Table of Contents

SQL AVANZATO

VIEW

Una **VIEW** è una tabella virtuale che viene creata a partire da una o più tabelle reali. Le VIEW possono essere utilizzate per nascondere o modificare la struttura delle tabelle reali, per fornire un'interfaccia più semplice agli utenti o per eseguire query complesse in modo più conciso.

Pseudocodice

```
CREATE VIEW [nome_view] AS
SELECT [colonne]
FROM [tabella]
[WHERE] [condizione]
```

Esempio

Supponiamo di avere la seguente tabella:

```
CREATE TABLE clienti (
  nome VARCHAR(255),
  cognome VARCHAR(255),
  email VARCHAR(255)
);
```

Per creare una VIEW che contenga solo i nomi e i cognomi dei clienti, possiamo utilizzare il seguente codice:

```
CREATE VIEW clienti_nomi_cognome AS
SELECT nome, cognome
FROM clienti;
```

CLUSTERED e NON CLUSTER INDEX

Un **INDEX** è una struttura di dati che viene utilizzata per velocizzare le query che accedono a una tabella. Gli INDEX sono costituiti da una serie di chiavi che vengono utilizzate per ordinare i dati della tabella.

Clustered index

Un clustered index è un tipo di indice che ordina fisicamente i dati della tabella. Questo significa che i dati della tabella vengono memorizzati in ordine di indice.

Solo una tabella può avere un clustered index. Se una tabella non ha un clustered index, i dati della tabella vengono memorizzati in ordine casuale.

I clustered index sono utili per le query che devono accedere ai dati in un ordine specifico. Ad esempio, una query che deve recuperare tutti i clienti ordinati per nome sarà più veloce se la tabella `clienti` ha un clustered index sulla colonna `nome`.

Pseudocodice

```
CREATE CLUSTERED INDEX [nome_indice] ON [tabella] ([colonna1], [colonna2], ...);
```

Esempio

Supponiamo di avere la seguente tabella:

```
CREATE TABLE clienti (
  nome VARCHAR(255),
  cognome VARCHAR(255),
  email VARCHAR(255)
);
```

Per creare un clustered index sulla colonna nome, possiamo utilizzare il seguente codice:

```
CREATE CLUSTERED INDEX idx_nome ON clienti (nome);
```

Non clustered index

Un non clustered index è un tipo di indice che ordina i dati della tabella in modo logico. Questo significa che i dati della tabella vengono memorizzati in ordine casuale, ma l'indice mantiene un elenco di valori di chiave e puntatori ai record corrispondenti.

Una tabella può avere più non clustered index.

I non clustered index sono utili per le query che devono accedere ai dati in un ordine specifico, ma non è necessario che i dati della tabella siano memorizzati in quell'ordine. Ad esempio, una query che deve recuperare tutti i clienti ordinati per cognome sarà più

veloce se la tabella `clienti` ha un non clustered index sulla colonna `cognome`.

Pseudocodice

```
CREATE NONCLUSTERED INDEX [nome_indice] ON [tabella] ([colonna1], [colonna2], ...);
```

Esempio

Per creare un non clustered index sulla colonna cognome, possiamo utilizzare il seguente codice:

```
CREATE NONCLUSTERED INDEX idx_cognome ON clienti (cognome);
```

TRIGGERS

Un **TRIGGER** è un blocco di codice che viene eseguito automaticamente quando si verifica un evento specifico, come l'inserimento, l'aggiornamento o l'eliminazione di un record da una tabella. I TRIGGER possono essere utilizzati per eseguire operazioni di validazione, controllo della consistenza dei dati o per eseguire operazioni automatiche.

Tipi di TRIGGER

Esistono due tipi di TRIGGER:

- **INSERT TRIGGER** viene eseguito quando viene inserito un nuovo record in una tabella.
- **UPDATE TRIGGER** viene eseguito quando viene aggiornato un record in una tabella.
- **DELETE TRIGGER** viene eseguito quando viene eliminato un record da una tabella.

Pseudocodice

```
CREATE TRIGGER [nome_trigger]
ON [tabella]
FOR [tipo_evento]
AS
BEGIN
  [blocco_di_codice]
END;
```

Esempio

Per creare un TRIGGER che validi il campo nome della tabella clienti, possiamo utilizzare il seguente codice:

```
CREATE TRIGGER validazione_nome
ON clienti
FOR INSERT
AS
BEGIN
  IF NEW.nome IS NULL
  BEGIN
    RAISERROR('Il campo "nome" non può essere vuoto.', 16, 1);
    ROLLBACK TRANSACTION;
  END
END;
```

In questo esempio, il TRIGGER verifica che il campo nome del record inserito non sia vuoto. Se il campo è vuoto, il TRIGGER genera un errore e annulla la transazione.

STORED PROCEDURES

Una **stored procedure** è un blocco di codice SQL che viene memorizzato nel database. Le stored procedure possono essere utilizzate per eseguire una serie di operazioni complesse in un unico comando.

Pseudocodice

```
CREATE PROCEDURE [nome_procedura]
(
  [parametro1] [tipo_dato],
  [parametro2] [tipo_dato],
  ...
)
AS
BEGIN
  [blocco_di_codice]
END;
```

Esempio

Supponiamo di avere la seguente tabella:

```
CREATE TABLE clienti (
  nome VARCHAR(255),
  cognome VARCHAR(255),
  email VARCHAR(255)
);
```

Ecco un esempio di una stored procedure che recupera tutti i clienti ordinati per nome:

```
CREATE PROCEDURE get_clienti_ordinati_per_nome
AS
BEGIN
  SELECT
    nome,
    cognome,
    email
  FROM
    clienti
  ORDER BY
    nome;
END;
```

Per eseguire questa stored procedure, possiamo utilizzare il seguente comando:

```
EXEC get_clienti_ordinati_per_nome;
```

IF

L'istruzione **IF** viene utilizzata per eseguire un blocco di codice solo se una condizione è soddisfatta.

Pseudocodice

```
IF [condizione]
THEN
  [blocco di codice]
END IF;
```

Esempio

```
DECLARE @Threshold DECIMAL(10, 2) = 1000;

 IF EXISTS (

      SELECT *

      FROM Production.Product

      WHERE Price > @Threshold )

BEGIN PRINT 'There are products with prices greater than $' +
CAST(@Threshold AS VARCHAR);

END;
```

WHILE

L'istruzione **WHILE** viene utilizzata per eseguire un blocco di codice fino a quando una condizione è soddisfatta.

Pseudocodice

```
WHILE [condizione]
BEGIN
  [blocco di codice]
END WHILE;
```

Esempio

Supponiamo di voler stampare tutti i numeri da 1 a 10:

```
DECLARE
  i INT;
BEGIN
  i := 1;
```

```
  WHILE i <= 10
  BEGIN
    PRINT i;
    i := i + 1;
  END WHILE;
END;
```

Questo codice inizializza la variabile i a 1. Quindi, esegue un ciclo WHILE mentre i è minore o uguale a 10. All'interno del ciclo, il codice stampa il valore di i e incrementa i di 1.

CONTINUE

L'istruzione **CONTINUE** viene utilizzata per saltare alla fine del ciclo corrente.

Pseudocodice

```
WHILE [condizione]
BEGIN
  [blocco di codice]
  IF [condizione]
  THEN
    CONTINUE;
  END IF;
END WHILE;
```

Esempio

Supponiamo di voler stampare tutti i numeri da 1 a 10, ma di saltare i numeri pari:

```
DECLARE
  i INT;

BEGIN
  i := 1;
  WHILE i <= 10
  BEGIN
    IF i % 2 = 0
    THEN
      CONTINUE;
    END IF;
    PRINT i;
```

```
    i := i + 1;
  END WHILE;
END;
```

Questo codice inizializza la variabile i a 1. Quindi, esegue un ciclo WHILE mentre i è minore o uguale a 10. All'interno del ciclo, il codice verifica se i è un numero pari. Se lo è, il codice salta alla fine del ciclo. In caso contrario, il codice stampa il valore di i e incrementa i di 1.

TRY CATCH

L'istruzione **TRY CATCH** viene utilizzata per gestire gli errori.

Pseudocodice

```
TRY
BEGIN
  [blocco di codice]
END TRY
CATCH [eccezione]
BEGIN
  [blocco di codice di gestione errori]
END CATCH;
```

Esempio

Supponiamo di voler inserire un nuovo record in una tabella, ma di voler gestire l'errore che si verifica se il record esiste già.

```
TRY
BEGIN
  INSERT INTO clienti (nome, cognome, email)
  VALUES ('Mario', 'Rossi', 'mario.rossi@example.com');
END TRY
CATCH [DuplicateKeyException]
BEGIN
  PRINT 'Il record esiste già.';
END CATCH;
```

Questo codice inizializza un blocco TRY CATCH. Il blocco TRY tenta di inserire il nuovo

record. Se l'operazione ha esito positivo, il blocco TRY viene eseguito e il codice prosegue. Se l'operazione fallisce, il blocco CATCH viene eseguito e il codice stampa un messaggio di errore.

TABLE VARIABLES

Una **TABLE VARIABLE** è una variabile che può contenere un set di dati. Le TABLE VARIABLES possono essere utilizzate per memorizzare i risultati di una query o per passare dati tra funzioni.

Pseudocodice

```
DECLARE
  [nome_variabile] TABLE (
    [colonna1] [tipo_dato],
    [colonna2] [tipo_dato],
    ...
  );
```

Esempio

Supponiamo di avere la seguente tabella:

```
CREATE TABLE clienti (
  nome VARCHAR(255),
  cognome VARCHAR(255),
  email VARCHAR(255)
);
```

Ecco un esempio di come utilizzare una TABLE VARIABLE per memorizzare i risultati di una query:

```
DECLARE
  clienti_selezionati TABLE (
    nome VARCHAR(255),
```

```
    cognome VARCHAR(255),
    email VARCHAR(255)
  );



BEGIN
  SELECT
    nome,
    cognome,
    email
  INTO
    clienti_selezionati
  FROM
    clienti
  WHERE
    età > 18;
END;
```

Questo codice dichiara una TABLE VARIABLE chiamata clienti_selezionati. Quindi, esegue una query per selezionare tutti i clienti che hanno più di 18 anni. I risultati della query vengono memorizzati nella TABLE VARIABLE clienti_selezionati.

SCALAR FUNCTIONS

Una **SCALAR FUNCTION** è una funzione che restituisce un solo valore. Le SCALAR FUNCTIONS possono essere utilizzate per eseguire operazioni matematiche, logiche o di stringhe.

Pseudocodice

```
CREATE FUNCTION [nome_funzione] (
  [parametro1] [tipo_dato],
  [parametro2] [tipo_dato],
  ...
)
RETURNS [tipo_dato]
AS
BEGIN
  [blocco_di_codice]
  RETURN [valore_di_ritorno];
```

```
END;
```

Esempio

Ecco un esempio di una SCALAR FUNCTION che restituisce la somma di due numeri:

```
CREATE FUNCTION somma (
  a INT,
  b INT
)
RETURNS INT
AS
BEGIN
  RETURN a + b;
END;
```

Questa funzione dichiara una SCALAR FUNCTION chiamata somma. La funzione accetta due parametri di tipo INT e restituisce un valore di tipo INT. Il blocco di codice della funzione semplicemente somma i due parametri e restituisce il risultato.

TABLE VALUE FUNCTIONS

Una **TABLE VALUE FUNCTION** è una funzione che restituisce un set di dati. Le TABLE VALUE FUNCTIONS possono essere utilizzate per eseguire operazioni complesse sui dati o per generare report.

Pseudocodice

```
CREATE FUNCTION [nome_funzione] (
  [parametro1] [tipo_dato],
  [parametro2] [tipo_dato],
  ...
)
RETURNS TABLE (
    [colonna1] [tipo_dato],
    [colonna2] [tipo_dato],
    ...
  )
```

```
AS
BEGIN
  [blocco_di_codice]
  RETURN [elenco_di_righe];
END;
```

Esempio

Ecco un esempio di una TABLE VALUE FUNCTION che restituisce tutti i clienti che hanno più di 18 anni:

```
CREATE FUNCTION clienti_maggiorenni ()
RETURNS TABLE (
  nome VARCHAR(255),
  cognome VARCHAR(255),
  email VARCHAR(255)
)
AS
BEGIN
  RETURN
    SELECT
      nome,
      cognome,
      email
    FROM
      clienti
    WHERE
      età > 18;
END;
```

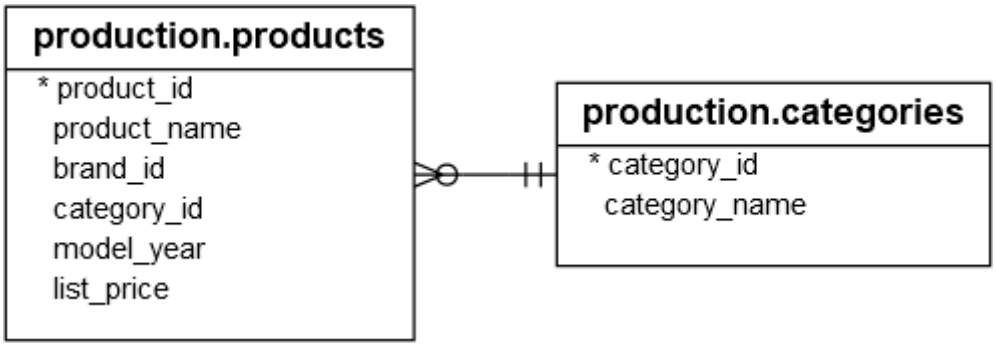
Questa funzione dichiara una TABLE VALUE FUNCTION chiamata clienti_maggiorenni. La funzione non accetta parametri e restituisce un elenco di righe di tipo TABLE. Il blocco di codice della funzione esegue una query per selezionare tutti i clienti che hanno più di 18 anni. I risultati della query vengono restituiti come elenco di righe.

PIVOT e UNPIVOT

https://www.sqlservertutorial.net/sql-server-basics/sql-server-pivot/

Setting up the goals

For the demonstration, we will use the `production.products` and `production.categories` tables from the sample database:



The following query finds the number of products for each product category:

```
SELECT

    category_name,

    COUNT(product_id) product_count

FROM

    production.products p

    INNER JOIN production.categories c

        ON c.category_id = p.category_id

GROUP BY

    category_name;
```

Code language: SQL (Structured Query Language) (sql)

Here is the output:

category_name	product_count
Children Bicycles	59
Comfort Bicycles	30
Cruisers Bicycles	78
Cyclocross Bicycles	10
Electric Bikes	24
Mountain Bikes	60
Road Bikes	60

Our goal is to turn the category names from the first column of the output into multiple columns and count the number of products for each category name as the following picture:

Children Bicycles	Comfort Bicycles	Cruisers Bicycles	Cyclocross Bicycles	Electric Bikes	Mountain Bikes	Road Bikes
59	30	78	10	24	60	60

In addition, we can add the model year to group the category by model year as shown in the following output:

model_year	Children Bicycles	Comfort Bicycles	Cruisers Bicycles	Cyclocross Bicycles	Electric Bikes	Mountain Bikes	Road Bikes
2016	3	3	9	2	1	8	0
2017	19	10	19	2	2	21	12
2018	37	17	50	6	21	31	42
2019	0	0	0	0	0	0	6

Introduction to SQL Server PIVOT operator

SQL Server `PIVOT` operator rotates a table-valued expression. It turns the unique values in one column into multiple columns in the output and performs aggregations on any remaining column values.

You follow these steps to make a query a pivot table:

- First, select a base dataset for pivoting.
- Second, create a temporary result by using a derived table or common table expression (CTE)
- Third, apply the `PIVOT` operator.

Let's apply these steps in the following example.

First, select category name and product id from the `production.products` and `production.categories` tables as the base data for pivoting:

```
SELECT
```

```
    category_name,
    product_id
FROM
    production.products p
    INNER JOIN production.categories c
        ON c.category_id = p.category_id
```

Code language: SQL (Structured Query Language) (sql)

Second, create a temporary result set using a derived table:

```
SELECT * FROM (
    SELECT
        category_name,
        product_id
    FROM
        production.products p
        INNER JOIN production.categories c
            ON c.category_id = p.category_id
) t
```

Code language: SQL (Structured Query Language) (sql)

Third, apply the PIVOT operator:

```
SELECT * FROM
(
    SELECT
        category_name,
        product_id
    FROM
        production.products p
        INNER JOIN production.categories c
            ON c.category_id = p.category_id
) t
PIVOT(
    COUNT(product_id)
    FOR category_name IN (
        [Children Bicycles],
        [Comfort Bicycles],
        [Cruisers Bicycles],
        [Cyclocross Bicycles],
        [Electric Bikes],
        [Mountain Bikes],
```

```
        [Road Bikes])

) AS pivot_table;
```

Code language: SQL (Structured Query Language) (sql)

This query generates the following output:

Children Bicycles	Comfort Bicycles	Cruisers Bicycles	Cyclocross Bicycles	Electric Bikes	Mountain Bikes	Road Bikes
59	30	78	10	24	60	60

Now, any additional column which you add to the select list of the query that returns the base data will automatically form row groups in the pivot table. For example, you can add the model year column to the above query:

```
SELECT * FROM

(

    SELECT

        category_name,

        product_id,

        model_year

    FROM

        production.products p

        INNER JOIN production.categories c

            ON c.category_id = p.category_id

) t

PIVOT(
```

```
    COUNT(product_id)
    FOR category_name IN (
        [Children Bicycles],
        [Comfort Bicycles],
        [Cruisers Bicycles],
        [Cyclocross Bicycles],
        [Electric Bikes],
        [Mountain Bikes],
        [Road Bikes])
) AS pivot_table;
```

Code language: SQL (Structured Query Language) (sql)

Here is the output:

model_year	Children Bicycles	Comfort Bicycles	Cruisers Bicycles	Cyclocross Bicycles	Electric Bikes	Mountain Bikes	Road Bikes
2016	3	3	9	2	1	8	0
2017	19	10	19	2	2	21	12
2018	37	17	50	6	21	31	42
2019	0	0	0	0	0	0	6

Generating column values

In the above query, you had to type each category name in the parentheses after the `IN` operator manually. To avoid this, you can use the `QUOTENAME()` function to generate the category name list and copy them over the query.

First, generate the category name list:

```
DECLARE
```

```
    @columns NVARCHAR(MAX) = '';


SELECT
    @columns += QUOTENAME(category_name) + ','
FROM
    production.categories
ORDER BY
    category_name;


SET @columns = LEFT(@columns, LEN(@columns) - 1);


PRINT @columns;
```

Code language: SQL (Structured Query Language) (sql)

The output will look like this:

```
[Children Bicycles],[Comfort Bicycles],[Cruisers
Bicycles],[Cyclocross Bicycles],[Electric Bikes],[Mountain
Bikes],[Road Bikes]
```

Code language: CSS (css)

In this snippet:

- The `QUOTENAME()` function wraps the category name by the square brackets e.g., `[Children Bicycles]`
- The `LEFT()` function removes the last comma from the @columns string.

Second, copy the category name list from the output and paste it to the query.

Dynamic pivot tables

If you add a new category name to the `production.categories` table, you need to rewrite your query, which is not ideal. To avoid doing this, you can use dynamic SQL to make the pivot table dynamic.

In this query, instead of passing a fixed list of category names to the `PIVOT` operator, we construct the category name list and pass it to an SQL statement, and then execute this statement dynamically using the stored procedure `sp_executesql`.

```
DECLARE

    @columns NVARCHAR(MAX) = '',

    @sql     NVARCHAR(MAX) = '';



-- select the category names

SELECT

    @columns+=QUOTENAME(category_name) + ','

FROM

    production.categories

ORDER BY

    category_name;



-- remove the last comma
```

```
SET @columns = LEFT(@columns, LEN(@columns) - 1);


-- construct dynamic SQL
SET @sql ='
SELECT * FROM
(
    SELECT
        category_name,
        model_year,
        product_id
    FROM
        production.products p
        INNER JOIN production.categories c
            ON c.category_id = p.category_id
) t
PIVOT(
    COUNT(product_id)
    FOR category_name IN ('+ @columns +')
) AS pivot_table;';
```

```
-- execute the dynamic SQL

EXECUTE sp_executesql @sql;
```