Persona, CF, Cognome, Nome, Età, Sesso);
        PK ID_Persona;
        UNIQUE (CF);
CANI (ID_Cane, Nome, Razza, Età, Sesso, CodiceChip, ID_Padrone);
        PK ID_Cane;
        FK ID_Padrone REF Persone (ID_Persona);
ESERCIZI-SINGOLI (ID_EXS, Nome, ID_Cane);
        PK ID_EXS;
        FK ID_Cane REF Cani (ID_Cane);
ESERCIZI-COPPIA (ID_EXC, Nome, ID_Cane, ID_Accompagnatore);
        PK ID_EXC;
        FK (ID_Cane, ID_Accompagnatore) REF Cani (ID_Cane, ID_Accompagnatore);
        -- fare attenzione che la chiave esterna è una coppia --
```

ESERCIZIO 9.

Si implementi in T-SQL il seguente schema relazionale:

```
GIOCATORI (ID_Giocatore, Cognome, Nome, Età, Sesso);
        PK ID_Giocatore;
PARTITE (ID_Partita, Data, ID_Bianco, ID_Nero, Stato, ID_Vincitore);
        PK ID_Partita;
        FK ID_Bianco REF Giocatori (ID_Giocatore);
        FK ID_Nero REF Giocatori (ID_Giocatore);
        FK ID_Vincitore REF Giocatori (ID_Giocatore);
        CHECK Stato IN ('Da fare', 'In corso', 'Patta', 'Stallo', 'Vinta');
        CHECK ID_Bianco <> ID_Nero;
        CHECK (ID_Vincitore IS NULL) OR (ID_Vincitore = ID_Bianco) OR (ID_Vincitore = ID_Nero);
        CHECK ( (Stato <> 'Vinta') OR (ID_Vincitore NOT IS NULL) );
```

ESERCIZI SULLE PROCEDURE

ESERCIZIO 10.

Sul seguente schema relazionale

```
CLASSI (ID_Classe, Anno, Sezione, Indirizzo);
       PK ID_Classe;
       UNIQUE (Anno, Sezione, Indirizzo);
STUDENTI (ID_Studente, Cognome, Nome, Età, Sesso, ID_Classe);
       PK ID_Studente;
       FK ID_Classe REF Classi (ID_Classe);
```

Si implementino le seguenti procedure:

- Inclusione di uno studente in una classe (**parametri**: codice classe e codice studente)
- Esclusione di uno studente dalla classe (**parametri**: codice studente)
- Spostamento di uno studente da classe a altra classe (**parametri**: da valutare)
- Inserimento nuova classe, se esiste già imporre NULL per la sezione

ESERCIZIO 11.

Sul seguente schema relazionale

```
PERSONE (ID_Persona, CF, Cognome, Nome, Età, Sesso);
       PK ID_Persona;
       UNIQUE (CF);
CANI (ID_Cane, Nome, Razza, Età, Sesso, CodiceChip, ID_Padrone);
       PK ID_Cane;
       FK ID_Padrone REF Persone (ID_Persona);
```

Si implementino le seguenti procedure:

- Inserimento di una persona e di un cane in parallelo (**parametri**: da valutare)
- Rimozione di un cane dal padrone con padrone NULL (**parametri**: codice cane)
- Spostamento di un cane da padrone a altro padrone (**parametri**: da valutare)
- Eliminazione di tutti i cani di un certo padrone (**parametri**: codice padrone)

ESERCIZIO 12.

Sul seguente schema relazionale

```
GIOCATORI (ID_Giocatore, Cognome, Nome, Età, Sesso);
       PK ID_Giocatore;
PARTITE (ID_Partita, Data, ID_Bianco, ID_Nero, Stato, ID_Vincitore);
       PK ID_Partita;
       FK ID_Bianco REF Giocatori (ID_Giocatore);
       FK ID_Nero REF Giocatori (ID_Giocatore);
       FK ID_Vincitore REF Giocatori (ID_Giocatore);
```

Si implementino le seguenti procedure:

- Inserimento di una partita odierna dove un giocatore affronta se stesso (**parametri**: da valutare)
- Il bianco vince una partita (**parametri**: codice partita)
- Il bianco perde una partita (**parametri**: codice partita)
- Eliminazione di tutte le partite giocate da una persona contro se stessa (**parametri**: codice persona)

ESERCIZIO 13.

Sul seguente schema relazionale

```
STUDENTI (ID_Studente, Cognome, Nome, Età, Sesso);
       PK ID_Studente;
VERIFICHE (ID_Verifica, ID_Studente, Data, Tipo, Materia, Voto);
       PK ID_Verifica;
       FK ID_Studente REF Studenti (ID_Studente);
```

Si implementino le seguenti procedure:

- Inserimento di una verifica odierna in Storia (**parametri**: codice persona, voto)
- Inserimento di una verifica odierna (**parametri**: codice persona, nome materia, voto)
- Inserimento di una verifica (**parametri**: codice persona, data, nome materia, voto)
- Eliminazione di tutte le verifiche di uno studente (**parametri**: codice persona)

ESERCIZIO 14.

Sul seguente schema relazionale

```
PIZZA (ID_Pizza, Nome, Prezzo);
       PK ID_Pizza;
       UNIQUE (Nome);
INGREDIENTI (ID_Ingrediente, Nome, Costo, Scorte);
       PK ID_ Ingrediente;
COMPOSIZIONI (ID_Composizione, ID_Pizza, ID_Ingrediente);
       PK ID_Composizione;
       FK ID_Pizza REF Pizze (ID_Pizza);
       FK ID_Ingrediente REF Ingredienti (ID_Ingrediente);
```

Si implementino le seguenti procedure:

- Inserimento di una nuova pizza (**parametri**: nome, prezzo)
- Assegnazione di un ingrediente a una pizza (**parametri**: codice pizza, codice ingrediente)
- Eliminazione di un ingrediente da una pizza (**parametri**: codice pizza, codice ingrediente)
- Incremento del 10% del prezzo delle pizze contenenti un ingrediente (**parametri**: codice ingrediente)

ESERCIZI SULLE FUNZIONI

ESERCIZIO 15.

Sul seguente schema relazionale

```
PIZZA (ID_Pizza, Nome, Prezzo);
       PK ID_Pizza;
       UNIQUE (Nome);
INGREDIENTI (ID_Ingrediente, Nome, Costo, Scorte);
       PK ID_ Ingrediente;
COMPOSIZIONI (ID_Composizione, ID_Pizza, ID_Ingrediente);
       PK ID_Composizione;
       FK ID_Pizza REF Pizze (ID_Pizza);
       FK ID_Ingrediente REF Ingredienti (ID_Ingrediente);
```

Si implementino le seguenti **funzioni**:

- Tabella listino pizze ordinato alfabeticamente (**parametri**: nessuno)
- Tabella listino pizze ordinato per prezzo crescente (**parametri**: nessuno)
- Tabella listino pizze (nome, prezzo) contenenti un ingrediente (**parametri**: codice ingrediente)
- Tabella listino pizze che non contengono un certo ingrediente (**parametri**: codice ingrediente)
- Calcolo numero pizze contenenti un ingrediente (**parametri**: codice ingrediente)
- Calcolo numero pizze non contengono un ingrediente (**parametri**: codice ingrediente)
- Calcolo numero ingredienti contenuti in una pizza (**parametri**: codice pizza)

SOMMARIO

SOMMARIO

MICROSOFT SQL SERVER 2
IL PRODOTTO 2
MICROSOFT SQL SERVER 2
DATA DESCRIPTION LANGUAGE 2
DATA MANIPULATION LANGUAGE 2
TSQL PER PROCEDURE E TRIGGER 3
TSQL PER GESTIRE STRINGHE E DATE 3
CREAZIONE DEL DATABASE 3
COMANDI CREATE 3
COMANDI DROP 3
TIPI DI DATO PREESISTENTI 4
NUOVI TIPI DI DATO 5
CREATE TYPE 5
DEFINIZIONE DI TABELLE 6
CREATE TABLE 6
CHIAVI PRIMARIE E AUTO-INCREMENTO 6
CHIAVI PRIMARIE SU PIÙ CAMPI 7
INDICE UNIVOCO 7
CHIAVI ESTERNE 9
CHIAVI ESTERNE SU PIÙ CAMPI 9
TABELLE ED INTEGRITÀ REFERENZIALE 10
GESTIONE DEI TENTATIVI DI VIOLAZIONE DI INTEGRITÀ REFERENZIALE 10
TABELLE E CONTROLLI SUI CAMPI 11
CONTROLLI (CHECK) 11
CREATE INDEX 11
OPZIONI SQL 12
ELIMINARE UN INDICE 13
CREATE VIEW 13
PERCHÉ USARE UNA VISTA? 14
LINGUAGGIO T-SQL E DICHIARAZIONI DI VARIABILI 14
T-SQL VARIABILI 14
DICHIARAZIONE DI VARIABILE 14
ASSEGNAZIONE DI VARIABILE 15
T-SQL VARIABILI GLOBALI 16
T-SQL FUNZIONI PREDEFINITE 17
FUNZIONI SCALARI 17
COMANDI IMPERATIVI IN LINGUAGGIO T-SQL 19
COMANDO SEQUENZA (BLOCCO DI ISTRUZIONI) 19
COMANDO DECISIONALE (ESECUZIONE CONDIZIONALE) 19
COMANDO ITERATIVO (ESECUZIONE CICLICA) 20
STORED PROCEDURE IN T-SQL 20
SINTASSI GENERALE 20
PARAMETRI 22
INVOCAZIONE 22
FUNCTION IN T-SQL 23
SINTASSI GENERALE 23
FUNZIONI SCALARI 23
FUNZIONI TABELLARI 24
TRIGGER IN T-SQL 25
CREATE TRIGGER 25
ASSERZIONI IN T-SQL 26
CREATE RULE 26
ESERCIZI SULLE TABELLE 27
ESERCIZIO 1. 27
ESERCIZIO 2. 27
ESERCIZIO 3. 27
ESERCIZIO 4. 27
ESERCIZIO 5. 27
ESERCIZIO 6. 28
ESERCIZIO 7. 28
ESERCIZIO 8. 28
ESERCIZIO 9. 28
ESERCIZI SULLE PROCEDURE 29
ESERCIZIO 10. 29
ESERCIZIO 11. 29
ESERCIZIO 12. 29
ESERCIZIO 13. 29
ESERCIZIO 14. 30
ESERCIZI SULLE FUNZIONI 30
ESERCIZIO 15. 30