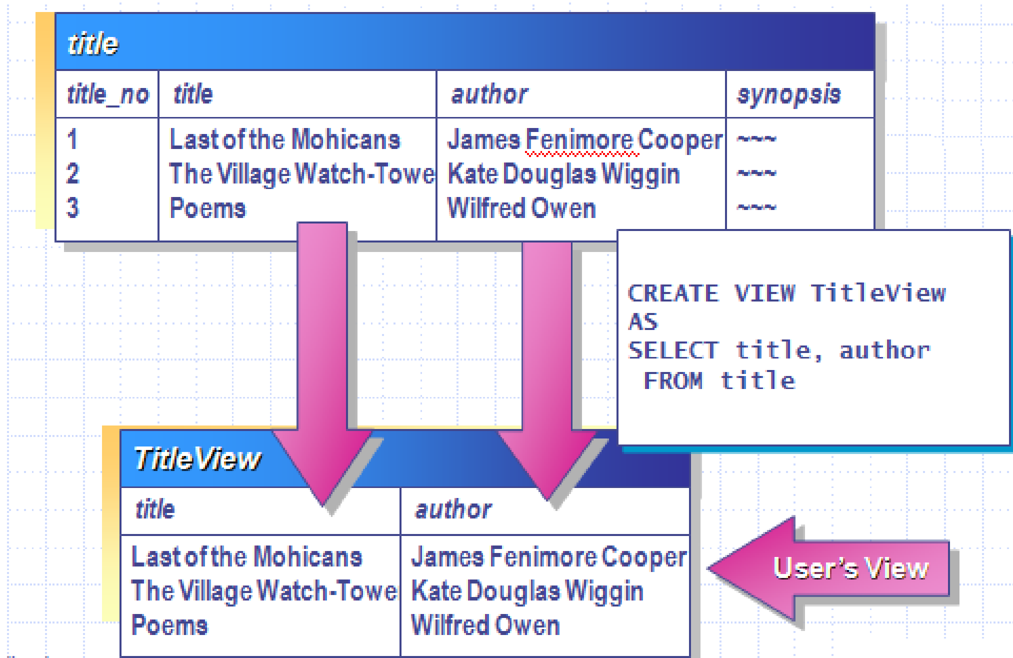
VISTE E ALTERNATIVE ALLE VISTE

Le viste servono a risolvere problemi che, spesso, possono essere risolti anche in altri modi.



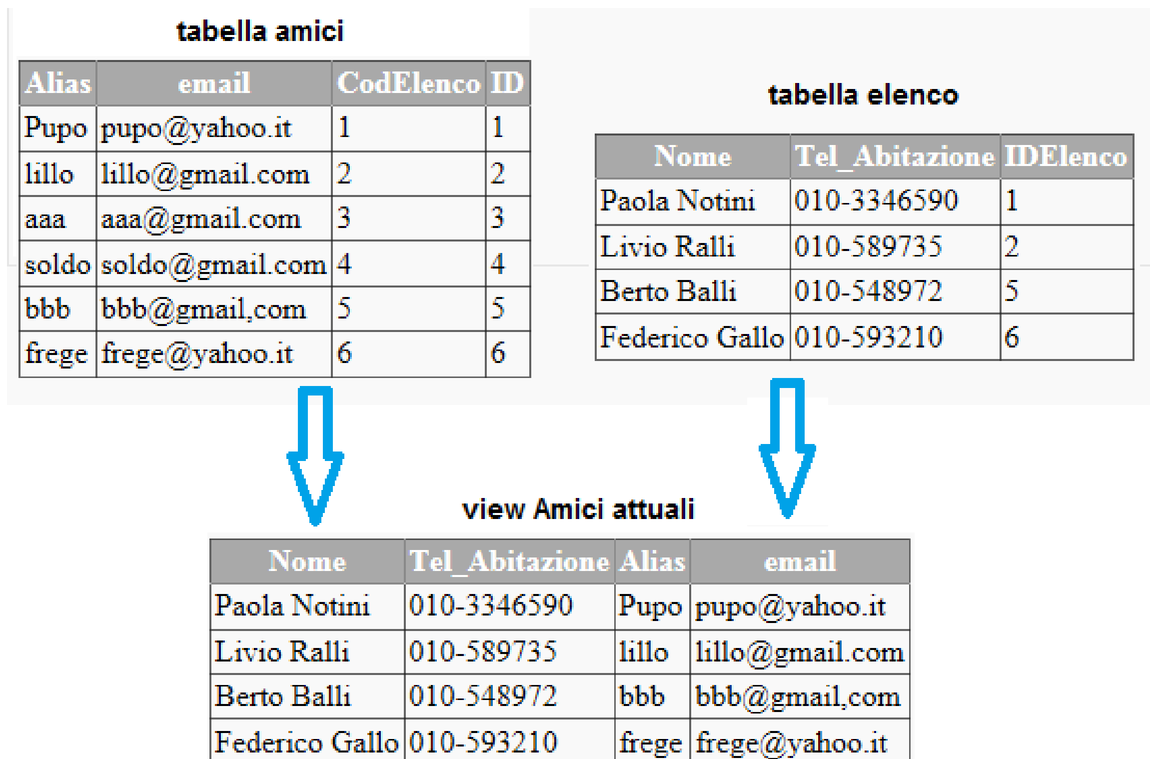


tabella amici

Alias	email	CodElenco	ID
Pupo	pupo@yahoo.it	1	1
lillo	lillo@gmail.com	2	2
aaa	aaa@gmail.com	3	3
soldo	soldo@gmail.com	4	4
bbb	bbb@gmail.com	5	5
frege	frege@yahoo.it	6	6

tabella elenco

Nome	Tel_Abitazione	IDElenco
Paola Notini	010-3346590	1
Livio Ralli	010-589735	2
Berto Balli	010-548972	5
Federico Gallo	010-593210	6

view Amici attuali

Nome	Tel_Abitazione	Alias	email
Paola Notini	010-3346590	Pupo	pupo@yahoo.it
Livio Ralli	010-589735	lillo	lillo@gmail.com
Berto Balli	010-548972	bbb	bbb@gmail.com
Federico Gallo	010-593210	frege	frege@yahoo.it

Se nome composto con spazi bianchi, nella sintassi si usano [] es: CREATE VIEW [Amici attuali] AS *query_tipo_SELECT*

CREATE VIEW [Amici attuali] AS
SELECT Elenco.Nome, Elenco.Tel_Abitazione,Amici.Alias, Amici.email
FROM Amici,Elenco
WHERE Elenco.IDElenco=Amici.CodElenco

Per eliminare se esiste **DROP** VIEW [Amici attuali]
Se non esiste[1] Error: **Cannot drop the view 'Amici attuali', because it does not exist or you do not have permission.**

[1] Esempi con uso SQL SERVER – provider free somee

```
CREATE VIEW estratto AS
SELECT Elenco.Nome, Elenco.Tel_Abitazione,Amici.Alias, Amici.email
FROM Amici,Elenco
WHERE Elenco.IDElenco=Amici.CodElenco
```

Se già esiste Error: **There is already an object named 'estratto' in the database.**

select **name** FROM **sys.views**

name
estratto
Amici attuali

Per modificare se esiste

```
ALTER VIEW estratto AS
SELECT Elenco.Nome, Amici.Alias, Amici.email
FROM Amici,Elenco
WHERE Elenco.IDElenco=Amici.CodElenco
```

Se non esiste Error: **Invalid object name 'estratto'.**

Con controllo sull'esistenza – non esistenza

```
IF OBJECT_ID ('estratto', 'V') IS NOT NULL
DROP VIEW estratto
```

Alternative:

```
IF EXISTS(select * FROM sys.views where name = 'estratto')
DROP View estratto

IF (EXISTS (SELECT 1 FROM sys.views WHERE name = 'estratto'))
DROP View estratto
```

Visualizzando con:

```
select * FROM sys.views where name = 'estratto'
```

SQL Query Result

name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date	i
estratto	1986106116		1	0	V	VIEW	12/13/2015 2:47:56 AM	12/13/2015 2:47:56 AM	

```
select * FROM sys.views
```

SQL Query Result

name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date	is_ms_shipped	is_published	is_schema_published	is_replicated	has_rep
estratto	1986106116		1	0	V	VIEW	12/13/2015 2:47:56 AM	12/13/2015 2:47:56 AM					
Amici attuali	2002106173		1	0	V	VIEW	12/13/2015 2:50:56 AM	12/13/2015 2:50:56 AM					

Non fornisce errore il tentativo di ***visualizzare*** VIEW ***inesistenti***

select * FROM sys.views where name='nuova'

SQL Query Result
Success!

Nb: sia CREATE che ALTER devono essere all'inizio dello statement

Quindi non accettata la sintassi :

```
IF OBJECT_ID ('estratto', 'V') IS NOT NULL
ALTER VIEW estratto AS ....
```

Error: **Incorrect syntax near the keyword 'VIEW'.**

```
IF OBJECT_ID('estratto') IS NULL
  CREATE VIEW estratto AS ....
```

oppure

```
IF (NOT EXISTS (SELECT 1 FROM sys.views WHERE name = 'estratto'))
   CREATE VIEW estratto AS....
```

oppure

```
IF NOT EXISTS(select * FROM sys.views where name = 'estratto')
  CREATE VIEW estratto AS ....
```

Error: **Incorrect syntax near the keyword 'VIEW'.**

Equivalenza:

<table>
<tr><td>SELECT Nome
FROM elenco
GROUP BY Nome
ORDER BY Nome ASC;</td><td>SELECT DISTINCT Nome
FROM elenco
ORDER BY Nome ASC;</td></tr>
</table>

Nella sua forma di utilizzo più semplice la **clausola *GROUP BY*** produce un risultato analogo a ***SELECT DISTINCT***.

Nome
Berto Balli
Federico Gallo
Livio Ralli
Paola Notini

Creazione tabelle senza PK:

<table>
<tr><td>

```
CREATE TABLE Persons
        (
        PersonID int,
        LastName varchar(255),
        FirstName varchar(255),
        Address varchar(255),
        City varchar(255)
        )
```

</td><td>

```
CREATE TABLE Ordini
        (
        id int,
        id_cliente int,
        prezzo int,
        prodotto varchar(255)
        )
```

</td></tr>
</table>

Popoliamo tabella Persons

INSERT INTO Persons VALUES(1,'Perla','Giada', 'via Giotto 7', 'Milano')

Select * FROM Persons

PersonID	LastName	FirstName	Address	City
1	Perla	Giada	via Giotto 7	Milano

Cancellando **DROP TABLE [Persons]** oppure **DROP TABLE Persons**

Reinserendo

INSERT INTO Persons VALUES(2,'Dori','Marco', 'via Pisa 6', 'Genova')

Select * FROM Persons

PersonID	LastName	FirstName	Address	City
1	Perla	Giada	via Giotto 7	Milano
2	Dori	Marco	via Pisa 6	Genova

Aggiorniamo

```
UPDATE table_name
SET column1=value1,column2=value2,...
WHERE some_column=some_value;
```

UPDATE Persons SET LastName='Perlo' WHERE PersonID=1

Select * FROM Persons

PersonID	LastName	FirstName	Address	City
1	Perlo	Giada	via Giotto 7	Milano
2	Dori	Marco	via Pisa 6	Genova

Popoliamo tabella ORDINI

INSERT INTO Ordini VALUES(1, 1, 50, 'Scarpe')
INSERT INTO Ordini VALUES(2, 1,1500, 'Vestito')
INSERT INTO Ordini VALUES(3, 2, 40, 'Scarpe')
INSERT INTO Ordini VALUES(4, 2, 500, 'Vestito')
INSERT INTO Ordini VALUES(5, 1, 60, 'Camicia')
INSERT INTO Ordini VALUES(6, 1,70, 'Cravatta')

select * from Ordini

id	id_cliente	prezzo	prodotto
1	1	50	Scarpe
2	1	1500	Vestito
3	2	40	Scarpe
4	2	500	Vestito
5	1	60	Camicia
6	1	70	Cravatta

Estraiamo con raggruppamento ed ordinamento

```
SELECT LastName
FROM Persons
GROUP BY LastName
ORDER BY LastName ASC;
```

LastName
Dori
Perlo

```
SELECT id_cliente, SUM(prezzo) AS spesa
FROM ordini
GROUP BY id_cliente
HAVING SUM(prezzo)>=1000     /*nb:  valori interi */
```

id_cliente	spesa
1	1680

```
SELECT id_cliente, Count(id) AS acquisti, SUM(prezzo) AS spesa
FROM ordini
GROUP BY id_cliente
HAVING SUM(prezzo) >= 1000 OR SUM(id) >= 2;  /*nb:  valori interi */
```

id_cliente	acquisti	spesa
1	4	1680
2	2	540

```
SELECT LastName, Count(id) AS acquisti, SUM(prezzo) AS spesa
FROM ordini, Persons
Where id_cliente = PersonID
GROUP BY LastName
HAVING (SUM(prezzo) >= 1000) OR (Count(id) >= 2);
```

LastName	acquisti	spesa
Dori	2	540
Perlo	4	1680

```
SELECT id_cliente, MAX(prezzo) AS spesa_massima
FROM ordini
GROUP BY id_cliente;
```

id_cliente	spesa_massima
1	1500
2	500

```
SELECT LastName, MAX(prezzo) AS spesa_massima
FROM ordini, Persons
Where id_cliente = PersonID
GROUP BY LastName
```

LastName	spesa_massima
Dori	500
Perlo	1500

```
SELECT prodotto,prezzo
FROM ordini
GROUP BY prodotto, prezzo
ORDER BY prodotto ASC;
```

prodotto	prezzo
Camicia	60
Cravatta	70
Scarpe	40
Scarpe	50
Vestito	500
Vestito	1500

```
SELECT prodotto,prezzo
FROM ordini
GROUP BY prodotto, prezzo
ORDER BY prezzo ASC;
```

prodotto	prezzo
Scarpe	40
Scarpe	50
Camicia	60
Cravatta	70
Vestito	500
Vestito	1500

```
SELECT prodotto,prezzo
FROM ordini
GROUP BY prodotto, prezzo
ORDER BY prezzo DESC;
```

prodotto	prezzo
Vestito	1500
Vestito	500
Cravatta	70
Camicia	60
Scarpe	50
Scarpe	40

```
SELECT LastName, AVG(prezzo) AS spesa_media
FROM ordini, Persons
Where id_cliente = PersonID
GROUP BY LastName
```

LastName	spesa_media
Dori	270
Perlo	420

```
SELECT LastName, AVG(prezzo) AS spesa_media
FROM ordini, Persons
Where id_cliente = PersonID
GROUP BY LastName
ORDER BY spesa_media DESC;
```

LastName	spesa_media
Perlo	420
Dori	270

Attuali tabelle

name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date
Persons	174623665		1	0	U	USER_TABLE	12/13/2015 3:40:34 AM	12/13/2015 3:40:34 AM
Elenco	245575913		1	0	U	USER_TABLE	11/25/2015 11:52:37 PM	11/25/2015 11:52:37 PM
Amici	277576027		1	0	U	USER_TABLE	11/25/2015 11:52:58 PM	11/25/2015 11:52:58 PM
Ordini	430624577		1	0	U	USER_TABLE	12/13/2015 4:11:19 AM	12/13/2015 4:11:19 AM

Esercizi:

1. Realizzare come VIEWS alcune delle estrazioni esemplificate precedentemente con raggruppamenti ed ordinamento
2. Modificare una View creata
3. Verificare l'archiviazione delle Views create
4. Eliminare le Views create con controllo di esistenza

Traccia soluzione:

1. CREATE VIEW *nomeView* AS SELECT …..
2. ALTER VIEW *nomeView* AS SELECT ….
3. SELECT * FROM sys.views
4. IF OBJECT_ID (' *nomeView* ', 'V') IS NOT NULL DROP VIEW *nomeView*

STORED PROCEDURE

Una stored procedure (SP) è **un insieme di comandi T-SQL compilati**, direttamente accessibili da SQL Server.

Tali comandi vengono eseguiti come un'unica unita di lavoro (**batch**) sul server e il beneficio è che il traffico di rete viene ridotto limitando la congestione della rete stessa.

CREATE PROC [EDURE] *nome_procedura*
AS *istruzione_sql* [...*n*]

STORED PROCEDURE (DDL) – creazione dinamica di tabella ANONIMA

```
CREATE PROCEDURE Test2
AS
   CREATE TABLE #t(x INT PRIMARY KEY);     /*crea tabella dinamica  anonima con x intero e PK*/
   INSERT INTO #t VALUES (2);        /* inserisce 2 come valore dell'unico campo x */
   SELECT Test2Col = x FROM #t;     /*estrae  rinomando l'unica colonna */
```

Per ESEGUIRE: `EXEC nome` **Exec Test2**

Test2Col
2

Se si tratta della prima istruzione in un batch oppure di uno script **osql** o **sqlcmd**, non è necessario specificare EXEC

select * from sys.procedures

name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date
Test2	750625717		1	0	P	SQL_STORED_PROCEDURE	12/13/2015 4:56:27 AM	12/13/2015 4:56:27 AM

```
CREATE PROC TestProc
AS
   CREATE TABLE #t(id INT PRIMARY KEY, Nome varchar (255) );
   INSERT INTO #t VALUES (1, 'Rossi');
   SELECT * FROM #t;
```

Per ESEGUIRE: **TestProc**

id	Nome
1	Rossi

Per eliminare: **DROP PROCEDURE Test2**
Sintassi: `DROP PROCEDURE <sp_name>`

```
IF OBJECT_ID('Test2','P')IS NOT NULL
DROP PROCEDURE Test2
```

o più semplicemente **IF OBJECT_ID('Test2')IS NOT NULL**
DROP PROCEDURE Test2

Evitare di MEMORIZZARE con NOME la tabella→ errore quando si riusa la procedura (tabella già creata)

```
CREATE PROC TabProc
AS CREATE TABLE Nuova(id INT PRIMARY KEY, Nome varchar (255) );
   INSERT INTO Nuova VALUES (1, 'Rossi');
   SELECT * FROM Nuova;
```

name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date
Persons	174623665		1	0	U	USER_TABLE	12/13/2015 03:40:34	12/13/2015 03:40:34
Elenco	245575913		1	0	U	USER_TABLE	11/25/2015 23:52:37	11/25/2015 23:52:37
Amici	277576027		1	0	U	USER_TABLE	11/25/2015 23:52:58	11/25/2015 23:52:58
Ordini	430624577		1	0	U	USER_TABLE	12/13/2015 04:11:19	12/13/2015 04:11:19
Nuova	1438628168		1	0	U	USER_TABLE	12/13/2015 05:51:23	12/13/2015 05:51:23

nb: verificabile l'archiviazione della tabella con **select * from Nuova**
da eliminare per riuso della SP: drop table Nuova

id	Nome
1	Rossi

STORED PROCEDURE: DML

Una SP può richiamare istruzioni SELECT, UPDATE e DELETE ma anche altre SP, può utilizzare istruzioni che controllano il flusso di esecuzione e funzioni di aggregazione.

E' importante sottolineare che oltre alle SP create dai programmatori esistono centinaia di SP di sistema all'interno di SQL Server (tutte iniziano con il prefisso sp_).

I benefici derivanti dall'utilizzo delle SP sono diversi. Ad esempio **evitano l'accesso diretto alle tabelle** da parte degli utenti ed inoltre vengono **ottimizzate tramite sistemi di caching** per essere più performanti possibile.

Si può dire che esse costituiscano una sorta di interfaccia tramite cui operare sui dati di un database senza conoscerne la struttura e le relazioni.

La sintassi completa per la creazione di una SP :

```
CREATE PROCEDURE procedure_name
[ { @parameter_name} datatype [= default_value] [OUTPUT]]
[ { WITH [RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION } ]
AS
[BEGIN]
   statements
[END]
```

Dopo aver inserito il **nome** occorre specificare i **parametri** (se necessari e comunque **preceduti dalla @**) che la SP richiede, con **relativo tipo** di dati ed eventualmente valore di default. E' anche possibile ritornare uno o più valori o una tabella di dati utilizzando un parametro per trasmettere tali informazioni. Tale parametro deve essere seguito dalla parola riservata OUTPUT e per esso non può essere definito un valore di default.

Nella sintassi generale possiamo notare anche le opzioni RECOMPILE e ENCRYPTION. La prima indica a SQL Server di ricompilare la SP ogni volta che viene eseguita per forzare ogni volta la rigenerazione del piano di esecuzione, cosa che abbiamo detto in precedenza migliora le prestazioni. La seconda comporta la crittografia del contenuto della SP in modo che esso non sia visibile e comprensibile da chi non ha diritto di farlo.

Dopo la parola riservata AS comincia poi un blocco BEGIN-END (***opzionali***) dove possiamo inserire le nostre istruzioni T-SQL. Di seguito un esempio di una SP

CREATE PROCEDURE ProceduraTest **@id_cliente int** **AS** **SELECT LastName, AVG(prezzo) AS spesa_media** **FROM ordini, Persons** **Where PersonID =@id_cliente int** **GROUP BY LastName**	**CREATE PROCEDURE ProceduraTest** **@id_cliente int** **AS** **begin** **SELECT LastName, AVG(prezzo) AS spesa_media** **FROM ordini, Persons** **Where PersonID =@id_cliente** **GROUP BY LastName** **End**

name	object_id	principal_id	schema_id	parent_object_id	type	type_desc	create_date	modify_date
ProceduraTest	1726629194		1	0	P	SQL_STORED_PROCEDURE	12/13/2015 06:43:00	12/13/2015 06:43:00

Per eseguire se ci sono parametri

```
EXEC nome @nomeParamentro = 25
```

oppure (se più parametri, in questo caso è essenziale il corretto ordine):

```
EXEC nome 25;
```

ProceduraTest 2

LastName	spesa_media
Dori	370

ProceduraTest @id_cliente = 1

LastName	spesa_media
Perlo	370

Trigger: esecuzione in caso di modifica dei dati **(DML)**

Un trigger è una **stored procedure specializzata** che può essere eseguita o in caso di modifica dei dati (trigger **DML** – Data Modification Language) o in caso di definizione del modello dei dati (trigger **DDL** – Data Definition Language) come nel caso di istruzioni CREATE TABLE.

Ci soffermeremo sui trigger **DML** che possono essere considerati come **parti di codice legate a tabelle specifiche** che vengono **eseguite automaticamente in conseguenza di istruzioni** INSERT, DELETE o UPDATE.

I trigger DML possono essere utilizzati per diversi scopi. Un utilizzo abbastanza diffuso è legato a diverse forme di **validazione e controllo sui dati**. Chiaramente l'utilizzo di chiavi e vincoli è preferibile dal punto di vista delle prestazioni ma i trigger sono la scelta migliore quando le validazioni da effettuare sono complesse.

Un altro uso molto diffuso si può riscontrare quando è necessario effettuare **modifiche** sui dati di altre tabelle nel momento in cui si verifica una variazione dei dati sulla tabella contenente un trigger.

È possibile creare trigger separati per ciascuna azione che si effettua su una tabella **tranne per le istruzioni SELECT** perché ovviamente tali istruzioni non comportano modifiche ai dati.

I trigger DML possono essere di tre tipi:

- INSERT trigger
- DELETE trigger
- UPDATE trigger

È possibile anche avere una combinazione dei tre tipi.

La sintassi T-SQL per la creazione di un trigger è abbastanza complessa ed è la seguente

```
CREATE TRIGGER [schema_name.]trigger_name
ON {table|view}
[WITH ENCRYPTION]
{
{{FOR {AFTER|INSTEAD OF} {[INSERT] [,] [UPDATE] [,] [DELETE]}
AS
[{IF [UPDATE (column)
[{AND|OR} UPDATE (column)]] ]
COLUMNS_UPDATE()]
sql_statements}}
```

```
CREATE TRIGGER nomeTrigger
ON nomeTabella
FOR DELETE                /* , INSERT, UPDATE più azioni contemporaneamente */
AS
INSERT TabellaInserimento
    SELECT campi AS Azione
FROM deleted
```

Create trigger è l'istruzione che fisicamente crea il trigger *nomeTrigger*, la parola chiave ON invece ci dice la tabella sulla quale viene ancorato mentre la parola chiave FOR indica a quali eventi viene associato.

In questo caso il trigger **si attiverà per ogni DELETE** riguardante la tabella *nomeTabella*.

Quando attivato, un trigger lavora su due tabelle particolari chiamate *inserted* e *deleted*.

La SELECT inclusa nel trigger *nomeTrigger* estrae le informazioni necessarie dalla tabella *deleted*.

Esempio

Avendo prima creato la seguente tabella:

```
CREATE TABLE CronologiaPersone
 (
PersonID int,
 LastName varchar(255),
 FirstName varchar(255),
 Address varchar(255),
 City varchar(255),
Stato varchar(255)
 )
```

si crea il trigger:

```
CREATE TRIGGER TR_DEL_Persone
ON Persons
FOR DELETE
AS
INSERT CronologiaPersone
   SELECT PersonID, LastName, FirstName, Address, City , ' Eliminato '  AS Azione
FROM deleted
```

Effetto di ***cancellazione***

delete from Persons
where PersonID = 1

in Tabella **Persons**

PersonID	LastName	FirstName	Address	City
2	Dori	Marco	via Pisa 6	Genova

select * from CronologiaPersone

PersonID	LastName	FirstName	Address	City	Stato
1	Perlo	Giada	via Giotto 7	Milano	Eliminato

È facile intuire che nel caso di una operazione di DELETE la tabella *deleted* conterrà le righe che sono state appena eliminate al contrario con una INSERT la tabella *inserted* conterrà le righe appena inserite, ma con una UPDATE?

In questa caso entrambe le tabelle contengono valori, perché la *deleted* conterrà i dati prima della modifica (le vecchie righe) mentre la *inserted* conterrà i dati dopo la modifica (le nuove righe). Per capire meglio il funzionamento vediamo un esempio

```
CREATE TRIGGER TR_UPD_Persone
ON Persons
FOR UPDATE
AS
DECLARE @LastName NVARCHAR(20)
SELECT @LastName = LastName FROM deleted
PRINT 'Prima: ' + @LastName
SELECT @LastName = LastName FROM inserted
PRINT 'Dopo: ' + @LastName
BEGIN TRAN
SET NOCOUNT ON
UPDATE Persons SET LastName = 'Rossi' WHERE PersonID = 1
ROLLBACK    /* Cancello la modifica appena fatta */
```

select * from sys.triggers

name	object_id	parent_class	parent_class_desc	parent_id	type	type_desc	create_date	modify_date
TR_DEL_Persone	203147769	1	OBJECT_OR_COLUMN	174623665	TR	SQL_TRIGGER	12/13/2015 7:51:03 AM	12/13/2015 7:51:03 AM
TR_UPD_Persone	2030630277	1	OBJECT_OR_COLUMN	174623665	TR	SQL_TRIGGER	12/13/2015 7:07:53 AM	12/13/2015 7:07:53 AM

Esercizio:

1. Creare un Trigger per reinserire la tuple eliminate nella tabella Persons ed aggiornare la tabella CronologiaPersone con lo stesso effetto dello statement seghuente:

UPDATE CronologiaPersone SET Stato='Repristino' WHERE PersonID=1

PersonID	LastName	FirstName	Address	City	Stato
1	Perlo	Giada	via Giotto 7	Milano	Repristino

***Altri esempi*:**

```
createtable copymcnvendors
 (
 vendorid int,
 vendornamevarchar(15),
 vendorcityvarchar(15),
 vendorstatevarchar(15)
 )

insertinto copymcnvendors values (20,'vipendra','noida','up')
insertinto copymcnvendors values (21,'deepak','lucknow','up')
insertinto copymcnvendors values (22,'rahul','kanpur','up')
insertinto copymcnvendors values (23,'malay','delhi','delhi')
insertinto copymcnvendors values (24,'mayank','noida','up')
insertinto copymcnvendors values (25,'shiva','delhi','delhi')
insertinto copymcnvendors values (26,'praveen','noida','up')

createtable copymcninvoices
(
invoiceidint notnull identityprimary key,
vendoridint notnull,
invoicenovarchar(15),
invoicetotalmoney,
paymenttotalmoney,
creadittotalmoney
)

insertinto copymcninvoices values (20,'e001',100,100,0.00)
insertinto copymcninvoices values (21,'e002',200,200,0.00)
insertinto copymcninvoices values (22,'e003',500,0.00,100)
insertinto copymcninvoices values (23,'e004',1000,100,100)
insertinto copymcninvoices values (24,'e005',1200,200,500)
```

Statement usato per creare un **AFTER trigger** in **SQL Server 2012:**

```
createtrigger copymcnvendors_del_up
on copymcnvendors
after delete,update
as
if exists ( select* from deleted join copymcninvoices
on deleted.vendorid=copymcninvoices.vendorid)
BEGIN
raiserror('Vendor ID is in Use in other table.',1,1)
rollback TRAN
END
```

Batch Aborted se

```
Delete  from copymcnvendors where vendorid = 21
Delete  from copymcnvendors where vendorid = 24
```

Ok: `Delete  from copymcnvendors where vendorid = 26`

Statement usato per creare un **AFTER trigger** in **SQL Server 2012:**

```
CREATETRIGGER copymcninvoices_ins_up
ON
copymcninvoices
AFTER INSERT,UPDATE
as
IF not exists( SELECT* FROM copymcnvendors WHERE vendorid IN (SELECT vendorid FROM inserted))
BEGIN
RAISERROR('Vendor is invalied',1,1)
ROLLBACK TRAN
END
```

```
INSERT INTO copymcninvoices
        ( vendorid ,
          invoiceno ,
          invoicetotal ,
          paymenttotal ,
          creadittotal
        )
VALUES  ( 21 , -- vendorid - int
          'e006' , -- invoiceno - varchar(15)
          10000 , -- invoicetotal - money
          2000 , -- paymenttotal - money
          1000  -- creadittotal - money
        )
```

Colonne Virtuali o Colonne calcolate

SQL Server supporta le **Colonne Virtuali**, ma le chiama **Computed Column** (colonne calcolate).

L'unico tipo supportato in SQL Server è **Persisted,** che equivale di fatto a Persistent.

Queste colonne vengono calcolate in base a un'espressione che utilizza altre colonne della stessa tabella.

L'espressione può essere un nome di colonna non calcolata, una costante, una funzione e qualsiasi combinazione di questi elementi collegati da uno o più operatori. L'espressione **non può essere una sottoquery**.

Le colonne calcolate sono colonne virtuali, e quindi **non "editabili"**, ognuna delle quali viene definita come espressione calcolata in base a una o più colonne specificate nell'istruzione CREATE TABLE, ad esempio:

```
CREATE TABLE Products
(
    ProductID int IDENTITY (1,1) NOT NULL,
    QtyAvailable smallint,
    UnitPrice money,
    InventoryValue AS QtyAvailable * UnitPrice
);

-- Insert values into the table.
INSERT INTO Products (QtyAvailable, UnitPrice)
VALUES (25, 2.00), (10, 1.5);

-- Display the rows in the table.
SELECT ProductID, QtyAvailable, UnitPrice, InventoryValue
FROM Products;
```

Per aggiungere una nuova **colonna calcolata** a una tabella esistente

```
ALTER TABLE Products ADD RetailValue AS (QtyAvailable * UnitPrice * 1.35);
```