Comandi base

OFFSET e Fetch

I comandi OFFSET e FETCH vengono utilizzati per restituire un sottoinsieme di righe da una tabella.

- **OFFSET** specifica il numero di righe da saltare prima di restituire i risultati.
- **FETCH** specifica il numero di righe da restituire.

Pseudocodice

```
SELECT
  *
FROM
  tabella
OFFSET
  offset_row_count
ROWS
FETCH
  fetch_row_count
ROWS
```

Esempio

Supponiamo di avere una tabella prodotti con i seguenti dati:

id	nome	prezzo
1	Cappello	10
2	Pantaloni	20
3	Camicia	30
4	Scarpe	40

La seguente query restituirà le prime 10 righe della tabella, partendo dalla quarta riga:

```
SELECT
  *
```

```
FROM
  prodotti
OFFSET
  3
ROWS
FETCH
  10
ROWS
```

TOP WITH TIES

Il comando TOP WITH TIES è simile al comando TOP, ma restituisce anche le righe con lo stesso valore del campo specificato nella clausola ORDER BY.

Pseudocodice

```
SELECT
  *
FROM
  tabella
ORDER BY
  campo
TOP
  n_righe
WITH TIES
```

Esempio

Supponiamo di avere una tabella studenti con i seguenti dati:

id	nome	cognome	voto
1	Mario	Rossi	20
2	Anna	Bianchi	20
3	Giovanni	Verdi	19

La seguente query restituirà i primi due studenti con il voto più alto:

```
SELECT
  *
FROM
  studenti
ORDER BY
  voto
TOP
  2
WITH TIES
```

GROUP BY

Il comando GROUP BY raggruppa le righe di una tabella in base a un valore comune.

Pseudocodice

```
SELECT
  *
FROM
  tabella
GROUP BY
  campo
```

Esempio

Supponiamo di avere una tabella ordini con i seguenti dati:

id	cliente	prodotto	quantità
1	Mario	Cappello	1
2	Anna	Pantaloni	2
3	Giovanni	Camicia	3

La seguente query restituirà il numero di ordini per ogni cliente:

```
SELECT
  cliente,
  COUNT(*) AS numero_ordini
```

```
FROM
  ordini
GROUP BY
  cliente
```

HAVING

La clausola HAVING viene utilizzata per filtrare i risultati di una query GROUP BY.

Pseudocodice

```
SELECT
  *
FROM
  tabella
GROUP BY
  campo
HAVING
  condizione
```

Esempio

La seguente query restituirà il numero di ordini per ogni cliente, ma solo se il numero di ordini è maggiore di 2:

```
SELECT
  cliente,
  COUNT(*) AS numero_ordini
FROM
  ordini
GROUP BY
  cliente
HAVING
  numero_ordini > 2
```

GROUPING SET

Il comando GROUPING SET viene utilizzato per restituire più combinazioni di risultati da una query GROUP BY.

Pseudocodice

```
SELECT
  *
FROM
  tabella
GROUP BY
GROUPING SET
  (campo1, campo2),
  (campo1),
  (campo2)
```

Esempio

Supponiamo di avere una tabella ordini con i seguenti dati:

id	cliente	prodotto	quantità
1	Mario	Cappello	1
2	Anna	Pantaloni	2
3	Giovanni	Camicia	3

La seguente query restituirà il numero di ordini per ogni cliente e prodotto, nonché il numero totale di ordini:

```
SELECT
  cliente,
  prodotto,
  COUNT(*) AS numero_ordini
FROM
  ordini
GROUP BY
GROUPING SET
  (cliente, prodotto),
  (cliente),
  ()
```

CUBE

Il comando CUBE viene utilizzato per restituire tutte le combinazioni possibili di risultati da una query GROUP BY.

Pseudocodice

```
SELECT
  *
FROM
  tabella
GROUP BY
  campo1,
  campo2
WITH CUBE
```

Esempio

La seguente query restituirà il numero di ordini per ogni cliente e prodotto, nonché il numero totale di ordini:

```
SELECT
  cliente,
  prodotto,
  COUNT(*) AS numero_ordini
FROM
  ordini
GROUP BY
  cliente,
  prodotto
WITH CUBE
```

ROLLUP

Il comando ROLLUP viene utilizzato per restituire tutte le combinazioni di risultati da una query GROUP BY, partendo dai gruppi più grandi fino ai gruppi più piccoli.

Pseudocodice

```
SELECT
  *
FROM
  tabella
GROUP BY
  campo1,
  campo2
WITH ROLLUP
```

Esempio

La seguente query restituirà il numero di ordini per ogni cliente, il numero totale di ordini per ogni prodotto, e il numero totale di ordini:

```
SELECT
  cliente,
  prodotto,
  COUNT(*) AS numero_ordini
FROM
  ordini
GROUP BY
  cliente,
  prodotto
WITH ROLLUP
```

DIFFERENZA TRA ROLLUP E CUBE

Funzione: `ROLLUP` genera subtotals in modo gerarchico, partendo dal set di colonne più a sinistra nel `GROUP BY` e aggiungendo progressivamente le colonne a destra.

Funzione: `CUBE` genera subtotals per tutte le combinazioni possibili di colonne specificate nel `GROUP BY`.

Esempio con ROLLUP:

```
SELECT Prodotto, Regione, Anno, SUM(Importo) AS TotaleVendite
FROM Vendite
GROUP BY Prodotto, Regione, Anno WITH ROLLUP;
```

Questo genererà un risultato con subtotali gerarchici, partendo da subtotali per `Prodotto`, poi per `Prodotto, Regione`, poi per `Prodotto, Regione, Anno`, e infine un totale complessivo.

Esempio con CUBE:

```
SELECT Prodotto, Regione, Anno, SUM(Importo) AS TotaleVendite
FROM Vendite
GROUP BY Prodotto, Regione, Anno WITH CUBE;
```

Questo genererà un risultato con tutte le possibili combinazioni di subtotali, considerando tutte le combinazioni di `Prodotto`, `Regione`, e `Anno`, e infine un totale complessivo.

In breve, mentre entrambi ROLLUP e CUBE sono utilizzati per generare subtotali e totali durante l'aggregazione dei dati, ROLLUP è gerarchico e segue l'ordine delle colonne nel GROUP BY, mentre CUBE considera tutte le possibili combinazioni di colonne.

CREATE/DROP DATABASE

I comandi CREATE DATABASE e DROP DATABASE vengono utilizzati per creare e rimuovere database.

Pseudocodice

```
CREATE DATABASE nome_database

DROP DATABASE nome_database
```

Esempio

La seguente query creerà un database chiamato mio_database:

```
CREATE DATABASE mio_database

DROP DATABASE mio_database
```

CREATE/ALTER/DROP SCHEMA

I comandi CREATE SCHEMA, ALTER SCHEMA e DROP SCHEMA vengono utilizzati per creare, modificare e rimuovere schemi.

Pseudocodice

```
CREATE SCHEMA nome_schema

ALTER SCHEMA nome_schema

DROP SCHEMA nome_schema
```

Esempio

La seguente query creerà uno schema chiamato mio_schema:

```
CREATE SCHEMA mio_schema
```

La seguente query aggiungerà una tabella chiamata ordini allo schema mio_schema:

```
ALTER SCHEMA mio_schema
```

```
ADD TABLE ordini (
  id INT,
  cliente VARCHAR(255),
  prodotto VARCHAR(255),
  quantità INT
);
```

La seguente query rimuoverà lo schema mio_schema:

```
DROP SCHEMA mio_schema
```

CREATE\DROP\ALTER\RENAME\TRUNCATE TABLE

I comandi CREATE TABLE, DROP TABLE, ALTER TABLE, RENAME TABLE e TRUNCATE TABLE vengono utilizzati per gestire le tabelle di un database.

CREATE TABLE

Il comando CREATE TABLE viene utilizzato per creare una nuova tabella.

Pseudocodice

```
CREATE TABLE nome_tabella (
  colonna1 tipo_dato,
  colonna2 tipo_dato,
  ...
);
```

Esempio

La seguente query creerà una tabella chiamata ordini con le seguenti colonne:

```
CREATE TABLE ordini (
  id INT,
  cliente VARCHAR(255),
  prodotto VARCHAR(255),
  quantità INT
);
```

DROP TABLE

Il comando DROP TABLE viene utilizzato per rimuovere una tabella.

Pseudocodice

```
DROP TABLE nome_tabella;
```

Esempio

La seguente query rimuoverà la tabella ordini:

```
DROP TABLE ordini;
```

ALTER TABLE

Il comando ALTER TABLE viene utilizzato per modificare una tabella.

Pseudocodice

```
ALTER TABLE nome_tabella
ADD colonna1 tipo_dato,
ADD colonna2 tipo_dato,
...;

ALTER TABLE nome_tabella
DROP colonna1,
DROP colonna2,
...;

ALTER TABLE nome_tabella
MODIFY colonna1 tipo_dato,
MODIFY colonna2 tipo_dato,
...;
```

Esempio

La seguente query aggiungerà una colonna chiamata data alla tabella ordini:

```
ALTER TABLE ordini
ADD data DATE;
```

RENAME TABLE

Il comando RENAME TABLE viene utilizzato per rinominare una tabella.

Pseudocodice

```
RENAME TABLE nome_tabella TO nuovo_nome_tabella;
```

Esempio

La seguente query rinominerà la tabella ordini in ordini_aggiornati:

```
RENAME TABLE ordini TO ordini_aggiornati;
```

TRUNCATE TABLE

Il comando TRUNCATE TABLE viene utilizzato per eliminare tutti i dati da una tabella.

Pseudocodice

```
TRUNCATE TABLE nome_tabella;
```

Esempio

La seguente query rimuoverà tutti i dati dalla tabella ordini:

```
TRUNCATE TABLE ordini;
```

SEQUENCE

Il comando SEQUENCE viene utilizzato per creare una sequenza.

Pseudocodice

```
CREATE SEQUENCE nome_sequenza
INCREMENT BY 1
START WITH 1
MINVALUE 1
MAXVALUE 9999999999
CYCLE
NO CACHE;
```

Esempio

La seguente query creerà una sequenza chiamata id_ordini che inizia a 1 e incrementa di 1 a ogni valore successivo:

```
CREATE SEQUENCE id_ordini
```

```
INCREMENT BY 1
START WITH 1
MINVALUE 1
MAXVALUE 9999999999
CYCLE
NO CACHE;
```

TEMPORARY TABLE

Il comando TEMPORARY TABLE viene utilizzato per creare una tabella temporanea.

una temporary table, o tabella temporanea, è una tabella che viene creata e utilizzata temporaneamente durante l'esecuzione di una sessione o di un blocco di codice in un database. Questa tabella esiste solo per la durata della sessione o del blocco di codice e scompare automaticamente alla fine di tale periodo. Le temporary table offrono un modo per archiviare temporaneamente dati intermedi o di lavoro senza dover creare una tabella permanente nel database.

Pseudocodice

```
CREATE TEMPORARY TABLE nome_tabella (
  colonna1 tipo_dato,
  colonna2 tipo_dato,
  ...
);
```

Esempio

La seguente query creerà una tabella temporanea chiamata ordini_temporanei:

```
CREATE TEMPORARY TABLE ordini_temporanei (
  id INT,
  cliente VARCHAR(255),
  prodotto VARCHAR(255),
  quantità INT
);
```

SYNONYMS

Il comando SYNONYMS viene utilizzato per creare un sinonimo per una tabella, una vista o un'altra risorsa di database.

in SQL, un sinonimo (synonym) è un oggetto del database che fornisce un alias o un nome alternativo per un altro oggetto del database, come una tabella, una vista, una stored procedure o un altro sinonimo. L'uso principale di un sinonimo è semplificare e rendere più flessibile l'accesso agli oggetti del database.

Pseudocodice

```
CREATE SYNONYM nome_sinonimo FOR nome_risorsa;
```

Esempio

La seguente query creerà un sinonimo chiamato ordini_sinonimo per la tabella ordini:

```
CREATE SYNONYM ordini_sinonimo FOR ordini;
```

COALESCE

La funzione COALESCE viene utilizzata per restituire il primo valore non NULL di una lista di valori.

Pseudocodice

```
COALESCE(valore1, valore2, ..., valore_n)
```

Esempio

Supponiamo di avere una tabella ordini con le seguenti colonne:

id	cliente	prodotto	quantità

La seguente query restituirà il nome del cliente per l'ordine con l'id 1, se il valore della colonna cliente è non NULL, altrimenti restituirà il valore "Anonimo":

```
SELECT COALESCE(cliente, 'Anonimo')
FROM ordini
WHERE id = 1;
```

CASE

La clausola CASE viene utilizzata per eseguire un'azione in base al valore di un'espressione.

Pseudocodice

```
CASE
WHEN condizione THEN azione1
WHEN condizione THEN azione2
...
ELSE azione_default
END
```

Esempio

Supponiamo di avere una tabella ordini con le seguenti colonne:

id	cliente	prodotto	quantità

La seguente query restituirà la categoria del prodotto per l'ordine con l'id 1, in base alla quantità ordinata:

```
  SQL
SELECT
  CASE
    WHEN quantità > 10 THEN 'Prodotto grande'
    WHEN quantità > 5 THEN 'Prodotto medio'
    ELSE 'Prodotto piccolo'
  END AS categoria
FROM ordini
WHERE id = 1;
```

NULLIF

La funzione NULLIF viene utilizzata per restituire NULL se due espressioni hanno lo stesso valore, altrimenti restituisce la prima espressione.

Pseudocodice

```
NULLIF(espressione1, espressione2)
```

Esempio

Supponiamo di avere una tabella ordini con le seguenti colonne:

id	cliente	prodotto	quantità

La seguente query restituirà NULL se il valore della colonna prodotto è uguale a "Cappello", altrimenti restituirà il valore della colonna prodotto:

```
SELECT NULLIF(prodotto, 'Cappello')
FROM ordini;
```

IIF

La funzione IIF è simile alla clausola CASE, ma è più concisa.

Pseudocodice

```
IIF(condizione, azione1, azione2)
```

Esempio

L'esempio della clausola CASE può essere riscritto utilizzando la funzione IIF come segue:

```
SELECT
  IIF(quantità > 10, 'Prodotto grande',
    IIF(quantità > 5, 'Prodotto medio', 'Prodotto piccolo')) AS
categoria
FROM ordini
WHERE id = 1;
```

INSERT INTO

Il comando INSERT INTO viene utilizzato per inserire nuove righe in una tabella.

Pseudocodice

```
INSERT INTO tabella (colonna1, colonna2, ...)
```

```
VALUES (valore1, valore2, ...);
```

Esempio

Supponiamo di avere una tabella ordini con le seguenti colonne:

id	cliente	prodotto	quantità

```
INSERT INTO ordini (id, cliente, prodotto, quantità)
VALUES (1, 'Mario', 'Cappello', 1);


-- Esempio di inserimento dati da una tabella a un'altra
INSERT INTO NuovaTabella (Colonna1, Colonna2, Colonna3, ...)
SELECT Colonna1, Colonna2, Colonna3, ...
FROM VecchiaTabella;

-- Esempio di inserimento di tutti i valori da una tabella a un'altra
INSERT INTO TabellaDestinazione
SELECT * FROM TabellaOrigine;
```

UPDATE

Il comando UPDATE viene utilizzato per aggiornare le righe di una tabella.

Pseudocodice

```
UPDATE tabella
SET colonna1 = valore1,
  colonna2 = valore2,
  ...
WHERE condizione;
```

Esempio

Supponiamo di avere una tabella ordini con le seguenti colonne:

id	cliente	prodotto	quantità

```
UPDATE ordini
SET quantità = 2
```

```
WHERE cliente = 'Mario'
AND prodotto = 'Cappello';
```

DELETE

Il comando DELETE viene utilizzato per eliminare le righe da una tabella.

Pseudocodice

```
DELETE FROM tabella
WHERE condizione;
```

Esempio

Supponiamo di avere una tabella ordini con le seguenti colonne:

id	cliente	prodotto	quantità

La seguente query eliminerà tutte le righe dalla tabella ordini:

```
DELETE FROM ordini;
```

MERGE

https://www.sqlshack.com/understanding-the-sql-merge-statement/

Il comando MERGE viene utilizzato per combinare i dati da due tabelle.

Il comando `MERGE` in SQL Server è utilizzato per eseguire operazioni di inserimento, aggiornamento o eliminazione basate su una condizione di corrispondenza specificata tra la tabella di destinazione (target table) e la tabella di origine (source table). Questo comando è utile quando è necessario sincronizzare o aggiornare dati tra due tabelle in base a una condizione specifica.

Pseudocodice

```
MERGE target_table AS target
USING source_table AS source
```

```
ON condition
WHEN MATCHED THEN
    UPDATE SET column1 = value1, column2 = value2, ...
WHEN NOT MATCHED THEN
    INSERT (column1, column2, ...)
    VALUES (value1, value2, ...);
```

Dove:

- `target_table`: È la tabella di destinazione, cioè la tabella in cui si desidera eseguire le operazioni di inserimento, aggiornamento o eliminazione.
- `source_table`: È la tabella di origine, cioè la tabella che fornisce i dati per l'inserimento, l'aggiornamento o l'eliminazione.
- `condition`: È la condizione di corrispondenza che determina quando le righe nella tabella di destinazione corrispondono alle righe nella tabella di origine.
- `WHEN MATCHED`: Specifica cosa fare quando esiste una corrispondenza tra la tabella di destinazione e la tabella di origine.
- `WHEN NOT MATCHED`: Specifica cosa fare quando non esiste una corrispondenza tra la tabella di destinazione e la tabella di origine.

Esempio

Supponiamo di avere due tabelle ordini_vecchi e ordini_nuovi con le seguenti colonne:

id	cliente	prodotto	quantità

La seguente query aggiornerà le righe della tabella ordini_vecchi con i dati della tabella ordini_nuovi, se le due righe hanno lo stesso valore per la colonna id:

```
MERGE INTO ordini_vecchi
USING ordini_nuovi
ON ordini_vecchi.id = ordini_nuovi.id
WHEN MATCHED THEN UPDATE SET
  ordini_vecchi.cliente = ordini_nuovi.cliente,
  ordini_vecchi.prodotto = ordini_nuovi.prodotto,
  ordini_vecchi.quantità = ordini_nuovi.quantità;
```

altro esempio:

Supponiamo di avere una tabella di destinazione chiamata `TargetTable` e una tabella di origine chiamata `SourceTable`, entrambe con una colonna `ID`. Vogliamo aggiornare le righe esistenti in `TargetTable` e inserire le nuove righe dalla `SourceTable` in base all'`ID`. La query `MERGE` potrebbe apparire così:

```
MERGE TargetTable AS target
USING SourceTable AS source
ON target.ID = source.ID
WHEN MATCHED THEN
    UPDATE SET target.ColumnName = source.ColumnName
WHEN NOT MATCHED THEN
    INSERT (ID, ColumnName)
    VALUES (source.ID, source.ColumnName);
```

In questo esempio, le righe con lo stesso `ID` vengono aggiornate, mentre le nuove righe con `ID` non presente in `TargetTable` vengono inserite. La clausola `UPDATE SET` specifica quali colonne aggiornare quando una corrispondenza è trovata.

SUBQUERIES

Le sottoquery sono query contenute all'interno di altre query.

Pseudocodice

```
SELECT
  colonna1,
  colonna2,
  ...
FROM
  tabella
WHERE condizione = (
    SELECT
      colonna1,
      colonna2,
      ...
    FROM
      tabella
    WHERE condizione
  );
```

Esempio

Supponiamo di avere una tabella ordini con le seguenti colonne:

id	cliente	prodotto	quantità

La seguente query restituirà i nomi dei clienti che hanno effettuato più di un ordine:

```sql
SELECT
  cliente
FROM
  (
    SELECT
      cliente,
      COUNT(*) AS numero_ordini
    FROM
      ordini
    GROUP BY
      cliente
    HAVING
      numero_ordini > 1
  ) AS t;
```

SUBQUERIES CORRELATE

Le subquery correlate in SQL sono sottointerrogazioni in cui il risultato di una query dipende dai risultati di un'altra query esterna. In altre parole, una subquery correlata viene eseguita per ogni riga restituita dalla query esterna.

Ecco un esempio per spiegare meglio:

Supponiamo di avere due tabelle: `Ordini` e `DettagliOrdine`. La tabella `Ordini` contiene informazioni sugli ordini, mentre la tabella `DettagliOrdine` contiene i dettagli specifici di ogni ordine.

```sql
CREATE TABLE Ordini (
IDOrdine INT PRIMARY KEY,
DataOrdine DATE,
ClienteID INT
);
```

```sql
CREATE TABLE DettagliOrdine (
    IDDettaglio INT PRIMARY KEY,
    IDOrdine INT,
    ProdottoID INT,
    Quantita INT,
    Prezzo DECIMAL(10, 2)
);
```

supponiamo che vogliamo trovare tutti gli ordini in cui la quantità totale di prodotti è superiore a una certa soglia. In questo caso, possiamo utilizzare una subquery correlata per ottenere questo risultato. La query potrebbe apparire così:

```sql
SELECT IDOrdine, DataOrdine
```

```
FROM Ordini o

WHERE (
    SELECT SUM(Quantita)
    FROM DettagliOrdine
    WHERE IDOrdine = o.IDOrdine
) > 100;
```

La query esterna seleziona gli ordini dalla tabella `Ordini`.

1. La subquery correlata viene eseguita per ogni riga restituita dalla query esterna. La subquery calcola la somma delle quantità di prodotti per l'ordine specifico associato alla riga corrente della query esterna.
2. La clausola `WHERE` della query esterna filtra gli ordini in base alla condizione che la somma delle quantità di prodotti sia superiore a 100.

Le subquery correlate sono utili quando è necessario eseguire una query che coinvolge più tabelle e quando la condizione di filtraggio dipende dai risultati di una sottoquery basata su valori della riga corrente della query esterna.

LIKE / NOT LIKE

l'operatore `LIKE` viene utilizzato per confrontare una colonna di testo con un modello di stringa utilizzando caratteri jolly (wildcard) per rappresentare parte o tutto del testo. Le wildcard più comuni sono `%` (percentuale) e _

Esempio 1: Utilizzo di `%` come wildcard

-- Seleziona tutte le righe dove la colonna 'nome' inizia con 'Jo'

SELECT * FROM utenti WHERE nome LIKE 'Jo%';

In questo caso, la query selezionerà tutte le righe dalla tabella "utenti" dove il valore nella colonna "nome" inizia con 'Jo'.

Esempio 2: Utilizzo di `%` come wildcard a metà della stringa

-- Seleziona tutte le righe dove la colonna 'città' contiene 'burg'

```
SELECT * FROM indirizzi WHERE città LIKE '%burg%';
```

Questa query seleziona tutte le righe dalla tabella "indirizzi" dove il valore nella colonna "città" contiene la sottostringa 'burg' da qualche parte nel mezzo.

Esempio 3: Utilizzo di _ come wildcard per un singolo carattere

```
-- Seleziona tutte le righe dove la colonna 'password' ha il secondo carattere 'a'

SELECT * FROM utenti WHERE password LIKE '_a%';
```

Questa query seleziona tutte le righe dalla tabella "utenti" dove il secondo carattere nella colonna "password" è 'a'.

La sintassi `[...]` all'interno di un'espressione `LIKE` in SQL viene utilizzata per definire un insieme di caratteri possibili in una posizione specifica di una stringa. Gli insiemi di caratteri possono contenere un elenco di caratteri o un intervallo di caratteri. Questo è noto come "character class" o "caratteri set" e può essere utilizzato per definire pattern più specifici nelle tue query.

Esempio 4: Utilizzo di `[...]` per un set di caratteri

```
-- Seleziona tutte le righe dove la colonna 'nome' inizia con 'A' o 'B'

SELECT * FROM utenti WHERE nome LIKE '[AB]%';
```

Questa query selezionerà tutte le righe dalla tabella "utenti" dove il valore nella colonna "nome" inizia con 'A' o 'B'.

Esempio 5: Utilizzo di ^ all'interno di `[...]` per negare un set di caratteri

```
-- Seleziona tutte le righe dove la colonna 'città' non inizia con 'N','e', o 'w'

SELECT * FROM indirizzi WHERE città LIKE '[^New]%';
```

Questa query selezionerà tutte le righe dalla tabella "indirizzi" dove il valore nella colonna "città" non inizia con 'New'.

Altri esempi:

```
-- Seleziona tutte le righe dove la colonna 'codice' inizia con un carattere compreso tra 'A' e 'F'

SELECT * FROM prodotti WHERE codice LIKE '[A-F]%';


-- Seleziona tutte le righe dove la colonna 'nome' inizia con 'A' o 'B'

SELECT * FROM utenti WHERE nome LIKE '[AB]%';


-- Seleziona tutte le righe dove la colonna 'città' inizia con una lettera diversa da 'N', 'e' o 'w'

SELECT * FROM indirizzi WHERE città LIKE '[^New]%';


-- Seleziona tutte le righe dove la colonna 'nome' contiene una vocale

SELECT * FROM utenti WHERE nome LIKE '[aeiou]%';
```