EBook Gratuito

APPRENDIMENTO SQL

Free unaffiliated eBook created from
Stack Overflow contributors.

#sql

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sql

It is an unofficial and free SQL ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official SQL.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con SQL

Osservazioni

SQL è Structured Query Language utilizzato per gestire i dati in un sistema di database relazionale. Diversi venditori hanno migliorato la lingua e hanno una varietà di sapori per la lingua.

NB: questo tag si riferisce esplicitamente allo **standard SQL ISO / ANSI** ; non a qualsiasi implementazione specifica di tale standard.

Versioni

Versione	Nome corto	Standard	Data di rilascio
1986	SQL-86	ANSI X3.135-1986, ISO 9075: 1987	1986-01-01
1989	SQL-89	ANSI X3.135-1989, ISO / IEC 9075: 1989	1989/01/01
1992	SQL-92	ISO / IEC 9075: 1992	1992/01/01
1999	SQL: 1999	ISO / IEC 9075: 1999	1999/12/16
2003	SQL: 2003	ISO / IEC 9075: 2003	2003-12-15
2006	SQL: 2006	ISO / IEC 9075: 2006	2006-06-01
2008	SQL: 2008	ISO / IEC 9075: 2008	2008-07-15
2011	SQL: 2011	ISO / IEC 9075: 2011	2011-12-15
2016	SQL: 2016	ISO / IEC 9075: 2016	2016/12/01

Examples

Panoramica

Structured Query Language (SQL) è un linguaggio di programmazione per scopi speciali progettato per gestire i dati contenuti in un sistema di gestione di database relazionali (RDBMS). I linguaggi di tipo SQL possono essere utilizzati anche in sistemi di gestione dei flussi di dati relazionali (RDSMS) o in database "non solo SQL" (NoSQL).

SQL comprende 3 principali sotto-lingue:

1. Data Definition Language (DDL): per creare e modificare la struttura del database;
2. Data Manipulation Language (DML): per eseguire operazioni di lettura, inserimento,

aggiornamento e cancellazione sui dati del database;
3. Data Control Language (DCL): per controllare l'accesso ai dati memorizzati nel database.

Articolo SQL su Wikipedia

Le operazioni core di DML sono Create, Read, Update e Delete (in breve CRUD) che vengono eseguite dalle istruzioni `INSERT` , `SELECT` , `UPDATE` e `DELETE` .
Esiste anche un'istruzione `MERGE` (aggiunta di recente) che può eseguire tutte e 3 le operazioni di scrittura (INSERT, UPDATE, DELETE).

Articolo CRUD su Wikipedia

Molti database SQL sono implementati come sistemi client / server; il termine "SQL server" descrive un tale database.
Allo stesso tempo, Microsoft crea un database denominato "SQL Server". Mentre quel database parla un dialetto di SQL, le informazioni specifiche per quel database non sono in argomento in questo tag ma appartengono alla documentazione di SQL Server .

Leggi Iniziare con SQL online: https://riptutorial.com/it/sql/topic/184/iniziare-con-sql

Capitolo 2: ADERIRE

introduzione

JOIN è un metodo per combinare (unire) informazioni da due tabelle. Il risultato è un insieme di colonne cucite da entrambe le tabelle, definite dal tipo di join (INTERNO / ESTERNO / CROCE e SINISTRA / DESTRA / PIENA, spiegato di seguito) e criteri di unione (come si riferiscono le righe di entrambe le tabelle).

Una tabella può essere unita a se stessa oa qualsiasi altra tabella. Se è necessario accedere a informazioni da più di due tabelle, è possibile specificare più join in una clausola FROM.

Sintassi

- `[ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } } ] JOIN`

Osservazioni

I join, come suggerisce il loro nome, sono un modo per interrogare i dati di più tabelle in modo congiunto, con le righe che mostrano colonne prese da più di una tabella.

Examples

Giuntura interna esplicita di base

Un join di base (chiamato anche "inner join") esegue query sui dati da due tabelle, con la loro relazione definita in una clausola `join` .

L'esempio seguente selezionerà i nomi dei dipendenti (FName) dalla tabella Impiegati e il nome del reparto per cui lavorano (Nome) dalla tabella Departments:

```
SELECT Employees.FName, Departments.Name
FROM   Employees
JOIN   Departments
ON Employees.DepartmentId = Departments.Id
```

Ciò restituirebbe il seguente dal database di esempio :

Employees.FName	Departments.Name
Giacomo	HR
John	HR
Richard	I saldi

Unirsi implicitamente

Join possono anche essere eseguite da avere più tabelle nel `from` clausola separati da virgole `,` e definisce il rapporto tra loro in `where` clausola. Questa tecnica è chiamata un'adesione implicita (poiché in realtà non contiene una clausola `join`).

Tutti gli RDBMS lo supportano, ma di solito la sintassi viene sconsigliata. I motivi per cui è una cattiva idea usare questa sintassi sono:

- È possibile ottenere crossover accidentali che restituiscono risultati errati, soprattutto se si hanno molti join nella query.
- Se intendevi un cross join, allora non è chiaro dalla sintassi (scrivi invece CROSS JOIN), e qualcuno potrebbe cambiarlo durante la manutenzione.

Nell'esempio seguente verranno selezionati i nomi dei dipendenti e il nome dei reparti per cui lavorano:

```
SELECT e.FName, d.Name
FROM   Employee e, Departments d
WHERE  e.DeptartmentId = d.Id
```

Ciò restituirebbe il seguente dal database di esempio :

d.nome	d.Name
Giacomo	HR
John	HR
Richard	I saldi

Left Outer Join

Un Join esterno sinistro (noto anche come Join sinistro o Join esterno) è un Join che assicura che tutte le righe della tabella sinistra siano rappresentate; se non esiste una riga corrispondente dalla tabella di destra, i suoi campi corrispondenti sono `NULL` .

L'esempio seguente selezionerà tutti i reparti e il primo nome dei dipendenti che lavorano in quel dipartimento. I reparti senza dipendenti sono ancora restituiti nei risultati, ma avranno NULL per il nome del dipendente:

```
SELECT          Departments.Name, Employees.FName
FROM            Departments
LEFT OUTER JOIN Employees
ON              Departments.Id = Employees.DepartmentId
```

Ciò restituirebbe il seguente dal database di esempio :

Departments.Name	Employees.FName
HR	Giacomo
HR	John
HR	Johnathon
I saldi	Michael
Tech	NULLO

Quindi come funziona?

Ci sono due tabelle nella clausola FROM:

Id	FName	LName	Numero di telefono	ManagerID	DepartmentID	Stipendio	Data di assunzione
1	Giacomo	fabbro	1234567890	NULLO	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	fabbro	1212121212	2	1	500	24-07-2016

e

Id	Nome
1	HR
2	I saldi
3	Tech

Innanzitutto un prodotto *cartesiano* viene creato dalle due tabelle dando una tabella intermedia. I record che soddisfano i criteri di join (*Departments.Id = Employees.DepartmentId*) sono evidenziati in grassetto; questi sono passati alla fase successiva della query.

Poiché questo è un GIÙ ESTERNO SINISTRO, tutti i record vengono restituiti dal lato SINISTRA del join (Dipartimenti), mentre a tutti i record sul lato DESTRO viene assegnato un indicatore NULL se non corrispondono ai criteri di join. Nella tabella seguente, questo restituirà **Tech** con `NULL`

Id	Nome	Id	FName	LName	Numero di telefono	ManagerID	DepartmentID	Stipendio	Da as
1	**HR**	**1**	**Giacomo**	**fabbro**	**1234567890**	**NULLO**	**1**	**1000**	**01**
1	**HR**	**2**	**John**	**Johnson**	**2468101214**	**1**	**1**	**400**	**23**
1	HR	3	Michael	Williams	1357911131	1	2	600	12
1	**HR**	**4**	**Johnathon**	**fabbro**	**1212121212**	**2**	**1**	**500**	**24**
2	I saldi	1	Giacomo	fabbro	1234567890	NULLO	1	1000	01
2	I saldi	2	John	Johnson	2468101214	1	1	400	23
2	**I saldi**	**3**	**Michael**	**Williams**	**1357911131**	**1**	**2**	**600**	**12**
2	I saldi	4	Johnathon	fabbro	1212121212	2	1	500	24
3	Tech	1	Giacomo	fabbro	1234567890	NULLO	1	1000	01
3	Tech	2	John	Johnson	2468101214	1	1	400	23
3	Tech	3	Michael	Williams	1357911131	1	2	600	12
3	Tech	4	Johnathon	fabbro	1212121212	2	1	500	24

Infine, ogni espressione utilizzata all'interno della clausola **SELECT** viene valutata per restituire il nostro tavolo finale:

Departments.Name	Employees.FName
HR	Giacomo
HR	John
I saldi	Richard
Tech	NULLO

Self Join

Una tabella può essere unita a se stessa, con righe diverse che si corrispondono a seconda delle condizioni. In questo caso d'uso, è necessario utilizzare alias per distinguere le due occorrenze della tabella.

Nell'esempio seguente, per ciascun Dipendente nella tabella Dipendenti del database di esempio , viene restituito un record contenente il nome del dipendente insieme al nome corrispondente

corrispondente del manager del dipendente. Poiché i manager sono anche dipendenti, la tabella è unita a se stessa:

```
SELECT
    e.FName AS "Employee",
    m.FName AS "Manager"
FROM
    Employees e
JOIN
    Employees m
    ON e.ManagerId = m.Id
```

Questa query restituirà i seguenti dati:

Dipendente	Manager
John	Giacomo
Michael	Giacomo
Johnathon	John

Quindi come funziona?

La tabella originale contiene questi record:

Id	FName	LName	Numero di telefono	ManagerID	DepartmentID	Stipendio	Data di assunzione
1	Giacomo	fabbro	1234567890	NULLO	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	fabbro	1212121212	2	1	500	24-07-2016

La prima azione consiste nel creare un prodotto *cartesiano* di tutti i record nelle tabelle utilizzate nella clausola **FROM** . In questo caso è la tabella Dipendenti due volte, quindi la tabella intermedia sarà simile a questa (ho rimosso tutti i campi non utilizzati in questo esempio):

e.Id	d.nome	e.ManagerId	m.Id	m.FName	m.ManagerId
1	Giacomo	NULLO	1	Giacomo	NULLO
1	Giacomo	NULLO	2	John	1
1	Giacomo	NULLO	3	Michael	1

e.Id	d.nome	e.ManagerId	m.Id	m.FName	m.ManagerId
1	Giacomo	NULLO	4	Johnathon	2
2	John	1	1	Giacomo	NULLO
2	John	1	2	John	1
2	John	1	3	Michael	1
2	John	1	4	Johnathon	2
3	Michael	1	1	Giacomo	NULLO
3	Michael	1	2	John	1
3	Michael	1	3	Michael	1
3	Michael	1	4	Johnathon	2
4	Johnathon	2	1	Giacomo	NULLO
4	Johnathon	2	2	John	1
4	Johnathon	2	3	Michael	1
4	Johnathon	2	4	Johnathon	2

L'azione successiva è quella di mantenere solo i record che soddisfano i criteri di **unire,** in modo che qualsiasi record in cui il alias `e` tavolo `ManagerId` uguale al alias `m` tavolo `Id` :

e.Id	d.nome	e.ManagerId	m.Id	m.FName	m.ManagerId
2	John	1	1	Giacomo	NULLO
3	Michael	1	1	Giacomo	NULLO
4	Johnathon	2	2	John	1

Quindi, ciascuna espressione utilizzata all'interno della clausola **SELECT** viene valutata per restituire questa tabella:

d.nome	m.FName
John	Giacomo
Michael	Giacomo
Johnathon	John

Infine, i nomi delle colonne `e.FName` e `m.FName` sono sostituiti dai loro nomi di colonne di alias, assegnati con l'operatore AS :

Dipendente	Manager
John	Giacomo
Michael	Giacomo
Johnathon	John

CROSS JOIN

Cross join fa un prodotto cartesiano dei due membri, un prodotto cartesiano significa che ogni riga di una tabella è combinata con ciascuna riga della seconda tabella nel join. Ad esempio, se `TABLEA` ha 20 righe e `TABLEB` ha 20 righe, il risultato sarà `20*20 = 400` righe di output.

Utilizzo del database di esempio

```
SELECT d.Name, e.FName
FROM   Departments d
CROSS JOIN Employees e;
```

Che restituisce:

d.Name	d.nome
HR	Giacomo
HR	John
HR	Michael
HR	Johnathon
I saldi	Giacomo
I saldi	John
I saldi	Michael
I saldi	Johnathon
Tech	Giacomo
Tech	John
Tech	Michael
Tech	Johnathon

Si consiglia di scrivere un CROSS JOIN esplicito se si vuole fare un join cartesiano, per evidenziare che questo è ciò che si desidera.

Partecipare a una sottoquery

Unire una sottoquery viene spesso utilizzato quando si desidera ottenere dati aggregati da una tabella figlio / dettagli e visualizzarli insieme ai record dalla tabella padre / intestazione. Ad esempio, è possibile che si desideri ottenere un conteggio dei record figlio, una media di alcune colonne numeriche nei record figlio o la riga superiore o inferiore in base a una data o un campo numerico. In questo esempio vengono utilizzati alias, il che rende più facile leggere le query quando sono coinvolte più tabelle. Ecco come appare un join di subquery piuttosto tipico. In questo caso, recuperiamo tutte le righe dalla tabella padre Ordini d'acquisto e recuperiamo solo la prima riga per ogni record padre della tabella figlio PurchaseOrderLineItems.

```
SELECT po.Id, po.PODate, po.VendorName, po.Status, item.ItemNo,
  item.Description, item.Cost, item.Price
FROM PurchaseOrders po
LEFT JOIN
     (
       SELECT l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price, Min(l.id) as Id
       FROM PurchaseOrderLineItems l
       GROUP BY l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price
     ) AS item ON item.PurchaseOrderId = po.Id
```

CROSS APPLY & LATERAL JOIN

Un tipo molto interessante di JOIN è il LATERAL JOIN (nuovo in PostgreSQL 9.3+), che è anche conosciuto come APPLICAZIONE CROSS / APPLICAZIONE ESTERNA in SQL-Server e Oracle.

L'idea di base è che una funzione valutata a livello di tabella (o subquery inline) venga applicata per ogni riga che aggiungi.

Ciò consente, ad esempio, di unire solo la prima voce corrispondente in un'altra tabella.
La differenza tra un join normale e uno laterale risiede nel fatto che puoi utilizzare una colonna che hai precedentemente aggiunto **alla sottoquery** che hai "CROSS APPLY".

Sintassi:

PostgreSQL 9.3+

> sinistra | giusto | inner JOIN **LATERAL**

Server SQL:

> CROCE | **DOMANDA** ESTERNA

`INNER JOIN LATERAL` è uguale a `CROSS APPLY`
e `LEFT JOIN LATERAL` è lo stesso di `OUTER APPLY`

Esempio di utilizzo (PostgreSQL 9.3+):

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989


LEFT JOIN LATERAL
(
    SELECT
         --MAP_CTCOU_UID
         MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
   FROM T_MAP_Contacts_Ref_OrganisationalUnit
   WHERE MAP_CTCOU_SoftDeleteStatus = 1
   AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
    AND
    (
        (__in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
        (__in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
   ORDER BY MAP_CTCOU_DateFrom
   LIMIT 1
) AS FirstOE
```

E per SQL-Server

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989

-- CROSS APPLY -- = INNER JOIN
OUTER APPLY    -- = LEFT JOIN
(
    SELECT TOP 1
         --MAP_CTCOU_UID
         MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
   FROM T_MAP_Contacts_Ref_OrganisationalUnit
   WHERE MAP_CTCOU_SoftDeleteStatus = 1
   AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
    AND
    (
        (@in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
```

```
        (@in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
   ORDER BY MAP_CTCOU_DateFrom
) AS FirstOE
```

FULL JOIN

Un tipo di JOIN meno conosciuto è FULL JOIN.
(Nota: FULL JOIN non è supportato da MySQL come da 2016)

Un FULL OUTER JOIN restituisce tutte le righe dalla tabella di sinistra e tutte le righe dalla tabella di destra.

Se nella tabella a sinistra ci sono delle righe che non hanno corrispondenze nella tabella di destra, o se ci sono delle righe nella tabella di destra che non hanno corrispondenze nella tabella di sinistra, anche quelle righe saranno elencate.

Esempio 1 :

```
SELECT * FROM Table1

FULL JOIN Table2
     ON 1 = 2
```

Esempio 2:

```
SELECT
     COALESCE(T_Budget.Year, tYear.Year) AS RPT_BudgetInYear
    ,COALESCE(T_Budget.Value, 0.0) AS RPT_Value
FROM T_Budget

FULL JOIN tfu_RPT_All_CreateYearInterval(@budget_year_from, @budget_year_to) AS tYear
      ON tYear.Year = T_Budget.Year
```

Si noti che se si utilizzano eliminazioni software, sarà necessario controllare nuovamente lo stato di eliminazione software nella clausola WHERE (poiché FULL JOIN si comporta in modo simile a UNION);
È facile trascurare questo piccolo fatto, dal momento che hai inserito AP_SoftDeleteStatus = 1 nella clausola join.

Inoltre, se stai facendo un FULL JOIN, di solito devi consentire NULL nella clausola WHERE; dimenticarsi di consentire a NULL su un valore avrà gli stessi effetti di un join INNER, che è qualcosa che non vuoi se stai facendo un FULL JOIN.

Esempio:

```
SELECT
     T_AccountPlan.AP_UID
    ,T_AccountPlan.AP_Code
    ,T_AccountPlan.AP_Lang_EN
```

```
    ,T_BudgetPositions.BUP_Budget
    ,T_BudgetPositions.BUP_UID
    ,T_BudgetPositions.BUP_Jahr
FROM T_BudgetPositions

FULL JOIN T_AccountPlan
    ON T_AccountPlan.AP_UID = T_BudgetPositions.BUP_AP_UID
    AND T_AccountPlan.AP_SoftDeleteStatus = 1

WHERE (1=1)
AND (T_BudgetPositions.BUP_SoftDeleteStatus = 1 OR T_BudgetPositions.BUP_SoftDeleteStatus IS
NULL)
AND (T_AccountPlan.AP_SoftDeleteStatus = 1 OR T_AccountPlan.AP_SoftDeleteStatus IS NULL)
```

JOIN ricorsivi

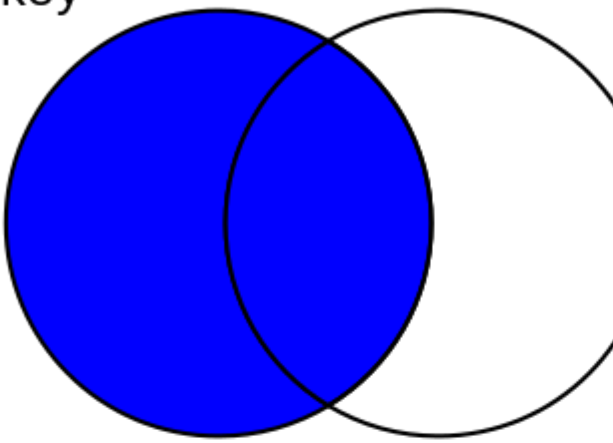
I join ricorsivi vengono spesso utilizzati per ottenere dati padre-figlio. In SQL, sono implementati con espressioni di tabella comuni ricorsive, ad esempio:

```
WITH RECURSIVE MyDescendants AS (
    SELECT Name
    FROM People
    WHERE Name = 'John Doe'

    UNION ALL

    SELECT People.Name
    FROM People
    JOIN MyDescendants ON People.Name = MyDescendants.Parent
)
SELECT * FROM MyDescendants;
```

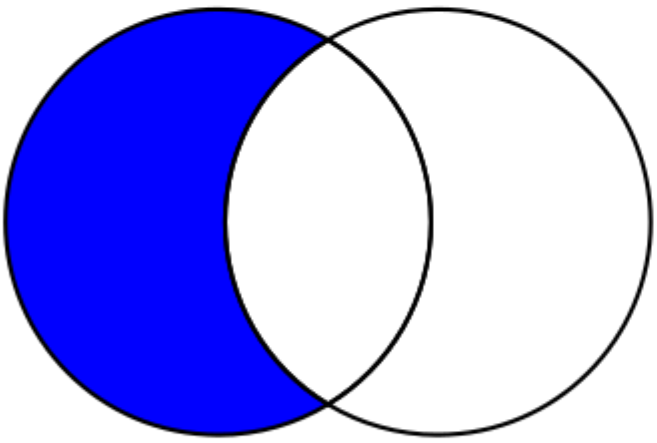
Differenze tra join interni / esterni

SQL ha vari tipi di join per specificare se le righe (non) corrispondenti sono incluse nel risultato: `INNER JOIN` , `LEFT OUTER JOIN` , `RIGHT OUTER JOIN` e `FULL OUTER JOIN` (le parole chiave `INNER` e `OUTER` sono facoltative). La figura seguente evidenzia le differenze tra questi tipi di join: l'area blu rappresenta i risultati restituiti dal join e l'area bianca rappresenta i risultati che il join non restituirà.

```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
```



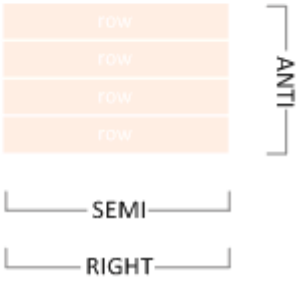
```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
WHERE B.key IS NULL
```



fai attenzione se stai usando NOT IN su una colonna NULABILE! Maggiori dettagli qui .

Giusto Anti Semi Join

Include le righe giuste che **non** corrispondono alle righe a sinistra.




```
SELECT * FROM B WHERE Y NOT IN (SELECT X FROM A);

Y
-------
Tim
Vincent
```

Come puoi vedere, non esiste una sintassi NOT IN dedicata per l'anti semi join sinistro e destro - otteniamo l'effetto semplicemente cambiando le posizioni della tabella all'interno del testo SQL.

Cross Join

Un prodotto cartesiano di tutti lasciato con tutte le righe giuste.

```
SELECT * FROM A CROSS JOIN B;

X      Y
-----  -------
Amy    Lisa
```

```
John   Lisa
Lisa   Lisa
Marco  Lisa
Phil   Lisa
Amy    Marco
John   Marco
Lisa   Marco
Marco  Marco
Phil   Marco
Amy    Phil
John   Phil
Lisa   Phil
Marco  Phil
Phil   Phil
Amy    Tim
John   Tim
Lisa   Tim
Marco  Tim
Phil   Tim
Amy    Vincent
John   Vincent
Lisa   Vincent
Marco  Vincent
Phil   Vincent
```

Cross join equivale a un join interno con condizione join che corrisponde sempre, quindi la query seguente avrebbe restituito lo stesso risultato:

```
SELECT * FROM A JOIN B ON 1 = 1;
```

Self-Join

Questo semplicemente indica una tabella che si unisce a se stessa. Un self-join può essere uno dei tipi di join discussi sopra. Ad esempio, questo è un self-join interno:

```
SELECT * FROM A A1 JOIN A A2 ON LEN(A1.X) < LEN(A2.X);

X     X
----  -----
Amy   John
Amy   Lisa
Amy   Marco
John  Marco
Lisa  Marco
Phil  Marco
Amy   Phil
```

Leggi ADERIRE online: https://riptutorial.com/it/sql/topic/261/aderire

Capitolo 3: AGGIORNARE

Sintassi

- *Tabella* UPDATE
  SET *nome_colonna* = *valore* , *nome_colonna2* = *valore_2* , ..., *nome_colonna_n* = *valore_n*
  Condizione WHERE (*operatore logico* condizione_n)

Examples

Aggiornamento di tutte le righe

Questo esempio utilizza la tabella Cars dai database di esempio.

```
UPDATE Cars
SET Status = 'READY'
```

Questa istruzione imposterà la colonna "status" di tutte le righe della tabella "Cars" in "READY" perché non ha una clausola `WHERE` per filtrare il set di righe.

Aggiornamento delle righe specificate

Questo esempio utilizza la tabella Cars dai database di esempio.

```
UPDATE
    Cars
SET
    Status = 'READY'
WHERE
    Id = 4
```

Questa affermazione imposta lo stato della riga di "Auto" con ID 4 su "PRONTO".

`WHERE` clausola `WHERE` contiene un'espressione logica che viene valutata per ogni riga. Se una riga soddisfa i criteri, il suo valore viene aggiornato. Altrimenti, una riga rimane invariata.

Modifica dei valori esistenti

Questo esempio utilizza la tabella Cars dai database di esempio.

```
UPDATE Cars
SET TotalCost = TotalCost + 100
WHERE Id = 3 or Id = 4
```

Le operazioni di aggiornamento possono includere i valori correnti nella riga aggiornata. In questo semplice esempio `TotalCost` viene incrementato di 100 per due righe:

- La TotalCost di Car # 3 è aumentata da 100 a 200
- La TotalCost di Car # 4 è aumentata da 1254 a 1354

Il nuovo valore di una colonna può essere derivato dal suo valore precedente o dal valore di qualsiasi altra colonna nella stessa tabella o tabella unita.

AGGIORNA con i dati di un'altra tabella

Gli esempi che seguono riempiono un `PhoneNumber` di `PhoneNumber` per qualsiasi dipendente che sia anche un `Customer` e attualmente non ha un numero di telefono impostato nella tabella dei `Employees` .

(Questi esempi utilizzano le tabelle Dipendenti e Clienti dai database di esempio.)

SQL standard

Aggiornamento utilizzando una sottoquery correlata:

```
UPDATE
    Employees
SET PhoneNumber =
    (SELECT
         c.PhoneNumber
     FROM
         Customers c
     WHERE
         c.FName = Employees.FName
         AND c.LName = Employees.LName)
WHERE Employees.PhoneNumber IS NULL
```

SQL: 2003

Aggiorna usando `MERGE` :

```
MERGE INTO
    Employees e
USING
    Customers c
ON
    e.FName = c.Fname
    AND e.LName = c.LName
    AND e.PhoneNumber IS NULL
WHEN MATCHED THEN
   UPDATE
      SET PhoneNumber = c.PhoneNumber
```

server SQL

Aggiornamento con `INNER JOIN` :

```
UPDATE
    Employees
SET
    PhoneNumber = c.PhoneNumber
FROM
    Employees e
INNER JOIN Customers c
        ON e.FName = c.FName
        AND e.LName = c.LName
WHERE
    PhoneNumber IS NULL
```

Catturare record aggiornati

A volte uno vuole catturare i record che sono stati appena aggiornati.

```
CREATE TABLE #TempUpdated(ID INT)

Update TableName SET Col1 = 42
    OUTPUT inserted.ID INTO #TempUpdated
    WHERE Id > 50
```

Leggi AGGIORNARE online: https://riptutorial.com/it/sql/topic/321/aggiornare

Capitolo 4: Algebra relazionale

Examples

Panoramica

Algebra relazionale non è un linguaggio *SQL in* piena *regola* , ma piuttosto un modo per acquisire una comprensione teorica dell'elaborazione relazionale. In quanto tale, non dovrebbe fare riferimento a entità fisiche come tabelle, record e campi; dovrebbe fare riferimenti a costrutti astratti come relazioni, tuple e attributi. Detto questo, non userò i termini accademici in questo documento e rimarrò fedele ai più noti termini laici - tabelle, record e campi.

Un paio di regole dell'algebra relazionale prima di iniziare:

- Gli operatori utilizzati nell'algebra relazionale funzionano su intere tabelle anziché su singoli record.
- Il risultato di un'espressione relazionale sarà sempre una tabella (questa è chiamata la *proprietà closure*)

In questo documento mi riferirò alle seguenti due tabelle:

Departments

ID	Dept
1	Production
2	Quality Control

People

ID	PersonName	StartYear	ManagerID	DepartmentID
1	Darren	2005		1
2	David	2006	1	1
3	Burt	2006	1	1
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

SELEZIONARE

L'operatore **select** restituisce un sottoinsieme della tabella principale.
seleziona *<table>* ***dove*** *<condizione>*

Ad esempio, esamina l'espressione:

seleziona Persone **dove** DepartmentID = 2

Questo può essere scritto come:

$$\sigma_{\text{DepartmentID} = 2}(\text{People})$$

Ciò comporterà una tabella i cui record comprendono tutti i record nella tabella *Persone in* cui il valore *DepartmentID* è uguale a 2:

ID	PersonName	StartYear	ManagerID	DepartmentID
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

Le condizioni possono anche essere unite per limitare ulteriormente l'espressione:

selezionare Persone **dove** StartYear> 2005 **e** DepartmentID = 2

risulterà nella seguente tabella:

ID	PersonName	StartYear	ManagerID	DepartmentID
5	Fred	2008	4	2

PROGETTO

L'operatore di **progetto** restituirà valori di campo distinti da una tabella.
progetto *<table>* ***su*** *<field list>*

Ad esempio, esaminare la seguente espressione:
progetto People **over** StartYear

Questo può essere scritto come:

$$\pi_{\text{StartYear}}\,(\text{People})$$

Ciò comporterà una tabella che comprende i valori distinti contenuti nel campo *StartYear* della tabella *People* .

StartYear
2005
2006
2004
2008

I valori duplicati vengono rimossi dalla tabella risultante a causa della *proprietà di chiusura che* crea una tabella relazionale: tutti i record in una tabella relazionale devono essere distinti.

Se l' *elenco dei campi* comprende più di un singolo campo, la tabella risultante è una versione distinta di questi campi.
progetto People **over** StartYear, DepartmentID restituirà:

StartYear	DepartmentID
2005	1
2006	1
2004	2
2008	2
2005	2

Un record viene rimosso a causa della duplicazione di *StartYear* 2006 e 1 *DepartmentID* .

DANDO

Le espressioni relazionali possono essere concatenate assegnando un nome alle singole espressioni utilizzando la parola chiave **giving** o incorporando un'espressione in un'altra.

*<espressione algebra relazionale> che **fornisce** <nome alias>*

Ad esempio, considera le seguenti espressioni:
seleziona Persone **dove** DepartmentID = 2 che **dà** A
progetto A **su** PersonName che **dà** B

Questo risulterà nella tabella B sottostante, con la tabella A che è il risultato della prima espressione.

A

ID	PersonName	StartYear	ManagerID	DepartmentID
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

B

PersonName
Sarah
Fred
Joanne

La prima espressione viene valutata e alla tabella risultante viene fornito l'alias A. Questa tabella viene quindi utilizzata all'interno della seconda espressione per fornire alla tabella finale un alias di B.

Un altro modo di scrivere questa espressione è di sostituire il nome alias della tabella nella seconda espressione con l'intero testo della prima espressione racchiuso tra parentesi:
progetto (**selezionare** People **where** DepartmentID = 2) **su** PersonName che **fornisce** B

Questa è chiamata *espressione annidata* .

JOIN NATURALE

Un join naturale attacca due tabelle usando un campo comune condiviso tra le tabelle.

unisciti a *<table 1>* ***e*** *<table 2>* ***dove*** *<field 1> = <field 2>*
supponendo che <campo 1> sia in <tabella 1> e <campo 2> sia in <tabella 2>.

Ad esempio, la seguente espressione di join si unirà a *People* and *Departments in* base alle colonne *DepartmentID* e *ID* nelle rispettive tabelle:
unisciti a Persone **e** dipartimenti **dove** DepartmentID = ID

ID	PersonName	StartYear	ManagerID	DepartmentID	Dept
1	Darren	2005		1	Production
2	David	2006	1	1	Production
3	Burt	2006	1	1	Production
4	Sarah	2004		2	Quality Control
5	Fred	2008	4	2	Quality Control
6	Joanne	2005	4	2	Quality Control

Si noti che viene visualizzato solo *DepartmentID* dalla tabella *People* e non l' *ID* dalla tabella *Department* . È necessario mostrare solo uno dei campi confrontati, che generalmente è il nome del campo della prima tabella dell'operazione di join.

Sebbene non mostrato in questo esempio, è possibile che le tabelle di unione possano generare due campi con la stessa intestazione. Ad esempio, se avessi usato il *nome dell'intestazione* per identificare i campi *PersonName* e *Dept* (ad es. Per identificare il nome della persona e il nome del dipartimento). Quando si verifica questa situazione, utilizziamo il nome della tabella per qualificare i nomi dei campi usando la notazione dot: *People.Name* e *Departments.Name*

join combinato con **select** e **project** possono essere usati insieme per ottenere informazioni:

unisciti a *Persone* ***e*** *dipartimenti* ***dove*** *DepartmentID = ID che* ***dà*** *A*
selezionare *A* ***dove*** *StartYear = 2005* ***e*** *Dept = 'Produzione'* ***dando*** *B*
progetto *B* ***su*** *PersonName che* ***dà*** *C*

o come espressione combinata:

progetto *(* ***selezionare*** *(* ***unire*** *People* ***and*** *Departments* ***dove*** *DepartmentID = ID)* ***dove*** *StartYear = 2005* ***e*** *Dept = 'Production')* ***su*** *PersonName che* ***fornisce*** *C*

Ciò comporterà questa tabella:

PersonName
Darren

ALIAS

DIVIDERE

UNIONE

INTERSEZIONE

DIFFERENZA

AGGIORNAMENTO (: =)

VOLTE

Leggi Algebra relazionale online: https://riptutorial.com/it/sql/topic/7311/algebra-relazionale

Capitolo 5: ALTER TABLE

introduzione

Il comando ALTER in SQL viene utilizzato per modificare la colonna / il vincolo in una tabella

Sintassi

- ALTER TABLE [nome_tabella] ADD [nome_colonna] [tipo dati]

Examples

Aggiungi colonna / e

```
ALTER TABLE Employees
ADD StartingDate date NOT NULL DEFAULT GetDate(),
    DateOfBirth date NULL
```

L'istruzione precedente aggiungerebbe colonne denominate `StartingDate` che non possono essere NULL con valore predefinito come data corrente e `DateOfBirth` che può essere NULL nella tabella Impiegati .

Drop Column

```
ALTER TABLE Employees
DROP COLUMN salary;
```

Questo non solo cancellerà le informazioni da quella colonna, ma farà cadere lo stipendio della colonna dai dipendenti della tabella (la colonna non esisterà più).

Drop Constraint

```
ALTER TABLE Employees
DROP CONSTRAINT DefaultSalary
```

Elimina un vincolo chiamato DefaultSalary dalla definizione della tabella dei dipendenti.

Nota: - Assicurarsi che i vincoli della colonna vengano eliminati prima di rilasciare una colonna.

Aggiungi vincolo

```
ALTER TABLE Employees
ADD CONSTRAINT DefaultSalary DEFAULT ((100)) FOR [Salary]
```

Questo aggiunge un vincolo chiamato DefaultSalary che specifica un valore predefinito di 100 per la colonna Salary.

È possibile aggiungere un vincolo a livello di tabella.

Tipi di vincoli

- Chiave primaria: impedisce un record duplicato nella tabella
- Chiave esterna: punta a una chiave primaria da un'altra tabella
- Not Null: impedisce l'immissione di valori nulli in una colonna
- Unico - identifica in modo univoco ogni record nella tabella
- Predefinito: specifica un valore predefinito
- Verifica: limita gli intervalli di valori che possono essere inseriti in una colonna

Per ulteriori informazioni sui vincoli, consultare la documentazione di Oracle .

Alter Column

```
ALTER TABLE Employees
ALTER COLUMN StartingDate DATETIME NOT NULL DEFAULT (GETDATE())
```

Questa query alterare il tipo di dati della colonna di `StartingDate` e modificarlo dal semplice `date` a `datetime` e impostare di default la data corrente.

Aggiungi chiave primaria

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID)
```

Questo aggiungerà una chiave primaria alla tabella Dipendenti `ID` del campo. Includendo più di un nome di colonna tra parentesi e ID verrà creata una chiave primaria composta. Quando si aggiungono più di una colonna, i nomi delle colonne devono essere separati da virgole.

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID, FName)
```

Leggi ALTER TABLE online: https://riptutorial.com/it/sql/topic/356/alter-table

Capitolo 6: applicare croce, applicare esterno

Examples

Domande di base su CROSS APPLY e ESTERNO APPLY

L'applicazione verrà utilizzata quando la tabella ha valore di funzione nell'espressione corretta.

creare una tabella dipartimento per contenere informazioni sui reparti. Quindi creare una tabella Employee che contiene informazioni sui dipendenti. Si noti che ogni dipendente appartiene a un reparto, pertanto la tabella Employee ha integrità referenziale con la tabella Department.

La prima query seleziona i dati dalla tabella del dipartimento e utilizza APPLICA CROSS per valutare la tabella Employee per ogni record della tabella Department. La seconda query unisce semplicemente la tabella Department con la tabella Employee e vengono prodotti tutti i record corrispondenti.

```
SELECT *
FROM Department D
CROSS APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
INNER JOIN Employee E
  ON D.DepartmentID = E.DepartmentID
```

Se guardi i risultati che hanno prodotto, è lo stesso identico set di risultati; In che cosa differisce da un JOIN e come aiuta a scrivere query più efficienti.

La prima query in Script # 2 seleziona i dati dalla tabella Department e utilizza OUTER APPLY per valutare la tabella Employee per ogni record della tabella Department. Per quelle righe per le quali non esiste una corrispondenza nella tabella Employee, tali righe contengono valori NULL come si può vedere nel caso delle righe 5 e 6. La seconda query utilizza semplicemente un JOINT OUTER SINISTRO tra la tabella Dipartimento e la tabella Employee. Come previsto, la query restituisce tutte le righe dalla tabella Department; anche per quelle righe per le quali non vi è alcuna corrispondenza nella tabella Dipendente.

```
SELECT *
FROM Department D
OUTER APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
```

```
FROM Department D
LEFT OUTER JOIN Employee E
  ON D.DepartmentID = E.DepartmentID
GO
```

Anche se le due query precedenti restituiscono le stesse informazioni, il piano di esecuzione sarà leggermente diverso. Ma a livello di costi non ci sarà molta differenza.

Ora arriva il momento di vedere dove è realmente richiesto l'operatore APPLY. In Script # 3, sto creando una funzione con valori di tabella che accetta DepartmentID come parametro e restituisce tutti i dipendenti che appartengono a questo reparto. La query successiva seleziona i dati dalla tabella di dipartimento e utilizza APPLICA CROSS per unirsi alla funzione che abbiamo creato. Passa il DepartmentID per ogni riga dall'espressione della tabella esterna (nel nostro caso la tabella Department) e valuta la funzione per ogni riga simile a una sottoquery correlata. La query successiva utilizza l'APPLICAZIONE ESTERNA al posto di APPLICAZIONI CROSS e quindi, a differenza di APPLICAZIONI CROSS, che restituisce solo dati correlati, l'APPLICAZIONE ESTERNA restituisce anche dati non correlati, posizionando NULL nelle colonne mancanti.

```
CREATE FUNCTION dbo.fn_GetAllEmployeeOfADepartment (@DeptID AS int)
RETURNS TABLE
AS
  RETURN
  (
  SELECT
    *
  FROM Employee E
  WHERE E.DepartmentID = @DeptID
  )
GO
SELECT
  *
FROM Department D
CROSS APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
SELECT
  *
FROM Department D
OUTER APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
```

Quindi ora, se ti stai chiedendo, possiamo usare un semplice join al posto delle query sopra? Quindi la risposta è NO, se si sostituisce CROSS / OUTER APPLY nelle query sopra con INTERNO JOIN / LEFT OUTER JOIN, specificare la clausola ON (qualcosa come 1 = 1) ed eseguire la query, si otterrà "L'identificatore di più parti" D.DepartmentID "non può essere vincolato." errore. Questo perché con JOINs il contesto di esecuzione della query esterna è diverso dal contesto di esecuzione della funzione (o una tabella derivata) e non è possibile associare un valore / variabile dalla query esterna alla funzione come parametro. Quindi l'operatore APPLY è richiesto per tali query.

Leggi applicare croce, applicare esterno online: https://riptutorial.com/it/sql/topic/2516/applicare-croce--applicare-esterno

Capitolo 7: ASTUCCIO

introduzione

L'espressione CASE viene utilizzata per implementare la logica if-then.

Sintassi

- CASE input_expression
  QUANDO compare1 POI risultato1
  [WHEN compare2 THEN result2] ...
  [Risultato ELSEX]
  FINE
- ASTUCCIO
  WHEN condition1 THEN result1
  [WHEN condition2 THEN result2] ...
  [Risultato ELSEX]
  FINE

Osservazioni

L' *espressione CASE semplice* restituisce il primo risultato il cui valore `compareX` è uguale a `input_expression` .

L' *espressione CASE cercata* restituisce il primo risultato di cui `conditionX` è vera.

Examples

CASO ricercato in SELEZIONA (corrisponde ad un'espressione booleana)

Il CASE *cercato* restituisce risultati quando un'espressione *booleana* è VERA.

(Questo differisce dal caso semplice, che può solo verificare l'equivalenza con un input.)

```
SELECT Id, ItemId, Price,
  CASE WHEN Price < 10 THEN 'CHEAP'
       WHEN Price < 20 THEN 'AFFORDABLE'
       ELSE 'EXPENSIVE'
  END AS PriceRating
FROM ItemSales
```

Id	Numero identificativo dell'oggetto	Prezzo	PriceRating
1	100	34.5	COSTOSO
2	145	2.3	A BUON MERCATO

Id	Numero identificativo dell'oggetto	Prezzo	PriceRating
3	100	34.5	COSTOSO
4	100	34.5	COSTOSO
5	145	10	CONVENIENTE

Usa CASE per COUNT il numero di righe in una colonna corrisponde a una condizione.

Caso d'uso

`CASE` può essere usato insieme a `SUM` per restituire un conteggio di solo quegli articoli che corrispondono a una condizione predefinita. (È simile a `COUNTIF` in Excel.)

Il trucco consiste nel restituire risultati binari che indicano le corrispondenze, quindi gli "1" restituiti per le voci corrispondenti possono essere sommati per un conteggio del numero totale di corrispondenze.

Data questa tabella `ItemSales` , supponiamo che tu voglia conoscere il numero totale di articoli che sono stati classificati come "costosi":

Id	Numero identificativo dell'oggetto	Prezzo	PriceRating
1	100	34.5	COSTOSO
2	145	2.3	A BUON MERCATO
3	100	34.5	COSTOSO
4	100	34.5	COSTOSO
5	145	10	CONVENIENTE

domanda

```
SELECT
    COUNT(Id) AS ItemsCount,
    SUM ( CASE
            WHEN PriceRating = 'Expensive' THEN 1
            ELSE 0
          END
        ) AS ExpensiveItemsCount
FROM ItemSales
```

risultati:

ItemsCount	ExpensiveItemsCount
5	3

Alternativa:

```
SELECT
    COUNT(Id) as ItemsCount,
    SUM (
        CASE PriceRating
            WHEN 'Expensive' THEN 1
            ELSE 0
        END
       ) AS ExpensiveItemsCount
FROM ItemSales
```

Stenografia CASE in SELECT

La variante abbreviata di `CASE` valuta un'espressione (solitamente una colonna) rispetto a una serie di valori. Questa variante è un po 'più corta e salva ripetutamente l'espressione valutata più volte. La clausola `ELSE` può ancora essere utilizzata, tuttavia:

```
SELECT Id, ItemId, Price,
  CASE Price WHEN 5  THEN 'CHEAP'
             WHEN 15 THEN 'AFFORDABLE'
             ELSE         'EXPENSIVE'
  END as PriceRating
FROM ItemSales
```

Una parola di cautela. È importante rendersi conto che quando si utilizza la variante breve l'intera istruzione viene valutata in ogni `WHEN` . Pertanto la seguente dichiarazione:

```
SELECT
    CASE ABS(CHECKSUM(NEWID())) % 4
        WHEN 0 THEN 'Dr'
        WHEN 1 THEN 'Master'
        WHEN 2 THEN 'Mr'
        WHEN 3 THEN 'Mrs'
    END
```

può produrre un risultato `NULL` . Questo perché in ogni `WHEN NEWID()` viene chiamato di nuovo con un nuovo risultato. Equivalente a:

```
SELECT
    CASE
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 0 THEN 'Dr'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 1 THEN 'Master'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 2 THEN 'Mr'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 3 THEN 'Mrs'
    END
```

Pertanto può perdere tutti i casi `WHEN` e risultare `NULL` .

CASO in una clausola ORDINE DI

Possiamo usare 1,2,3 .. per determinare il tipo di ordine:

```
SELECT * FROM DEPT
ORDER BY
CASE DEPARTMENT
      WHEN 'MARKETING' THEN  1
      WHEN 'SALES' THEN 2
      WHEN 'RESEARCH' THEN 3
      WHEN 'INNOVATION' THEN 4
      ELSE        5
      END,
      CITY
```

ID	REGIONE	CITTÀ	DIPARTIMENTO	EMPLOYEES_NUMBER
12	Nuova Inghilterra	Boston	MARKETING	9
15	ovest	San Francisco	MARKETING	12
9	Midwest	Chicago	I SALDI	8
14	Mid-Atlantic	New York	I SALDI	12
5	ovest	Los Angeles	RICERCA	11
10	Mid-Atlantic	Philadelphia	RICERCA	13
4	Midwest	Chicago	INNOVAZIONE	11
2	Midwest	Detroit	RISORSE UMANE	9

Utilizzo di CASE in UPDATE

campione sugli aumenti di prezzo:

```
UPDATE ItemPrice
SET Price = Price *
  CASE ItemId
    WHEN 1 THEN 1.05
    WHEN 2 THEN 1.10
    WHEN 3 THEN 1.15
    ELSE 1.00
  END
```

Utilizzare CASE per i valori NULL ordinati per ultimi

in questo modo '0' che rappresenta i valori noti sono classificati per primi, '1' che rappresenta i valori NULL sono ordinati per ultimi:

```
SELECT ID
      ,REGION
      ,CITY
      ,DEPARTMENT
      ,EMPLOYEES_NUMBER
  FROM DEPT
  ORDER BY
  CASE WHEN REGION IS NULL THEN 1
  ELSE 0
  END,
  REGION
```

ID	REGIONE	CITTÀ	DIPARTIMENTO	EMPLOYEES_NUMBER
10	Mid-Atlantic	Philadelphia	RICERCA	13
14	Mid-Atlantic	New York	I SALDI	12
9	Midwest	Chicago	I SALDI	8
12	Nuova Inghilterra	Boston	MARKETING	9
5	ovest	Los Angeles	RICERCA	11
15	NULLO	San Francisco	MARKETING	12
4	NULLO	Chicago	INNOVAZIONE	11
2	NULLO	Detroit	RISORSE UMANE	9

CASE nella clausola ORDER BY per ordinare i record per il valore più basso di 2 colonne

Immagina di aver bisogno di ordinare i record per il valore più basso di una delle due colonne. Alcuni database potrebbero utilizzare una funzione `MIN()` o `LEAST()` non aggregata per questo (`... ORDER BY MIN(Date1, Date2)`), ma in SQL standard, è necessario utilizzare un'espressione `CASE` .

Il `CASE` espressione nella domanda sotto esamina i `Date1` e `Date2` colonne, assegni quale colonna ha il valore più basso, e ordina i record a seconda di questo valore.

Dati di esempio

Id	Data1	Data2
1	2017/01/01	2017/01/31
2	2017/01/31	2017/01/03
3	2017/01/31	2017/01/02

Id	Data1	Data2
4	2017/01/06	2017/01/31
5	2017/01/31	2017/01/05
6	2017/01/04	2017/01/31

domanda

```
SELECT Id, Date1, Date2
FROM YourTable
ORDER BY CASE
           WHEN COALESCE(Date1, '1753-01-01') < COALESCE(Date2, '1753-01-01') THEN Date1
           ELSE Date2
         END
```

risultati

Id	Data1	Data2
1	**2017/01/01**	2017/01/31
3	2017/01/31	**2017/01/02**
2	2017/01/31	**2017/01/03**
6	**2017/01/04**	2017/01/31
5	2017/01/31	**2017/01/05**
4	**2017/01/06**	2017/01/31

Spiegazione

Come vedi riga con `Id = 1` è il primo, che a causa `Date1` hanno record di più basso da tutta la tabella `2017-01-01` , riga in cui `Id = 3` è il secondo che, a causa `Date2` equivale a `2017-01-02` che è secondo valore più basso da tavolo e così via.

Così abbiamo ordinato i record dal `2017-01-01` al `2017-01-06` ascendente e nessuna cura su cui una colonna `Date1` o `Date2` sono quei valori.

Leggi ASTUCCIO online: https://riptutorial.com/it/sql/topic/456/astuccio

Capitolo 8: Blocchi di esecuzione

Examples

Usando BEGIN ... END

```
BEGIN
  UPDATE Employees SET PhoneNumber = '5551234567' WHERE Id = 1;
  UPDATE Employees SET Salary = 650 WHERE Id = 3;
END
```

Leggi Blocchi di esecuzione online: https://riptutorial.com/it/sql/topic/1632/blocchi-di-esecuzione

Capitolo 9: Chiavi esterne

Examples

Creazione di una tabella con una chiave esterna

In questo esempio abbiamo una tabella esistente, `SuperHeros` .

Questa tabella contiene un `ID` chiave primaria.

Aggiungeremo un nuovo tavolo per memorizzare i poteri di ciascun supereroe:

```
CREATE TABLE HeroPowers
(
    ID int NOT NULL PRIMARY KEY,
    Name nvarchar(MAX) NOT NULL,
    HeroId int REFERENCES SuperHeros(ID)
)
```

La colonna `HeroId` è una **chiave estranea** al tavolo `SuperHeros` .

Chiavi esterne

I vincoli Foreign Keys assicurano l'integrità dei dati, facendo rispettare i valori in una tabella che devono corrispondere ai valori di un'altra tabella.

Un esempio di dove è richiesta una chiave straniera è: in un'università, un corso deve appartenere a un dipartimento. Il codice per questo scenario è:

```
CREATE TABLE Department (
    Dept_Code        CHAR (5)     PRIMARY KEY,
    Dept_Name        VARCHAR (20) UNIQUE
);
```

Inserire valori con la seguente dichiarazione:

```
INSERT INTO Department VALUES ('CS205', 'Computer Science');
```

La seguente tabella conterrà le informazioni degli argomenti offerti dalla sezione Informatica:

```
CREATE TABLE Programming_Courses (
    Dept_Code       CHAR(5),
    Prg_Code        CHAR(9) PRIMARY KEY,
    Prg_Name        VARCHAR (50) UNIQUE,
    FOREIGN KEY (Dept_Code) References Department(Dept_Code)
);
```

(Il tipo di dati della chiave esterna deve corrispondere al tipo di dati della chiave di riferimento.)

Il vincolo di chiave esterna nella colonna `Dept_Code` consente valori solo se sono già presenti nella tabella di riferimento, `Department` . Ciò significa che se provi ad inserire i seguenti valori:

```
INSERT INTO Programming_Courses Values ('CS300', 'FDB-DB001', 'Database Systems');
```

il database genererà un errore di violazione di chiave esterna, poiché `CS300` non esiste nella tabella del `Department` . Ma quando provi un valore chiave che esiste:

```
INSERT INTO Programming_Courses VALUES ('CS205', 'FDB-DB001', 'Database Systems');
INSERT INTO Programming_Courses VALUES ('CS205', 'DB2-DB002', 'Database Systems II');
```

quindi il database consente questi valori.

Alcuni suggerimenti per l'utilizzo di chiavi esterne

- Una chiave esterna deve fare riferimento a una chiave UNICA (o PRIMARIA) nella tabella padre.
- L'inserimento di un valore NULL in una colonna Chiave esterna non genera un errore.
- I vincoli di chiave esterna possono fare riferimento a tabelle all'interno dello stesso database.
- I vincoli di chiave esterna possono fare riferimento a un'altra colonna nella stessa tabella (riferimento automatico).

Leggi Chiavi esterne online: https://riptutorial.com/it/sql/topic/1533/chiavi-esterne

Capitolo 10: Chiavi primarie

Sintassi

- MySQL: CREATE TABLE Employees (Id int NOT NULL, PRIMARY KEY (Id), ...);
- Altro: CREATE TABLE Employees (Id int NOT NULL PRIMARY KEY, ...);

Examples

Creazione di una chiave primaria

```
CREATE TABLE Employees (
    Id int NOT NULL,
    PRIMARY KEY (Id),
    ...
);
```

Questo creerà la tabella Impiegati con 'Id' come chiave primaria. La chiave primaria può essere utilizzata per identificare in modo univoco le righe di una tabella. È consentita una sola chiave primaria per tabella.

Una chiave può anche essere composta da uno o più campi, la cosiddetta chiave composta, con la seguente sintassi:

```
CREATE TABLE EMPLOYEE (
    e1_id INT,
    e2_id INT,
    PRIMARY KEY (e1_id, e2_id)
)
```

Utilizzo dell'incremento automatico

Molti database consentono di aumentare automaticamente il valore della chiave primaria quando viene aggiunta una nuova chiave. Questo assicura che ogni chiave sia diversa.

MySQL

```
CREATE TABLE Employees (
    Id int NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (Id)
);
```

PostgreSQL

```
CREATE TABLE Employees (
    Id SERIAL PRIMARY KEY
);
```

server SQL

```
CREATE TABLE Employees (
    Id int NOT NULL IDENTITY,
    PRIMARY KEY (Id)
);
```

SQLite

```
CREATE TABLE Employees (
    Id INTEGER PRIMARY KEY
);
```

Leggi Chiavi primarie online: https://riptutorial.com/it/sql/topic/505/chiavi-primarie

Capitolo 11: Clausola

Examples

Semplice clausola IN

Per ottenere i record con **uno qualsiasi** degli `id` indicati

```
select *
from products
where id in (1,8,3)
```

La query sopra è uguale a

```
select *
from products
where id = 1
   or id = 8
   or id = 3
```

Utilizzo della clausola IN con una sottoquery

```
SELECT *
FROM customers
WHERE id IN (
    SELECT DISTINCT customer_id
    FROM orders
);
```

Quanto sopra ti darà tutti i clienti che hanno ordini nel sistema.

Leggi Clausola online: https://riptutorial.com/it/sql/topic/3169/clausola

Capitolo 12: CLAUSOLA ESISTENTE

Examples

CLAUSOLA ESISTENTE

Tabella clienti

Id	Nome di battesimo	Cognome
1	Ozgur	Ozturk
2	Youssef	Medi
3	Henry	Tai

Tabella degli ordini

Id	Identificativo del cliente	Quantità
1	2	123.50
2	3	14.80

Ottieni tutti i clienti con almeno un ordine

```
SELECT * FROM Customer WHERE EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId=Customer.Id
)
```

Risultato

Id	Nome di battesimo	Cognome
2	Youssef	Medi
3	Henry	Tai

Ottieni tutti i clienti senza alcun ordine

```
SELECT * FROM Customer WHERE NOT EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId = Customer.Id
)
```

Risultato

Id	Nome di battesimo	Cognome
1	Ozgur	Ozturk

Scopo

`EXISTS` , `IN` e `JOIN` potrebbero a volte essere utilizzati per lo stesso risultato, tuttavia, non sono uguali:

- `EXISTS` dovrebbero essere usati per verificare se un valore esiste in un'altra tabella
- `IN` dovrebbe essere usato per la lista statica
- `JOIN` dovrebbe essere usato per recuperare i dati dalla (e) tabella (e) di altri (s)

Leggi CLAUSOLA ESISTENTE online: https://riptutorial.com/it/sql/topic/7933/clausola-esistente

Capitolo 13: Commenti

Examples

Commenti a riga singola

I commenti a riga singola sono preceduti da `--` , e vanno fino alla fine della riga:

```
SELECT *
FROM Employees -- this is a comment
WHERE FName = 'John'
```

Commenti su più righe

I commenti del codice multilinea sono racchiusi in `/* ... */` :

```
/* This query
   returns all employees */
SELECT *
FROM Employees
```

È anche possibile inserire un commento di questo tipo nel mezzo di una riga:

```
SELECT /* all columns: */ *
FROM Employees
```

Leggi Commenti online: https://riptutorial.com/it/sql/topic/1597/commenti

Capitolo 14: CONCEDERE e REVOCARE

Sintassi

- GRANT [privilege1] [, [privilege2] ...] ON [table] TO [grantee1] [, [grantee2] ...] [CON OPZIONE DI CONCESSIONE]
- REVOKE [privilege1] [, [privilege2] ...] ON [table] FROM [grantee1] [, [grantee2] ...]

Osservazioni

Concedere permessi agli utenti. Se viene specificata l' `WITH GRANT OPTION` , il beneficiario ottiene inoltre il privilegio di concedere l'autorizzazione o revocare le autorizzazioni concesse in precedenza.

Examples

Concedere / revocare i privilegi

```
GRANT SELECT, UPDATE
ON Employees
TO User1, User2;
```

Concedere l'autorizzazione `User1` e `User2` per eseguire operazioni `SELECT` e `UPDATE` sul tavolo `Employees` .

```
REVOKE SELECT, UPDATE
ON Employees
FROM User1, User2;
```

Revoca da `User1` e `User2` l'autorizzazione per eseguire operazioni `SELECT` e `UPDATE` sulla tabella Dipendenti.

Leggi CONCEDERE e REVOCARE online: https://riptutorial.com/it/sql/topic/5574/concedere-e-revocare

Capitolo 15: CREA FUNZIONE

Sintassi

- CREATE FUNCTION nome_funzione ([list_of_paramenters]) RETURNS return_data_type AS BEGIN function_body RETURN scalar_expression END

Parametri

Discussione	Descrizione
function_name	il nome della funzione
list_of_paramenters	parametri che la funzione accetta
return_data_type	digita quella funzione. Alcuni tipi di dati SQL
function_body	il codice della funzione
scalar_expression	valore scalare restituito dalla funzione

Osservazioni

CREATE FUNCTION crea una funzione definita dall'utente che può essere utilizzata quando si esegue una query SELECT, INSERT, UPDATE o DELETE. Le funzioni possono essere create per restituire una singola variabile o una singola tabella.

Examples

Crea una nuova funzione

```
CREATE FUNCTION FirstWord (@input varchar(1000))
RETURNS varchar(1000)
AS
BEGIN
    DECLARE @output varchar(1000)
    SET @output = SUBSTRING(@input, 0, CASE CHARINDEX(' ', @input)
        WHEN 0 THEN LEN(@input) + 1
        ELSE CHARINDEX(' ', @input)
    END)

    RETURN @output
END
```

Questo esempio crea una funzione denominata **FirstWord** , che accetta un parametro varchar e restituisce un altro valore varchar.

Leggi CREA FUNZIONE online: https://riptutorial.com/it/sql/topic/2437/crea-funzione

Capitolo 16: CREA il database

Sintassi

- CREATE DATABASE dbname;

Examples

CREA il database

Un database viene creato con il seguente comando SQL:

```
CREATE DATABASE myDatabase;
```

Ciò creerebbe un database vuoto denominato myDatabase in cui è possibile creare tabelle.

Leggi CREA il database online: https://riptutorial.com/it/sql/topic/2744/crea-il-database

Capitolo 17: CREA TABELLA

introduzione

L'istruzione CREATE TABLE viene utilizzata per creare una nuova tabella nel database. Una definizione di tabella consiste in un elenco di colonne, i loro tipi e eventuali vincoli di integrità.

Sintassi

- CREATE TABLE tableName ([ColumnName1] [datatype1] [, [ColumnName2] [datatype2] ...])

Parametri

Parametro	Dettagli
tableName	Il nome del tavolo
colonne	Contiene una "enumerazione" di tutte le colonne della tabella. Vedi **Creare una nuova tabella** per maggiori dettagli.

Osservazioni

I nomi delle tabelle devono essere unici.

Examples

Crea una nuova tabella

È possibile creare una tabella `Employees` base contenente un ID e il nome e il cognome del dipendente insieme al relativo numero di telefono

```
CREATE TABLE Employees(
    Id int identity(1,1) primary key not null,
    FName varchar(20) not null,
    LName varchar(20) not null,
    PhoneNumber varchar(10) not null
);
```

Questo esempio è specifico di Transact-SQL

`CREATE TABLE` crea una nuova tabella nel database, seguita dal nome della tabella, `Employees`

Questo è seguito dall'elenco dei nomi delle colonne e delle loro proprietà, come l'ID

```
Id int identity(1,1) not null
```

Valore	Senso
`Id`	il nome della colonna.
`int`	è il tipo di dati.
`identity(1,1)`	afferma che la colonna avrà valori generati automaticamente a partire da 1 e incrementando di 1 per ogni nuova riga.
`primary key`	afferma che tutti i valori in questa colonna avranno valori univoci
`not null`	afferma che questa colonna non può avere valori nulli

Crea tabella da Seleziona

Potresti voler creare un duplicato di una tabella:

```
CREATE TABLE ClonedEmployees AS SELECT * FROM Employees;
```

È possibile utilizzare qualsiasi altra funzionalità di un'istruzione SELECT per modificare i dati prima di passarli alla nuova tabella. Le colonne della nuova tabella vengono create automaticamente in base alle righe selezionate.

```
CREATE TABLE ModifiedEmployees AS
SELECT Id, CONCAT(FName," ",LName) AS FullName FROM Employees
WHERE Id > 10;
```

Duplica un tavolo

Per duplicare una tabella, fai semplicemente quanto segue:

```
CREATE TABLE newtable LIKE oldtable;
INSERT newtable SELECT * FROM oldtable;
```

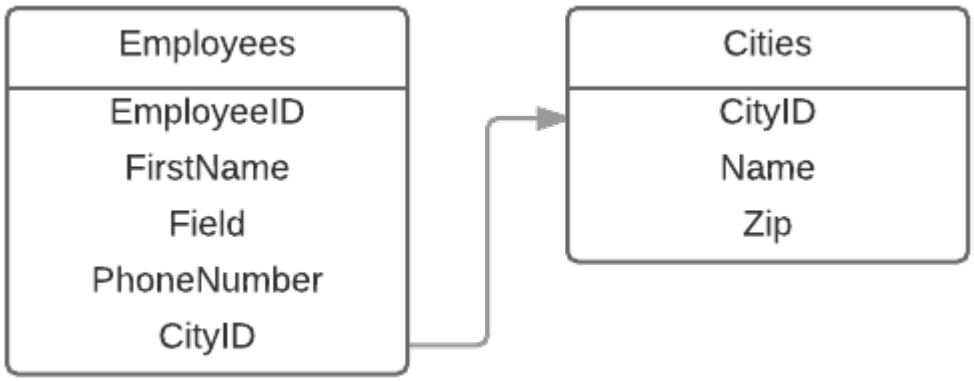
CREA TABELLA CON CHIAVE STRANIERA

Di seguito è possibile trovare il tavolo `Employees` con un riferimento alla tabella `Cities` .

```
CREATE TABLE Cities(
    CityID INT IDENTITY(1,1) NOT NULL,
    Name VARCHAR(20) NOT NULL,
    Zip VARCHAR(10) NOT NULL
);

CREATE TABLE Employees(
    EmployeeID INT IDENTITY (1,1) NOT NULL,
    FirstName VARCHAR(20) NOT NULL,
```

```
    LastName VARCHAR(20) NOT NULL,
    PhoneNumber VARCHAR(10) NOT NULL,
    CityID INT FOREIGN KEY REFERENCES Cities(CityID)
);
```

Qui puoi trovare un diagramma del database.



La colonna `CityID` della tabella `Employees` farà riferimento alla colonna `CityID` di Table `Cities` . Sotto puoi trovare la sintassi per farlo.

```
CityID INT FOREIGN KEY REFERENCES Cities(CityID)
```

Valore	Senso
`CityID`	Nome della colonna
`int`	tipo della colonna
`FOREIGN KEY`	Rende la chiave esterna *(opzionale)*
`REFERENCES Cities(CityID)`	Fa il riferimento alla tabella `Cities` colonna `CityID`

Importante: non è possibile creare un riferimento a una tabella che non esiste nel database. Sii la fonte per rendere prima il tavolo `Cities` e secondo il tavolo `Employees` . Se lo fai, viceversa, genererà un errore.

Crea una tabella temporanea o in memoria

PostgreSQL e SQLite

Per creare una tabella temporanea locale alla sessione:

```
CREATE TEMP TABLE MyTable(...);
```

server SQL

Per creare una tabella temporanea locale alla sessione:

```
CREATE TABLE #TempPhysical(...);
```

Per creare una tabella temporanea visibile a tutti:

```
CREATE TABLE ##TempPhysicalVisibleToEveryone(...);
```

Per creare una tabella in memoria:

```
DECLARE @TempMemory TABLE(...);
```

Leggi CREA TABELLA online: https://riptutorial.com/it/sql/topic/348/crea-tabella

Capitolo 18: DROP o DELETE Database

Sintassi

- Sintassi MSSQL:
- DROP DATABASE [IF EXISTS] {database_name | database_snapshot_name} [, ... n] [;]
- Sintassi MySQL:
- DROP {DATABASE | SCHEMA} [IF EXISTS] db_name

Osservazioni

`DROP DATABASE` viene utilizzato per rilasciare un database da SQL. Assicurati di creare un backup del tuo database prima di rilasciarlo per evitare la perdita accidentale di informazioni.

Examples

Database DROP

Eliminare il database è una semplice dichiarazione one-liner. Drop database cancellerà il database, quindi assicurati sempre di avere un backup del database, se necessario.

Di seguito è riportato il comando per eliminare il database dei dipendenti

```
DROP DATABASE [dbo].[Employees]
```

Leggi DROP o DELETE Database online: https://riptutorial.com/it/sql/topic/3974/drop-o-delete-database

Capitolo 19: ELIMINA

introduzione

L'istruzione DELETE viene utilizzata per eliminare i record da una tabella.

Sintassi

1. DELETE FROM *TableName* [WHERE *Condition*] [LIMIT *count*]

Examples

CANCELLARE determinate righe con DOVE

Questo eliminerà tutte le righe che corrispondono ai criteri `WHERE` .

```
DELETE FROM Employees
WHERE FName = 'John'
```

CANCELLARE tutte le righe

L'omissione di una clausola `WHERE` cancellerà tutte le righe da una tabella.

```
DELETE FROM Employees
```

Consultare la documentazione TRUNCATE per i dettagli su come le prestazioni di TRUNCATE possono essere migliori perché ignorano i trigger, gli indici e i registri per eliminare solo i dati.

Clausola TRUNCATE

Utilizzare questo per ripristinare la tabella alla condizione in cui è stata creata. Questo elimina tutte le righe e reimposta valori come l'incremento automatico. Inoltre non registra ogni singola eliminazione di riga.

```
TRUNCATE TABLE Employees
```

CANCELLA determinate righe in base al confronto con altre tabelle

È possibile `DELETE` dati da una tabella se corrispondono (o non corrispondono) determinati dati in altre tabelle.

Supponiamo di voler `DELETE` dati da Source una volta caricati in Target.

```
DELETE FROM Source
```

```
WHERE  EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
                FROM Target
                Where Source.ID = Target.ID )
```

Le implementazioni RDBMS più comuni (es. MySQL, Oracle, PostgresSQL, Teradata) consentono di unire le tabelle durante `DELETE` consentendo un confronto più complesso in una sintassi compatta.

Aggiungendo complessità allo scenario originale, assumiamo che Aggregate sia stato creato da Target una volta al giorno e non contenga lo stesso ID ma contenga la stessa data. Supponiamo inoltre di voler cancellare i dati da Source *solo* dopo che l'aggregato è stato compilato per il giorno.

Su MySQL, Oracle e Teradata questo può essere fatto usando:

```
DELETE FROM Source
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.Date = AggregateSchema.Aggregate.Date
```

Nell'uso di PostgreSQL:

```
DELETE FROM Source
USING  TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

Ciò si traduce essenzialmente in INNER JOIN tra Source, Target e Aggregate. La cancellazione viene eseguita su Source quando esistono gli stessi ID nella destinazione E la data presente in Target per quegli ID esiste anche in Aggregate.

La stessa query può anche essere scritta (su MySQL, Oracle, Teradata) come:

```
DELETE Source
FROM   Source, TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

I join espliciti possono essere menzionati nelle istruzioni `Delete` su alcune implementazioni RDBMS (ad esempio Oracle, MySQL) ma non supportati su tutte le piattaforme (ad esempio Teradata non li supporta)

I confronti possono essere progettati per verificare gli scenari di mismatch invece di quelli corrispondenti con tutti gli stili di sintassi (osservare `NOT EXISTS` sotto)

```
DELETE FROM Source
WHERE NOT EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
                FROM Target
                Where Source.ID = Target.ID )
```

Leggi ELIMINA online: https://riptutorial.com/it/sql/topic/1105/elimina

Capitolo 20: Elimina in cascata

Examples

ON DELETE CASCADE

Supponi di avere un'applicazione che amministri le stanze.
Supponiamo inoltre che la tua applicazione funzioni su base client (inquilino).
Hai diversi clienti.
Quindi il tuo database conterrà una tabella per i clienti e una per le stanze.

Ora, ogni cliente ha N stanze.

Questo dovrebbe significare che hai una chiave esterna nella tabella della stanza, facendo riferimento alla tabella del cliente.

```
ALTER TABLE dbo.T_Room  WITH CHECK ADD  CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
GO
```

Supponendo che un cliente passi ad altri software, dovrai cancellare i suoi dati nel tuo software.
Ma se lo fai

```
DELETE FROM T_Client WHERE CLI_ID = x
```

Quindi riceverai una violazione di chiave esterna, perché non puoi eliminare il client quando ha ancora stanze.

Ora avresti il codice di scrittura nella tua applicazione che cancella le stanze del cliente prima che cancelli il client. Supponiamo inoltre che in futuro verranno aggiunte molte più dipendenze da chiavi esterne nel tuo database, poiché la funzionalità dell'applicazione si espande. Orribile. Per ogni modifica nel tuo database, dovrai adattare il codice dell'applicazione in N posti.
Probabilmente dovrai adattare il codice anche ad altre applicazioni (ad es. Interfacce con altri sistemi).

C'è una soluzione migliore rispetto a farlo nel tuo codice.
Puoi semplicemente aggiungere `ON DELETE CASCADE` alla tua chiave esterna.

```
ALTER TABLE dbo.T_Room  -- WITH CHECK -- SQL-Server can specify WITH CHECK/WITH NOCHECK
ADD  CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
ON DELETE CASCADE
```

Ora puoi dire

```
DELETE FROM T_Client WHERE CLI_ID = x
```

e le stanze vengono cancellate automaticamente quando il client viene cancellato.
Problema risolto - senza modifiche al codice dell'applicazione.

Una parola di cautela: in Microsoft SQL-Server, questo non funzionerà se si dispone di una tabella che fa riferimento a se stessa. Quindi, se provi a definire una cascata di eliminazione su una struttura ad albero ricorsiva, in questo modo:

```
IF NOT EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation]  WITH CHECK ADD  CONSTRAINT
[FK_T_FMS_Navigation_T_FMS_Navigation] FOREIGN KEY([NA_NA_UID])
REFERENCES [dbo].[T_FMS_Navigation] ([NA_UID])
ON DELETE CASCADE
GO

IF  EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation] CHECK CONSTRAINT [FK_T_FMS_Navigation_T_FMS_Navigation]
GO
```

non funzionerà, perché Microsoft-SQL-server non ti permette di impostare una chiave esterna con `ON DELETE CASCADE` su una struttura ad albero ricorsiva. Una ragione per questo è che l'albero è possibilmente ciclico e che potrebbe portare a una situazione di stallo.

D'altra parte, PostgreSQL può farlo;
il requisito è che l'albero non sia ciclico.
Se l'albero è ciclico, otterrai un errore di runtime.
In tal caso, dovrai solo implementare la funzione di cancellazione.

Una parola di cautela:
Questo significa che non puoi semplicemente cancellare e reinserire la tabella del client, perché se lo fai, cancellerà tutte le voci in "T_Room" ... (nessun aggiornamento non delta più)

Leggi Elimina in cascata online: https://riptutorial.com/it/sql/topic/3518/elimina-in-cascata

Capitolo 21: Esempi di database e tabelle

Examples

Database del negozio automatico

Nell'esempio seguente - Database per un'azienda di auto shop, abbiamo un elenco di reparti, dipendenti, clienti e auto clienti. Utilizziamo le chiavi esterne per creare relazioni tra i vari tavoli.

Esempio dal vivo: violino SQL

Relazioni tra tabelle

- Ogni dipartimento può avere 0 o più dipendenti
- Ogni Dipendente può avere 0 o 1 Manager
- Ogni cliente può avere 0 o più automobili

---

dipartimenti

Id	Nome
1	HR
2	I saldi
3	Tech

Istruzioni SQL per creare la tabella:

```
CREATE TABLE Departments (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(25) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Departments
    ([Id], [Name])
VALUES
    (1, 'HR'),
    (2, 'Sales'),
    (3, 'Tech')
;
```

I dipendenti

Id	FName	LName	Numero di telefono	ManagerID	DepartmentID	Stipendio	Data di assunzione
1	Giacomo	fabbro	1234567890	NULLO	1	1000	01-01-2002
2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Michael	Williams	1357911131	1	2	600	12-05-2009
4	Johnathon	fabbro	1212121212	2	1	500	24-07-2016

Istruzioni SQL per creare la tabella:

```
CREATE TABLE Employees (
    Id INT NOT NULL AUTO_INCREMENT,
    FName VARCHAR(35) NOT NULL,
    LName VARCHAR(35) NOT NULL,
    PhoneNumber VARCHAR(11),
    ManagerId INT,
    DepartmentId INT NOT NULL,
    Salary INT NOT NULL,
    HireDate DATETIME NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY (ManagerId) REFERENCES Employees(Id),
    FOREIGN KEY (DepartmentId) REFERENCES Departments(Id)
);

INSERT INTO Employees
    ([Id], [FName], [LName], [PhoneNumber], [ManagerId], [DepartmentId], [Salary], [HireDate])
VALUES
    (1, 'James', 'Smith', 1234567890, NULL, 1, 1000, '01-01-2002'),
    (2, 'John', 'Johnson', 2468101214, '1', 1, 400, '23-03-2005'),
    (3, 'Michael', 'Williams', 1357911131, '1', 2, 600, '12-05-2009'),
    (4, 'Johnathon', 'Smith', 1212121212, '2', 1, 500, '24-07-2016')
;
```

Clienti

Id	FName	LName	E-mail	Numero di telefono	PreferredContact
1	William	Jones	william.jones@example.com	3347927472	TELEFONO
2	David	Mugnaio	dmiller@example.net	2137921892	E-MAIL
3	Richard	Davis	richard0123@example.com	NULLO	E-MAIL

Istruzioni SQL per creare la tabella:

```
CREATE TABLE Customers (
    Id INT NOT NULL AUTO_INCREMENT,
    FName VARCHAR(35) NOT NULL,
```

```
    LName VARCHAR(35) NOT NULL,
    Email varchar(100) NOT NULL,
    PhoneNumber VARCHAR(11),
    PreferredContact VARCHAR(5) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Customers
    ([Id], [FName], [LName], [Email], [PhoneNumber], [PreferredContact])
VALUES
    (1, 'William', 'Jones', 'william.jones@example.com', '3347927472', 'PHONE'),
    (2, 'David', 'Miller', 'dmiller@example.net', '2137921892', 'EMAIL'),
    (3, 'Richard', 'Davis', 'richard0123@example.com', NULL, 'EMAIL')
;
```

Macchine

Id	Identificativo del cliente	Numero Identità dell'impiegato	Modello	Stato	Costo totale
1	1	2	Ford F-150	PRONTO	230
2	1	2	Ford F-150	PRONTO	200
3	2	1	Ford Mustang	IN ATTESA	100
4	3	3	Toyota Prius	LAVORO	1254

Istruzioni SQL per creare la tabella:

```
CREATE TABLE Cars (
    Id INT NOT NULL AUTO_INCREMENT,
    CustomerId INT NOT NULL,
    EmployeeId INT NOT NULL,
    Model varchar(50) NOT NULL,
    Status varchar(25) NOT NULL,
    TotalCost INT NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY (CustomerId) REFERENCES Customers(Id),
    FOREIGN KEY (EmployeeId) REFERENCES Employees(Id)
);

INSERT INTO Cars
    ([Id], [CustomerId], [EmployeeId], [Model], [Status], [TotalCost])
VALUES
    ('1', '1', '2', 'Ford F-150', 'READY', '230'),
    ('2', '1', '2', 'Ford F-150', 'READY', '200'),
    ('3', '2', '1', 'Ford Mustang', 'WAITING', '100'),
    ('4', '3', '3', 'Toyota Prius', 'WORKING', '1254')
;
```

Database delle biblioteche

In questo database di esempio per una libreria, abbiamo tabelle *Autori* , *Libri* e *LibriAutori* .

Esempio dal vivo: violino SQL

Gli autori e i *libri* sono noti come **tabelle di base** , poiché contengono la definizione di colonna e i dati per le entità effettive nel modello relazionale. *BooksAuthors* è noto come **tabella delle relazioni** , poiché questa tabella definisce la relazione tra la tabella *Libri* e *Autori* .

Relazioni tra tabelle

- Ogni autore può avere 1 o più libri
- Ogni libro può avere 1 o più autori

---

autori

(guarda *la tabella*)

Id	Nome	Nazione
1	JD Salinger	Stati Uniti d'America
2	F. Scott. Fitzgerald	Stati Uniti d'America
3	Jane Austen	UK
4	Scott Hanselman	Stati Uniti d'America
5	Jason N. Gaylord	Stati Uniti d'America
6	Pranav Rastogi	India
7	Todd Miranda	Stati Uniti d'America
8	Christian Wenz	Stati Uniti d'America

SQL per creare la tabella:

```
CREATE TABLE Authors (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(70) NOT NULL,
    Country VARCHAR(100) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Authors
```

```
    (Name, Country)
VALUES
    ('J.D. Salinger', 'USA'),
    ('F. Scott. Fitzgerald', 'USA'),
    ('Jane Austen', 'UK'),
    ('Scott Hanselman', 'USA'),
    ('Jason N. Gaylord', 'USA'),
    ('Pranav Rastogi', 'India'),
    ('Todd Miranda', 'USA'),
    ('Christian Wenz', 'USA')
;
```

Libri

(guarda *la tabella*)

Id	Titolo
1	Il cacciatore nella segale
2	Nove storie
3	Franny e Zooey
4	Il grande Gatsby
5	Tender id the Night
6	Orgoglio e pregiudizio
7	Professional ASP.NET 4.5 in C # e VB

SQL per creare la tabella:

```
CREATE TABLE Books (
    Id INT NOT NULL AUTO_INCREMENT,
    Title VARCHAR(50) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Books
    (Id, Title)
VALUES
    (1, 'The Catcher in the Rye'),
    (2, 'Nine Stories'),
    (3, 'Franny and Zooey'),
    (4, 'The Great Gatsby'),
    (5, 'Tender id the Night'),
    (6, 'Pride and Prejudice'),
    (7, 'Professional ASP.NET 4.5 in C# and VB')
;
```

BooksAuthors

(guarda *la tabella*)

BookID	AuthorID
1	1
2	1
3	1
4	2
5	2
6	3
7	4
7	5
7	6
7	7
7	8

SQL per creare la tabella:

```
CREATE TABLE BooksAuthors (
    AuthorId INT NOT NULL,
    BookId  INT NOT NULL,
    FOREIGN KEY (AuthorId) REFERENCES Authors(Id),
    FOREIGN KEY (BookId) REFERENCES Books(Id)
);

INSERT INTO BooksAuthors
    (BookId, AuthorId)
VALUES
    (1, 1),
    (2, 1),
    (3, 1),
    (4, 2),
    (5, 2),
    (6, 3),
    (7, 4),
    (7, 5),
    (7, 6),
    (7, 7),
    (7, 8)
;
```

Esempi

Visualizza tutti gli autori (vedi l'esempio dal vivo):

```
SELECT * FROM Authors;
```

Visualizza tutti i titoli dei libri (vedi esempio dal vivo):

```
SELECT * FROM Books;
```

Visualizza tutti i libri e i loro autori (vedi l'esempio dal vivo):

```
SELECT
  ba.AuthorId,
  a.Name AuthorName,
  ba.BookId,
  b.Title BookTitle
FROM BooksAuthors ba
  INNER JOIN Authors a ON a.id = ba.authorid
  INNER JOIN Books b ON b.id = ba.bookid
;
```

Tabella Paesi

In questo esempio, abbiamo una tabella **Paesi** . Una tabella per i paesi ha molti usi, specialmente nelle applicazioni finanziarie che riguardano valute e tassi di cambio.

Esempio dal vivo: violino SQL

Alcune applicazioni software di dati di mercato come Bloomberg e Reuters richiedono di fornire alla propria API un codice paese a 2 o 3 caratteri insieme al codice valuta. Quindi questa tabella di esempio ha sia la colonna del codice `ISO` 2 caratteri che le colonne del codice `ISO3` 3 caratteri.

---

paesi

(guarda *la tabella*)

Id	ISO	ISO3	ISONumeric	Nome del paese	Capitale	ContinentCode	Codice valuta
1	AU	AUS	36	Australia	Canberra	OC	AUD
2	DE	DEU	276	Germania	Berlino	Unione Europea	euro
2	NEL	IND	356	India	Nuova Delhi	COME	INR

Id	ISO	ISO3	ISONumeric	Nome del paese	Capitale	ContinentCode	Codice valuta
3	LA	LAO	418	Laos	Vientiane	COME	LAK
4	NOI	Stati Uniti d'America	840	stati Uniti	Washington	N / A	Dollaro statunitense
5	Z W	ZWE	716	Zimbabwe	Harare	AF	ZWL

SQL per creare la tabella:

```
CREATE TABLE Countries (
    Id INT NOT NULL AUTO_INCREMENT,
    ISO VARCHAR(2) NOT NULL,
    ISO3 VARCHAR(3) NOT NULL,
    ISONumeric INT NOT NULL,
    CountryName VARCHAR(64) NOT NULL,
    Capital VARCHAR(64) NOT NULL,
    ContinentCode VARCHAR(2) NOT NULL,
    CurrencyCode VARCHAR(3) NOT NULL,
    PRIMARY KEY(Id)
)
;

INSERT INTO Countries
    (ISO, ISO3, ISONumeric, CountryName, Capital, ContinentCode, CurrencyCode)
VALUES
    ('AU', 'AUS', 36, 'Australia', 'Canberra', 'OC', 'AUD'),
    ('DE', 'DEU', 276, 'Germany', 'Berlin', 'EU', 'EUR'),
    ('IN', 'IND', 356, 'India', 'New Delhi', 'AS', 'INR'),
    ('LA', 'LAO', 418, 'Laos', 'Vientiane', 'AS', 'LAK'),
    ('US', 'USA', 840, 'United States', 'Washington', 'NA', 'USD'),
    ('ZW', 'ZWE', 716, 'Zimbabwe', 'Harare', 'AF', 'ZWL')
;
```

Leggi Esempi di database e tabelle online: https://riptutorial.com/it/sql/topic/280/esempi-di-database-e-tabelle

Capitolo 22: Espressioni di tabella comuni

Sintassi

- WITH QueryName [(ColumnName, ...)] AS (
  SELEZIONA
)
  SELECT ... FROM QueryName ...;

- WITH RECURSIVE QueryName [(ColumnName, ...)] AS (
  SELEZIONA
  UNION [TUTTI]
  SELEZIONA ... DA QueryName ...
)
  SELECT ... FROM QueryName ...;

Osservazioni

Documentazione ufficiale: clausola WITH

Un'espressione tabella comune è un set di risultati temporaneo e può essere il risultato di una subquery complessa. È definito utilizzando la clausola WITH. CTE migliora la leggibilità e viene creato in memoria anziché nel database TempDB in cui viene creata la variabile Tabella temporanea e Tabella.

Concetti chiave delle espressioni di tabella comuni:

- Può essere utilizzato per suddividere query complesse, in particolare join e sottoquery complessi.
- È un modo per incapsulare una definizione di query.
- Persistere solo fino all'esecuzione della query successiva.
- L'uso corretto può portare a miglioramenti in termini di qualità del codice / manutenibilità e velocità.
- Può essere utilizzato per fare riferimento alla tabella risultante più volte nella stessa istruzione (eliminare la duplicazione in SQL).
- Può essere un sostituto di una vista quando non è richiesto l'uso generale di una vista; cioè, non è necessario memorizzare la definizione nei metadati.
- Verrà eseguito quando chiamato, non quando definito. Se il CTE viene utilizzato più volte in una query, verrà eseguito più volte (probabilmente con risultati diversi).

Examples

Query temporanea

Si comportano allo stesso modo delle subquery nidificate ma con una sintassi diversa.

```
WITH ReadyCars AS (
  SELECT *
  FROM Cars
  WHERE Status = 'READY'
)
SELECT ID, Model, TotalCost
FROM ReadyCars
ORDER BY TotalCost;
```

ID	Modello	Costo totale
1	Ford F-150	200
2	Ford F-150	230

Sintassi di subquery equivalente

```
SELECT ID, Model, TotalCost
FROM (
  SELECT *
  FROM Cars
  WHERE Status = 'READY'
) AS ReadyCars
ORDER BY TotalCost
```

salendo ricorsivamente su un albero

```
WITH RECURSIVE ManagersOfJonathon AS (
    -- start with this row
    SELECT *
    FROM Employees
    WHERE ID = 4

    UNION ALL

    -- get manager(s) of all previously selected rows
    SELECT Employees.*
    FROM Employees
    JOIN ManagersOfJonathon
        ON Employees.ID = ManagersOfJonathon.ManagerID
)
SELECT * FROM ManagersOfJonathon;
```

Id	FName	LName	Numero di telefono	ManagerID	DepartmentID
4	Johnathon	fabbro	1212121212	**2**	1
2	John	Johnson	2468101214	**1**	1
1	Giacomo	fabbro	1234567890	NULLO	1

generare valori

La maggior parte dei database non ha un modo nativo di generare una serie di numeri per l'uso ad-hoc; tuttavia, le espressioni di tabella comuni possono essere utilizzate con la ricorsione per emulare tale tipo di funzione.

L'esempio seguente genera un'espressione di tabella comune denominata `Numbers` con una colonna `i` che ha una riga per i numeri 1-5:

```
--Give a table name `Numbers" and a column `i` to hold the numbers
WITH Numbers(i) AS (
    --Starting number/index
    SELECT 1
    --Top-level UNION ALL operator required for recursion
    UNION ALL
    --Iteration expression:
    SELECT i + 1
    --Table expression we first declared used as source for recursion
    FROM Numbers
    --Clause to define the end of the recursion
    WHERE i < 5
)
--Use the generated table expression like a regular table
SELECT i FROM Numbers;
```

io
1
2
3
4
5

Questo metodo può essere utilizzato con qualsiasi intervallo di numeri, nonché con altri tipi di dati.

enumerare ricorsivamente una sottostruttura

```
WITH RECURSIVE ManagedByJames(Level, ID, FName, LName) AS (
    -- start with this row
    SELECT 1, ID, FName, LName
    FROM Employees
    WHERE ID = 1

    UNION ALL

    -- get employees that have any of the previously selected rows as manager
    SELECT ManagedByJames.Level + 1,
           Employees.ID,
           Employees.FName,
           Employees.LName
    FROM Employees
    JOIN ManagedByJames
```

```
        ON Employees.ManagerID = ManagedByJames.ID

    ORDER BY 1 DESC   -- depth-first search
)
SELECT * FROM ManagedByJames;
```

Livello	ID	FName	LName
1	1	Giacomo	fabbro
2	2	John	Johnson
3	4	Johnathon	fabbro
2	3	Michael	Williams

Funzionalità Oracle CONNECT BY con CTE ricorsive

La funzionalità CONNECT BY di Oracle offre molte funzionalità utili e non banali che non sono integrate quando si utilizzano CTE ricorsive standard SQL. Questo esempio replica queste funzioni (con alcune aggiunte per motivi di completezza), utilizzando la sintassi di SQL Server. È molto utile per gli sviluppatori Oracle che trovano molte funzionalità mancanti nelle loro query gerarchiche su altri database, ma serve anche a mostrare cosa si può fare con una query gerarchica in generale.

```
  WITH tbl AS (
       SELECT id, name, parent_id
         FROM mytable)
     , tbl_hierarchy AS (
       /* Anchor */
       SELECT 1 AS "LEVEL"
            --, 1 AS CONNECT_BY_ISROOT
            --, 0 AS CONNECT_BY_ISBRANCH
            , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS
CONNECT_BY_ISLEAF
            , 0 AS CONNECT_BY_ISCYCLE
            , '/' + CAST(t.id   AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_id
            , '/' + CAST(t.name AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_name
            , t.id AS root_id
            , t.*
         FROM tbl t
        WHERE t.parent_id IS NULL                              -- START WITH parent_id IS NULL
       UNION ALL
       /* Recursive */
       SELECT th."LEVEL" + 1 AS "LEVEL"
            --, 0 AS CONNECT_BY_ISROOT
            --, CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 1 ELSE 0 END AS
CONNECT_BY_ISBRANCH
            , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS
CONNECT_BY_ISLEAF
            , CASE WHEN th.SYS_CONNECT_BY_PATH_id LIKE '%/' + CAST(t.id AS VARCHAR(MAX)) +
'/%' THEN 1 ELSE 0 END AS CONNECT_BY_ISCYCLE
            , th.SYS_CONNECT_BY_PATH_id   + CAST(t.id   AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_id
            , th.SYS_CONNECT_BY_PATH_name + CAST(t.name AS VARCHAR(MAX)) + '/' AS
```

```
SYS_CONNECT_BY_PATH_name
            , th.root_id
            , t.*
         FROM tbl t
              JOIN tbl_hierarchy th ON (th.id = t.parent_id) -- CONNECT BY PRIOR id =
parent_id
        WHERE th.CONNECT_BY_ISCYCLE = 0)                     -- NOCYCLE
SELECT th.*
     --, REPLICATE(' ', (th."LEVEL" - 1) * 3) + th.name AS tbl_hierarchy
  FROM tbl_hierarchy th
       JOIN tbl CONNECT_BY_ROOT ON (CONNECT_BY_ROOT.id = th.root_id)
 ORDER BY th.SYS_CONNECT_BY_PATH_name;                       -- ORDER SIBLINGS BY name
```

CONNECT BY caratteristiche illustrate sopra, con spiegazioni:

- clausole
  - CONNECT BY: specifica la relazione che definisce la gerarchia.
  - INIZIA CON: specifica i nodi radice.
  - ORDINA I SEGNI DI: Ordini risultati correttamente.
- parametri
  - NOCYCLE: interrompe l'elaborazione di un ramo quando viene rilevato un loop. Le gerarchie valide sono i grafici aciclici diretti e i riferimenti circolari violano questo costrutto.
- operatori
  - PRIOR: ottiene i dati dal genitore del nodo.
  - CONNECT_BY_ROOT: ottiene i dati dalla radice del nodo.
- pseudocolonne
  - LIVELLO: indica la distanza del nodo dalla sua radice.
  - CONNECT_BY_ISLEAF: indica un nodo senza figli.
  - CONNECT_BY_ISCYCLE: indica un nodo con un riferimento circolare.
- funzioni
  - SYS_CONNECT_BY_PATH: restituisce una rappresentazione appiattita / concatenata del percorso del nodo dalla sua radice.

Generare in modo ricorsivo date, estese per includere la generazione di squadre come esempio

```
DECLARE @DateFrom DATETIME = '2016-06-01 06:00'
DECLARE @DateTo DATETIME = '2016-07-01 06:00'
DECLARE @IntervalDays INT = 7

-- Transition Sequence = Rest & Relax into Day Shift into Night Shift
-- RR (Rest & Relax) = 1
-- DS (Day Shift) = 2
-- NS (Night Shift) = 3

;WITH roster AS
(
   SELECT @DateFrom AS RosterStart, 1 AS TeamA, 2 AS TeamB, 3 AS TeamC
   UNION ALL
   SELECT DATEADD(d, @IntervalDays, RosterStart),
          CASE TeamA WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamA,
          CASE TeamB WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamB,
```

```
          CASE TeamC WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamC
   FROM roster WHERE RosterStart < DATEADD(d, -@IntervalDays, @DateTo)
)

SELECT RosterStart,
       ISNULL(LEAD(RosterStart) OVER (ORDER BY RosterStart), RosterStart + @IntervalDays) AS
RosterEnd,
       CASE TeamA WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamA,
       CASE TeamB WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamB,
       CASE TeamC WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamC
FROM roster
```

Risultato

Cioè per la settimana 1 TeamA è su R & R, TeamB su Day Shift e TeamC su Night Shift.

Results | Messages

	RosterStart	RosterEnd	TeamA	TeamB	TeamC
1	2016-06-01 06:00:00.000	2016-06-08 06:00:00.000	RR	DS	NS
2	2016-06-08 06:00:00.000	2016-06-15 06:00:00.000	DS	NS	RR
3	2016-06-15 06:00:00.000	2016-06-22 06:00:00.000	NS	RR	DS
4	2016-06-22 06:00:00.000	2016-06-29 06:00:00.000	RR	DS	NS
5	2016-06-29 06:00:00.000	2016-07-06 06:00:00.000	DS	NS	RR

Rifattorizzare una query per utilizzare le espressioni di tabella comuni

Supponiamo di voler ottenere tutte le categorie di prodotti con vendite totali superiori a 20.

Ecco una query senza Common Table Expressions:

```
SELECT category.description, sum(product.price) as total_sales
FROM sale
LEFT JOIN product on sale.product_id = product.id
LEFT JOIN category on product.category_id = category.id
GROUP BY category.id, category.description
HAVING sum(product.price) > 20
```

E una query equivalente utilizzando Common Table Expressions:

```
WITH all_sales AS (
  SELECT product.price, category.id as category_id, category.description as
category_description
  FROM sale
  LEFT JOIN product on sale.product_id = product.id
  LEFT JOIN category on product.category_id = category.id
)
, sales_by_category AS (
  SELECT category_description, sum(price) as total_sales
  FROM all_sales
  GROUP BY category_id, category_description
)
SELECT * from sales_by_category WHERE total_sales > 20
```

Esempio di un SQL complesso con Common Table Expression

Supponiamo di voler interrogare i "prodotti più economici" dalle "categorie principali".

Ecco un esempio di query utilizzando Common Table Expressions

```
-- all_sales: just a simple SELECT with all the needed JOINS
WITH all_sales AS (
  SELECT
  product.price as product_price,
  category.id as category_id,
  category.description as category_description
  FROM sale
  LEFT JOIN product on sale.product_id = product.id
  LEFT JOIN category on product.category_id = category.id
)
-- Group by category
, sales_by_category AS (
  SELECT category_id, category_description,
  sum(product_price) as total_sales
  FROM all_sales
  GROUP BY category_id, category_description
)
-- Filtering total_sales > 20
, top_categories AS (
  SELECT * from sales_by_category WHERE total_sales > 20
)
-- all_products: just a simple SELECT with all the needed JOINS
, all_products AS (
  SELECT
  product.id as product_id,
  product.description as product_description,
  product.price as product_price,
  category.id as category_id,
  category.description as category_description
  FROM product
  LEFT JOIN category on product.category_id = category.id
)
-- Order by product price
, cheapest_products AS (
  SELECT * from all_products
  ORDER by product_price ASC
)
-- Simple inner join
, cheapest_products_from_top_categories AS (
  SELECT product_description, product_price
  FROM cheapest_products
  INNER JOIN top_categories ON cheapest_products.category_id = top_categories.category_id
)
--The main SELECT
SELECT * from cheapest_products_from_top_categories
```

Leggi Espressioni di tabella comuni online: https://riptutorial.com/it/sql/topic/747/espressioni-di-tabella-comuni

Capitolo 23: Filtra i risultati usando WHERE e HAVING

Sintassi

- SELEZIONA nome_colonna
  FROM nome_tabella
  WHERE nome_operatore valore dell'operatore
- SELECT column_name, aggregate_function (column_name)
  FROM nome_tabella
  GROUP BY nome_colonna
  Valore operatore HAVING aggregate_function (column_name)

Examples

La clausola WHERE restituisce solo le righe che corrispondono ai suoi criteri

Steam ha un gioco sotto la sezione $ 10 della loro pagina del negozio. Da qualche parte nel profondo dei loro sistemi, c'è probabilmente una query che assomiglia a qualcosa:

```
SELECT *
FROM Items
WHERE Price < 10
```

Utilizzare IN per restituire righe con un valore contenuto in un elenco

Questo esempio utilizza la tabella Car dai database di esempio.

```
SELECT *
FROM Cars
WHERE TotalCost IN (100, 200, 300)
```

Questa query restituirà Car # 2 che costa 200 e Car # 3 che costa 100. Si noti che questo è equivalente all'utilizzo di più clausole con `OR` , ad esempio:

```
SELECT *
FROM Cars
WHERE TotalCost = 100 OR TotalCost = 200 OR TotalCost = 300
```

Usa LIKE per trovare stringhe e sottostringhe corrispondenti

Vedi la documentazione completa sull'operatore LIKE .

Questo esempio utilizza la tabella Impiegati dai database di esempio.

```
SELECT *
FROM Employees
WHERE FName LIKE 'John'
```

Questa query restituirà solo Employee # 1 il cui nome di battesimo corrisponde esattamente a "John".

```
SELECT *
FROM Employees
WHERE FName like 'John%'
```

L'aggiunta di `%` ti consente di cercare una sottostringa:

- `John%` - restituirà qualsiasi Dipendente il cui nome inizia con "John", seguito da qualsiasi quantità di caratteri
- `%John` - restituirà qualsiasi Dipendente il cui nome termina con "John", preceduto da qualsiasi quantità di caratteri
- `%John%` - restituirà qualsiasi Dipendente il cui nome contenga "John" in qualsiasi punto all'interno del valore

In questo caso, la query restituirà Employee # 2 il cui nome è 'John' così come Employee # 4 il cui nome è 'Johnathon'.

Clausola WHERE con valori NULL / NOT NULL

```
SELECT *
FROM Employees
WHERE ManagerId IS NULL
```

Questa istruzione restituirà tutti i record Employee in cui il valore della colonna `ManagerId` è `NULL` .

Il risultato sarà:

```
Id    FName    LName    PhoneNumber    ManagerId    DepartmentId
1     James    Smith    1234567890     NULL         1
```

```
SELECT *
FROM Employees
WHERE ManagerId IS NOT NULL
```

Questa istruzione restituirà tutti i record Employee in cui il valore di `ManagerId` *non* è `NULL` .

Il risultato sarà:

```
Id    FName       LName     PhoneNumber    ManagerId    DepartmentId
2     John        Johnson   2468101214     1            1
3     Michael     Williams  1357911131     1            2
4     Johnathon   Smith     1212121212     2            1
```

Nota: la stessa query non restituirà risultati se si modifica la clausola `WHERE ManagerId = NULL` in `WHERE ManagerId = NULL` o `WHERE ManagerId <> NULL` .

Utilizzare HAVING con le funzioni aggregate

A differenza della clausola `WHERE` , `HAVING` può essere utilizzato con funzioni aggregate.

> Una funzione aggregata è una funzione in cui i valori di più righe sono raggruppati come input su determinati criteri per formare un singolo valore di significato o misura più significativo (Wikipedia).

Le funzioni di aggregazione comuni includono `COUNT()` , `SUM()` , `MIN()` e `MAX()` .

Questo esempio utilizza la tabella Car dai database di esempio.

```
SELECT CustomerId, COUNT(Id) AS [Number of Cars]
FROM Cars
GROUP BY CustomerId
HAVING COUNT(Id) > 1
```

Questa query restituirà il conteggio `CustomerId` e `Number of Cars` di qualsiasi cliente che abbia più di un'auto. In questo caso, l'unico cliente che ha più di un'auto è il Cliente n. 1.

I risultati saranno simili a:

Identificativo del cliente	Numero di auto
1	2

Utilizzare TRA per filtrare i risultati

Negli esempi seguenti vengono utilizzati i database di esempio Item Sales and Customers .

> Nota: l'operatore BETWEEN è incluso.

Utilizzando l'operatore BETWEEN con i numeri:

```
SELECT * From ItemSales
WHERE Quantity BETWEEN 10 AND 17
```

Questa query restituirà tutti i record `ItemSales` che hanno una quantità maggiore o uguale a 10 e minore o uguale a 17. I risultati saranno simili a:

Id	Data di vendita	Numero identificativo dell'oggetto	Quantità	Prezzo
1	2013/07/01	100	10	34.5
4	2013/07/23	100	15	34.5

Id	Data di vendita	Numero identificativo dell'oggetto	Quantità	Prezzo
5	2013/07/24	145	10	34.5

Utilizzo dell'operatore BETWEEN con i valori di data:

```
SELECT * From ItemSales
WHERE SaleDate BETWEEN '2013-07-11' AND '2013-05-24'
```

Questa query restituirà tutti i record `ItemSales` con un `SaleDate` maggiore o uguale all'11 luglio 2013 e inferiore o uguale al 24 maggio 2013.

Id	Data di vendita	Numero identificativo dell'oggetto	Quantità	Prezzo
3	2013/07/11	100	20	34.5
4	2013/07/23	100	15	34.5
5	2013/07/24	145	10	34.5

> Quando si confrontano i valori datetime anziché le date, potrebbe essere necessario convertire i valori datetime in valori data o aggiungere o sottrarre 24 ore per ottenere i risultati corretti.

Utilizzo dell'operatore BETWEEN con valori di testo:

```
SELECT Id, FName, LName FROM Customers
WHERE LName BETWEEN 'D' AND 'L';
```

Esempio dal vivo: violino SQL

Questa query restituirà tutti i clienti il cui nome cade alfabeticamente tra le lettere "D" e "L". In questo caso, i clienti n. 1 e n. 3 verranno restituiti. Il cliente n. 2, il cui nome inizia con una "M" non sarà incluso.

Id	FName	LName
1	William	Jones
3	Richard	Davis

Uguaglianza

```
SELECT * FROM Employees
```

Questa dichiarazione restituirà tutte le righe dalla tabella `Employees` .

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

L'utilizzo di un `WHERE` alla fine `SELECT` consente di limitare le righe restituite a una condizione. In questo caso, dove c'è una corrispondenza esatta usando il segno `=` :

```
SELECT * FROM Employees WHERE DepartmentId = 1
```

Restituirà solo le righe in cui `DepartmentId` è uguale a `1` :

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

AND e OR

Puoi anche combinare più operatori per creare condizioni `WHERE` più complesse. Gli esempi seguenti utilizzano la tabella `Employees` :

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

E

```
SELECT * FROM Employees WHERE DepartmentId = 1 AND ManagerId = 1
```

Tornerà:

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
```

```
2005    01-01-2002
```

O

```
SELECT * FROM Employees WHERE DepartmentId = 2 OR ManagerId = 2
```

Tornerà:

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

Utilizzare HAVING per verificare più condizioni in un gruppo

Tabella degli ordini

Identificativo del cliente	Codice prodotto	Quantità	Prezzo
1	2	5	100
1	3	2	200
1	4	1	500
2	1	4	50
3	5	6	700

Per verificare i clienti che hanno ordinato entrambi - ProductID 2 e 3, è possibile utilizzare HAVING

```
select customerId
from orders
where productID in (2,3)
group by customerId
having count(distinct productID) = 2
```

Valore di ritorno:

identificativo del cliente
1

La query seleziona solo i record con ID prodotto nelle domande e con la clausola HAVING controlla se i gruppi hanno 2 ID prodotto e non solo uno.

Un'altra possibilità sarebbe

```
select customerId
from orders
group by customerId
having sum(case when productID = 2 then 1 else 0 end) > 0
   and sum(case when productID = 3 then 1 else 0 end) > 0
```

Questa query seleziona solo i gruppi che hanno almeno un record con ID prodotto 2 e almeno uno con ID prodotto 3.

Dove ESISTE

Selezionerà i record in `TableName` con i record corrispondenti in `TableName1` .

```
SELECT * FROM TableName t WHERE EXISTS (
    SELECT 1 FROM TableName1 t1 where t.Id = t1.Id)
```

Leggi Filtra i risultati usando WHERE e HAVING online: https://riptutorial.com/it/sql/topic/636/filtra-i-risultati-usando-where-e-having

Capitolo 24: Funzioni (aggregato)

Sintassi

- Funzione (espressione [*DISTINCT*]) -DISTINCT è un parametro facoltativo
- AVG ([ALL | DISTINCT] espressione)
- COUNT ({[ALL | DISTINCT] espressione] | *})
- RAGGRUPPAMENTO (<espressione_colonna>)
- MAX (espressione [ALL | DISTINCT])
- MIN (espressione [ALL | DISTINCT])
- SUM (espressione [ALL | DISTINCT])
- VAR (espressione [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause)
- VARP ([ALL | DISTINCT] espressione)
  OVER ([partition_by_clause] order_by_clause
- STDEV ([ALL | DISTINCT] espressione)
  OVER ([partition_by_clause] order_by_clause)
- STDEVP ([ALL | DISTINCT] espressione)
  OVER ([partition_by_clause] order_by_clause)

Osservazioni

Nella gestione del database una funzione aggregata è una funzione in cui i valori di più righe sono raggruppati come input su determinati criteri per formare un singolo valore di significato o misura più significativo come un set, una borsa o un elenco.

```
MIN         returns the smallest value in a given column
MAX         returns the largest value in a given column
SUM         returns the sum of the numeric values in a given column
AVG         returns the average value of a given column
COUNT       returns the total number of values in a given column
COUNT(*)    returns the number of rows in a table
GROUPING    Is a column or an expression that contains a column in a GROUP BY clause.
STDEV       returns the statistical standard deviation of all values in the specified
expression.
STDEVP      returns the statistical standard deviation for the population for all values in the
specified expression.
VAR         returns the statistical variance of all values in the specified expression. may be
followed by the OVER clause.
VARP        returns the statistical variance for the population for all values in the specified
expression.
```

> Le funzioni aggregate vengono utilizzate per calcolare una "colonna restituita di dati numerici" `SELECT` . In pratica riassumono i risultati di una particolare colonna di dati selezionati. - SQLCourse2.com

Tutte le funzioni aggregate ignorano i valori NULL.

Examples

SOMMA

`Sum` funzione sommare il valore di tutti i righe nel gruppo. Se la clausola group by viene omessa, somma tutte le righe.

```
select sum(salary) TotalSalary
from employees;
```

TotalSalary
2500

```
select DepartmentId, sum(salary) TotalSalary
from employees
group by DepartmentId;
```

DepartmentID	TotalSalary
1	2000
2	500

Aggregazione condizionale

Tabella dei pagamenti

Cliente	Modalità di pagamento	Quantità
Peter	Credito	100
Peter	Credito	300
John	Credito	1000
John	Addebito	500

```
select customer,
       sum(case when payment_type = 'credit' then amount else 0 end) as credit,
       sum(case when payment_type = 'debit' then amount else 0 end) as debit
from payments
group by customer
```

Risultato:

Cliente	Credito	Addebito
Peter	400	0
John	1000	500

```
select customer,
       sum(case when payment_type = 'credit' then 1 else 0 end) as credit_transaction_count,
       sum(case when payment_type = 'debit' then 1 else 0 end) as debit_transaction_count
from payments
group by customer
```

Risultato:

Cliente	credit_transaction_count	debit_transaction_count
Peter	2	0
John	1	1

AVG ()

La funzione di aggregazione AVG () restituisce la media di una determinata espressione, in genere valori numerici in una colonna. Supponiamo di avere una tabella contenente il calcolo annuale della popolazione nelle città di tutto il mondo. I record di New York City sono simili a quelli seguenti:

TABELLA ESEMPIO

nome della città	popolazione	anno
New York City	8.550.405	2015
New York City	...	...
New York City	8.000.906	2005

Per selezionare la popolazione media della città di New York, Stati Uniti, da una tabella contenente nomi di città, misure di popolazione e anni di misurazione per gli ultimi dieci anni:

QUERY

```
select city_name, AVG(population) avg_population
from city_population
where city_name = 'NEW YORK CITY';
```

Si noti come l'anno di misura è assente dalla query poiché la popolazione viene mediata nel

tempo.

RISULTATI

nome della città	avg_population
New York City	8.250.754

> Nota: la funzione AVG () convertirà i valori in tipi numerici. Questo è particolarmente importante da ricordare quando si lavora con le date.

Elenco di concatenazione

Credito parziale a questa risposta SO.

List Concatenazione aggrega una colonna o un'espressione combinando i valori in una singola stringa per ciascun gruppo. Una stringa per delimitare ogni valore (vuoto o una virgola quando omesso) e l'ordine dei valori nel risultato può essere specificato. Sebbene non faccia parte dello standard SQL, tutti i principali fornitori di database relazionali lo supportano a modo loro.

MySQL

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB ORDER BY ColumnB SEPARATOR ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

Oracle e DB2

```
SELECT ColumnA
     , LISTAGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

PostgreSQL

```
SELECT ColumnA
     , STRING_AGG(ColumnB, ',' ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

server SQL

SQL Server 2016 e versioni precedenti

(CTE inclusa per incoraggiare il principio DRY)

```
  WITH CTE_TableName AS (
       SELECT ColumnA, ColumnB
         FROM TableName)
SELECT t0.ColumnA
     , STUFF((
       SELECT ',' + t1.ColumnB
         FROM CTE_TableName t1
        WHERE t1.ColumnA = t0.ColumnA
        ORDER BY t1.ColumnB
          FOR XML PATH('')), 1, 1, '') AS ColumnBs
  FROM CTE_TableName t0
 GROUP BY t0.ColumnA
 ORDER BY ColumnA;
```

SQL Server 2017 e SQL Azure

```
SELECT ColumnA
     , STRING_AGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

SQLite

senza ordinare:

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

l'ordine richiede una subquery o CTE:

```
  WITH CTE_TableName AS (
       SELECT ColumnA, ColumnB
         FROM TableName
        ORDER BY ColumnA, ColumnB)
SELECT ColumnA
     , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
  FROM CTE_TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

Contare

Puoi contare il numero di righe:

```
SELECT count(*) TotalRows
FROM employees;
```

TotalRows
4

O contare i dipendenti per dipartimento:

```
SELECT DepartmentId, count(*) NumEmployees
FROM employees
GROUP BY DepartmentId;
```

DepartmentID	NumEmployees
1	3
2	1

Puoi contare su una colonna / espressione con l'effetto che non conteggia i valori `NULL` :

```
SELECT count(ManagerId) mgr
FROM EMPLOYEES;
```

mgr
3

(C'è una colonna ID gestore valori null)

Puoi anche utilizzare **DISTINCT** all'interno di un'altra funzione come **COUNT** per trovare solo i membri **DISTINCT** del set su cui eseguire l'operazione.

Per esempio:

```
SELECT COUNT(ContinentCode) AllCount
,      COUNT(DISTINCT ContinentCode) SingleCount
FROM Countries;
```

Restituirà valori diversi. Il *SingleCount* conterà solo una volta i singoli Continenti, mentre l' *AllCount* includerà duplicati.

ContinentCode
OC
Unione Europea
COME
N / A
N / A
AF
AF

AllCount: 7 SingleCount: 5

Max

Trova il valore massimo della colonna:

```
select max(age) from employee;
```

L'esempio sopra restituirà il valore più grande per l' `age` della colonna della tabella dei `employee` .

Sintassi:

```
SELECT MAX(column_name) FROM table_name;
```

min

Trova il valore più basso della colonna:

```
 select min(age) from employee;
```

L'esempio sopra restituirà il valore più piccolo per l' `age` della colonna della tabella dei `employee` .

Sintassi:

```
 SELECT MIN(column_name) FROM table_name;
```

Leggi Funzioni (aggregato) online: https://riptutorial.com/it/sql/topic/1002/funzioni--aggregato-

Capitolo 25: Funzioni (analitico)

introduzione

Si utilizzano le funzioni analitiche per determinare i valori in base a gruppi di valori. Ad esempio, è possibile utilizzare questo tipo di funzione per determinare totali parziali, percentuali o il risultato migliore all'interno di un gruppo.

Sintassi

1. FIRST_VALUE (scalar_expression) OVER ([partition_by_clause] order_by_clause [rows_range_clause])
2. LAST_VALUE (scalar_expression) OVER ([partition_by_clause] order_by_clause [rows_range_clause])
3. LAG (scalar_expression [, offset] [, default]) OVER ([partition_by_clause] order_by_clause)
4. LEAD (scalar_expression [, offset], [default]) OVER ([partition_by_clause] order_by_clause)
5. PERCENT_RANK () OVER ([partition_by_clause] order_by_clause)
6. CUME_DIST () OVER ([partition_by_clause] order_by_clause)
7. PERCENTILE_DISC (numeric_literal) WITHIN GROUP (ORDER BY order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])
8. PERCENTILE_CONT (numeric_literal) WITHIN GROUP (ORDER BY order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])

Examples

FIRST_VALUE

Si utilizza la funzione `FIRST_VALUE` per determinare il primo valore in un set di risultati ordinato, che si identifica utilizzando un'espressione scalare.

```
SELECT StateProvinceID, Name, TaxRate,
       FIRST_VALUE(StateProvinceID)
        OVER(ORDER BY TaxRate ASC) AS FirstValue
FROM SalesTaxRate;
```

In questo esempio, la funzione `FIRST_VALUE` viene utilizzata per restituire l' `ID` dello stato o della provincia con l'aliquota d'imposta più bassa. La clausola `OVER` viene utilizzata per ordinare le aliquote fiscali per ottenere il tasso più basso.

StateProvinceID	Nome	Aliquota fiscale	FirstValue
74	Tassa di vendita dello stato dell'Utah	5.00	74
36	Imposta sul reddito dello stato del	6.75	74

StateProvinceID	Nome	Aliquota fiscale	FirstValue
	Minnesota		
30	Imposta sulle vendite dello stato del Massachusetts	7.00	74
1	GST canadese	7.00	74
57	GST canadese	7.00	74
63	GST canadese	7.00	74

LAST_VALUE

La funzione `LAST_VALUE` fornisce l'ultimo valore in un set di risultati ordinato, che si specifica utilizzando un'espressione scalare.

```
SELECT TerritoryID, StartDate, BusinessentityID,
       LAST_VALUE(BusinessentityID)
        OVER(ORDER BY TerritoryID) AS LastValue
FROM SalesTerritoryHistory;
```

Questo esempio utilizza la funzione `LAST_VALUE` per restituire l'ultimo valore per ogni set di righe nei valori ordinati.

TerritoryID	Data d'inizio	BusinessEntityID	LastValue
1	2005-07-01 00.00.00.000	280	283
1	2006-11-01 00.00.00.000	284	283
1	2005-07-01 00.00.00.000	283	283
2	2007-01-01 00.00.00.000	277	275
2	2005-07-01 00.00.00.000	275	275
3	2007-01-01 00.00.00.000	275	277

GAL e LEAD

La funzione `LAG` fornisce i dati sulle righe prima della riga corrente nello stesso set di risultati. Ad esempio, in un'istruzione `SELECT` , è possibile confrontare i valori nella riga corrente con i valori di una riga precedente.

Si utilizza un'espressione scalare per specificare i valori da confrontare. Il parametro offset è il numero di righe prima della riga corrente che verrà utilizzata nel confronto. Se non si specifica il

numero di righe, viene utilizzato il valore predefinito di una riga.

Il parametro predefinito specifica il valore che deve essere restituito quando l'espressione all'offset ha un valore `NULL` . Se non si specifica un valore, viene restituito un valore di `NULL` .

La funzione `LEAD` fornisce i dati sulle righe dopo la riga corrente nel set di righe. Ad esempio, in un'istruzione `SELECT` , è possibile confrontare i valori nella riga corrente con i valori nella seguente riga.

Specificare i valori che dovrebbero essere confrontati utilizzando un'espressione scalare. Il parametro offset è il numero di righe dopo la riga corrente che verrà utilizzata nel confronto.

Specificare il valore che deve essere restituito quando l'espressione all'offset ha un valore `NULL` utilizzando il parametro predefinito. Se non si specificano questi parametri, viene utilizzato il valore predefinito di una riga e viene restituito un valore di `NULL` .

```
SELECT BusinessEntityID, SalesYTD,
       LEAD(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lead value",
       LAG(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lag value"
FROM SalesPerson;
```

Questo esempio utilizza le funzioni LEAD e LAG per confrontare i valori di vendita di ciascun dipendente con quelli degli impiegati elencati sopra e sotto, con i record ordinati in base alla colonna BusinessEntityID.

BusinessEntityID	SalesYTD	Valore guida	Valore di ritardo
274	559697.5639	3763178.1787	0.0000
275	3763178.1787	4251368.5497	559697.5639
276	4251368.5497	3189418.3662	3763178.1787
277	3189418.3662	1453719.4653	4251368.5497
278	1453719.4653	2315185.6110	3189418.3662
279	2315185.6110	1352577.1325	1453719.4653

PERCENT_RANK e CUME_DIST

La funzione `PERCENT_RANK` calcola la classifica di una riga relativa al set di righe. La percentuale si basa sul numero di righe nel gruppo che hanno un valore inferiore rispetto alla riga corrente.

Il primo valore nel set di risultati ha sempre una percentuale di zero. Il valore per il valore più alto o ultimo del set è sempre uno.

La funzione `CUME_DIST` calcola la posizione relativa di un valore specificato in un gruppo di valori, determinando la percentuale di valori inferiore o uguale a quel valore. Questa è chiamata la distribuzione cumulativa.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
PERCENT_RANK() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
       AS "Percent Rank",
CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
       AS "Cumulative Distribution"
FROM Employee;
```

In questo esempio, si utilizza una clausola `ORDER` per partizionare - o raggruppare - le righe recuperate `SELECT` base ai titoli di lavoro dei dipendenti, con i risultati in ogni gruppo ordinati in base al numero di ore di congedo per malattia che i dipendenti hanno utilizzato.

BusinessEntityID	Titolo di lavoro	SickLeaveHours	Classifica percentuale	Distribuzione cumulativa
267	Specialista dell'applicazione	57	0	0.25
268	Specialista dell'applicazione	56	0,333333333333333	0.75
269	Specialista dell'applicazione	56	0,333333333333333	0.75
272	Specialista dell'applicazione	55	1	1
262	Assitant al funzionario finanziario di Cheif	48	0	1
239	Specialista dei vantaggi	45	0	1
252	Acquirente	50	0	0,111111111111111
251	Acquirente	49	0,125	0,333333333333333
256	Acquirente	49	0,125	0,333333333333333
253	Acquirente	48	0.375	0,555555555555555
254	Acquirente	48	0.375	0,555555555555555

La funzione `PERCENT_RANK` classifica le voci all'interno di ciascun gruppo. Per ciascuna voce,

restituisce la percentuale di voci nello stesso gruppo con valori inferiori.

La funzione `CUME_DIST` è simile, tranne che restituisce la percentuale di valori inferiore o uguale al valore corrente.

PERCENTILE_DISC e PERCENTILE_CONT

La funzione `PERCENTILE_DISC` elenca il valore della prima voce in cui la distribuzione cumulativa è superiore al percentile fornito utilizzando il parametro `numeric_literal` .

I valori sono raggruppati per set di righe o partizioni, come specificato dalla clausola `WITHIN GROUP` .

La funzione `PERCENTILE_CONT` è simile alla funzione `PERCENTILE_DISC` , ma restituisce la media della somma della prima voce corrispondente e della voce successiva.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
       CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
       AS "Cumulative Distribution",
       PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Discreet"
FROM Employee;
```

Per trovare il valore esatto dalla riga che corrisponde o supera lo 0,5 percentile, si passa il percentile come valore letterale numerico nella funzione `PERCENTILE_DISC` . La colonna Percentile Discreta in un set di risultati elenca il valore della riga in cui la distribuzione cumulativa è superiore al percentile specificato.

BusinessEntityID	Titolo di lavoro	SickLeaveHours	Distribuzione cumulativa	Percentile discreto
272	Specialista dell'applicazione	55	0.25	**56**
268	Specialista dell'applicazione	56	0.75	**56**
269	Specialista dell'applicazione	56	0.75	**56**
267	Specialista dell'applicazione	57	1	**56**

Per basare il calcolo su un set di valori, si utilizza la funzione `PERCENTILE_CONT` . La colonna "Percentile continuo" nei risultati elenca il valore medio della somma del valore del risultato e il successivo valore di corrispondenza più alto.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
       CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
```

```
       AS "Cumulative Distribution",
       PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Discreet",
       PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Continuous"
FROM Employee;
```

BusinessEntityID	Titolo di lavoro	SickLeaveHours	Distribuzione cumulativa	Percentile discreto	Percentile continuo
272	Specialista dell'applicazione	55	0.25	56	**56**
268	Specialista dell'applicazione	56	0.75	56	**56**
269	Specialista dell'applicazione	56	0.75	56	**56**
267	Specialista dell'applicazione	57	1	56	**56**

Leggi Funzioni (analitico) online: https://riptutorial.com/it/sql/topic/8811/funzioni--analitico-

Capitolo 26: Funzioni (scalare / riga singola)

introduzione

SQL offre diverse funzioni scalari incorporate. Ogni funzione scalare prende un valore come input e restituisce un valore come output per ogni riga in un set di risultati.

Le funzioni scalari vengono utilizzate ovunque sia consentita un'espressione all'interno di un'istruzione T-SQL.

Sintassi

- CAST (espressione AS data_type [(lunghezza)])
- CONVERT (data_type [(length)], expression [, style])
- PARSE (string_value AS data_type [USING culture])
- DATENAME (datapart, data)
- GETDATE ()
- DATEDIFF (datapart, startdate, enddate)
- DATEADD (datapart, numero, data)
- SCEGLI (indice, val_1, val_2 [, val_n])
- IIF (boolean_expression, true_value, false_value)
- SIGN (espressione numerica)
- POTENZA (float_expression, y)

Osservazioni

Le funzioni scalari o a riga singola vengono utilizzate per gestire ciascuna riga di dati nel set di risultati, al contrario delle funzioni aggregate che operano sull'intero set di risultati.

Esistono dieci tipi di funzioni scalari.

1. Le funzioni di configurazione forniscono informazioni sulla configurazione dell'istanza SQL corrente.
2. Le funzioni di conversione convertono i dati nel tipo di dati corretto per una determinata operazione. Ad esempio, questi tipi di funzioni possono riformattare le informazioni convertendo una stringa in una data o un numero per consentire la comparazione di due diversi tipi.
3. Le funzioni di data e ora manipolano i campi contenenti i valori di data e ora. Possono restituire valori numerici, di data o di stringa. Ad esempio, è possibile utilizzare una funzione per recuperare il giorno corrente della settimana o dell'anno o recuperare solo l'anno dalla data.

I valori restituiti dalle funzioni di data e ora dipendono dalla data e dall'ora impostate per il sistema operativo del computer che esegue l'istanza SQL.

4. Funzione logica che esegue operazioni utilizzando operatori logici. Valuta un insieme di condizioni e restituisce un singolo risultato.
5. Le funzioni matematiche eseguono operazioni matematiche, o calcoli, su espressioni numeriche. Questo tipo di funzione restituisce un singolo valore numerico.
6. Le funzioni di metadati recuperano le informazioni su un database specificato, come il nome e gli oggetti del database.
7. Le funzioni di sicurezza forniscono informazioni che è possibile utilizzare per gestire la sicurezza di un database, ad esempio informazioni su utenti e ruoli del database.
8. Le funzioni stringa eseguono operazioni su valori stringa e restituiscono valori numerici o stringa.

Utilizzando le funzioni di stringa, è possibile, ad esempio, combinare dati, estrarre una sottostringa, confrontare stringhe o convertire una stringa in caratteri maiuscoli o minuscoli.

9. Le funzioni di sistema eseguono operazioni e restituiscono informazioni su valori, oggetti e impostazioni per l'istanza SQL corrente
10. Le funzioni statistiche di sistema forniscono varie statistiche sull'istanza SQL corrente, ad esempio per monitorare i livelli di prestazioni correnti del sistema.

Examples

Modifiche del personaggio

Le funzioni di modifica dei caratteri includono la conversione di caratteri in caratteri maiuscoli o minuscoli, la conversione di numeri in numeri formattati, l'esecuzione di manipolazioni di caratteri, ecc.

La funzione `lower(char)` converte il dato parametro di carattere in caratteri a caratteri bassi.

```
SELECT customer_id, lower(customer_last_name) FROM customer;
```

restituire il cognome del cliente modificato da "SMITH" a "smith".

Data e ora

In SQL, si utilizzano tipi di dati di data e ora per memorizzare le informazioni del calendario. Questi tipi di dati includono ora, data, ora minima, datetime, datetime2 e datetimeoffset. Ogni tipo di dati ha un formato specifico.

Tipo di dati	Formato
tempo	hh: mm: ss [.nnnnnnn]
Data	AAAA-MM-DD
smalldatetime	AAAA-MM-GG hh: mm: ss

Tipo di dati	Formato
appuntamento	AAAA-MM-GG hh: mm: ss [.nnn]
datetime2	AAAA-MM-GG hh: mm: ss [.nnnnnnn]
datetimeoffset	AAAA-MM-GG hh: mm: ss [.nnnnnnn] [+/-] hh: mm

La funzione `DATENAME` restituisce il nome o il valore di una parte specifica della data.

```
SELECT DATENAME (weekday,'2017-01-14') as Datename
```

DATENAME
Sabato

Si utilizza la funzione `GETDATE` per determinare la data e l'ora correnti del computer che esegue l'istanza SQL corrente. Questa funzione non include la differenza del fuso orario.

```
SELECT GETDATE() as Systemdate
```

Systemdate
2017-01-14 11: 11: 47.7230728

La funzione `DATEDIFF` restituisce la differenza tra due date.

Nella sintassi, datepart è il parametro che specifica quale parte della data che si desidera utilizzare per calcolare la differenza. La datapart può essere anno, mese, settimana, giorno, ora, minuto, secondo o millisecondo. Quindi si specifica la data di inizio nel parametro startdate e la data di fine nel parametro enddate per il quale si desidera trovare la differenza.

```
SELECT SalesOrderID, DATEDIFF(day, OrderDate, ShipDate)
AS 'Processing time'
FROM Sales.SalesOrderHeader
```

SalesOrderID	Tempo di elaborazione
43659	7
43660	7
43661	7
43662	7

La funzione `DATEADD` consente di aggiungere un intervallo a una parte di una data specifica.

```
SELECT DATEADD (day, 20, '2017-01-14') AS Added20MoreDays
```

Added20MoreDays
2017-02-03 00: 00: 00.000

Funzione di configurazione e conversione

Un esempio di una funzione di configurazione in SQL è la funzione `@@SERVERNAME` . Questa funzione fornisce il nome del server locale su cui è in esecuzione SQL.

```
SELECT @@SERVERNAME AS 'Server'
```

server
SQL064

In SQL, la maggior parte delle conversioni di dati si verificano implicitamente, senza alcun intervento da parte dell'utente.

Per eseguire conversioni che non possono essere completate implicitamente, è possibile utilizzare le funzioni `CAST` o `CONVERT` .

La sintassi della funzione `CAST` è più semplice della sintassi della funzione `CONVERT` , ma è limitata in ciò che può fare.

Qui, utilizziamo entrambe le funzioni `CAST` e `CONVERT` per convertire il tipo di dati datetime nel tipo di dati `varchar` .

La funzione `CAST` usa sempre l'impostazione di stile predefinita. Ad esempio, rappresenterà le date e le ore utilizzando il formato AAAA-MM-GG.

La funzione `CONVERT` usa lo stile di data e ora specificato. In questo caso, 3 specifica il formato data gg / mm / aa.

```
USE AdventureWorks2012
GO
SELECT FirstName + ' ' + LastName + ' was hired on ' +
       CAST(HireDate AS varchar(20)) AS 'Cast',
       FirstName + ' ' + LastName + ' was hired on ' +
       CONVERT(varchar, HireDate, 3) AS 'Convert'
FROM Person.Person AS p
JOIN HumanResources.Employee AS e
ON p.BusinessEntityID = e.BusinessEntityID
GO
```

lanciare	Convertire
David Hamiltion è stato assunto il 2003-02-04	David Hamiltion è stato assunto il 04/02/03

Un altro esempio di una funzione di conversione è la funzione `PARSE` . Questa funzione converte una stringa in un tipo di dati specificato.

Nella sintassi della funzione, si specifica la stringa che deve essere convertita, la parola chiave `AS` e quindi il tipo di dati richiesto. Facoltativamente, è anche possibile specificare la cultura in cui il valore stringa deve essere formattato. Se non lo si specifica, viene utilizzata la lingua per la sessione.

Se il valore stringa non può essere convertito in formato numerico, data o ora, si verificherà un errore. Dovrai quindi utilizzare `CAST` o `CONVERT` per la conversione.

```
SELECT PARSE('Monday, 13 August 2012' AS datetime2 USING 'en-US') AS 'Date in English'
```

Data in inglese
2012-08-13 00: 00: 00.0000000

Funzione logica e matematica

SQL ha due funzioni logiche: `CHOOSE` e `IIF` .

La funzione `CHOOSE` restituisce un elemento da un elenco di valori, in base alla sua posizione nell'elenco. Questa posizione è specificata dall'indice.

Nella sintassi, il parametro index specifica l'elemento ed è un numero intero o intero. Il parametro val_1 ... val_n identifica l'elenco di valori.

```
SELECT CHOOSE(2, 'Human Resources', 'Sales', 'Admin', 'Marketing' ) AS Result;
```

Risultato
I saldi

In questo esempio, si utilizza la funzione `CHOOSE` per restituire la seconda voce in un elenco di reparti.

La funzione `IIF` restituisce uno di due valori, in base a una particolare condizione. Se la condizione è vera, restituirà il valore vero. Altrimenti restituirà un valore falso.

Nella sintassi, il parametro boolean_expression specifica l'espressione booleana. Il parametro true_value specifica il valore che deve essere restituito se boolean_expression restituisce true e il

parametro false_value specifica il valore che deve essere restituito se boolean_expression restituisce false.

```
SELECT BusinessEntityID, SalesYTD,
       IIF(SalesYTD > 200000, 'Bonus', 'No Bonus') AS 'Bonus?'
FROM Sales.SalesPerson
GO
```

BusinessEntityID	SalesYTD	Bonus?
274	559697.5639	indennità
275	3763178.1787	indennità
285	172524.4512	Nessun bonus

In questo esempio, si utilizza la funzione IIF per restituire uno dei due valori. Se le vendite da inizio anno a un venditore sono superiori a 200.000, questa persona avrà diritto a un bonus. Valori inferiori a 200.000 significano che i dipendenti non si qualificano per i bonus.

SQL include diverse funzioni matematiche che è possibile utilizzare per eseguire calcoli sui valori di input e restituire risultati numerici.

Un esempio è la funzione `SIGN` , che restituisce un valore che indica il segno di un'espressione. Il valore di -1 indica un'espressione negativa, il valore di +1 indica un'espressione positiva e 0 indica zero.

```
SELECT SIGN(-20) AS 'Sign'
```

Cartello
-1

Nell'esempio, l'input è un numero negativo, quindi il riquadro Risultati elenca il risultato -1.

Un'altra funzione matematica è la funzione `POWER` . Questa funzione fornisce il valore di un'espressione elevata a una potenza specificata.

Nella sintassi, il parametro float_expression specifica l'espressione e il parametro y specifica la potenza a cui si desidera aumentare l'espressione.

```
SELECT POWER(50, 3) AS Result
```

Risultato
125000

Leggi Funzioni (scalare / riga singola) online: https://riptutorial.com/it/sql/topic/6898/funzioni--scalare---riga-singola-

Capitolo 27: Funzioni della finestra

Examples

Aggiungere le righe totali selezionate a ogni riga

```
SELECT your_columns, COUNT(*) OVER() as Ttl_Rows FROM your_data_set
```

id	nome	Ttl_Rows
1	esempio	5
2	foo	5
3	bar	5
4	baz	5
5	quux	5

Invece di utilizzare due query per ottenere un conteggio della riga, è possibile utilizzare una funzione di aggregazione come finestra e utilizzare il set di risultati completo come finestra. Questo può essere usato come base per ulteriori calcoli senza la complessità degli extra self join.

Impostazione di un flag se altre righe hanno una proprietà comune

Diciamo che ho questi dati:

Articoli da tavola

id	nome	etichetta
1	esempio	unique_tag
2	foo	semplice
42	bar	semplice
3	baz	Ciao
51	quux	mondo

Mi piacerebbe avere tutte quelle righe e sapere se un tag è usato da altre linee

```
SELECT id, name, tag, COUNT(*) OVER (PARTITION BY tag) > 1 AS flag FROM items
```

Il risultato sarà:

id	nome	etichetta	bandiera
1	esempio	unique_tag	falso
2	foo	semplice	vero
42	bar	semplice	vero
3	baz	Ciao	falso
51	quux	mondo	falso

Nel caso in cui il tuo database non abbia OVER e PARTITION puoi usarlo per produrre lo stesso risultato:

```
SELECT id, name, tag, (SELECT COUNT(tag) FROM items B WHERE tag = A.tag) > 1 AS flag FROM
items A
```

Ottenere un totale parziale

Dati questi dati:

Data	quantità
2016/03/12	200
2016/03/11	-50
2016/03/14	100
2016/03/15	100
2016/03/10	-250

```
SELECT date, amount, SUM(amount) OVER (ORDER BY date ASC) AS running
FROM operations
ORDER BY date ASC
```

ti darò

Data	quantità	in esecuzione
2016/03/10	-250	-250
2016/03/11	-50	-300
2016/03/12	200	-100

Data	quantità	in esecuzione
2016/03/14	100	0
2016/03/15	100	-100

Ottenere le N righe più recenti su più raggruppamenti

Dati questi dati

ID utente	Data di completamento
1	2016/07/20
1	2016/07/21
2	2016/07/20
2	2016/07/21
2	2016/07/22

```
;with CTE as
(SELECT *,
        ROW_NUMBER() OVER (PARTITION BY User_ID
                           ORDER BY Completion_Date DESC) Row_Num
FROM    Data)
SELECT * FORM CTE WHERE Row_Num <= n
```

Usando n = 1, otterrai la riga più recente per `user_id` :

ID utente	Data di completamento	row_num
1	2016/07/21	1
2	2016/07/22	1

Ricerca di record "fuori sequenza" utilizzando la funzione LAG ()

Dati questi dati di esempio:

ID	STATO	STATUS_TIME	STATUS_BY
1	UNO	2016-09-28-19.47.52.501398	USER_1
3	UNO	2016-09-28-19.47.52.501511	Utente_2
1	TRE	2016-09-28-19.47.52.501517	Utente_3

ID	STATO	STATUS_TIME	STATUS_BY
3	DUE	2016-09-28-19.47.52.501521	Utente_2
3	TRE	2016-09-28-19.47.52.501524	USER_4

Gli articoli identificati dai valori `ID` devono passare da `STATUS` 'UNO' a 'DUE' a 'TRE' in sequenza, senza saltare gli stati. Il problema è trovare i valori degli utenti (`STATUS_BY`) che violano la regola e passare da "UNO" immediatamente a "TRE".

La funzione analitica di `LAG()` aiuta a risolvere il problema restituendo per ogni riga il valore nella riga precedente:

```
SELECT * FROM (
 SELECT
  t.*,
  LAG(status) OVER (PARTITION BY id ORDER BY status_time) AS prev_status
  FROM test t
) t1 WHERE status = 'THREE' AND prev_status != 'TWO'
```

Nel caso in cui il tuo database non abbia il LAG () puoi usarlo per produrre lo stesso risultato:

```
SELECT A.id, A.status, B.status as prev_status, A.status_time, B.status_time as
prev_status_time
FROM Data A, Data B
WHERE A.id = B.id
AND   B.status_time = (SELECT MAX(status_time) FROM Data where status_time < A.status_time and
id = A.id)
AND   A.status = 'THREE' AND NOT B.status = 'TWO'
```

Leggi Funzioni della finestra online: https://riptutorial.com/it/sql/topic/647/funzioni-della-finestra

Capitolo 28: Funzioni di stringa

introduzione

Le funzioni stringa eseguono operazioni su valori stringa e restituiscono valori numerici o stringa.

Utilizzando le funzioni di stringa, è possibile, ad esempio, combinare dati, estrarre una sottostringa, confrontare stringhe o convertire una stringa in caratteri maiuscoli o minuscoli.

Sintassi

- CONCAT (string_value1, string_value2 [, string_valueN])
- LTRIM (character_expression)
- RTRIM (character_expression)
- SUBSTRING (espressione, inizio, lunghezza)
- ASCII (character_expression)
- REPLICATE (string_expression, integer_expression)
- REVERSE (string_expression)
- UPPER (character_expression)
- TRIM ([caratteri FROM] stringa)
- STRING_SPLIT (stringa, separatore)
- STUFF (character_expression, start, length, replaceWith_expression)
- REPLACE (string_expression, string_pattern, string_replacement)

Osservazioni

Riferimento di funzioni stringa per Transact-SQL / Microsoft

Riferimento a funzioni di stringa per MySQL

Riferimento a funzioni di stringa per PostgreSQL

Examples

Taglia gli spazi vuoti

Trim è usato per rimuovere lo spazio di scrittura all'inizio o alla fine della selezione

In MSSQL non esiste un singolo `TRIM()`

```
SELECT LTRIM('  Hello  ') --returns 'Hello  '
SELECT RTRIM('  Hello  ') --returns '  Hello'
SELECT LTRIM(RTRIM('  Hello  ')) --returns 'Hello'
```

MySql e Oracle

```
SELECT TRIM('  Hello  ') --returns 'Hello'
```

Concatenare

In SQL (standard ANSI / ISO), l'operatore per la concatenazione di stringhe è `||` . Questa sintassi è supportata da tutti i principali database tranne SQL Server:

```
SELECT 'Hello' || 'World' || '!'; --returns HelloWorld!
```

Molti database supportano una funzione `CONCAT` per unire le stringhe:

```
SELECT CONCAT('Hello', 'World'); --returns 'HelloWorld'
```

Alcuni database supportano l'uso di `CONCAT` per unire più di due stringhe (Oracle non lo fa):

```
SELECT CONCAT('Hello', 'World', '!'); --returns 'HelloWorld!'
```

In alcuni database, i tipi non stringa devono essere espressi o convertiti:

```
SELECT CONCAT('Foo', CAST(42 AS VARCHAR(5)), 'Bar'); --returns 'Foo42Bar'
```

Alcuni database (ad esempio Oracle) eseguono conversioni lossless implicite. Ad esempio, un `CONCAT` su un `CLOB` e `NCLOB` produce un `NCLOB` . Un `CONCAT` su un numero e un `varchar2` produce un `varchar2` , ecc .:

```
SELECT CONCAT(CONCAT('Foo', 42), 'Bar') FROM dual; --returns Foo42Bar
```

Alcuni database possono utilizzare l'operatore non standard `+` (ma in più, `+` funziona solo per i numeri):

```
SELECT 'Foo' + CAST(42 AS VARCHAR(5)) + 'Bar';
```

Su SQL Server <2012, dove `CONCAT` non è supportato, `+` è l'unico modo per unire le stringhe.

Maiuscole e minuscole

```
SELECT UPPER('HelloWorld') --returns 'HELLOWORLD'
SELECT LOWER('HelloWorld') --returns 'helloworld'
```

substring

La sintassi è: `SUBSTRING ( string_expression, start, length )` . Si noti che le stringhe SQL sono 1-indicizzate.

```
SELECT SUBSTRING('Hello', 1, 2) --returns 'He'
SELECT SUBSTRING('Hello', 3, 3) --returns 'llo'
```

Questo è spesso usato in congiunzione con la funzione `LEN()` per ottenere gli ultimi `n` caratteri di una stringa di lunghezza sconosciuta.

```
DECLARE @str1 VARCHAR(10) = 'Hello', @str2 VARCHAR(10) = 'FooBarBaz';
SELECT SUBSTRING(@str1, LEN(@str1) - 2, 3) --returns 'llo'
SELECT SUBSTRING(@str2, LEN(@str2) - 2, 3) --returns 'Baz'
```

Diviso

Divide un'espressione di stringa usando un separatore di caratteri. Nota che `STRING_SPLIT()` è una funzione con valori di tabella.

```
SELECT value FROM STRING_SPLIT('Lorem ipsum dolor sit amet.', ' ');
```

Risultato:

```
value
-----
Lorem
ipsum
dolor
sit
amet.
```

Cose

Inserisci una stringa in un'altra, sostituendo 0 o più caratteri in una determinata posizione.

Nota: la posizione `start` è 1-indicizzata (si inizia l'indicizzazione a 1, non a 0).

Sintassi:

```
STUFF ( character_expression , start , length , replaceWith_expression )
```

Esempio:

```
SELECT STUFF('FooBarBaz', 4, 3, 'Hello') --returns 'FooHelloBaz'
```

Lunghezza

server SQL

La LEN non conta lo spazio finale.

```
SELECT LEN('Hello') -- returns 5

SELECT LEN('Hello '); -- returns 5
```

Il DATALENGTH conta lo spazio finale.

```
SELECT DATALENGTH('Hello') -- returns 5

SELECT DATALENGTH('Hello '); -- returns 6
```

Si noti tuttavia che DATALENGTH restituisce la lunghezza della rappresentazione di byte sottostante della stringa, che dipende, tra l'altro, dal set di caratteri utilizzato per memorizzare la stringa.

```
DECLARE @str varchar(100) = 'Hello ' --varchar is usually an ASCII string, occupying 1 byte
per char
SELECT DATALENGTH(@str) -- returns 6

DECLARE @nstr nvarchar(100) = 'Hello ' --nvarchar is a unicode string, occupying 2 bytes per
char
SELECT DATALENGTH(@nstr) -- returns 12
```

Oracolo

Sintassi: lunghezza (carattere)

Esempi:

```
SELECT Length('Bible') FROM dual; --Returns 5
SELECT Length('righteousness') FROM dual; --Returns 13
SELECT Length(NULL) FROM dual; --Returns NULL
```

Vedi anche: LunghezzaB, LunghezzaC, Lunghezza2, Lunghezza4

Sostituire

Sintassi:

`REPLACE(` stringa da cercare `,` stringa da cercare e sostituire `,` stringa da inserire nella stringa originale `)`

Esempio:

```
SELECT REPLACE( 'Peter Steve Tom', 'Steve', 'Billy' ) --Return Values: Peter Billy Tom
```

SINISTRA DESTRA

La sintassi è:
SINISTRA (espressione-stringa, intero)
RIGHT (espressione-stringa, intero)

```
SELECT LEFT('Hello',2)  --return He
SELECT RIGHT('Hello',2) --return lo
```

Oracle SQL non ha le funzioni LEFT e RIGHT. Possono essere emulati con SUBSTR e LENGTH.
SUBSTR (string-expression, 1, intero)
SUBSTR (string-expression, length (string-expression) -integer + 1, intero)

```
SELECT SUBSTR('Hello',1,2)  --return He
SELECT SUBSTR('Hello',LENGTH('Hello')-2+1,2) --return lo
```

INVERSO

La sintassi è: REVERSE (espressione di stringa)

```
SELECT REVERSE('Hello') --returns olleH
```

REPLICARE

La funzione `REPLICATE` concatena una stringa con se stessa un numero specificato di volte.

La sintassi è: REPLICATE (string-expression, integer)

```
SELECT REPLICATE ('Hello',4) --returns 'HelloHelloHelloHello'
```

REGEXP

MySQL 3.19

Controlla se una stringa corrisponde a un'espressione regolare (definita da un'altra stringa).

```
SELECT 'bedded' REGEXP '[a-f]' -- returns True

SELECT 'beam' REGEXP '[a-f]' -- returns False
```

Sostituisci la funzione in sql Seleziona e aggiorna la query

La funzione Sostituisci in SQL viene utilizzata per aggiornare il contenuto di una stringa. La chiamata di funzione è REPLACE () per MySQL, Oracle e SQL Server.

La sintassi della funzione Replace è:

```
REPLACE (str, find, repl)
```

L'esempio seguente sostituisce le occorrenze di `South` e `Southern` nella tabella Impiegati:

Nome di battesimo	Indirizzo
Giacomo	South New York
John	South Boston

Nome di battesimo	Indirizzo
Michael	South San Diego

Seleziona istruzione:

Se applichiamo la seguente funzione di sostituzione:

```
SELECT
    FirstName,
    REPLACE (Address, 'South', 'Southern') Address
FROM Employees
ORDER BY FirstName
```

Risultato:

Nome di battesimo	Indirizzo
Giacomo	Southern New York
John	Boston meridionale
Michael	San Diego del sud

Dichiarazione di aggiornamento:

Possiamo utilizzare una funzione di sostituzione per apportare modifiche permanenti nella nostra tabella attraverso il seguente approccio.

```
Update Employees
Set city = (Address, 'South', 'Southern');
```

Un approccio più comune è quello di utilizzare questo in combinazione con una clausola WHERE come questa:

```
Update Employees
Set Address = (Address, 'South', 'Southern')
Where Address LIKE 'South%';
```

ParseName

DATABASE : SQL Server

La funzione **PARSENAME** restituisce la parte specifica della stringa data (nome dell'oggetto). il nome dell'oggetto può contenere stringhe come nome oggetto, nome proprietario, nome database e nome server.

Maggiori dettagli `MSDN: PARSENAME`

Sintassi

```
PARSENAME('NameOfStringToParse',PartIndex)
```

Esempio

Per ottenere il nome dell'oggetto utilizzare l'indice della parte `1`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',1)  // returns `ObjectName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',1)     // returns `Student`
```

Per ottenere il nome dello schema utilizzare l'indice della parte `2`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',2)  // returns `SchemaName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',2)     // returns `school`
```

Per ottenere il nome del database utilizzare l'indice della parte `3`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',3) // returns `DatabaseName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',3)    // returns `SchoolDatabase`
```

Per ottenere il nome del server utilizzare l'indice della parte `4`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',4)  // returns `ServerName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',4)     // returns `[1012-1111]`
```

PARSENAME restituisce null è specificato parte non è presente nella stringa nome oggetto specificata

INSTR

Restituisce l'indice della prima occorrenza di una sottostringa (zero se non trovata)

Sintassi: INSTR (stringa, sottostringa)

```
SELECT INSTR('FooBarBar', 'Bar') -- return 4
SELECT INSTR('FooBarBar', 'Xar') -- return 0
```

Leggi Funzioni di stringa online: https://riptutorial.com/it/sql/topic/1120/funzioni-di-stringa

Capitolo 29: Identifier

introduzione

Questo argomento riguarda gli identificatori, ovvero le regole di sintassi per i nomi di tabelle, colonne e altri oggetti di database.

Laddove appropriato, gli esempi dovrebbero coprire le variazioni utilizzate dalle diverse implementazioni SQL o identificare l'implementazione SQL dell'esempio.

Examples

Identificatori non quotati

Gli identificatori non quotati possono utilizzare lettere (`a` - `z`), cifre (`0` - `9`) e caratteri di sottolineatura (`_`) e devono iniziare con una lettera.

A seconda dell'implementazione SQL e / o delle impostazioni del database, possono essere consentiti altri caratteri, alcuni anche come primo carattere, ad es

- MS SQL: `@` , `$` , `#` e altre lettere Unicode *(fonte)*
- MySQL: `$` *(fonte)*
- Oracle: `$` , `#` e altre lettere dal set di caratteri del database *(origine)*
- PostgreSQL: `$` e altre lettere Unicode *(fonte)*

Gli identificatori non quotati non fanno distinzione tra maiuscole e minuscole. Il modo in cui viene gestito dipende molto dall'implementazione SQL:

- MS SQL: conservazione delle maiuscole, sensibilità definita dal set di caratteri del database, quindi può essere sensibile al maiuscolo / minuscolo.
- MySQL: conservazione delle maiuscole, la sensibilità dipende dall'impostazione del database e dal file system sottostante.
- Oracle: convertito in maiuscolo, quindi gestito come identificatore quotato.
- PostgreSQL: convertito in lettere minuscole, quindi gestito come identificatore quotato.
- SQLite: conservazione delle maiuscole; maiuscole e minuscole solo per caratteri ASCII.

Leggi Identifier online: https://riptutorial.com/it/sql/topic/9677/identifier

Capitolo 30: indici

introduzione

Gli indici sono una struttura dati che contiene puntatori ai contenuti di una tabella disposti in un ordine specifico, per aiutare il database a ottimizzare le query. Sono simili all'indice del libro, dove le pagine (righe della tabella) sono indicizzate dal loro numero di pagina.

Esistono diversi tipi di indici e possono essere creati su una tabella. Quando un indice esiste sulle colonne utilizzate nella clausola WHERE di una query, clausola JOIN o clausola ORDER BY, può migliorare sostanzialmente le prestazioni della query.

Osservazioni

Gli indici sono un modo per accelerare le query di lettura ordinando le righe di una tabella in base a una colonna.

L'effetto di un indice non è evidente per i piccoli database come nell'esempio, ma se c'è un numero elevato di righe, può migliorare notevolmente le prestazioni. Invece di controllare ogni riga della tabella, il server può eseguire una ricerca binaria sull'indice.

Il compromesso per la creazione di un indice è la velocità di scrittura e la dimensione del database. La memorizzazione dell'indice richiede spazio. Inoltre, ogni volta che viene eseguito un INSERT o la colonna viene aggiornata, l'indice deve essere aggiornato. Questa operazione non è costosa quanto la scansione dell'intera tabella su una query SELECT, ma è ancora qualcosa da tenere a mente.

Examples

Creare un indice

```
CREATE INDEX ix_cars_employee_id ON Cars (EmployeeId);
```

Questo creerà un indice per la colonna *EmployeeId* nella tabella *Cars* . Questo indice migliorerà la velocità delle query che chiedono al server di ordinare o selezionare in base ai valori in *EmployeeId* , ad esempio:

```
SELECT * FROM Cars WHERE EmployeeId = 1
```

L'indice può contenere più di 1 colonna, come di seguito;

```
CREATE INDEX ix_cars_e_c_o_ids ON Cars (EmployeeId, CarId, OwnerId);
```

In questo caso, l'indice sarebbe utile per le domande che richiedono di ordinare o selezionare da

tutte le colonne incluse, se l'insieme di condizioni è ordinato allo stesso modo. Ciò significa che quando si recuperano i dati, è possibile trovare le righe da recuperare utilizzando l'indice, anziché esaminare l'intera tabella.

Ad esempio, il caso seguente utilizzerà il secondo indice;

```
SELECT * FROM Cars WHERE EmployeeId = 1 Order by CarId DESC
```

Se l'ordine è diverso, tuttavia, l'indice non presenta gli stessi vantaggi, come nel seguito;

```
SELECT * FROM Cars WHERE OwnerId = 17 Order by CarId DESC
```

L'indice non è così utile perché il database deve recuperare l'intero indice, attraverso tutti i valori di EmployeeId e CarID, per trovare quali elementi hanno `OwnerId = 17` .

(L'indice può ancora essere utilizzato, potrebbe essere il caso in cui Query Optimizer rileva che il recupero dell'indice e del filtro su `OwnerId` , quindi il recupero solo delle righe necessarie è più veloce del recupero dell'intera tabella, specialmente se la tabella è grande.)

Indici raggruppati, univoci e ordinati

Gli indici possono avere diverse caratteristiche che possono essere impostate alla creazione o alterando gli indici esistenti.

```
CREATE CLUSTERED INDEX ix_clust_employee_id ON Employees(EmployeeId, Email);
```

L'istruzione SQL sopra riportata crea un nuovo indice cluster su Dipendenti. Gli indici raggruppati sono indici che dettano la struttura effettiva della tabella; la tabella stessa è ordinata per corrispondere alla struttura dell'indice. Ciò significa che può esserci al massimo un indice cluster su un tavolo. Se un indice cluster esiste già sulla tabella, l'istruzione sopra avrà esito negativo. (Le tabelle senza indici cluster sono anche chiamate heap).

```
CREATE UNIQUE INDEX uq_customers_email ON Customers(Email);
```

Ciò creerà un indice univoco per la colonna *Email* nella tabella *Clienti* . Questo indice, oltre a velocizzare le query come un normale indice, imporrà anche l'unicità di ogni indirizzo email in quella colonna. Se una riga viene inserita o aggiornata con un valore di *E - mail* non univoco, l'inserimento o l'aggiornamento, per impostazione predefinita, falliranno.

```
CREATE UNIQUE INDEX ix_eid_desc ON Customers(EmployeeID);
```

Ciò crea un indice su Clienti che crea anche un vincolo di tabella che EmployeeID deve essere univoco. (Questo fallirà se la colonna non è al momento unica - in questo caso, se ci sono impiegati che condividono un ID.)

```
CREATE INDEX ix_eid_desc ON Customers(EmployeeID Desc);
```

Questo crea un indice che è ordinato in ordine decrescente. Per impostazione predefinita, gli indici (almeno nel server MSSQL) sono in ordine crescente, ma possono essere modificati.

Inserimento con un indice unico

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1;
```

Ciò fallirà se un indice univoco è impostato sulla colonna *Email* dei *Clienti* . Tuttavia, per questo caso è possibile definire un comportamento alternativo:

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1 ON DUPLICATE KEY;
```

SAP ASE: Drop index

Questo comando farà cadere l'indice nella tabella. Funziona sul server `SAP ASE` .

Sintassi:

```
DROP INDEX [table name].[index name]
```

Esempio:

```
DROP INDEX Cars.index_1
```

Indice ordinato

Se si utilizza un indice ordinato nel modo in cui lo si recupera, l'istruzione `SELECT` non eseguirà ulteriori ordinamenti durante il recupero.

```
CREATE INDEX ix_scoreboard_score ON scoreboard (score DESC);
```

Quando si esegue la query

```
SELECT * FROM scoreboard ORDER BY score DESC;
```

Il sistema di database non eseguirà ulteriori ordinamenti, poiché può eseguire una ricerca dell'indice in tale ordine.

Eliminazione di un indice o disattivazione e ricostruzione

```
DROP INDEX ix_cars_employee_id ON Cars;
```

Possiamo usare il comando `DROP` per cancellare il nostro indice. In questo esempio `DROP` l'indice chiamato *ix_cars_employee_id* sul tavolo *Cars* .

Questo elimina completamente l'indice e, se l'indice è in cluster, rimuoverà qualsiasi cluster. Non

può essere ricostruito senza ricreare l'indice, che può essere lento e computazionalmente costoso. In alternativa, l'indice può essere disabilitato:

```
ALTER INDEX ix_cars_employee_id ON Cars DISABLE;
```

Ciò consente alla tabella di mantenere la struttura, insieme ai metadati relativi all'indice.

Criticamente, questo mantiene le statistiche dell'indice, in modo che sia possibile valutare facilmente la modifica. Se giustificato, l'indice può successivamente essere ricostruito, invece di essere ricreato completamente;

```
ALTER INDEX ix_cars_employee_id ON Cars REBUILD;
```

Indice univoco che consente NULLS

```
CREATE UNIQUE INDEX idx_license_id
   ON Person(DrivingLicenseID) WHERE DrivingLicenseID IS NOT NULL
GO
```

Questo schema consente una relazione 0..1 - le persone possono avere zero o una patente di guida e ogni licenza può appartenere a una sola persona

Ricostruisci indice

Nel corso del tempo gli indici B-Tree potrebbero diventare frammentati a causa dell'aggiornamento / eliminazione / inserimento dei dati. Nella terminologia di SQLServer possiamo avere interno (pagina indice che è mezzo vuoto) ed esterno (l'ordine logico delle pagine non corrisponde all'ordine fisico). L'indice di ricostruzione è molto simile a lasciarlo cadere e ricrearlo.

Possiamo ricostruire un indice con

```
ALTER INDEX index_name REBUILD;
```

L'indice di ricostruzione predefinito è un'operazione offline che blocca la tabella e impedisce DML contro di essa, ma molti RDBMS consentono la ricostruzione in linea. Inoltre, alcuni produttori di DB offrono alternative alla ricostruzione degli indici come `REORGANIZE` (SQLServer) o `COALESCE` / `SHRINK SPACE` (Oracle).

Indice raggruppato

Quando si utilizza l'indice cluster, le righe della tabella vengono ordinate in base alla colonna a cui viene applicato l'indice cluster. Pertanto, può esistere un solo indice cluster sulla tabella poiché non è possibile ordinare la tabella in base a due colonne diverse.

In genere, è preferibile utilizzare l'indice cluster durante l'esecuzione di letture su tabelle di big data. Il rastrello dell'indice cluster è quando si scrive sulla tabella e i dati devono essere

riorganizzati (ricorsi).

Un esempio di creazione di un indice cluster su una tabella Dipendenti sulla colonna Employee_Surname:

```
CREATE CLUSTERED INDEX ix_employees_name ON Employees(Employee_Surname);
```

Indice non raggruppato

Gli indici non cluster sono memorizzati separatamente dalla tabella. Ogni indice in questa struttura contiene un puntatore alla riga nella tabella che rappresenta.

Questi indicatori sono chiamati localizzatori di riga. La struttura del localizzatore di riga dipende dal fatto che le pagine di dati siano archiviate in un heap o in una tabella in cluster. Per un heap, un localizzatore di riga è un puntatore alla riga. Per una tabella in cluster, il localizzatore di riga è la chiave di indice cluster.

Un esempio di creazione di un indice non cluster sulla tabella Dipendenti e colonna Employee_Surname:

```
CREATE NONCLUSTERED INDEX ix_employees_name ON Employees(Employee_Surname);
```

Ci possono essere più indici non cluster sulla tabella. Le operazioni di lettura sono generalmente più lente con gli indici non clusterizzati rispetto agli indici cluster, in quanto è necessario innanzitutto passare all'indice e alla tabella. Non ci sono tuttavia restrizioni nelle operazioni di scrittura.

Indice parziale o filtrato

SQL Server e SQLite consentono di creare indici che contengono non solo un sottoinsieme di colonne, ma anche un sottoinsieme di righe.

Considera una quantità crescente di ordini con `order_state_id` uguale a finished (2) e una quantità stabile di ordini con `order_state_id equal` a started (1).

Se la tua azienda fa uso di query come questa:

```
SELECT id, comment
  FROM orders
 WHERE order_state_id =  1
   AND product_id = @some_value;
```

L'indicizzazione parziale consente di limitare l'indice, inclusi solo gli ordini incompleti:

```
CREATE INDEX Started_Orders
          ON orders(product_id)
       WHERE order_state_id = 1;
```

Questo indice sarà più piccolo di un indice non filtrato, che consente di risparmiare spazio e riduce

il costo di aggiornamento dell'indice.

Leggi indici online: https://riptutorial.com/it/sql/topic/344/indici

Capitolo 31: INSERIRE

Sintassi

- INSERISCI IN nome_tabella (colonna1, colonna2, colonna3, ...) VALORI (valore1, valore2, valore3, ...);
- INSERIRE IN nome_tabella (colonna1, colonna2 ...) SELEZIONA valore1, valore2 ... da altro_table

Examples

Inserisci nuova riga

```
INSERT INTO Customers
VALUES ('Zack', 'Smith', 'zack@example.com', '7049989942', 'EMAIL');
```

Questa affermazione inserirà una nuova riga nella tabella `Customers` . Si noti che non è stato specificato un valore per la colonna `Id` , poiché verrà aggiunto automaticamente. Tuttavia, tutti gli altri valori di colonna devono essere specificati.

Inserisci solo colonne specificate

```
INSERT INTO Customers (FName, LName, Email, PreferredContact)
VALUES ('Zack', 'Smith', 'zack@example.com', 'EMAIL');
```

Questa affermazione inserirà una nuova riga nella tabella `Customers` . I dati verranno inseriti solo nelle colonne specificate - si noti che non è stato fornito alcun valore per la colonna `PhoneNumber` . Si noti, tuttavia, che tutte le colonne contrassegnate come `not null` devono essere incluse.

INSERIRE i dati da un'altra tabella usando SELECT

```
INSERT INTO Customers (FName, LName, PhoneNumber)
SELECT FName, LName, PhoneNumber FROM Employees
```

Questo esempio inserirà tutti i Dipendenti nella tabella Clienti . Poiché le due tabelle hanno campi diversi e non si desidera spostare tutti i campi sopra, è necessario impostare quali campi inserire e quali campi selezionare. I nomi dei campi correlati non devono essere chiamati la stessa cosa, ma devono essere lo stesso tipo di dati. Questo esempio presuppone che il campo Id abbia un set di Identity Specification e verrà incrementato automaticamente.

Se hai due tabelle che hanno esattamente lo stesso nome di campo e vuoi semplicemente spostare tutti i record su di te puoi usare:

```
INSERT INTO Table1
SELECT * FROM Table2
```

Inserisci più righe contemporaneamente

È possibile inserire più righe con un solo comando di inserimento:

```
INSERT INTO tbl_name (field1, field2, field3)

VALUES (1,2,3), (4,5,6), (7,8,9);
```

Per l'inserimento di grandi quantità di dati (bulk insert) allo stesso tempo, esistono caratteristiche e raccomandazioni specifiche per DBMS.

MySQL - LOAD DATA INFILE

MSSQL - BULK INSERT

Leggi INSERIRE online: https://riptutorial.com/it/sql/topic/465/inserire

Capitolo 32: Le transazioni

Osservazioni

Una transazione è un'unità logica di lavoro che contiene uno o più passaggi, ognuno dei quali deve essere completato correttamente affinché la transazione possa eseguire il commit nel database. Se ci sono errori, tutte le modifiche ai dati vengono cancellate e il database viene riportato allo stato iniziale all'inizio della transazione.

Examples

Transazione semplice

```
BEGIN TRANSACTION
    INSERT INTO DeletedEmployees(EmployeeID, DateDeleted, User)
        (SELECT 123, GetDate(), CURRENT_USER);
    DELETE FROM Employees WHERE EmployeeID = 123;
COMMIT TRANSACTION
```

Transazione di rollback

Quando qualcosa non funziona nel tuo codice transazione e vuoi annullarlo, puoi eseguire il rollback della transazione:

```
BEGIN TRY
    BEGIN TRANSACTION
        INSERT INTO Users(ID, Name, Age)
        VALUES(1, 'Bob', 24)

        DELETE FROM Users WHERE Name = 'Todd'
   COMMIT TRANSACTION
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION
END CATCH
```

Leggi Le transazioni online: https://riptutorial.com/it/sql/topic/2424/le-transazioni

Capitolo 33: MERGE

introduzione

MERGE (spesso chiamato anche UPSERT per "aggiornamento o inserimento") consente di inserire nuove righe o, se esiste già una riga, di aggiornare la riga esistente. Il punto è di eseguire l'intero set di operazioni atomicamente (per garantire che i dati rimangano coerenti) e per prevenire l'overhead di comunicazione per più istruzioni SQL in un sistema client / server.

Examples

MERGE per rendere Target match Source

```
MERGE INTO targetTable t
    USING sourceTable s
        ON t.PKID = s.PKID
    WHEN MATCHED AND NOT EXISTS (
            SELECT s.ColumnA, s.ColumnB, s.ColumnC
            INTERSECT
            SELECT t.ColumnA, t.ColumnB, s.ColumnC
            )
        THEN UPDATE SET
            t.ColumnA = s.ColumnA
            ,t.ColumnB = s.ColumnB
            ,t.ColumnC = s.ColumnC
    WHEN NOT MATCHED BY TARGET
        THEN INSERT (PKID, ColumnA, ColumnB, ColumnC)
        VALUES (s.PKID, s.ColumnA, s.ColumnB, s.ColumnC)
    WHEN NOT MATCHED BY SOURCE
        THEN DELETE
    ;
```

Nota: la parte `AND NOT EXISTS` impedisce l'aggiornamento di record che non sono stati modificati. L'utilizzo del costrutto `INTERSECT` consente di confrontare le colonne nullable senza una gestione speciale.

MySQL: contando gli utenti per nome

Supponiamo di voler sapere quanti utenti hanno lo stesso nome. Cerchiamo di creare `users` tabelle come segue:

```
create table users(
    id int primary key auto_increment,
    name varchar(8),
    count int,
    unique key name(name)
);
```

Ora, abbiamo appena scoperto un nuovo utente di nome Joe e vorrei tenerlo in considerazione.

Per riuscirci, dobbiamo determinare se esiste una riga esistente con il suo nome e, in tal caso, aggiornarla per incrementare il conteggio; d'altra parte, se non esiste una riga esistente, dovremmo crearla.

MySQL utilizza la seguente sintassi: inserire ... sull'aggiornamento della chiave duplicato .... In questo caso:

```
insert into users(name, count)
       values ('Joe', 1)
       on duplicate key update count=count+1;
```

PostgreSQL: conteggio degli utenti per nome

Supponiamo di voler sapere quanti utenti hanno lo stesso nome. Cerchiamo di creare `users` tabelle come segue:

```
create table users(
    id serial,
    name varchar(8) unique,
    count int
);
```

Ora, abbiamo appena scoperto un nuovo utente di nome Joe e vorrei tenerlo in considerazione. Per riuscirci, dobbiamo determinare se esiste una riga esistente con il suo nome e, in tal caso, aggiornarla per incrementare il conteggio; d'altra parte, se non esiste una riga esistente, dovremmo crearla.

PostgreSQL usa la seguente sintassi: insert ... on conflict ... do update .... In questo caso:

```
insert into users(name, count)
    values('Joe', 1)
    on conflict (name) do update set count = users.count + 1;
```

Leggi MERGE online: https://riptutorial.com/it/sql/topic/1470/merge

Capitolo 34: NULLO

introduzione

`NULL` in SQL, oltre alla programmazione in generale, significa letteralmente "niente". In SQL, è più facile capire come "l'assenza di qualsiasi valore".

È importante distinguerlo da valori apparentemente vuoti, come la stringa vuota `''` o il numero `0` , nessuno dei quali è in realtà `NULL` .

È inoltre importante fare attenzione a non includere `NULL` tra virgolette, come `'NULL'` , che è consentito nelle colonne che accettano il testo, ma non è `NULL` e può causare errori e set di dati errati.

Examples

Filtro per NULL nelle query

La sintassi per il filtraggio di `NULL` (ovvero l'assenza di un valore) nei blocchi `WHERE` è leggermente diversa rispetto al filtro per valori specifici.

```
SELECT * FROM Employees WHERE ManagerId IS NULL ;
SELECT * FROM Employees WHERE ManagerId IS NOT NULL ;
```

Si noti che poiché `NULL` non è uguale a nulla, nemmeno a se stesso, usando gli operatori di uguaglianza `= NULL` o `<> NULL` (o `!= NULL`) produrrà sempre il valore di verità di `UNKNOWN` che verrà rifiutato da `WHERE` .

`WHERE` filtra tutte le righe in cui la condizione è `FALSE` o `UKNOWN` e mantiene solo le righe in cui la condizione è `TRUE` .

Colonne di Nullable nei tavoli

Quando si creano tabelle è possibile dichiarare una colonna come nullable o non annullabile.

```
CREATE TABLE MyTable
(
    MyCol1 INT NOT NULL, -- non-nullable
    MyCol2 INT NULL      -- nullable
) ;
```

Di default ogni colonna (eccetto quelle nel vincolo della chiave primaria) è nullable a meno che non si imposti esplicitamente il vincolo `NOT NULL` .

Se si tenta di assegnare `NULL` a una colonna non nullable, si verificherà un errore.

```
INSERT INTO MyTable (MyCol1, MyCol2) VALUES (1, NULL) ;  -- works fine

INSERT INTO MyTable (MyCol1, MyCol2) VALUES (NULL, 2) ;
        -- cannot insert
        -- the value NULL into column 'MyCol1', table 'MyTable';
        -- column does not allow nulls. INSERT fails.
```

Aggiornamento dei campi su NULL

L'impostazione di un campo su `NULL` funziona esattamente come con qualsiasi altro valore:

```
UPDATE Employees
SET ManagerId = NULL
WHERE Id = 4
```

Inserimento di righe con campi NULL

Ad esempio inserendo un dipendente senza numero di telefono e nessun manager nella tabella di esempio Dipendenti :

```
INSERT INTO Employees
    (Id, FName, LName, PhoneNumber, ManagerId, DepartmentId, Salary, HireDate)
VALUES
    (5, 'Jane', 'Doe', NULL, NULL, 2, 800, '2016-07-22') ;
```

Leggi NULLO online: https://riptutorial.com/it/sql/topic/3421/nullo

Capitolo 35: Numero di riga

Sintassi

- ROW_NUMBER ()
- OVER ([PARTITION BY value_expression, ... [n]] order_by_clause)

Examples

Numeri di riga senza partizioni

Includere un numero di riga in base all'ordine specificato.

```
SELECT
  ROW_NUMBER() OVER(ORDER BY Fname ASC) AS RowNumber,
  Fname,
  LName
FROM Employees
```

Numeri di riga con partizioni

Utilizza un criterio di partizione per raggruppare la numerazione delle righe in base ad essa.

```
SELECT
  ROW_NUMBER() OVER(PARTITION BY DepartmentId ORDER BY DepartmentId ASC) AS RowNumber,
  DepartmentId, Fname, LName
FROM Employees
```

Elimina tutto tranne l'ultimo record (da 1 a molti tabella)

```
WITH cte AS (
  SELECT ProjectID,
         ROW_NUMBER() OVER (PARTITION BY ProjectID ORDER BY InsertDate DESC) AS rn
  FROM ProjectNotes
)
DELETE FROM cte WHERE rn > 1;
```

Leggi Numero di riga online: https://riptutorial.com/it/sql/topic/1977/numero-di-riga

Capitolo 36: Operatore LIKE

Sintassi

- **Wild Card con%:** SELECT * FROM [table] WHERE [column_name] Like '% Value%'

  Wild Card con _: SELECT * FROM [table] WHERE [column_name] Come 'V_n%'

  Wild Card con [charlist]: SELECT * FROM [table] WHERE [column_name] Like 'V [abc] n%'

Osservazioni

La condizione LIKE nella clausola WHERE viene utilizzata per cercare valori di colonna corrispondenti al modello specificato. I modelli sono formati usando i seguenti due caratteri jolly

- % (Simbolo percentuale): utilizzato per rappresentare zero o più caratteri
- _ (Sottolineatura) - Usato per rappresentare un singolo carattere

Examples

Abbina il modello aperto

Il carattere jolly `%` aggiunto all'inizio o alla fine (o entrambi) di una stringa consentirà che 0 o più di qualsiasi carattere prima dell'inizio o dopo la fine del modello corrisponda.

L'utilizzo di '%' nel mezzo consentirà che 0 o più caratteri tra le due parti del modello corrispondano.

Utilizzeremo questa tabella dei dipendenti:

Id	FName	LName	Numero di telefono	ManagerID	DepartmentID	Stipendio	Data di assunzione
1	John	Johnson	2468101214	1	1	400	23-03-2005
2	Sophie	Amudsen	2479100211	1	1	400	11-01-2010
3	Ronny	fabbro	2462544026	2	1	600	2015/06/08
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005
5	Hilde	Knag	2468021911	2	1	800	01-01-2000

Seguendo le corrispondenze delle istruzioni per tutti i record con FName **contenente la** stringa "on" dalla tabella dei dipendenti.

```
SELECT * FROM Employees WHERE FName LIKE '%on%';
```

Id	FName	LName	Numero di telefono	ManagerID	DepartmentID	Stipendio	Data di assunzione
3	R **on** ny	fabbro	2462544026	2	1	600	2015/06/08
4	J **on**	Sanchez	2454124602	1	1	400	23-03-2005

La seguente istruzione corrisponde a tutti i record che hanno PhoneNumber che **inizia con la** stringa "246" dei dipendenti.

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '246%';
```

Id	FName	LName	Numero di telefono	ManagerID	DepartmentID	Stipendio	Data di assunzione
1	John	Johnson	**246** 8101214	1	1	400	23-03-2005
3	Ronny	fabbro	**246** 2544026	2	1	600	2015/06/08
5	Hilde	Knag	**246** 8021911	2	1	800	01-01-2000

La seguente istruzione corrisponde a tutti i record che hanno PhoneNumber che **termina con la** stringa "11" di Employees.

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '%11'
```

Id	FName	LName	Numero di telefono	ManagerID	DepartmentID	Stipendio	Data di assunzione
2	Sophie	Amudsen	24791002 **11**	1	1	400	11-01-2010
5	Hilde	Knag	24680219 **11**	2	1	800	01-01-2000

Tutti i record in cui Fname **3 ° carattere** è 'n' da Impiegati.

```
SELECT * FROM Employees WHERE FName LIKE '__n%';
```

(due trattini bassi vengono usati prima di 'n' per saltare i primi 2 caratteri)

Id	FName	LName	Numero di telefono	ManagerID	DepartmentID	Stipendio	Data di assunzione
3	Ronny	fabbro	2462544026	2	1	600	2015/06/08
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005

Partita a singolo carattere

Per ampliare le selezioni di un'istruzione di linguaggio di query strutturata (SQL-SELECT), è possibile utilizzare i caratteri jolly, il segno di percentuale (%) e il carattere di sottolineatura (_).

Il carattere _ (carattere di sottolineatura) può essere utilizzato come carattere jolly per ogni singolo carattere in una corrispondenza di modello.

Trova tutti i dipendenti i cui Fname iniziano con 'j' e terminano con 'n' e ha esattamente 3 caratteri in Fname.

```
SELECT * FROM Employees WHERE FName LIKE 'j_n'
```

_ (carattere di sottolineatura) può essere utilizzato anche più di una volta come carattere jolly per abbinare i motivi.

Ad esempio, questo modello corrisponderebbe a "jon", "jan", "jen", ecc.

Questi nomi non verranno visualizzati "jn", "john", "jordan", "justin", "jason", "julian", "jillian", "joann" perché nella nostra query viene usato un carattere di sottolineatura e può saltare esattamente un personaggio, quindi il risultato deve essere di 3 caratteri Fname.

Ad esempio, questo modello corrisponderebbe a "LaSt", "LoSt", "HaLt", ecc.

```
SELECT * FROM Employees WHERE FName LIKE '_A_T'
```

Corrispondenza per intervallo o set

Abbina qualsiasi carattere singolo nell'intervallo specificato (es: `[af]`) o impostato (es: `[abcdef]`).

Questo modello di gamma corrisponderebbe a "gary" ma non a "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[a-g]ary'
```

Questo schema impostato corrisponderebbe a "mary" ma non a "gary":

```
SELECT * FROM Employees WHERE Fname LIKE '[lmnop]ary'
```

L'intervallo o l'insieme possono anche essere negati aggiungendo il carattere `^` prima dell'intervallo o del set:

Questo modello di intervallo *non* corrisponderebbe a "gary" ma corrisponderà a "mary":

```
SELECT * FROM Employees WHERE FName LIKE '[^a-g]ary'
```

Questo schema impostato *non* corrisponderebbe a "mary" ma corrisponderà a "gary":

```
SELECT * FROM Employees WHERE Fname LIKE '[^lmnop]ary'
```

Abbina QUALSIASI contro TUTTI

Abbina qualsiasi:
Deve corrispondere ad almeno una stringa. In questo esempio il tipo di prodotto deve essere "elettronica", "libri" o "video".

```
SELECT *
FROM   purchase_table
WHERE  product_type LIKE ANY ('electronics', 'books', 'video');
```

Abbina tutto (deve soddisfare tutti i requisiti).
In questo esempio, sia "regno unito" *che* "londra" *e* "strada orientale" (comprese le varianti) devono essere abbinati.

```
SELECT *
FROM   customer_table
WHERE  full_address LIKE ALL ('%united kingdom%', '%london%', '%eastern road%');
```

Selezione negativa:
Usa TUTTO per escludere tutti gli elementi.
Questo esempio produce tutti i risultati in cui il tipo di prodotto non è "elettronica" e non "libri" e non "video".

```
SELECT *
FROM   customer_table
WHERE  product_type NOT LIKE ALL ('electronics', 'books', 'video');
```

Cerca un intervallo di caratteri

La seguente istruzione corrisponde a tutti i record con FName che inizia con una lettera da A a F dalla tabella dei dipendenti .

```
SELECT * FROM Employees WHERE FName LIKE '[A-F]%'
```

Istruzione ESCAPE nella query LIKE

Se implementi una ricerca di testo come `LIKE` -query, di solito lo fai in questo modo:

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%')
```

Tuttavia, (a parte il fatto che non si dovrebbe utilizzare necessariamente `LIKE` quando è possibile utilizzare la ricerca fulltext), ciò crea un problema quando qualcuno inserisce un testo come "50%" o "a_b".

Quindi (invece di passare alla ricerca fulltext), puoi risolvere il problema usando l'istruzione `LIKE` - escape:

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%') ESCAPE '\'
```

Ciò significa che `\` ora verrà trattato come carattere ESCAPE. Ciò significa, è ora possibile solo anteporre `\` ad ogni carattere della stringa si cerca, ed i risultati inizieranno a essere corretti, anche quando l'utente inserisce un carattere speciale come `%` o `_` .

per esempio

```
string stringToSearch = "abc_def 50%";
string newString = "";
foreach(char c in stringToSearch)
     newString += @"\" + c;

sqlCmd.Parameters.Add("@in_SearchText", newString);
// instead of sqlCmd.Parameters.Add("@in_SearchText", stringToSearch);
```

Nota: l'algoritmo di cui sopra è solo a scopo dimostrativo. Non funzionerà nei casi in cui 1 grafema è composto da più caratteri (utf-8). eg `string stringToSearch = "Les Mise\u0301rables";` Dovrai farlo per ogni grafema, non per ogni personaggio. Non si dovrebbe usare l'algoritmo di cui sopra se si ha a che fare con lingue asiatiche / est-asiatiche / sud-asiatiche. O meglio, se vuoi che il codice corretto abbia inizio, dovresti farlo per ogni graphemeCluster.

Vedi anche ReverseString, una domanda-intervista in C #

Caratteri jolly

i caratteri jolly vengono utilizzati con l'operatore SQL LIKE. I caratteri jolly SQL vengono utilizzati per cercare dati all'interno di una tabella.

I caratteri jolly in SQL sono:%, _, [charlist], [^ charlist]

% - Un sostituto per zero o più caratteri

```
   Eg:  //selects all customers with a City starting with "Lo"
        SELECT * FROM Customers
```

```
        WHERE City LIKE 'Lo%';

       //selects all customers with a City containing the pattern "es"
      SELECT * FROM Customers
       WHERE City LIKE '%es%';
```

_ - Un sostituto per un singolo personaggio

```
Eg://selects all customers with a City starting with any character, followed by "erlin"
SELECT * FROM Customers
WHERE City LIKE '_erlin';
```

[charlist] - Set e intervalli di caratteri da abbinare

```
Eg://selects all customers with a City starting with "a", "d", or "l"
SELECT * FROM Customers
WHERE City LIKE '[adl]%';

//selects all customers with a City starting with "a", "d", or "l"
SELECT * FROM Customers
WHERE City LIKE '[a-c]%';
```

[^ charlist] - Corrisponde solo a un carattere NON specificato tra parentesi

```
Eg://selects all customers with a City starting with a character that is not "a", "p", or "l"
SELECT * FROM Customers
WHERE City LIKE '[^apl]%';

or

SELECT * FROM Customers
WHERE City NOT LIKE '[apl]%' and city like '_%';
```

Leggi Operatore LIKE online: https://riptutorial.com/it/sql/topic/860/operatore-like

Capitolo 37: Operatori AND & OR

Sintassi

1. SELECT * FROM table WHERE (condition1) AND (condition2);
2. SELECT * FROM table WHERE (condition1) OR (condition2);

Examples

E O Esempio

Avere un tavolo

Nome	Età	Città
peso	10	Parigi
Stuoia	20	Berlino
Maria	24	Praga

```
select Name from table where Age>10 AND City='Prague'
```

dà

Nome
Maria

```
select Name from table where Age=10 OR City='Prague'
```

dà

Nome
peso
Maria

Leggi Operatori AND & OR online: https://riptutorial.com/it/sql/topic/1386/operatori-and--amp--or

Capitolo 38: ORDINATO DA

Examples

Usa ORDER BY con TOP per restituire le prime x righe in base al valore di una colonna

In questo esempio, possiamo utilizzare GROUP BY non solo determinato il *tipo* di righe restituite, ma anche quali righe *vengono* restituite, poiché stiamo usando TOP per limitare il set di risultati.

Diciamo che vogliamo restituire i primi 5 utenti con la reputazione più alta da un sito di domande e risposte popolare senza nome.

Senza ORDINA DI

Questa query restituisce le 5 righe principali ordinate per impostazione predefinita, che in questo caso è "Id", la prima colonna della tabella (anche se non è una colonna mostrata nei risultati).

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
```

ritorna...

Nome da visualizzare	Reputazione
Comunità	1
Geoff Dalgas	12567
Jarrod Dixon	11739
Jeff Atwood	37628
Joel Spolsky	25784

Con ORDINA DI

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
ORDER BY Reputation desc
```

ritorna...

Nome da visualizzare	Reputazione
JonSkeet	**865.023**

Nome da visualizzare	Reputazione
Darin Dimitrov	**661.741**
BalusC	**650.237**
Hans Passant	**625.870**
Marc Gravell	**601.636**

Osservazioni

Alcune versioni di SQL (come MySQL) utilizzano una clausola `LIMIT` alla fine di una `SELECT` , invece di `TOP` all'inizio, ad esempio:

```
SELECT DisplayName, Reputation
FROM Users
ORDER BY Reputation DESC
LIMIT 5
```

Ordinamento per più colonne

```
SELECT DisplayName, JoinDate, Reputation
FROM Users
ORDER BY JoinDate, Reputation
```

Nome da visualizzare	Data di iscrizione	Reputazione
Comunità	**2008-09-15**	**1**
Jeff Atwood	**2008-09-16**	**25784**
Joel Spolsky	2008-09-16	**37628**
Jarrod Dixon	**2008-10-03**	**11739**
Geoff Dalgas	2008-10-03	**12567**

Ordinamento per numero di colonna (anziché nome)

Puoi usare il numero di una colonna (dove la colonna più a sinistra è "1") per indicare su quale colonna basare l'ordinamento, invece di descrivere la colonna con il suo nome.

Pro: Se pensi che sia probabile che tu possa cambiare i nomi delle colonne in un secondo momento, così facendo non infrangerà questo codice.

Con: Questo in genere ridurrà la leggibilità della query (è immediatamente chiaro che cosa significa 'ORDER BY Reputation', mentre 'ORDER BY 14' richiede qualche conteggio, probabilmente con un dito sullo schermo.)

Questa query ordina il risultato in base alla posizione relativa della colonna `3` dell'istruzione select anziché al nome della colonna `Reputation` .

```
SELECT DisplayName, JoinDate, Reputation
FROM Users
ORDER BY 3
```

Nome da visualizzare	Data di iscrizione	Reputazione
Comunità	2008-09-15	**1**
Jarrod Dixon	2008-10-03	**11739**
Geoff Dalgas	2008-10-03	**12567**
Joel Spolsky	2008-09-16	**25784**
Jeff Atwood	2008-09-16	**37628**

Ordine di Alias

A causa dell'ordine di elaborazione delle query logiche, l'alias può essere utilizzato in ordine da.

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY jd, rep
```

E può usare l'ordine relativo delle colonne nell'istruzione select. Considera lo stesso esempio come sopra e invece di usare l'alias usa l'ordine relativo come per il nome visualizzato è 1, per Jd è 2 e così via

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY 2, 3
```

Ordinamento personalizzato

Per ordinare questa tabella `Employee` per dipartimento, si utilizzerà `ORDER BY Department` . Tuttavia, se si desidera un diverso ordinamento che non è alfabetico, è necessario mappare i valori del `Department` in diversi valori che ordinano correttamente; questo può essere fatto con un'espressione CASE:

Nome	Dipartimento
Hasan	IT
Yusuf	HR
Hillary	HR

Nome	Dipartimento
Joe	IT
allegro	HR
comprensione	Contabile

```
SELECT *
FROM Employee
ORDER BY CASE Department
         WHEN 'HR'         THEN 1
         WHEN 'Accountant' THEN 2
         ELSE                   3
         END;
```

Nome	Dipartimento
Yusuf	**HR**
Hillary	**HR**
allegro	**HR**
comprensione	**Contabile**
Hasan	**IT**
Joe	**IT**

Leggi ORDINATO DA online: https://riptutorial.com/it/sql/topic/620/ordinato-da

Capitolo 39: Ordine di esecuzione

Examples

Ordine logico di elaborazione delle query in SQL

```
/*(8)*/  SELECT /*9*/ DISTINCT /*11*/ TOP
/*(1)*/  FROM
/*(3)*/       JOIN
/*(2)*/      ON
/*(4)*/  WHERE
/*(5)*/  GROUP BY
/*(6)*/  WITH {CUBE | ROLLUP}
/*(7)*/  HAVING
/*(10)*/ ORDER BY
/*(11)*/ LIMIT
```

L'ordine in cui una query viene elaborata e la descrizione di ogni sezione.

VT sta per 'Virtual Table' e mostra come vengono prodotti i vari dati man mano che la query viene elaborata

1. FROM: Un prodotto cartesiano (cross join) viene eseguito tra le prime due tabelle nella clausola FROM e, di conseguenza, viene generata la tabella virtuale VT1.

2. ON: il filtro ON viene applicato a VT1. Solo le righe per cui è VERO sono inserite in VT2.

3. OUTER (join): Se viene specificato un JOIN OUTER (al contrario di un CROSS JOIN o UNO JOIN INTERNO), le righe della tabella o delle tabelle protette per le quali non è stata trovata una corrispondenza vengono aggiunte alle righe da VT2 come righe esterne, generando VT3. Se nella clausola FROM vengono visualizzate più di due tabelle, i passaggi da 1 a 3 vengono applicati ripetutamente tra il risultato dell'ultimo join e la successiva tabella nella clausola FROM fino a quando tutte le tabelle vengono elaborate.

4. DOVE: Il filtro WHERE è applicato a VT3. Solo le righe per cui è VERO sono inserite in VT4.

5. GROUP BY: le righe da VT4 sono organizzate in gruppi in base all'elenco di colonne specificato nella clausola GROUP BY. VT5 è generato.

6. CUBO | ROLLUP: i supergroup (gruppi di gruppi) vengono aggiunti alle righe da VT5, generando VT6.

7. HAVING: il filtro HAVING è applicato a VT6. Solo i gruppi per cui è VERO sono inseriti in VT7.

8. SELEZIONA: l'elenco SELECT viene elaborato, generando VT8.

9. DISTINCT: le file duplicate vengono rimosse da VT8. VT9 è generato.

10. ORDER BY: le righe da VT9 vengono ordinate in base all'elenco di colonne specificato nella clausola ORDER BY. Viene generato un cursore (VC10).

11. TOP: il numero specificato o la percentuale di righe viene selezionata dall'inizio di VC10. La tabella VT11 viene generata e restituita al chiamante. LIMIT ha la stessa funzionalità di TOP in alcuni dialetti SQL come Postgres e Netezza.

Leggi Ordine di esecuzione online: https://riptutorial.com/it/sql/topic/3671/ordine-di-esecuzione

Capitolo 40: Procedura di archiviazione

Osservazioni

Le stored procedure sono istruzioni SQL memorizzate nel database che possono essere eseguite o richiamate nelle query. L'utilizzo di una stored procedure consente l'incapsulamento di una logica complicata o utilizzata frequentemente e migliora le prestazioni della query utilizzando i piani di query memorizzati nella cache. Possono restituire qualsiasi valore restituito da una query standard.

Altri vantaggi rispetto alle espressioni SQL dinamiche sono elencati su Wikipeida .

Examples

Crea e chiama una stored procedure

Le stored procedure possono essere create tramite una GUI di gestione del database (esempio di SQL Server) o tramite un'istruzione SQL come segue:

```
-- Define a name and parameters
CREATE PROCEDURE Northwind.getEmployee
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS

-- Define the query to be run
SELECT FirstName, LastName, Department
FROM Northwind.vEmployeeDepartment
WHERE FirstName = @FirstName AND LastName = @LastName
AND EndDate IS NULL;
```

Chiamando la procedura:

```
EXECUTE Northwind.getEmployee N'Ackerman', N'Pilar';

-- Or
EXEC Northwind.getEmployee @LastName = N'Ackerman', @FirstName = N'Pilar';
GO

-- Or
EXECUTE Northwind.getEmployee @FirstName = N'Pilar', @LastName = N'Ackerman';
GO
```

Leggi Procedura di archiviazione online: https://riptutorial.com/it/sql/topic/1701/procedura-di-archiviazione

Capitolo 41: PROVA A PRENDERE

Osservazioni

TRY / CATCH è un costrutto linguistico specifico per T-SQL di MS SQL Server.

Permette la gestione degli errori all'interno di T-SQL, simile a quella vista nel codice .NET.

Examples

Transazione in un tentativo / CATCH

Ciò ripristinerà entrambi gli inserimenti a causa di un datetime non valido:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, 'not a date', 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

Questo imposterà entrambi gli inserti:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

Leggi PROVA A PRENDERE online: https://riptutorial.com/it/sql/topic/4420/prova-a-prendere

Capitolo 42: Pulisci codice in SQL

introduzione

Come scrivere query SQL buone e leggibili ed esempi di buone pratiche.

Examples

Formattazione e ortografia di parole chiave e nomi

Nomi tabella / colonna

Due modi comuni di formattazione dei nomi di tabelle / colonne sono `CamelCase` e `snake_case` :

```
SELECT FirstName, LastName
FROM Employees
WHERE Salary > 500;
```

```
SELECT first_name, last_name
FROM employees
WHERE salary > 500;
```

I nomi dovrebbero descrivere ciò che è memorizzato nel loro oggetto. Ciò implica che i nomi delle colonne di solito dovrebbero essere singolari. Se i nomi dei tavoli debbano usare singolare o plurale è una domanda molto discussa , ma in pratica è più comune usare nomi di tabelle plurali.

L'aggiunta di prefissi o suffissi come `tbl` o `col` riduce la leggibilità, quindi evitali. Tuttavia, a volte vengono utilizzati per evitare conflitti con le parole chiave SQL e spesso utilizzati con trigger e indici (i cui nomi di solito non sono menzionati nelle query).

parole

Le parole chiave SQL non sono case sensitive. Tuttavia, è prassi comune scriverli in maiuscolo.

SELEZIONA

`SELECT *` restituisce tutte le colonne nello stesso ordine in cui sono definite nella tabella.

Quando si utilizza `SELECT *` , i dati restituiti da una query possono cambiare ogni volta che cambia la definizione della tabella. Ciò aumenta il rischio che diverse versioni della tua applicazione o del tuo database siano incompatibili tra loro.

Inoltre, la lettura di più colonne del necessario può aumentare la quantità di I / O del disco e della rete.

Pertanto, devi sempre specificare esplicitamente le colonne che vuoi effettivamente recuperare:

```
--SELECT *                                  don't
  SELECT ID, FName, LName, PhoneNumber  -- do
  FROM Emplopees;
```

(Quando si eseguono query interattive, queste considerazioni non si applicano).

Tuttavia, `SELECT *` non fa male nella sottoquery di un operatore EXISTS, perché EXISTS ignora comunque i dati effettivi (controlla solo se è stata trovata almeno una riga). Per lo stesso motivo, non è significativo elencare colonne specifiche per EXISTS, quindi `SELECT *` effettivamente più senso:

```
-- list departments where nobody was hired recently
SELECT ID,
       Name
FROM Departments
WHERE NOT EXISTS (SELECT *
                  FROM Employees
                  WHERE DepartmentID = Departments.ID
                    AND HireDate >= '2015-01-01');
```

Rientro

Non esiste uno standard ampiamente accettato. Ciò su cui tutti sono d'accordo è che spremere tutto in una singola riga è sbagliato:

```
SELECT d.Name, COUNT(*) AS Employees FROM Departments AS d JOIN Employees AS e ON d.ID =
e.DepartmentID WHERE d.Name != 'HR' HAVING COUNT(*) > 10 ORDER BY COUNT(*) DESC;
```

Come minimo, inserire ogni clausola in una nuova riga e dividere le righe se diventeranno troppo lunghe altrimenti:

```
SELECT d.Name,
       COUNT(*) AS Employees
FROM Departments AS d
JOIN Employees AS e ON d.ID = e.DepartmentID
WHERE d.Name != 'HR'
HAVING COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

A volte, tutto dopo che la parola chiave SQL che introduce una clausola viene rientrata nella stessa colonna:

```
SELECT   d.Name,
         COUNT(*) AS Employees
FROM     Departments AS d
JOIN     Employees AS e ON d.ID = e.DepartmentID
WHERE    d.Name != 'HR'
HAVING   COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

(Questo può essere fatto anche allineando correttamente le parole chiave SQL).

Un altro stile comune consiste nel mettere le parole chiave importanti sulle proprie linee:

```
SELECT
    d.Name,
    COUNT(*) AS Employees
FROM
    Departments AS d
JOIN
    Employees AS e
    ON d.ID = e.DepartmentID
WHERE
    d.Name != 'HR'
HAVING
    COUNT(*) > 10
ORDER BY
    COUNT(*) DESC;
```

L'allineamento verticale di più espressioni simili migliora la leggibilità:

```
SELECT Model,
       EmployeeID
FROM Cars
WHERE CustomerID = 42
  AND Status     = 'READY';
```

L'uso di più linee rende più difficile incorporare comandi SQL in altri linguaggi di programmazione. Tuttavia, molte lingue hanno un meccanismo per stringhe multi-linea, ad es. `@"..."` in C #, `"""..."""` in Python o `R"(...)"` in C ++.

Si unisce

Dovrebbero essere sempre utilizzati join espliciti; i join impliciti hanno diversi problemi:

- La condizione di join è da qualche parte nella clausola WHERE, mescolata con qualsiasi altra condizione di filtro. Ciò rende più difficile vedere quali tabelle sono unite e come.
- A causa di quanto sopra, c'è un rischio più elevato di errori, ed è più probabile che vengano trovati più tardi.
- In SQL standard, i join espliciti sono l'unico modo per utilizzare i join esterni :

  ```
  SELECT d.Name,
         e.Fname || e.LName AS EmpName
  FROM      Departments AS d
  LEFT JOIN Employees   AS e ON d.ID = e.DepartmentID;
  ```

- I join espliciti consentono di utilizzare la clausola USING:

  ```
  SELECT RecipeID,
  ```

```
       Recipes.Name,
       COUNT(*) AS NumberOfIngredients
FROM      Recipes
LEFT JOIN Ingredients USING (RecipeID);
```

(Ciò richiede che entrambe le tabelle utilizzino lo stesso nome di colonna.
USING rimuove automaticamente la colonna duplicata dal risultato, ad esempio, il join in questa query restituisce una singola colonna `RecipeID` .)

Leggi Pulisci codice in SQL online: https://riptutorial.com/it/sql/topic/9843/pulisci-codice-in-sql

Capitolo 43: RAGGRUPPA PER

introduzione

I risultati di una query SELECT possono essere raggruppati da una o più colonne utilizzando l'istruzione `GROUP BY` : tutti i risultati con lo stesso valore nelle colonne raggruppate vengono aggregati insieme. Questo genera una tabella di risultati parziali, invece di un risultato. GROUP BY può essere utilizzato insieme alle funzioni di aggregazione utilizzando l'istruzione `HAVING` per definire come aggregare le colonne non raggruppate.

Sintassi

- RAGGRUPPA PER {
  Colonna-espressione
  | ROLLUP (<group_by_expression> [, ... n])
  | CUBE (<group_by_expression> [, ... n])
  | GRUPPI DI GRUPPO ([, ... n])
  | () - calcola il totale generale
  } [, ... n]

- <group_by_expression> :: =
  Colonna-espressione
  | (espressione della colonna [, ... n])

- <grouping_set> :: =
  () - calcola il totale generale
  | <Grouping_set_item>
  | (<grouping_set_item> [, ... n])

- <grouping_set_item> :: =
  <Group_by_expression>
  | ROLLUP (<group_by_expression> [, ... n])
  | CUBE (<group_by_expression> [, ... n])

Examples

USA GROUP BY per COUNT il numero di righe per ogni voce univoca in una determinata colonna

Supponiamo di voler generare conteggi o subtotali per un dato valore in una colonna.

Dato questo tavolo, "Westerosians":

Nome	GreatHouseAllegience
Arya	rigido
cercei	Lannister
Myrcella	Lannister
yara	Greyjoy
Catelyn	rigido
Sansa	rigido

Senza GROUP BY, COUNT restituirà semplicemente un numero totale di righe:

```
SELECT Count(*) Number_of_Westerosians
FROM Westerosians
```

ritorna...

Number_of_Westerosians
6

Ma aggiungendo GROUP BY, possiamo COUNT gli utenti per ogni valore in una determinata colonna, per restituire il numero di persone in una determinata Grande Casa, per esempio:

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
```

ritorna...

Casa	Number_of_Westerosians
rigido	3
Greyjoy	1
Lannister	2

È comune combinare GROUP BY con ORDER BY per ordinare i risultati per categoria più grande o più piccola:

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
ORDER BY Number_of_Westerosians Desc
```

ritorna...

Casa	Number_of_Westerosians
rigido	3
Lannister	2
Greyjoy	1

Filtra i risultati GROUP BY utilizzando una clausola HAVING

Una clausola HAVING filtra i risultati di un'espressione GROUP BY. Nota: i seguenti esempi utilizzano il database di esempio Library .

Esempi:

Restituisci tutti gli autori che hanno scritto più di un libro (esempio dal vivo).

```
SELECT
  a.Id,
  a.Name,
  COUNT(*) BooksWritten
FROM BooksAuthors ba
  INNER JOIN Authors a ON a.id = ba.authorid
GROUP BY
  a.Id,
  a.Name
HAVING COUNT(*) > 1    -- equals to HAVING BooksWritten > 1
;
```

Restituisci tutti i libri che hanno più di tre autori (esempio dal vivo).

```
SELECT
  b.Id,
  b.Title,
  COUNT(*) NumberOfAuthors
FROM BooksAuthors ba
  INNER JOIN Books b ON b.id = ba.bookid
GROUP BY
  b.Id,
  b.Title
HAVING COUNT(*) > 3    -- equals to HAVING NumberOfAuthors > 3
;
```

Esempio di base GROUP BY

Potrebbe essere più semplice se si pensa a GROUP BY come "per ciascuno" a scopo di spiegazione. La query qui sotto:

```
SELECT EmpID, SUM (MonthlySalary)
FROM Employee
```

```
GROUP BY EmpID
```

sta dicendo:

"Dammi la somma di MonthlySalary's **per ogni** EmpID"

Quindi se il tuo tavolo fosse così:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Risultato:

```
+-+---+
|1|200|
+-+---+
|2|300|
+-+---+
```

Sum non sembra fare nulla perché la somma di un numero è quel numero. D'altra parte se sembra così:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|1    |300          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Risultato:

```
+-+---+
|1|500|
+-+---+
|2|300|
+-+---+
```

Allora sarebbe perché ci sono due EmpID 1 da sommare insieme.

Aggregazione ROLAP (data mining)

Descrizione

Lo standard SQL fornisce due operatori aggregati aggiuntivi. Questi usano il valore polimorfico

"ALL" per indicare l'insieme di tutti i valori che un attributo può assumere. I due operatori sono:

- `with data cube` che fornisce tutte le combinazioni possibili rispetto agli attributi argomento della clausola.
- `with roll up` che fornisce gli aggregati ottenuti considerando gli attributi nell'ordine da sinistra a destra rispetto a come sono elencati nell'argomento della clausola.

Versioni standard SQL che supportano queste funzionalità: 1999,2003,2006,2008,2011.

Esempi

Considera questa tabella:

Cibo	Marca	Importo totale
Pasta	Brand1	100
Pasta	brand2	250
Pizza	brand2	300

Con il cubo

```
select Food,Brand,Total_amount
from Table
group by Food,Brand,Total_amount with cube
```

Cibo	Marca	Importo totale
Pasta	Brand1	100
Pasta	brand2	250
Pasta	TUTTI	350
Pizza	brand2	300
Pizza	TUTTI	300
TUTTI	Brand1	100
TUTTI	brand2	550
TUTTI	TUTTI	650

Con roll up

```
select Food,Brand,Total_amount
from Table
group by Food,Brand,Total_amount with roll up
```

Cibo	Marca	Importo totale
Pasta	Brand1	100
Pasta	brand2	250
Pizza	brand2	300
Pasta	TUTTI	350
Pizza	TUTTI	300
TUTTI	TUTTI	650

Leggi RAGGRUPPA PER online: https://riptutorial.com/it/sql/topic/627/raggruppa-per

Capitolo 44: Ricerca di duplicati su un sottoinsieme di colonne con dettagli

Osservazioni

- Per selezionare le righe senza duplicati, modificare la clausola WHERE su "RowCnt = 1"
- Per selezionare una riga da ogni set, utilizzare Rank () anziché Sum () e modificare la clausola WHERE esterna per selezionare le righe con Rank () = 1

Examples

Studenti con lo stesso nome e data di nascita

```
WITH CTE (StudentId, Fname, LName, DOB, RowCnt)
as (
SELECT StudentId, FirstName, LastName, DateOfBirth as DOB, SUM(1) OVER (Partition By
FirstName, LastName, DateOfBirth) as RowCnt
FROM tblStudent
)
SELECT * from CTE where RowCnt > 1
ORDER BY DOB, LName
```

Questo esempio utilizza un'espressione di tabella comune e una funzione di finestra per mostrare tutte le righe duplicate (su un sottoinsieme di colonne) affiancate.

Leggi Ricerca di duplicati su un sottoinsieme di colonne con dettagli online: https://riptutorial.com/it/sql/topic/1585/ricerca-di-duplicati-su-un-sottoinsieme-di-colonne-con-dettagli

Capitolo 45: SALVO

Osservazioni

`EXCEPT` restituisce tutti i valori distinti dal set di dati a sinistra dell'operatore EXCEPT che non vengono restituiti anche dall'insieme di dati corretto.

Examples

Seleziona set di dati tranne dove i valori si trovano in questo altro set di dati

```
--dataset schemas must be identical
SELECT 'Data1' as 'Column' UNION ALL
SELECT 'Data2' as 'Column' UNION ALL
SELECT 'Data3' as 'Column' UNION ALL
SELECT 'Data4' as 'Column' UNION ALL
SELECT 'Data5' as 'Column'
EXCEPT
SELECT 'Data3' as 'Column'
--Returns Data1, Data2, Data4, and Data5
```

Leggi SALVO online: https://riptutorial.com/it/sql/topic/4082/salvo

Capitolo 46: Schema di informazioni

Examples

Ricerca dello schema di informazioni di base

Una delle query più utili per gli utenti finali di RDBMS di grandi dimensioni è la ricerca di uno schema di informazioni.

Tale query consente agli utenti di trovare rapidamente tabelle di database contenenti colonne di interesse, ad esempio quando si tenta di correlare i dati da 2 tabelle indirettamente tramite una terza tabella, senza la conoscenza di quali tabelle potrebbero contenere chiavi o altre colonne utili in comune con le tabelle di destinazione .

Utilizzando T-SQL per questo esempio, lo schema di informazioni di un database può essere cercato come segue:

```
SELECT *
FROM INFORMATION_SCHEMA.COLUMNS
WHERE COLUMN_NAME LIKE '%Institution%'
```

Il risultato contiene un elenco di colonne corrispondenti, i nomi delle loro tabelle e altre informazioni utili.

Leggi Schema di informazioni online: https://riptutorial.com/it/sql/topic/3151/schema-di-informazioni

Capitolo 47: SELEZIONARE

introduzione

L'istruzione SELECT è al centro della maggior parte delle query SQL. Definisce quale set di risultati deve essere restituito dalla query e viene quasi sempre utilizzato insieme alla clausola FROM, che definisce quale parte del database deve essere interrogata.

Sintassi

- SELEZIONA [DISTINCT] [column1] [, [column2] ...]
  DA [tavolo]
  [WHERE condition]
  [GROUP BY [column1] [, [column2] ...]

  [HAVING [column1] [, [column2] ...]

  [ORDINA PER ASC | DESC]

Osservazioni

SELECT determina i dati delle colonne da restituire e in quale ordine FROM una determinata tabella (dato che corrispondono agli altri requisiti della tua query specificatamente - dove e avendo filtri e join).

```
SELECT Name, SerialNumber
FROM ArmyInfo
```

per esempio restituirà risultati dalle colonne `Name` e `Serial Number` , ma non dalla colonna chiamata `Rank`

```
SELECT *
FROM ArmyInfo
```

indica che **tutte le** colonne saranno restituite. Tuttavia, tieni presente che non è consigliabile selezionare `SELECT *` poiché stai letteralmente restituendo tutte le colonne di un tavolo.

Examples

Utilizzando il carattere jolly per selezionare tutte le colonne in una query.

Considera un database con le seguenti due tabelle.

Tabella dei dipendenti:

Id	FName	LName	deptid
1	Giacomo	fabbro	3
2	John	Johnson	4

Tabella dei dipartimenti:

Id	Nome
1	I saldi
2	Marketing
3	Finanza
4	IT

Semplice affermazione di selezione

`*` è il **carattere jolly** utilizzato per selezionare tutte le colonne disponibili in una tabella.

Quando viene utilizzato come un sostituto per i nomi di colonna espliciti, restituisce tutte le colonne in tutte le tabelle che una query sta selezionando `FROM` . Questo effetto si applica a **tutte le tabelle** la query accede tramite le sue clausole `JOIN` .

Considera la seguente query:

```
SELECT * FROM Employees
```

Restituirà tutti i campi di tutte le righe della tabella `Employees` :

Id	FName	LName	deptid
1	Giacomo	fabbro	3
2	John	Johnson	4

Notazione a punti

Per selezionare tutti i valori da una tabella specifica, il carattere jolly può essere applicato alla tabella con *notazione a punti* .

Considera la seguente query:

```
SELECT
    Employees.*,
    Departments.Name
```

```
FROM
    Employees
JOIN
    Departments
    ON Departments.Id = Employees.DeptId
```

Ciò restituirà un set di dati con tutti i campi nella tabella `Employee` , seguito solo dal campo `Name` nella tabella `Departments` :

Id	FName	LName	deptid	Nome
1	Giacomo	fabbro	3	Finanza
2	John	Johnson	4	IT

Avvertenze contro l'uso

Si consiglia in genere che l'utilizzo di `*` venga evitato nel codice di produzione laddove possibile, in quanto può causare una serie di potenziali problemi, tra cui:

1. Eccesso di I / O, carico di rete, utilizzo della memoria e così via, a causa del fatto che il motore di database non legge i dati necessari e li trasmette al codice di front-end. Questo è particolarmente un problema in cui potrebbero esserci campi grandi come quelli usati per memorizzare note lunghe o file allegati.
2. Ulteriore sovraccarico di I / O se il database deve eseguire lo spool dei risultati interni sul disco come parte dell'elaborazione per una query più complessa di `SELECT <columns> FROM <table>` .
3. Elaborazione extra (e / o anche più IO) se alcune delle colonne non necessarie sono:
    - colonne calcolate nei database che li supportano
    - in caso di selezione da una vista, colonne da una tabella / vista che altrimenti potrebbero essere ottimizzate da Query Optimiser
4. Il potenziale di errori imprevisti se le colonne vengono aggiunte successivamente alle tabelle e alle viste che risultano in nomi di colonne ambigue. Ad esempio, `SELECT * FROM orders JOIN people ON people.id = orders.personid ORDER BY displayname` - se una colonna colonna denominata `displayname` viene aggiunta alla tabella degli ordini per consentire agli utenti di dare ai loro ordini nomi significativi per riferimenti futuri, il nome della colonna verrà visualizzato due volte nell'output quindi la clausola `ORDER BY` sarà ambigua e potrebbe causare errori ("nome colonna ambiguo" nelle recenti versioni di MS SQL Server) e, se non in questo esempio, il codice dell'applicazione potrebbe iniziare a visualizzare il nome dell'ordine in cui il nome della persona è inteso perché la nuova colonna è il primo di quel nome restituito e così via.

Quando puoi usare `*` , tenendo presente l'avvertimento sopra?

Mentre è meglio evitare nel codice di produzione, l'uso di `*` va bene come una scorciatoia quando si eseguono query manuali sul database per indagini o lavori di prototipazione.

A volte le decisioni di progettazione nella tua applicazione rendono inevitabile (in tali circostanze,

preferisci le `tablealias.*` solo su `*` dove possibile).

Quando si utilizzano `EXISTS` , come `SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID)` , non si restituiscono dati da B. Pertanto un join non è necessario e il motore non riconosce valori di B da restituire, quindi nessun risultato di prestazioni per l'utilizzo di `*` . Allo stesso modo, `COUNT(*)` va bene, in quanto non restituisce alcuna colonna, quindi è sufficiente leggere ed elaborare quelli utilizzati per scopi di filtraggio.

Selezione con condizione

La sintassi di base della clausola SELECT with WHERE è:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

La *[condizione]* può essere qualsiasi espressione SQL, specificata mediante confronto o operatori logici come>, <, =, <>,> =, <=, LIKE, NOT, IN, BETWEEN ecc.

L'istruzione seguente restituisce tutte le colonne dalla tabella 'Auto' in cui la colonna dello stato è 'PRONTA':

```
SELECT * FROM Cars WHERE status = 'READY'
```

Guarda WHERE e HAVING per altri esempi.

Seleziona singole colonne

```
SELECT
    PhoneNumber,
    Email,
    PreferredContact
FROM Customers
```

Questa dichiarazione restituirà le colonne `PhoneNumber` , `Email` e `PreferredContact` da tutte le righe della tabella `Customers` . Anche le colonne verranno restituite nella sequenza in cui appaiono nella clausola `SELECT` .

Il risultato sarà:

Numero di telefono	E-mail	PreferredContact
3347927472	william.jones@example.com	TELEFONO
2137921892	dmiller@example.net	E-MAIL
NULLO	richard0123@example.com	E-MAIL

Se più tabelle vengono unite, è possibile selezionare le colonne da tabelle specifiche specificando

il nome della tabella prima del nome della colonna: `[table_name].[column_name]`

```
SELECT
    Customers.PhoneNumber,
    Customers.Email,
    Customers.PreferredContact,
    Orders.Id AS OrderId
FROM
    Customers
LEFT JOIN
    Orders ON Orders.CustomerId = Customers.Id
```

* `AS OrderId` significa che il campo `Id` della tabella `Orders` verrà restituito come una colonna denominata `OrderId` . Vedi selezionare con l'alias della colonna per ulteriori informazioni.

Per evitare l'utilizzo di nomi di tabelle lunghe, è possibile utilizzare alias di tabella. Ciò riduce il dolore di scrivere nomi di tabelle lunghe per ogni campo selezionato nei join. Se stai eseguendo un self join (un join tra due istanze della *stessa* tabella), devi utilizzare gli alias di tabella per distinguere le tue tabelle. Possiamo scrivere un alias di tabella come `Customers c` o `Customers AS c` . Qui `c` funziona come alias per i `Customers` e possiamo selezionare diciamo `Email` come questa: `c.Email` .

```
SELECT
    c.PhoneNumber,
    c.Email,
    c.PreferredContact,
    o.Id AS OrderId
FROM
    Customers c
LEFT JOIN
    Orders o ON o.CustomerId = c.Id
```

SELEZIONARE Usando Colonna Alias

Gli alias di colonne vengono utilizzati principalmente per abbreviare il codice e rendere più leggibili i nomi di colonna.

Il codice si accorcia quando i nomi delle tabelle lunghi e l'identificazione non necessaria delle colonne *(ad esempio, possono esserci 2 ID nella tabella, ma solo uno è utilizzato nell'istruzione)* possono essere evitati. Insieme agli alias di tabella, questo consente di utilizzare nomi descrittivi più lunghi nella struttura del database mantenendo al contempo le query su quella struttura concisa.

Inoltre, a volte sono *necessari* , ad esempio nelle viste, per denominare i risultati calcolati.

Tutte le versioni di SQL

Gli alias possono essere creati in tutte le versioni di SQL usando virgolette doppie (`"`).

```
SELECT
    FName AS "First Name",
```

```
    MName AS "Middle Name",
    LName AS "Last Name"
FROM Employees
```

Diverse versioni di SQL

È possibile utilizzare virgolette singole (`'`), doppie virgolette (`"`) e parentesi quadre (`[]`) per creare un alias in Microsoft SQL Server.

```
SELECT
    FName AS "First Name",
    MName AS 'Middle Name',
    LName AS [Last Name]
FROM Employees
```

Entrambi si tradurranno in:

Nome di battesimo	Secondo nome	Cognome
Giacomo	John	fabbro
John	Giacomo	Johnson
Michael	Marcus	Williams

Questa istruzione restituirà le colonne `FName` e `LName` con un nome specifico (un alias). Ciò viene ottenuto utilizzando l'operatore `AS` seguito dall'alias o semplicemente scrivendo alias direttamente dopo il nome della colonna. Ciò significa che la seguente query ha lo stesso risultato di cui sopra.

```
SELECT
    FName "First Name",
    MName "Middle Name",
    LName "Last Name"
FROM Employees
```

Nome di battesimo	Secondo nome	Cognome
Giacomo	John	fabbro
John	Giacomo	Johnson
Michael	Marcus	Williams

Tuttavia, la versione esplicita (cioè, utilizzando l'operatore `AS`) è più leggibile.

Se l'alias ha una singola parola che non è una parola riservata, possiamo scriverla senza virgolette singole, virgolette doppie o parentesi:

```
SELECT
```

```
    FName AS FirstName,
    LName AS LastName
FROM Employees
```

Nome di battesimo	Cognome
Giacomo	fabbro
John	Johnson
Michael	Williams

Un'altra variante disponibile in MS SQL Server è `<alias> = <column-or-calculation>` , ad esempio:

```
SELECT FullName = FirstName + ' ' + LastName,
       Addr1    = FullStreetAddress,
       Addr2    = TownName
FROM CustomerDetails
```

che è equivalente a:

```
SELECT FirstName + ' ' + LastName As FullName
       FullStreetAddress          As Addr1,
       TownName                   As Addr2
FROM CustomerDetails
```

Entrambi si tradurranno in:

Nome e cognome	Addr1	Addr2
James Smith	123 AnyStreet	Townville
John Johnson	668 MyRoad	Anytown
Michael Williams	999 High End Dr	Williamsburgh

Alcuni trovano usando `=` invece di `As` più facile da leggere, sebbene molti raccomandino contro questo formato, principalmente perché non è standard quindi non ampiamente supportato da tutti i database. Potrebbe causare confusione con altri usi del carattere `=` .

Tutte le versioni di SQL

Inoltre, se è *necessario* utilizzare parole riservate, è possibile utilizzare parentesi o virgolette per evitare:

```
SELECT
    FName as "SELECT",
    MName as "FROM",
    LName as "WHERE"
```

```
FROM Employees
```

Diverse versioni di SQL

Allo stesso modo, puoi sfuggire le parole chiave in MSSQL con tutti i diversi approcci:

```
SELECT
    FName AS "SELECT",
    MName AS 'FROM',
    LName AS [WHERE]
FROM Employees
```

SELEZIONARE	A PARTIRE DAL	DOVE
Giacomo	John	fabbro
John	Giacomo	Johnson
Michael	Marcus	Williams

Inoltre, un alias di colonna può essere utilizzato una qualsiasi delle clausole finali della stessa query, ad esempio un `ORDER BY` :

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM
    Employees
ORDER BY
    LastName DESC
```

Tuttavia, *non* si può usare

```
SELECT
    FName AS SELECT,
    LName AS FROM
FROM
    Employees
ORDER BY
    LastName DESC
```

Per creare un alias da queste parole riservate (`SELECT` e `FROM`).

Ciò causerà numerosi errori durante l'esecuzione.

Selezione con risultati ordinati

```
SELECT * FROM Employees ORDER BY LName
```

Questa dichiarazione restituirà tutte le colonne dalla tabella `Employees` .

Id	FName	LName	Numero di telefono
2	John	Johnson	2468101214
1	Giacomo	fabbro	1234567890
3	Michael	Williams	1357911131

```
SELECT * FROM Employees ORDER BY LName DESC
```

O

```
 SELECT * FROM Employees ORDER BY LName ASC
```

Questa affermazione cambia la direzione di smistamento.

Si può anche specificare più colonne di ordinamento. Per esempio:

```
SELECT * FROM Employees ORDER BY LName ASC, FName ASC
```

Questo esempio `LName` i risultati prima per `LName` e poi, per i record che hanno lo stesso `LName` , per `FName` . Questo ti darà un risultato simile a quello che troveresti in una rubrica telefonica.

Per salvare la ridigitazione del nome della colonna nella clausola `ORDER BY` , è possibile utilizzare invece il numero della colonna. Si noti che i numeri delle colonne iniziano da 1.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY 3
```

È inoltre possibile incorporare un'istruzione `CASE` nella clausola `ORDER BY` .

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY CASE WHEN LName='Jones` THEN 0
ELSE 1 END ASC
```

Questo `LName` i risultati per avere tutti i record con il `LName` di "Jones" in alto.

Seleziona le colonne che prendono il nome da parole chiave riservate

Quando un nome di colonna corrisponde a una parola chiave riservata, SQL standard richiede che lo si racchiuda tra virgolette doppie:

```
SELECT
    "ORDER",
    ID
FROM ORDERS
```

Si noti che rende il nome della colonna case-sensitive.

Alcuni DBMS hanno modi proprietari di elencare i nomi. Ad esempio, SQL Server utilizza parentesi quadre per questo scopo:

```
SELECT
    [Order],
    ID
FROM ORDERS
```

mentre MySQL (e MariaDB) utilizzano di default i backtick:

```
SELECT
    `Order`,
    id
FROM orders
```

Selezione del numero specificato di record

Lo standard SQL 2008 definisce la clausola `FETCH FIRST` per limitare il numero di record restituiti.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
FETCH FIRST 10 ROWS ONLY
```

Questo standard è supportato solo nelle versioni recenti di alcuni RDMS. La sintassi non standard specifica del fornitore viene fornita in altri sistemi. Progress OpenEdge 11.x supporta anche la sintassi `FETCH FIRST <n> ROWS ONLY` .

Inoltre, `OFFSET <m> ROWS` prima di `FETCH FIRST <n> ROWS ONLY` consente di saltare le righe prima di recuperare le righe.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
OFFSET 5 ROWS
FETCH FIRST 10 ROWS ONLY
```

La seguente query è supportata in SQL Server e MS Access:

```
SELECT TOP 10 Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
```

Per fare lo stesso in MySQL o PostgreSQL, è necessario utilizzare la parola chiave `LIMIT` :

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
LIMIT 10
```

In Oracle lo stesso può essere fatto con `ROWNUM` :

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
```

```
WHERE ROWNUM <= 10
ORDER BY UnitPrice DESC
```

Risultati : **10** record.

```
Id    ProductName               UnitPrice             Package
38    Côte de Blaye             263.50                12 - 75 cl bottles
29    Thüringer Rostbratwurst   123.79                50 bags x 30 sausgs.
9    Mishi Kobe Niku            97.00                 18 - 500 g pkgs.
20    Sir Rodney's Marmalade    81.00                 30 gift boxes
18    Carnarvon Tigers          62.50                 16 kg pkg.
59    Raclette Courdavault      55.00                 5 kg pkg.
51    Manjimup Dried Apples     53.00                 50 - 300 g pkgs.
62    Tarte au sucre            49.30                 48 pies
43    Ipoh Coffee               46.00                 16 - 500 g tins
28    Rössle Sauerkraut         45.60                 25 - 825 g cans
```

Sfumature del venditore:

È importante notare che il `TOP` in Microsoft SQL opera dopo la clausola `WHERE` e restituirà il numero specificato di risultati se esistono in qualsiasi punto della tabella, mentre `ROWNUM` funziona come parte della clausola `WHERE` , quindi se altre condizioni non esistono nel numero di righe specificato all'inizio della tabella, otterrai zero risultati quando potrebbero essere trovati altri.

Selezione con alias di tabella

```
SELECT e.Fname, e.LName
FROM Employees e
```

Alla tabella Dipendenti viene assegnato l'alias 'e' direttamente dopo il nome della tabella. Ciò aiuta a rimuovere l'ambiguità in scenari in cui più tabelle hanno lo stesso nome di campo e devi essere specifico su quale tabella vuoi restituire i dati.

```
SELECT e.Fname, e.LName, m.Fname AS ManagerFirstName
FROM Employees e
    JOIN Managers m ON e.ManagerId = m.Id
```

Si noti che una volta definito un alias, non è più possibile utilizzare il nome della tabella canonica. vale a dire,

```
SELECT e.Fname, Employees.LName, m.Fname AS ManagerFirstName
FROM Employees e
JOIN Managers m ON e.ManagerId = m.Id
```

genererebbe un errore

Vale la pena notare che gli alias di tabelle - più formalmente "variabili di intervallo" - sono stati introdotti nel linguaggio SQL per risolvere il problema delle colonne duplicate causate da `INNER JOIN` . Lo standard SQL del 1992 ha corretto questa precedente imperfezione progettuale introducendo `NATURAL JOIN` (implementato in MySQL, PostgreSQL e Oracle ma non ancora in SQL

Server), il cui risultato non ha mai nomi di colonne duplicati. L'esempio sopra è interessante in quanto le tabelle sono unite su colonne con nomi diversi (`Id` e `ManagerId`) ma non dovrebbero essere unite sulle colonne con lo stesso nome (`LName` , `FName`), richiedendo la ridenominazione delle colonne da eseguire *prima* del join:

```
SELECT Fname, LName, ManagerFirstName
FROM Employees
     NATURAL JOIN
     ( SELECT Id AS ManagerId, Fname AS ManagerFirstName
       FROM Managers ) m;
```

Si noti che sebbene una variabile alias / intervallo debba essere dichiarata per la tabella dervied (altrimenti SQL genererà un errore), non ha mai senso usarla effettivamente nella query.

Seleziona le righe da più tabelle

```
SELECT *
FROM
    table1,
    table2
```

```
SELECT
    table1.column1,
    table1.column2,
    table2.column1
FROM
    table1,
    table2
```

Questo è chiamato prodotto incrociato in SQL è uguale al prodotto incrociato in serie

Queste istruzioni restituiscono le colonne selezionate da più tabelle in una query.

Non esiste una relazione specifica tra le colonne restituite da ciascuna tabella.

Selezione con le funzioni di aggregazione

Media

La funzione di aggregazione `AVG()` restituirà la media dei valori selezionati.

```
SELECT AVG(Salary) FROM Employees
```

Le funzioni aggregate possono anche essere combinate con la clausola where.

```
SELECT AVG(Salary) FROM Employees where DepartmentId = 1
```

Le funzioni aggregate possono anche essere combinate con la clausola group by.

Se il dipendente è suddiviso in più reparti e vogliamo trovare uno stipendio medio per ogni reparto,

possiamo utilizzare la seguente query.

```
SELECT AVG(Salary) FROM Employees GROUP BY DepartmentId
```

Minimo

La funzione di aggregazione `MIN()` restituirà il minimo dei valori selezionati.

```
SELECT MIN(Salary) FROM Employees
```

Massimo

La funzione di aggregazione `MAX()` restituirà il massimo dei valori selezionati.

```
SELECT MAX(Salary) FROM Employees
```

Contare

La funzione di aggregazione `COUNT()` restituirà il conteggio dei valori selezionati.

```
SELECT Count(*) FROM Employees
```

Può anche essere combinato con le condizioni in cui ottenere il conteggio delle righe che soddisfano condizioni specifiche.

```
SELECT Count(*) FROM Employees where ManagerId IS NOT NULL
```

È anche possibile specificare colonne specifiche per ottenere il numero di valori nella colonna. Si noti che i valori `NULL` non vengono conteggiati.

```
Select Count(ManagerId) from Employees
```

Il conteggio può anche essere combinato con la parola chiave distinta per un conteggio distinto.

```
Select Count(DISTINCT DepartmentId) from Employees
```

Somma

La funzione di aggregazione `SUM()` restituisce la somma dei valori selezionati per tutte le righe.

```
SELECT SUM(Salary) FROM Employees
```

Selezione con null

```
SELECT Name FROM Customers WHERE PhoneNumber IS NULL
```

La selezione con valori null richiede una sintassi diversa. Non usare `=` , utilizzare `IS NULL` o `IS NOT NULL` .

Selezione con CASE

Quando i risultati devono avere una logica applicata "al volo", è possibile utilizzare l'istruzione CASE per implementarla.

```
SELECT CASE WHEN Col1 < 50 THEN 'under' ELSE 'over' END threshold
FROM TableName
```

anche può essere incatenato

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         WHEN Col1 > 50 AND Col1 <100 THEN 'between'
         ELSE 'over'
    END threshold
FROM TableName
```

uno può anche avere `CASE` in un'altra istruzione `CASE`

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         ELSE
            CASE WHEN Col1 > 50 AND Col1 <100 THEN Col1
            ELSE 'over' END
    END threshold
FROM TableName
```

Selezionare senza bloccare la tabella

A volte quando le tabelle vengono utilizzate principalmente (o solo) per le letture, l'indicizzazione non aiuta più e ogni piccolo bit conta, si potrebbe usare selects senza LOCK per migliorare le prestazioni.

---

server SQL

```
SELECT * FROM TableName WITH (nolock)
```

MySQL

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
```

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Oracolo

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
```

DB2

```
SELECT * FROM TableName WITH UR;
```

dove `UR` sta per "lettura non salvata".

Se utilizzato su una tabella con modifiche ai record in corso potrebbe avere risultati imprevedibili.

Seleziona distinto (solo valori unici)

```
SELECT DISTINCT ContinentCode
FROM Countries;
```

Questa query restituirà tutti i valori `DISTINCT` (unici, diversi) dalla colonna `ContinentCode` dalla tabella `Countries`

ContinentCode
OC
Unione Europea
COME
N / A
AF

Demo di SQLFiddle

Selezionare con condizione di più valori dalla colonna

```
SELECT * FROM Cars  WHERE status IN ( 'Waiting', 'Working' )
```

Questo è semanticamente equivalente a

```
SELECT * FROM Cars  WHERE ( status = 'Waiting' OR status = 'Working' )
```

cioè il `value IN ( <value list> )` è una scorciatoia per disgiunzione (`OR` logico).

Ottieni risultati aggregati per i gruppi di righe

Conteggio delle righe in base a un valore di colonna specifico:

```
SELECT category, COUNT(*) AS item_count
FROM item
GROUP BY category;
```

Ottenere un reddito medio per dipartimento:

```
SELECT department, AVG(income)
FROM employees
GROUP BY department;
```

L'importante è selezionare solo le colonne specificate nella clausola `GROUP BY` o utilizzate con funzioni di aggregazione .

La clausola `WHERE` può essere utilizzata anche con `GROUP BY` , ma `WHERE` filtra i record *prima di* ogni raggruppamento:

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department;
```

Se è necessario filtrare i risultati dopo aver eseguito il raggruppamento, ad esempio per visualizzare solo i dipartimenti il cui reddito medio è superiore a 1000, è necessario utilizzare la clausola `HAVING` :

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department
HAVING avg(income) > 1000;
```

Selezione con più di 1 condizione.

La parola chiave `AND` viene utilizzata per aggiungere più condizioni alla query.

Nome	Età	Genere
Sam	18	M
John	21	M
peso	22	M

Nome	Età	Genere
Maria	23	F

```
SELECT name FROM persons WHERE gender = 'M' AND age > 20;
```

Questo ritornerà:

Nome
John
peso

usando la parola chiave `OR`

```
SELECT name FROM persons WHERE gender = 'M' OR age < 20;
```

Questo ritornerà:

nome
Sam
John
peso

Queste parole chiave possono essere combinate per consentire combinazioni di criteri più complessi:

```
SELECT name
FROM persons
WHERE (gender = 'M' AND age < 20)
   OR (gender = 'F' AND age > 20);
```

Questo ritornerà:

nome
Sam
Maria

Leggi SELEZIONARE online: https://riptutorial.com/it/sql/topic/222/selezionare

Capitolo 48: Sequenza

Examples

Crea sequenza

```
CREATE SEQUENCE orders_seq
START WITH     1000
INCREMENT BY   1;
```

Crea una sequenza con un valore iniziale di 1000 che viene incrementato di 1.

Usando le sequenze

un riferimento a *seq_name.* NEXTVAL viene utilizzato per ottenere il valore successivo in una sequenza. Una singola istruzione può generare solo un singolo valore di sequenza. Se ci sono più riferimenti a NEXTVAL in una dichiarazione, utilizzeranno lo stesso numero generato.

NEXTVAL può essere utilizzato per INSERTI

```
INSERT INTO Orders (Order_UID, Customer)
        VALUES (orders_seq.NEXTVAL, 1032);
```

Può essere utilizzato per gli AGGIORNAMENTI

```
UPDATE Orders
SET Order_UID = orders_seq.NEXTVAL
WHERE Customer = 581;
```

Può anche essere usato per SELEZIONI

```
SELECT Order_seq.NEXTVAL FROM dual;
```

Leggi Sequenza online: https://riptutorial.com/it/sql/topic/1586/sequenza

Capitolo 49: Sinonimi

Examples

Crea sinonimo

```
CREATE SYNONYM EmployeeData
FOR MyDatabase.dbo.Employees
```

Leggi Sinonimi online: https://riptutorial.com/it/sql/topic/2518/sinonimi

Capitolo 50: SKIP TAKE (impaginazione)

Examples

Saltare alcune righe dal risultato

ISO / ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 42424242424242;
-- skips 20 for take use very large number that is more than rows in table
```

Oracolo:

```
SELECT Id,
   Col1
FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber > 20
```

PostgreSQL:

```
SELECT * FROM TableName OFFSET 20;
```

SQLite:

```
SELECT * FROM TableName LIMIT -1 OFFSET 20;
```

Quantità limitante di risultati

ISO / ANSI SQL:

```
SELECT * FROM TableName FETCH FIRST 20 ROWS ONLY;
```

MySQL; PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20;
```

Oracolo:

```
SELECT Id,
   Col1
FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber <= 20
```

Server SQL:

```
SELECT TOP 20 *
FROM dbo.[Sale]
```

Saltare quindi con alcuni risultati (impaginazione)

ISO / ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 20; -- offset, limit
```

Oracolo; Server SQL:

```
SELECT Id,
   Col1
 FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber BETWEEN 21 AND 40
```

PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20 OFFSET 20;
```

Leggi SKIP TAKE (impaginazione) online: https://riptutorial.com/it/sql/topic/2927/skip-take--impaginazione-

Capitolo 51: SPIEGARE e DESCRIVERE

Examples

DESCRIZIONE tablename;

`DESCRIBE` e `EXPLAIN` sono sinonimi. `DESCRIBE` in un tablename restituisce la definizione delle colonne.

```
DESCRIBE tablename;
```

Exmple Result:

```
COLUMN_NAME     COLUMN_TYPE     IS_NULLABLE     COLUMN_KEY     COLUMN_DEFAULT    EXTRA
id              int(11)         NO              PRI            0
auto_increment
test            varchar(255)    YES                            (null)
```

Qui vedete i nomi delle colonne, seguiti dal tipo di colonna. Mostra se è consentito il `null` nella colonna e se la colonna utilizza un indice. viene anche visualizzato il valore predefinito e se la tabella contiene un comportamento speciale come un `auto_increment` .

SPIEGARE Seleziona query

Un `Explain` di una query di `select` mostra come verrà eseguita la query. In questo modo è possibile verificare se la query utilizza un indice o se è possibile ottimizzare la query aggiungendo un indice.

Query di esempio:

```
explain select * from user join data on user.test = data.fk_user;
```

Esempio di risultato:

```
id  select_type  table   type    possible_keys  key     key_len ref       rows  Extra
1   SIMPLE       user    index   test           test    5       (null)    1     Using where;
Using index
1   SIMPLE       data    ref     fk_user        fk_user 5       user.test 1     (null)
```

sul `type` si vede se è stato utilizzato un indice. Nella colonna `possible_keys` si vede se il piano di esecuzione può scegliere tra diversi indici se nessuno esiste. `key` ti dice l'indice usato acutal. `key_len` mostra la dimensione in byte per un oggetto indice. Più basso è questo valore, più elementi dell'indice si adattano alla stessa dimensione di memoria e possono essere elaborati più velocemente. `rows` mostra il numero previsto di righe che la query deve analizzare, minore è il migliore.

Leggi SPIEGARE e DESCRIVERE online: https://riptutorial.com/it/sql/topic/2928/spiegare-e-descrivere

Capitolo 52: SQL CURSOR

Examples

Esempio di un cursore che interroga tutte le righe per indice per ciascun database

Qui, un cursore viene utilizzato per scorrere tutti i database.
Inoltre, un cursore da sql dinamico viene utilizzato per interrogare ogni database restituito dal primo cursore.
Questo per dimostrare l'ambito di connessione di un cursore.

```
DECLARE @db_name nvarchar(255)
DECLARE @sql nvarchar(MAX)

DECLARE @schema nvarchar(255)
DECLARE @table nvarchar(255)
DECLARE @column nvarchar(255)




DECLARE db_cursor CURSOR FOR
SELECT name FROM sys.databases


OPEN db_cursor
FETCH NEXT FROM db_cursor INTO @db_name

WHILE @@FETCH_STATUS = 0
BEGIN
    SET @sql = 'SELECT * FROM ' + QUOTENAME(@db_name) + '.information_schema.columns'
    PRINT ''
    PRINT ''
    PRINT ''
    PRINT @sql
    -- EXECUTE(@sql)



    -- For each database

    DECLARE @sqlstatement nvarchar(4000)
    --move declare cursor into sql to be executed
    SET @sqlstatement = 'DECLARE  columns_cursor CURSOR FOR SELECT TABLE_SCHEMA, TABLE_NAME,
COLUMN_NAME FROM ' + QUOTENAME(@db_name) + '.information_schema.columns ORDER BY TABLE_SCHEMA,
TABLE_NAME, ORDINAL_POSITION'



    EXEC sp_executesql @sqlstatement


    OPEN columns_cursor
    FETCH NEXT FROM columns_cursor
```

```
    INTO @schema, @table, @column

    WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT @schema + '.' + @table + '.' + @column
        --EXEC asp_DoSomethingStoredProc @UserId

    FETCH NEXT FROM columns_cursor --have to fetch again within loop
    INTO @schema, @table, @column

    END
    CLOSE columns_cursor
    DEALLOCATE columns_cursor

    -- End for each database




    FETCH NEXT FROM db_cursor INTO @db_name
END

CLOSE db_cursor
DEALLOCATE db_cursor
```

Leggi SQL CURSOR online: https://riptutorial.com/it/sql/topic/8895/sql-cursor

Capitolo 53: SQL Group per contro distinto

Examples

Differenza tra GROUP BY e DISTINCT

`GROUP BY` viene utilizzato in combinazione con le funzioni di aggregazione. Considera la seguente tabella:

ID ordine	ID utente	nome del negozio	orderValue	data dell'ordine
1	43	Negozio A	25	20-03-2016
2	57	Negozio B	50	22-03-2016
3	43	Negozio A	30	25-03-2016
4	82	Negozio C	10	26-03-2016
5	21	Negozio A	45	29-03-2016

La query seguente utilizza `GROUP BY` per eseguire calcoli aggregati.

```
SELECT
    storeName,
    COUNT(*) AS total_nr_orders,
    COUNT(DISTINCT userId) AS nr_unique_customers,
    AVG(orderValue) AS average_order_value,
    MIN(orderDate) AS first_order,
    MAX(orderDate) AS lastOrder
FROM
    orders
GROUP BY
    storeName;
```

e restituirà le seguenti informazioni

nome del negozio	total_nr_orders	nr_unique_customers	average_order_value	primo ordine	ultimo ordine
Negozio A	3	2	33.3	20-03-2016	29-03-2016
Negozio B	1	1	50	22-03-2016	22-03-2016
Negozio	1	1	10	26-03-	26-03-

nome del negozio	total_nr_orders	nr_unique_customers	average_order_value	primo ordine	ultimo ordine
C				2016	2016

Mentre `DISTINCT` viene utilizzato per elencare una combinazione univoca di valori distinti per le colonne specificate.

```
SELECT DISTINCT
    storeName,
    userId
FROM
    orders;
```

nome del negozio	ID utente
Negozio A	43
Negozio B	57
Negozio C	82
Negozio A	21

Leggi SQL Group per contro distinto online: https://riptutorial.com/it/sql/topic/2499/sql-group-per-contro-distinto

Capitolo 54: SQL Injection

introduzione

SQL injection è un tentativo di accedere alle tabelle del database di un sito Web iniettando SQL in un campo modulo. Se un server Web non protegge dagli attacchi SQL injection, un hacker può ingannare il database nell'esecuzione del codice SQL aggiuntivo. Eseguendo il proprio codice SQL, gli hacker possono aggiornare l'accesso al proprio account, visualizzare le informazioni private di qualcun altro o apportare altre modifiche al database.

Examples

Campione di iniezione SQL

Supponendo che la chiamata al gestore di accesso dell'applicazione Web assomiglia a questo:

```
https://somepage.com/ajax/login.ashx?username=admin&password=123
```

Ora in login.ashx, leggi questi valori:

```
strUserName = getHttpsRequestParameterString("username");
strPassword = getHttpsRequestParameterString("password");
```

e interrogare il database per determinare se esiste un utente con quella password.

Quindi costruisci una stringa di query SQL:

```
txtSQL = "SELECT * FROM Users WHERE username = '" + strUserName + "' AND password = '"+
strPassword +"'";
```

Questo funzionerà se il nome utente e la password non contengono un preventivo.

Tuttavia, se uno dei parametri contiene una citazione, l'SQL che viene inviato al database sarà simile a questo:

```
-- strUserName = "d'Alambert";
txtSQL = "SELECT * FROM Users WHERE username = 'd'Alambert' AND password = '123'";
```

Ciò si tradurrà in un errore di sintassi, perché la citazione dopo la `d` in `d'Alambert` termina la stringa SQL.

Puoi correggere ciò evitando le virgolette in username e password, ad esempio:

```
strUserName = strUserName.Replace("'", "''");
strPassword = strPassword.Replace("'", "''");
```

Tuttavia, è più appropriato utilizzare i parametri:

```
cmd.CommandText = "SELECT * FROM Users WHERE username = @username AND password = @password";

cmd.Parameters.Add("@username", strUserName);
cmd.Parameters.Add("@password", strPassword);
```

Se non si utilizzano i parametri e si dimentica di sostituire il preventivo anche in uno dei valori, un utente malintenzionato (noto anche come hacker) può utilizzarlo per eseguire comandi SQL sul proprio database.

Ad esempio, se un utente malintenzionato è malvagio, verrà impostata la password

```
lol'; DROP DATABASE master; --
```

e quindi l'SQL sarà simile a questo:

```
"SELECT * FROM Users WHERE username = 'somebody' AND password = 'lol'; DROP DATABASE master; --'";
```

Sfortunatamente per te, questo è SQL valido e il DB lo eseguirà!

Questo tipo di exploit è chiamato un'iniezione SQL.

Ci sono molte altre cose che un utente malintenzionato potrebbe fare, come rubare l'indirizzo email di ogni utente, rubare la password di tutti, rubare numeri di carte di credito, rubare qualsiasi quantità di dati nel database, ecc.

Questo è il motivo per cui hai sempre bisogno di sfuggire alle tue corde.
E il fatto che tu ti dimenticerai immancabilmente di farlo prima o poi è esattamente il motivo per cui dovresti usare i parametri. Perché se si usano i parametri, il framework del linguaggio di programmazione farà tutto il necessario per evadere.

campione di iniezione semplice

Se l'istruzione SQL è costruita in questo modo:

```
SQL = "SELECT * FROM Users WHERE username = '" + user + "' AND password ='" + pw + "'";
db.execute(SQL);
```

Quindi un hacker potrebbe recuperare i tuoi dati fornendo una password come `pw' or '1'='1` ; la dichiarazione SQL risultante sarà:

```
SELECT * FROM Users WHERE username = 'somebody' AND password ='pw' or '1'='1'
```

Questo passerà il controllo della password per tutte le righe nella tabella `Users` perché `'1'='1'` è sempre true.

Per evitare ciò, utilizzare i parametri SQL:

```
SQL = "SELECT * FROM Users WHERE username = ? AND password = ?";
db.execute(SQL, [user, pw]);
```

Leggi SQL Injection online: https://riptutorial.com/it/sql/topic/3517/sql-injection

Capitolo 55: subquery

Osservazioni

Le sottoquery possono apparire in diverse clausole di una query esterna o nell'operazione set.

Devono essere racchiusi tra parentesi `()` . Se il risultato della sottoquery viene confrontato con qualcos'altro, il numero di colonne deve corrispondere. Gli alias di tabella sono necessari per le sottoquery nella clausola FROM per denominare la tabella temporanea.

Examples

Sottoquery nella clausola WHERE

Utilizzare una sottoquery per filtrare il set di risultati. Ad esempio, questo restituirà tutti i dipendenti con uno stipendio pari all'impiegato più pagato.

```
SELECT *
FROM Employees
WHERE Salary = (SELECT MAX(Salary) FROM Employees)
```

Sottoquery nella clausola FROM

Una sottoquery in una clausola `FROM` agisce in modo simile a una tabella temporanea generata durante l'esecuzione di una query e successivamente persa.

```
SELECT Managers.Id, Employees.Salary
FROM (
  SELECT Id
  FROM Employees
  WHERE ManagerId IS NULL
) AS Managers
JOIN Employees ON Managers.Id = Employees.Id
```

Sottoquery nella clausola SELECT

```
SELECT
  Id,
  FName,
  LName,
  (SELECT COUNT(*) FROM Cars WHERE Cars.CustomerId = Customers.Id) AS NumberOfCars
FROM Customers
```

Sottoquery nella clausola FROM

È possibile utilizzare le sottoquery per definire una tabella temporanea e utilizzarla nella clausola FROM di una query "esterna".

```
SELECT * FROM (SELECT city, temp_hi - temp_lo AS temp_var FROM weather) AS w
WHERE temp_var > 20;
```

Quanto sopra trova le città dalla tabella del tempo la cui variazione di temperatura giornaliera è maggiore di 20. Il risultato è:

città	temp_var
ST. LOUIS	21
LOS ANGELES	31
LOS ANGELES	23
LOS ANGELES	31
LOS ANGELES	27
LOS ANGELES	28
LOS ANGELES	28
LOS ANGELES	32

.

Sottoquery nella clausola WHERE

L'esempio seguente trova le città (dall'esempio delle città) la cui popolazione è inferiore alla temperatura media (ottenuta tramite una subquery):

```
SELECT name, pop2000 FROM cities
WHERE pop2000 < (SELECT avg(pop2000)  FROM cities);
```

Qui: la sottoquery (SELECT avg (pop2000) FROM cities) viene utilizzata per specificare le condizioni nella clausola WHERE. Il risultato è:

nome	pop2000
San Francisco	776.733
ST. LOUIS	348.189
Kansas City	146.866

Sottoquery nella clausola SELECT

Le sottoquery possono anche essere utilizzate nella parte `SELECT` della query esterna. La seguente

query mostra tutte le colonne della tabella meteo con gli stati corrispondenti dalla tabella delle città .

```
SELECT w.*,  (SELECT c.state FROM cities AS c WHERE c.name = w.city ) AS state
FROM weather AS w;
```

Filtra i risultati della query utilizzando la query su una tabella diversa

Questa query seleziona tutti i dipendenti non presenti nella tabella Supervisori.

```
SELECT *
FROM Employees
WHERE EmployeeID not in (SELECT EmployeeID
                            FROM Supervisors)
```

Gli stessi risultati possono essere ottenuti usando un SINISTRA SINISTRA.

```
SELECT *
FROM Employees AS e
LEFT JOIN Supervisors AS s ON s.EmployeeID=e.EmployeeID
WHERE s.EmployeeID is NULL
```

Subquery correlate

Le sottoquery correlate (note anche come sincronizzate o coordinate) sono query nidificate che fanno riferimento alla riga corrente della query esterna:

```
SELECT EmployeeId
    FROM Employee AS eOuter
    WHERE Salary > (
       SELECT AVG(Salary)
       FROM Employee eInner
       WHERE eInner.DepartmentId = eOuter.DepartmentId
    )
```

Subquery `SELECT AVG(Salary) ...` è *correlato* perché fa riferimento alla riga `Employee eOuter` dalla sua query esterna.

Leggi subquery online: https://riptutorial.com/it/sql/topic/1606/subquery

Capitolo 56: Tabella DROP

Osservazioni

DROP TABLE rimuove la definizione della tabella dallo schema insieme alle righe, agli indici, alle autorizzazioni e ai trigger.

Examples

Semplice caduta

```
Drop Table MyTable;
```

Verifica dell'esistenza prima di lasciar cadere

MySQL 3.19

```
DROP TABLE IF EXISTS MyTable;
```

PostgreSQL 8.x

```
DROP TABLE IF EXISTS MyTable;
```

SQL Server 2005

```
If Exists(Select * From Information_Schema.Tables
          Where Table_Schema = 'dbo'
            And Table_Name = 'MyTable')
  Drop Table dbo.MyTable
```

SQLite 3.0

```
DROP TABLE IF EXISTS MyTable;
```

Leggi Tabella DROP online: https://riptutorial.com/it/sql/topic/1832/tabella-drop

Capitolo 57: Tavolo di design

Osservazioni

The Open University (1999) Relational Database Systems: Block 2 Relational Theory, Milton Keynes, The Open University.

Examples

Proprietà di un tavolo ben progettato.

Un vero database relazionale deve andare oltre il lancio di dati in poche tabelle e la scrittura di alcune istruzioni SQL per estrarre tali dati.
Nella migliore delle ipotesi una struttura di tabelle mal progettata rallenterà l'esecuzione delle query e potrebbe rendere impossibile il funzionamento del database come previsto.

Una tabella di database non deve essere considerata solo come un'altra tabella; deve seguire un insieme di regole per essere considerato veramente relazionale. Accademicamente è indicato come una "relazione" per fare la distinzione.

Le cinque regole di una tabella relazionale sono:

1. Ogni valore è *atomico* ; il valore in ogni campo in ogni riga deve essere un singolo valore.
2. Ogni campo contiene valori che hanno lo stesso tipo di dati.
3. Ogni intestazione di campo ha un nome univoco.
4. Ogni riga nella tabella deve avere almeno un valore che la rende unica tra gli altri record nella tabella.
5. L'ordine delle righe e delle colonne non ha alcun significato.

Una tabella conforme alle cinque regole:

Id	Nome	DOB	Manager
1	Fred	1971/11/02	3
2	Fred	1971/11/02	3
3	Citare in giudizio	1975/08/07	2

- Regola 1: ogni valore è atomico. `Id` , `Name` , `DOB` e `Manager` contengono solo un singolo valore.
- Regola 2: `Id` contiene solo numeri interi, `Name` contiene testo (potremmo aggiungere che è testo di quattro caratteri o meno), `DOB` contiene date di un tipo valido e `Manager` contiene numeri interi (potremmo aggiungere che corrisponde a un campo Chiave primaria in un gestore tavolo).
- Regola 3: `Id` , `Name` , `DOB` e `Manager` sono nomi di intestazione univoci all'interno della tabella.

- Regola 4: l'inclusione del campo `Id` assicura che ogni record sia distinto da qualsiasi altro record all'interno della tabella.

Un tavolo mal progettato:

Id	Nome	DOB	Nome
1	Fred	1971/11/02	3
1	Fred	1971/11/02	3
3	Citare in giudizio	Venerdì 18 luglio 1975	2, 1

- Regola 1: il campo del secondo nome contiene due valori: 2 e 1.
- Regola 2: il campo DOB contiene date e testo.
- Regola 3: Ci sono due campi chiamati 'nome'.
- Regola 4: il primo e il secondo record sono esattamente gli stessi.
- Regola 5: questa regola non è infranta.

Leggi Tavolo di design online: https://riptutorial.com/it/sql/topic/2515/tavolo-di-design

Capitolo 58: Tipi di dati

Examples

DECIMALE e NUMERICO

Precisione fissa e numeri decimali in scala. `DECIMAL` e `NUMERIC` sono funzionalmente equivalenti.

Sintassi:

```
DECIMAL ( precision [ , scale] )
NUMERIC ( precision [ , scale] )
```

Esempi:

```
SELECT CAST(123 AS DECIMAL(5,2)) --returns 123.00
SELECT CAST(12345.12 AS NUMERIC(10,5)) --returns 12345.12000
```

FLOAT e REAL

Tipi di dati con numero approssimativo da utilizzare con dati numerici in virgola mobile.

```
SELECT CAST( PI() AS FLOAT) --returns 3.14159265358979
SELECT CAST( PI() AS REAL) --returns 3.141593
```

Interi

Tipi di dati con numero esatto che utilizzano dati interi.

Tipo di dati	Gamma	Conservazione
bigint	-2 ^ 63 (-9.223.372.036.854.775.808) a 2 ^ 63-1 (9.223.372.036.854.775.807)	8 byte
int	-2 ^ 31 (-2.147.483.648) a 2 ^ 31-1 (2.147.483.647)	4 byte
smallint	-2 ^ 15 (-32.768) a 2 ^ 15-1 (32.767)	2 byte
tinyint	Da 0 a 255	1 byte

SOLDI e SMALLMONEY

Tipi di dati che rappresentano valori monetari o valutari.

Tipo di dati	Gamma	Conservazione
i soldi	-922,337,203,685,477,5808 a 922,337,203,685,477,5807	8 byte
smallmoney	Da -214.748.3648 a 214.748.3647	4 byte

BINARIO e VARBINARIO

Tipi di dati binari di lunghezza fissa o lunghezza variabile.

Sintassi:

```
BINARY [ ( n_bytes ) ]
VARBINARY [ ( n_bytes | max ) ]
```

`n_bytes` può essere qualsiasi numero compreso tra 1 e 8000 byte. `max` indica che lo spazio di archiviazione massimo è 2 ^ 31-1.

Esempi:

```
SELECT CAST(12345 AS BINARY(10)) -- 0x0000000000000003039
SELECT CAST(12345 AS VARBINARY(10)) -- 0x00003039
```

CHAR e VARCHAR

Tipi di dati di stringa di lunghezza fissa o lunghezza variabile.

Sintassi:

```
CHAR [ ( n_chars ) ]
VARCHAR [ ( n_chars ) ]
```

Esempi:

```
SELECT CAST('ABC' AS CHAR(10)) -- 'ABC       ' (padded with spaces on the right)
SELECT CAST('ABC' AS VARCHAR(10)) -- 'ABC' (no padding due to variable character)
SELECT CAST('ABCDEFGHIJKLMNOPQRSTUVWXYZ' AS CHAR(10))  -- 'ABCDEFGHIJ' (truncated to 10
characters)
```

NCHAR e NVARCHAR

Tipi di dati stringa UNICODE di lunghezza fissa o lunghezza variabile.

Sintassi:

```
NCHAR [ ( n_chars ) ]
NVARCHAR [ ( n_chars | MAX ) ]
```

Usa `MAX` per stringhe molto lunghe che possono superare gli 8000 caratteri.

IDENTIFICATIVO UNICO

Un GUID / UUID a 16 byte.

```
DECLARE @GUID UNIQUEIDENTIFIER = NEWID();
SELECT @GUID -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
DECLARE @bad_GUID_string VARCHAR(100) = 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
SELECT
    @bad_GUID_string,   -- 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
    CONVERT(UNIQUEIDENTIFIER, @bad_GUID_string) -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
```

Leggi Tipi di dati online: https://riptutorial.com/it/sql/topic/1166/tipi-di-dati

Capitolo 59: trigger

Examples

CREARE TRIGGER

In questo esempio viene creato un trigger che inserisce un record in una seconda tabella (MyAudit) dopo che un record è stato inserito nella tabella in cui è definito il trigger su (MyTable). Qui la tabella "inserted" è una tabella speciale utilizzata da Microsoft SQL Server per memorizzare le righe interessate durante le istruzioni INSERT e UPDATE; c'è anche una speciale tabella "eliminata" che esegue la stessa funzione per le istruzioni DELETE.

```
CREATE TRIGGER MyTrigger
    ON MyTable
    AFTER INSERT

AS

BEGIN
    -- insert audit record to MyAudit table
    INSERT INTO MyAudit(MyTableId, User)
    (SELECT MyTableId, CURRENT_USER FROM inserted)
END
```

Usa Trigger per gestire un "Cestino" per gli elementi eliminati

```
CREATE TRIGGER BooksDeleteTrigger
    ON MyBooksDB.Books
    AFTER DELETE
AS
  INSERT INTO BooksRecycleBin
    SELECT *
    FROM deleted;
GO
```

Leggi trigger online: https://riptutorial.com/it/sql/topic/1432/trigger

Capitolo 60: TRONCARE

introduzione

L'istruzione TRUNCATE elimina tutti i dati da una tabella. È simile a DELETE senza filtro, ma, a seconda del software del database, presenta alcune restrizioni e ottimizzazioni.

Sintassi

- TRUNCATE TABLE nome_tabella;

Osservazioni

TRUNCATE è un comando DDL (Data Definition Language) e, come tale, esistono differenze significative tra esso e DELETE (un linguaggio di manipolazione dei dati, DML, comando). Mentre TRUNCATE può essere un mezzo per rimuovere rapidamente grandi volumi di record da un database, queste differenze dovrebbero essere comprese al fine di decidere se l'uso di un comando TRUNCATE è adatto alla tua particolare situazione.

- TRUNCATE è un'operazione di pagina di dati. Pertanto i trigger DML (ON DELETE) associati alla tabella non si attivano quando si esegue un'operazione TRUNCATE. Mentre questo farà risparmiare un sacco di tempo per le massicce operazioni di cancellazione, tuttavia potrebbe essere necessario eliminare manualmente i dati correlati.
- TRUNCATE rilascerà lo spazio su disco utilizzato dalle righe eliminate, DELETE rilascerà spazio
- Se la tabella da troncare utilizza colonne Identity (MS SQL Server), il seed viene resettato dal comando TRUNCATE. Ciò potrebbe causare problemi di integrità referenziale
- A seconda dei ruoli di sicurezza esistenti e della variante di SQL in uso, è possibile che non si disponga delle autorizzazioni necessarie per eseguire un comando TRUNCATE

Examples

Rimozione di tutte le righe dalla tabella Impiegato

```
TRUNCATE TABLE Employee;
```

Utilizzare la tabella troncata è spesso meglio quindi utilizzare DELETE TABLE poiché ignora tutti gli indici e i trigger e rimuove semplicemente tutto.

Elimina tabella è un'operazione basata su righe, ciò significa che ogni riga viene eliminata. Tronca tabella è un'operazione di una pagina di dati l'intera pagina di dati viene riallocata. Se si dispone di una tabella con un milione di righe, sarà molto più veloce troncare la tabella di quanto non sarebbe utilizzare un'istruzione di tabella di eliminazione.

Sebbene possiamo eliminare righe specifiche con DELETE, non possiamo TRONCARE righe specifiche, possiamo solo TRONCARE tutti i record contemporaneamente. Eliminando Tutte le righe e quindi inserendo un nuovo record, si continuerà ad aggiungere il valore della chiave primaria incrementale Auto dal valore precedentemente inserito, dove, come in Tronca, anche il valore della chiave primaria Incrementale automatica verrà ripristinato e verrà avviato da 1.

Si noti che quando si tronca una tabella, **non devono essere presenti chiavi esterne** , altrimenti si otterrà un errore.

Leggi TRONCARE online: https://riptutorial.com/it/sql/topic/1466/troncare

Capitolo 61: UNIONE / UNIONE TUTTI

introduzione

La parola chiave ***UNION*** in SQL viene utilizzata per combinare i risultati dell'istruzione ***SELECT*** con qualsiasi duplicato. Per utilizzare UNION e combinare i risultati, entrambe le istruzioni SELECT devono avere lo stesso numero di colonne con lo stesso tipo di dati nello stesso ordine, ma la lunghezza della colonna può essere diversa.

Sintassi

- SELECT column_1 [, column_2] FROM table_1 [, table_2] [WHERE condition]
  UNIONE | UNIONE TUTTI
  SELECT column_1 [, column_2] FROM table_1 [, table_2] [WHERE condition]

Osservazioni

`UNION` clausole `UNION` e `UNION ALL` combinano il set di risultati di due o più istruzioni SELECT identicamente strutturate in un singolo risultato / tabella.

Sia il numero di colonne che i tipi di colonna per ogni query devono corrispondere affinché `UNION` `UNION ALL` `UNION` / `UNION ALL` funzioni.

La differenza tra una query `UNION` e `UNION ALL` è che la clausola `UNION` rimuoverà qualsiasi riga duplicata nel risultato in cui `UNION ALL` non lo farà.

Questa distinta rimozione dei record può rallentare in modo significativo le query anche se non ci sono righe distinte da rimuovere a causa di questo se si sa che non ci saranno duplicati (o non importa) sempre predefinito su `UNION ALL` per una query più ottimizzata.

Examples

Basic UNION ALL query

```
CREATE TABLE HR_EMPLOYEES
(
    PersonID int,
    LastName VARCHAR(30),
    FirstName VARCHAR(30),
    Position VARCHAR(30)
);

CREATE TABLE FINANCE_EMPLOYEES
(
    PersonID INT,
    LastName VARCHAR(30),
    FirstName VARCHAR(30),
```

```
    Position VARCHAR(30)
);
```

Diciamo che vogliamo estrarre i nomi di tutti i `managers` dai nostri reparti.

Utilizzando `UNION` , possiamo ottenere tutti i dipendenti delle risorse umane e finanziarie, che detengono la `position` di `manager`

```
SELECT
    FirstName, LastName
FROM
    HR_EMPLOYEES
WHERE
    Position = 'manager'
UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES
WHERE
    Position = 'manager'
```

L'istruzione `UNION` rimuove le righe duplicate dai risultati della query. Dal momento che è possibile avere persone con lo stesso nome e posizione in entrambi i reparti, stiamo utilizzando `UNION ALL` , per non rimuovere i duplicati.

Se si desidera utilizzare un alias per ciascuna colonna di output, è sufficiente inserirli nella prima istruzione select, come segue:

```
SELECT
    FirstName as 'First Name', LastName as 'Last Name'
FROM
    HR_EMPLOYEES
WHERE
    Position = 'manager'
UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES
WHERE
    Position = 'manager'
```

Semplice spiegazione ed esempio

In parole povere:

- `UNION` unisce 2 set di risultati rimuovendo i duplicati dal set di risultati
- `UNION ALL` unisce 2 set di risultati senza tentare di rimuovere i duplicati

  Un errore che molte persone fanno è usare `UNION` quando non è necessario rimuovere i duplicati. Il costo aggiuntivo delle prestazioni rispetto a set di risultati di grandi dimensioni può essere molto significativo.

Quando potresti aver bisogno di `UNION`

Supponiamo che sia necessario filtrare una tabella con 2 diversi attributi e che siano stati creati indici separati non in cluster per ogni colonna. UN `UNION` ti consente di sfruttare entrambi gli indici evitando al contempo duplicati.

```
SELECT C1, C2, C3 FROM Table1 WHERE C1 = @Param1
UNION
SELECT C1, C2, C3 FROM Table1 WHERE C2 = @Param2
```

Questo semplifica l'ottimizzazione delle prestazioni poiché sono necessari solo indici semplici per eseguire queste query in modo ottimale. Si può anche essere in grado di cavarsela con un numero un po 'inferiore di indici non in cluster, migliorando anche le prestazioni di scrittura complessive rispetto alla tabella di origine.

Quando potresti aver bisogno di `UNION ALL`

Supponiamo che sia necessario filtrare una tabella con 2 attributi, ma non è necessario filtrare i record duplicati (perché non importa o i dati non produrranno duplicati durante l'unione a causa della progettazione del modello di dati).

```
SELECT C1 FROM Table1
UNION ALL
SELECT C1 FROM Table2
```

Ciò è particolarmente utile durante la creazione di viste che uniscono dati progettati per essere fisicamente partizionati su più tabelle (forse per motivi di prestazioni, ma che comunque desidera registrare i record). Poiché i dati sono già divisi, il fatto che il motore di database rimuova i duplicati non aggiunge alcun valore e aggiunge semplicemente un tempo di elaborazione aggiuntivo alle query.

Leggi UNIONE / UNIONE TUTTI online: https://riptutorial.com/it/sql/topic/349/unione---unione-tutti

Capitolo 62: Viste materializzate

introduzione

Una vista materializzata è una vista i cui risultati sono archiviati fisicamente e devono essere periodicamente aggiornati per rimanere aggiornati. Sono quindi utili per archiviare i risultati di query complesse e di lunga durata quando non sono richiesti risultati in tempo reale. Le viste materializzate possono essere create in Oracle e PostgreSQL. Altri sistemi di database offrono funzionalità simili, come le viste indicizzate di SQL Server o le tabelle di query materializzate di DB2.

Examples

Esempio di PostgreSQL

```
CREATE TABLE mytable (number INT);
INSERT INTO mytable VALUES (1);

CREATE MATERIALIZED VIEW myview AS SELECT * FROM mytable;

SELECT * FROM myview;
 number
--------
      1
(1 row)

INSERT INTO mytable VALUES(2);

SELECT * FROM myview;
 number
--------
      1
(1 row)

REFRESH MATERIALIZED VIEW myview;

SELECT * FROM myview;
 number
--------
      1
      2
(2 rows)
```

Leggi Viste materializzate online: https://riptutorial.com/it/sql/topic/8367/viste-materializzate

Capitolo 63: Visualizzazioni

Examples

Viste semplici

Una vista può filtrare alcune righe dalla tabella di base o proiettare solo alcune colonne da essa:

```
CREATE VIEW new_employees_details AS
SELECT E.id, Fname, Salary, Hire_date
FROM Employees E
WHERE hire_date > date '2015-01-01';
```

Se selezioni forma la vista:

```
select * from new_employees_details
```

Id	FName	Stipendio	Data di assunzione
4	Johnathon	500	24-07-2016

Viste complesse

Una vista può essere una query davvero complessa (aggregazioni, join, sottoquery, ecc.). Assicurati di aggiungere i nomi delle colonne per tutto ciò che selezioni:

```
Create VIEW dept_income AS
SELECT d.Name as DepartmentName, sum(e.salary) as TotalSalary
FROM Employees e
JOIN Departments d on e.DepartmentId = d.id
GROUP BY d.Name;
```

Ora puoi selezionarlo come da qualsiasi tabella:

```
SELECT *
FROM dept_income;
```

Nome Dipartimento	TotalSalary
HR	1900
I saldi	600

Leggi Visualizzazioni online: https://riptutorial.com/it/sql/topic/766/visualizzazioni

Capitolo 64: XML

Examples

Query dal tipo di dati XML

```
DECLARE @xmlIN XML = '<TableData>
<aaa Main="First">
  <row name="a" value="1" />
  <row name="b" value="2" />
  <row name="c" value="3" />
</aaa>
<aaa Main="Second">
  <row name="a" value="3" />
  <row name="b" value="4" />
  <row name="c" value="5" />
</aaa>
<aaa Main="Third">
  <row name="a" value="10" />
  <row name="b" value="20" />
  <row name="c" value="30" />
</aaa>
</TableData>'

SELECT t.col.value('../@Main', 'varchar(10)') [Header],
t.col.value('@name', 'VARCHAR(25)') [name],
t.col.value('@value',  'VARCHAR(25)') [Value]
FROM    @xmlIn.nodes('//TableData/aaa/row') AS t (col)
```

risultati

```
Header    name    Value
First      a        1
First      b        2
First      c        3
Second     a        3
Second     b        4
Second     c        5
Third      a        10
Third      b        20
Third      c        30
```

Leggi XML online: https://riptutorial.com/it/sql/topic/4421/xml

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con SQL	Arjan Einbu, brichins, Burkhard, cale_b, CL., Community, Devmati Wadikar, Epodax, geeksal, H. Pauwelyn, Hari, Joey, JohnLBevan, Jon Ericson, Lankymart, Laurel, Mureinik, Nathan, omini data, PeterRing, Phrancis, Prateek, RamenChef, Ray, Simone Carletti, SZenC, t1gor, ypercube
2	ADERIRE	A_Arnold, Akshay Anand, Andy G, bignose, Branko Dimitrijevic, Casper Spruit, CL., Daniel Langemann, Darren Bartrup-Cook, Dipesh Poudel, enrico.bacis, Florin Ghita, forsvarir, Franck Dernoncourt, hairboat, Hari K M, HK1, HLGEM, inquisitive_mind , John C, John Odom, John Slegers, Mark Iannucci, Marvin, Mureinik, Phrancis, raholling, Raidri, Saroj Sasmal, Stefan Steiger, sunkuet02, Tot Zam, xenodevil, ypercube, Рахул Маквана
3	AGGIORNARE	Akshay Anand, CL., Daniel Vérité, Dariusz, Dipesh Poudel, FlyingPiMonster, Gidil, H. Pauwelyn, Jon Chan, KIRAN KUMAR MATAM, Matas Vaitkevicius, Matt, Phrancis, Sanjay Bharwani, sunkuet02, Tot Zam, TriskalJM, vmaroli, WesleyJohnson
4	Algebra relazionale	CL., Darren Bartrup-Cook, Martin Smith
5	ALTER TABLE	Aidan, blackbishop, bluefeet, CL., Florin Ghita, Francis Lord, guiguiblitz, Joe W, KIRAN KUMAR MATAM, Lexi, mithra chintha , Ozair Kafray, Simon Foster, Siva Rama Krishna
6	applicare croce, applicare esterno	Karthikeyan, RamenChef
7	ASTUCCIO	ɐləx, Christos, CL., Dariusz, Fenton, Infinity, Jaydles, Matt, MotKohn, Mureinik, Peter Lang, Stanislovas Kalašnikovas
8	Blocchi di esecuzione	Phrancis
9	Chiavi esterne	CL., Harjot, Yehuda Shapira
10	Chiavi primarie	Andrea Montanari, CL., FlyingPiMonster, KjetilNordin
11	Clausola	CL., juergen d, walid, Zaga
12	CLAUSOLA	Blag, Özgür Öztürk

	ESISTENTE	
13	Commenti	CL., Phrancis
14	CONCEDERE e REVOCARE	RamenChef, user2314737
15	CREA FUNZIONE	John Odom, Ricardo Pontual
16	CREA il database	Emil Rowland
17	CREA TABELLA	Aidan, alex9311, Almir Vuk, Ares, CL., drunken_monkey, Dylan Vander Berg, Franck Dernoncourt, H. Pauwelyn, Jojodmo, KIRAN KUMAR MATAM, Matas Vaitkevicius, Prateek
18	DROP o DELETE Database	Abhilash R Vankayala, John Odom
19	ELIMINA	Batsu, Chip, CL., Dylan Vander Berg, fredden, Joel, KIRAN KUMAR MATAM, Phrancis, Umesh, xenodevil, Zoyd
20	Elimina in cascata	Stefan Steiger
21	Esempi di database e tabelle	Abhilash R Vankayala, Arulkumar, Athafoud, bignose, Bostjan, Brad Larson, Christian, CL., Dariusz, Dr. J. Testington, enrico.bacis, Florin Ghita, FlyingPiMonster, forsvarir, Franck Dernoncourt, hairboat, JavaHopper, Jaydles, Jon Ericson, Magisch, Matt, Mureinik, Mzzzzzz, Prateek, rdans, Shiva, tinlyx, Tot Zam, WesleyJohnson
22	Espressioni di tabella comuni	CL., Daniel, dd4711, fuzzy_logic, Gidil, Luis Lema, ninesided, Peter K, Phrancis, Sibeesh Venu
23	Filtra i risultati usando WHERE e HAVING	Arulkumar, Bostjan, CL., Community, Franck Dernoncourt, H. Pauwelyn, Jon Chan, Jon Ericson, juergen d, Matas Vaitkevicius, Mureinik, Phrancis, Tot Zam
24	Funzioni (aggregato)	ashja99, CL., Florin Ghita, Ian Kenney, Imran Ali Khan, Jon Chan, juergen d, KIRAN KUMAR MATAM, Mark Stewart, Maverick, Nathan, omini data, Peter K, Reboot, Tot Zam, William Ledbetter, winseybash, Алексей Неудачин
25	Funzioni (analitico)	CL., omini data
26	Funzioni (scalare / riga singola)	CL., Kewin Björk Nielsen, Mark Stewart
27	Funzioni della finestra	Arkh, beercohol, bhs, Gidil, Jerry Jeremiah, Mureinik, mustaccio

28	Funzioni di stringa	ɐlǝx, Allan S. Hansen, Arthur D, Arulkumar, Batsu, Chris, CL., Damon Smithies, Franck Dernoncourt, Golden Gate, hatchet, Imran Ali Khan, IncrediApp, Jaydip Jadhav, Jones Joseph, Kewin Björk Nielsen, Leigh Riffel, Matas Vaitkevicius, Mateusz Piotrowski, Neria Nachum, Phrancis, RamenChef, Robert Columbia, vmaroli, ypercube
29	Identifier	Andreas, CL.
30	indici	a1ex07, Almir Vuk, carlosb, CL., David Manheim, FlyingPiMonster, forsvarir, Franck Dernoncourt, Horaciux, Jenism, KIRAN KUMAR MATAM, mauris, Parado, Paulo Freitas , Ryan
31	INSERIRE	Ameya Deshpande, CL., Daniel Langemann, Dipesh Poudel, inquisitive_mind, KIRAN KUMAR MATAM, rajarshig, Tot Zam, zplizzi
32	Le transazioni	Amir Pourmand, CL., Daryl, John Odom
33	MERGE	Abhilash R Vankayala, CL., Kyle Hale, SQLFox, Zoyd
34	NULLO	Bart Schuijt, CL., dd4711, Devmati Wadikar, Phrancis, Saroj Sasmal, StanislavL, walid, ypercube
35	Numero di riga	CL., Phrancis, user1221533
36	Operatore LIKE	Abhilash R Vankayala, Aidan, ashja99, Bart Schuijt, CL., Cristian Abelleira, guiguiblitz, Harish Gyanani, hellyale, Jenism, Lohitha Palagiri, Mark Perera, Mr. Developer, Ojen, Phrancis, RamenChef, Redithion, Stefan Steiger, Tot Zam, Vikrant, vmaroli
37	Operatori AND & OR	guiguiblitz
38	ORDINATO DA	Andi Mohr, CL., Cristian Abelleira, Jaydles, mithra chintha, nazark, Özgür Öztürk, Parado, Phrancis, Wolfgang
39	Ordine di esecuzione	a1ex07, Gallus, Ryan Rockey, ypercube
40	Procedura di archiviazione	brichins, John Odom, Lamak, Ryan
41	PROVA A PRENDERE	Uberzen1
42	Pulisci codice in SQL	CL., Stivan
43	RAGGRUPPA PER	3N1GM4, Abe Miessler, Bostjan, Devmati Wadikar, Filipe Manuel, Frank, Gidil, Jaydles, juergen d, Nathaniel Ford, Peter

		Gordon, Simone - Ali One, WesleyJohnson, Zahiro Mor, Zoyd
44	Ricerca di duplicati su un sottoinsieme di colonne con dettagli	Darrel Lee, mnoronha
45	SALVO	LCIII
46	Schema di informazioni	Hack-R
47	SELEZIONARE	Abhilash R Vankayala, aholmes, Alok Singh, Amnon, Andrii Abramov, apomene, Arpit Solanki, Arulkumar, AstraSerg, Brent Oliver, Charlie West, Chris, Christian Sagmüller, Christos, CL., controller, dariru, Daryl, David Pine, David Spillett, day_dreamer , Dean Parker, DeepSpace, Dipesh Poudel, Dror, Durgpal Singh , Epodax, Eric VB, FH-Inway, Florin Ghita, FlyingPiMonster, Franck Dernoncourt, geeksal, George Bailey, Hari K M, HoangHieu, iliketocode, Imran Ali Khan, Inca, Jared Hooper, Jaydles, John Odom, John Slegers, Jojodmo, JonH, Kapep, KartikKannapur, Lankymart, Mark Iannucci, Mark Perera, Mark Wojciechowicz, Matas Vaitkevicius, Matt, Matt S, Mattew Whitt, Matthew Moisen, MegaTom, Mihai-Daniel Virna, Mureinik, mustaccio, mxmissile, Oded, Ojen, onedaywhen, Paul Bambury, penderi, Peter Gordon, Prateek, Praveen Tiwari, Přemysl Šťastný, Preuk, Racil Hilan, Robert Columbia, Ronnie Wang, Ryan, Saroj Sasmal, Shiva, SommerEngineering, sqluser, stark, sunkuet02, ThisIsImpossible, Timothy, user1336087, user1605665, waqasahmed, wintersolider, WMios, xQbert, Yury Fedorov, Zahiro Mor, zedfoxus
48	Sequenza	John Smith
49	Sinonimi	Daryl
50	SKIP TAKE (impaginazione)	CL., Karl Blacquiere, Matas Vaitkevicius, RamenChef
51	SPIEGARE e DESCRIVERE	Simulant
52	SQL CURSOR	Stefan Steiger
53	SQL Group per contro distinto	carlosb
54	SQL Injection	120196, CL., Clomp, Community, Epodax, Knickerless-Noggins, Stefan Steiger

55	subquery	CL., dasblinkenlight, KIRAN KUMAR MATAM, Nunie123, Phrancis, RamenChef, tinlyx
56	Tabella DROP	CL., Joel, KIRAN KUMAR MATAM, Stu
57	Tavolo di design	Darren Bartrup-Cook
58	Tipi di dati	bluefeet, Jared Hooper, John Odom, Jon Chan, JonMark Perry, Phrancis
59	trigger	Daryl, IncrediApp
60	TRONCARE	Abhilash R Vankayala, CL., Cristian Abelleira, DaImTo, Hynek Bernard, inquisitive_mind, KIRAN KUMAR MATAM, Paul Bambury, ss005
61	UNIONE / UNIONE TUTTI	Andrea, Athafoud, Daniel Langemann, Jason W, Jim, Joe Taras , KIRAN KUMAR MATAM, Lankymart, Mihai-Daniel Virna, sunkuet02
62	Viste materializzate	dmfay
63	Visualizzazioni	Amir978, CL., Florin Ghita
64	XML	Steven