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.

la principale differenza tra una **vista materializzata** e una **non materializzata** è che la prima **conserva fisicamente una copia dei dati, mentre la seconda esegue la query ogni volta che viene chiamata.** L**e viste materializzate possono portare a tempi di risposta più veloci, ma richiedono la gestione di aggiornamenti periodici per mantenere i dati sincronizzati con la fonte. Le viste non materializzate forniscono dati in tempo reale, ma possono richiedere più tempo per eseguire query complesse.** La scelta tra i due dipende dalle esigenze specifiche di prestazioni e aggiornamenti dei dati nel contesto dell'applicazione.

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

in SQL, un indice (index) è una struttura di dati che migliora la velocità delle operazioni di ricerca su una tabella. Gli indici vengono utilizzati per accelerare il recupero dei dati durante le query, riducendo il numero di letture del disco o di operazioni di scansione di una tabella.

Esistono due tipi principali di indici in SQL: clustered e non clustered.

1. **Clustered Index:**
    - In un indice raggruppato (clustered index), le righe della tabella sono ordinate fisicamente sulla base dell'ordine di indice. Ciò significa che i dati nella tabella sono fisicamente organizzati in base alla chiave dell'indice.
    - Una tabella può avere un solo indice raggruppato perché la disposizione fisica dei dati è determinata dall'ordine dell'indice. Se crei un nuovo indice raggruppato, sostituirà l'indice raggruppato esistente, se presente.
2. **Non-Clustered Index:**
    - In un indice non raggruppato (non-clustered index), le righe della tabella non sono ordinate fisicamente in base all'ordine dell'indice. L'indice fornisce una struttura separata che contiene un elenco di riferimenti alle posizioni delle righe nella tabella.
    - Una tabella può avere più di un indice non raggruppato. Gli indici non raggruppati forniscono una via di accesso rapido ai dati senza influire sull'ordine fisico delle righe nella tabella.

La principale differenza tra un indice raggruppato e uno non raggruppato è che, nel primo caso, le righe della tabella sono organizzate fisicamente in base all'ordine dell'indice, mentre, nel secondo caso, l'indice fornisce un modo efficiente per accedere ai dati senza specificare l'ordine fisico.

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;
```

VARIABILE

dichiarazione di variabili in SQL varia a seconda del sistema di gestione del database (DBMS) che stai utilizzando. Tuttavia, molti DBMS seguono uno standard SQL di base per dichiarare variabili. Ecco un esempio di come dichiarare una variabile in SQL usando la sintassi standard:

```
DECLARE @NomeVariabile TipoDato;
```

ESEMPIO:

```
DECLARE @MioNumero INT;
SET @MioNumero = 42;
SELECT @MioNumero AS Result;
```

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

Supponiamo di avere la seguente tabella:

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

Ecco un esempio di come utilizzare l'istruzione IF per verificare se un cliente è maggiore di 18 anni:

```
SELECT
```

```
  nome,
  cognome,
  email
FROM
  clienti
WHERE
  età > 18;
```

Questo comando restituisce tutti i clienti che hanno più di 18 anni.

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/

TABELLA ORIGINE

car_id	make	type	style	cost_$
1	Honda	Civic	Sedan	30000
2	Toyota	Corolla	Hatchback	25000
3	Ford	Explorer	SUV	40000
4	Chevrolet	Camaro	Coupe	36000
5	BMW	X5	SUV	55000
6	Audi	A4	Sedan	48000
7	Mercedes	C-Class	Coupe	60000
8	Nissan	Altima	Sedan	26000

Query:

```
SELECT '#Guadagni' AS Categoria,
    [Audi], [BMW], [Chevrolet], [Ford], [Honda], [Mercedes], [Nissan], [Toyota]
FROM (
    SELECT cost_$, make AS Tipi
    FROM cars
) AS t
PIVOT (
    SUM(cost_$)
    FOR Tipi IN ([Audi], [BMW], [Chevrolet], [Ford], [Honda], [Mercedes], [Nissan],
[Toyota])
) AS PIVOTTABLE
```

Result:

tipi	Audi	BMW	Chevrolet	Ford	Honda	Mercedes	Nissan	Toyota
#Guadagni	48000	55000	36000	40000	30000	60000	26000	25000

UNPIVOT:

Partendo dalla tabella prodotta dalla pivot

```
-- Unpivot dei dati
SELECT Categoria, Tipi, Costo
FROM tabella_pivot
UNPIVOT (
    Costo FOR Tipi IN ([Audi], [BMW], [Chevrolet], [Ford], [Honda], [Mercedes], [Nissan],
[Toyota])
) AS UnpivotedData;
```

Risultato:

Categoria	Tipi	Costo
#Guadagni	Audi	48000
#Guadagni	BMW	55000
#Guadagni	Chevrolet	36000

```
#Guadagni     Ford          40000
#Guadagni     Honda         30000
#Guadagni     Mercedes      60000
#Guadagni     Nissan        26000
#Guadagni     Toyota        25000
```