

14-11-2022

STORED PROCEDURE

Le Stored Procedure sono delle **varianti procedurali** di *SQL* (*che è un linguaggio dichiarativo*). Essendo SQL è un linguaggio dichiarativo, le *Stored Procedures* rappresentano una sua **estensione procedurale** grazie ai suoi costrutti: `BEGIN`, `END`, `DECLARE`, `FOR`, `WHILE`, `LOOP`, `IF`, etc..

Per esempio le `SELECT` sono funzioni e non *istruzioni separate*.

Si possono anche definire variabili ecc..

Per la creazione, alcuni database managers impiegano linguaggi procedurali:

- *PL/pgSQL* di PostgreSQL
- *PL/SQL* di Oracle
- *SQL PL* di DB2
- *Transact-SQL* di SQL Server
- *MySql Stored Procedure* di MySQL,

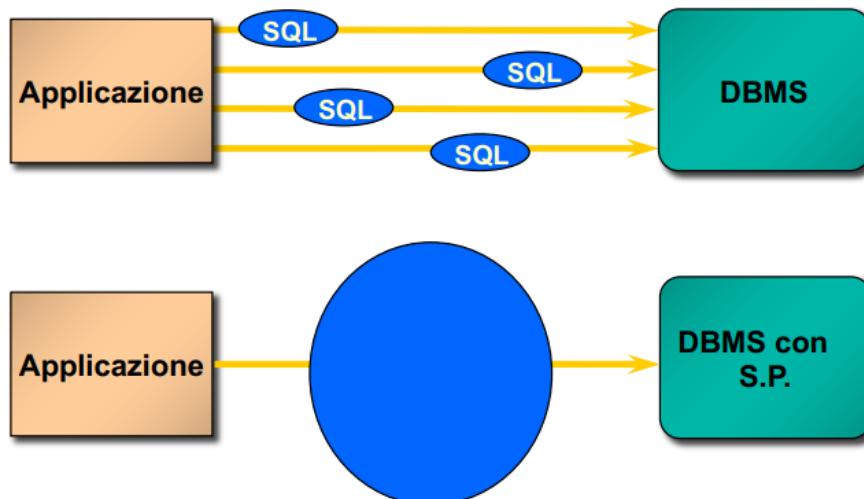
Infatti, in **SQL**, si hanno solo **query** ed esse non si influenzano a vicenda.

Le Stored Procedure sono suddivise in due gruppi di sotto-programmi dotati di **caratteristiche** differenti:

- **Procedure: Accetta parametri di input e non restituisce valori.** L'unico modo per farlo è attraverso una variabile di output passata in input per riferimento
 - cioè: sequenza di istruzioni richiamate che producono effetti sul database o in output. *Non vengono aggiunte nuove funzionalità*
- **Funzioni (UDF): Restituiscono un valore e accettano parametri di input ed output**
 - cioè: restituiscono in output qualcosa ma **non lo fanno da "sole"**. Sono **estensioni** delle istruzioni di *SQL* da aggiungere alle funzioni di default.

Caratteristiche Stored Procedure (SP)

- Le SP sono **riusabili** e **trasparenti** a qualsiasi applicazione dato che girano sul server. E' come se si definissero librerie per altri utenti.
- Migliorano l'**astrazione** (chi invoca la procedura può ignorarne i dettagli implementativi). **Non si deve sapere COME** vengono effettuate determinate funzioni ma **serve sapere il RISULTATO**.
- Accesso controllato alle tabelle in quanto solo gli utenti a cui è concesso l'uso della procedura potranno effettuare letture o modifiche alla tabella
- **Controllo centralizzato** su certi **vincoli d'integrità** non esprimibili nelle tabelle. Semplifica il processo di sicurezza



- **NON** si fanno più **single query** ma **basta una funzione** che lo fa al posto nostro.
- MySQL permette di definire **soltanto procedure scritte in linguaggio SQL**. Quindi non si ha libertà di scegliere il linguaggio da utilizzare ma tipicamente si possono usare anche in altri linguaggi mediante delle estensioni specifiche

Definizione di una procedura e di una funzione(UDF)

CREATE

[DEFINER = { user | CURRENT_USER }]

PROCEDURE nome_procedura ([parametri[,...]])

[caratteristiche ...] corpo_della_routine

#definizione UDF

CREATE [DEFINER = { user | CURRENT_USER }]

FUNCTION nome_procedura ([parametri[,...]])

RETURNS type

[caratteristiche ...] corpo_della_routine

dove **RETURNS** è valida per la sintassi delle funzioni (UDF). Esse, infatti, restituiscono un valore e per questo ad esso è associato il relativo tipo di dato.

Tipi di parametri di una S.P in MySQL

- **IN**: rappresenta gli **argomenti in ingresso** della routine; a questo parametro viene assegnato un valore quando viene invocata la S.P; il **parametro** utilizzato **non subirà** in seguito **modifiche**.
 - sono i parametri letti dalla funzione. (**parametro** passato ad una funzione **per valore in C++**)
- **OUT** (*non è standard*): è il parametro relativo ai **valori** che vengono **assegnati con l'uscita dalla procedura**; questi parametri diventano disponibili per gli utenti per eseguire ulteriori elaborazioni.
 - sono i parametri scritti in output dalla SP (è un **puntatore**, che è **READ-ONLY**)
- **INOUT** (*non è standard*): rappresenta una **combinazione** tra i due parametri precedenti.
 - sono parametri letti che si possono leggere e scrivere(**puntatore R/W** {Read-Write})

Delimitatore

In MySQL ogni istruzione finisce con punto e virgola ;

- Una funzione vuota con solo un ; è una funzione ma non va bene per MySQL
- Allora è possibile cambiare il delimitatore delle istruzioni, per esempio un //

Esempio:

```
mysql> delimiter //
```

```
mysql> CREATE PROCEDURE nome_procedura (p1 INT)
```

```
-> BEGIN
```

```
-> blocco istruzioni
```

```
-> END
```

```
-> //
```

```
mysql> delimiter ;
```

In particolare:

- Nel blocco istruzioni BEGIN/END metto i ; normali.
- Termino il blocco di istruzioni con il "nuovo delimitatore //"
- Dopodiché ripristino il delimitatore al ;

Questa sintassi si usa anche per i *trigger*.

- In pratica l'istruzione **DELIMITER** iniziale ha lo scopo di comunicare a MySQL che (fino a quando non verrà ordinato diversamente) il delimitatore utilizzato alla fine dell'istruzione non sarà più il "punto e virgola" ma proprio il delimitatore specificato.
- Per riportare il delimitatore al suo standard, ricordiamoci di dare il comando: ... `mysql> delimiter ;`

Variabili

Per dichiarare una variabile è possibile utilizzare:

```
DECLARE variable_name datatype(size) DEFAULT default_value;
```

- Quando una variabile viene creata essa avrà valore **NULL**.
- Per inizializzare una variabile è possibile utilizzare l'istruzione **DEFAULT**

Esempio:

1. `DECLARE total_sale INT DEFAULT 0`
2. `DECLARE x, y INT DEFAULT 0`

Per **modificare il valore di una variabile** è possibile farlo con un `SET` oppure con `INTO` (mediante un'istruzione `SQL`):

```
DECLARE total_count INT DEFAULT 0
SET total_count = 10;
```

```
DECLARE total_products INT DEFAULT 0
SELECT COUNT(*) INTO total_products
FROM products
```

Consigli

- Seguire una convenzione per i nomi. (quindi usare **nomi significativi**)
- Inizializzare variabili a **NOT NULL**.
- Inizializzare identificatori attraverso l'uso dell'operatore di assegnamento (SET) o con la parola chiave **DEFAULT**.
- Dichiarare al più un identificatore per linea.

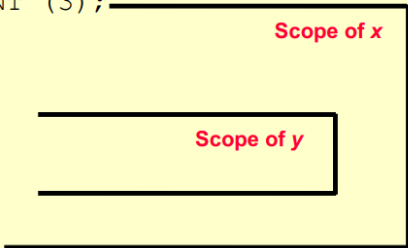
Scope delle variabili

Ogni variabile ha il suo proprio scope (visibilità).

- Se dichiariamo una variabile dentro una *S.P.*, questa sarà visibile (ed utilizzabile) fin quando non si conclude la procedura con il comando **END**.
- E' possibile dichiarare due o più variabili con lo **stesso nome in scope differenti**;
- La variabile potrà essere **usata all'interno del suo scope**.
- Una variabile preceduta dal simbolo **@** è una variabile di sessione (**GLOBALE**) e persisterà fino alla chiusura di ogni sessione.

Esempio

```
• ...
• DECLARE x INT (3);
• BEGIN
•   ...
•   DECLARE
•     y INT;
•   BEGIN
•     ...
•   END;
•   ...
• END;
```



```
mysql> CREATE PROCEDURE proc_ciao () SELECT 'Ciao Mondo' //
mysql> CALL proc_ciao ()//
+-----+
| Ciao Mondo |
+-----+
| Ciao Mondo |
+-----+
```

Uso del parametro IN

```
mysql> CREATE PROCEDURE proc_in (IN p INT(3)) SET @x = p//
```

IN solitamente è di default, quindi si potrebbe anche omettere!

```
mysql> CALL proc_in (14) //
```

```
mysql> SELECT @x//
```

```
+-----+  
| @x    |  
+-----+  
| 14    |  
+-----+
```

Uso del parametro OUT

- Il parametro **deve essere una variabile**. Come una *variabile per riferimento* che può essere solo scritta.

```
mysql> CREATE PROCEDURE proc_out (OUT p INT)  
-> SET p = -2 //
```

```
mysql> CALL proc_out (@y) //
```

```
mysql> SELECT @y//
```

```
+-----+  
| @y    |  
+-----+  
| -2    |  
+-----+
```

In questo caso `p` rappresenta il nome di un parametro di output che viene **passato come valore** alla variabile `y` introdotta nel momento in cui viene espresso il comando che invoca la procedura.

Nel corpo della routine il valore del parametro viene indicato come pari all'intero negativo -2, a questo punto spetta ad OUT segnalare a MySQL che il valore sarà associato tramite la procedura

Uso dei parametri IN e OUT

```
DELIMITER $$
CREATE PROCEDURE CountOrderByStatus
( IN orderStatus VARCHAR(25), OUT total INT)
BEGIN
    SELECT count(orderNumber) INTO total
    FROM orders
    WHERE status = orderStatus;
END$$
DELIMITER ;
```

```
mysql> CALL CountOrderByStatus('Shipped',@total);
mysql> SELECT @total AS total_shipped;
```

Uso del parametro INOUT

```
DELIMITER $$
CREATE PROCEDURE 'Capitalize'(INOUT str VARCHAR(1024))
BEGIN
    DECLARE i INT DEFAULT 1;
    DECLARE myc, pc CHAR(1);
    DECLARE outstr VARCHAR(1000) DEFAULT str;
    WHILE i <= CHAR_LENGTH(str) DO
        SET myc = SUBSTRING(str, i, 1);
        SET pc = CASE WHEN i = 1 THEN ' '
        ELSE SUBSTRING(str, i - 1, 1) END;
        IF pc IN (' ', '&', '!', '_', '?', ';',
        ':', '!', ',', '-', '/', '(', '.') THEN
            SET outstr = INSERT(outstr, i, 1, UPPER(myc));
        END IF;
        SET i = i + 1;
    END WHILE;
    SET str = outstr;
END$$
DELIMITER ;
```

- La procedura si può richiamare in questo modo:

```
mysql> SET @str = 'mysql stored procedure tutorial';
mysql> CALL Capitalize(@str);
mysql> SELECT @str;
```

Controlli condizionali

I controlli condizionali permettono di eseguire parti di codice secondo il valore di un'espressione logica (espressione che utilizza operatori logici e che può essere vera o falsa).

MySQL consente due dichiarazioni condizionali:

- **IF ... THEN ... ELSE ...**
- **CASE WHEN ... THEN ... ELSE ...**

```
IF expression THEN commands
    [ELSEIF expression THEN commands]
    [ELSE commands]
END IF;
```

```
IF expression THEN commands
END IF;
```

```
IF expression THEN commands
    ELSEIF expression THEN commands
    ELSE commands
END IF;
```


L'utilizzo del comando IF in alcuni casi riduce la leggibilità del codice (ad es. quando vengono scritti IF annidati o di seguito). In questi casi è consigliabile usare il comando **CASE**.

```
CASE
    WHEN expression THEN commands
    ...
    WHEN expression THEN commands
    ELSE commands
END CASE;
```

Le stored procedure di MySQL consentono la definizione di loop per consentire di processare comandi iterativamente.

I cicli consentiti sono:

- **WHILE... DO... ELSE ...**
- **REPEAT ... UNTIL ...**

REPEAT è come il do while: Esegue un'istruzione e poi esegue il loop.

```
WHILE espressione DO
    Istruzione
END WHILE
```

La prima cosa è valutare l'espressione: se è vera viene eseguita l'istruzione fino a che l'espressione non diventa falsa.

Esempio di ciclo infinito

```
loop_label: LOOP
    IF x > 10 THEN
        LEAVE loop_label;
    END IF;
    SET x = x + 1;
    IF (x mod 2) THEN
        ITERATE loop_label;
    ELSE
        SET str = CONCAT(str,x,',');
    END IF;
END LOOP;
```

dove:

- `LEAVE` = break di C++ e altri linguaggi -> **ferma** il loop;
- `ITERATE` = continue di C++ e altri linguaggi -> **salta** alla prossima iterazione;

Cursori

In MySQL non ci sono array solitamente.

Ci sono invece i **CURSORI**. Essi servono per iteratori di C o PHP o JAVA. Sono strumenti che hanno molte potenzialità rispetto a un array standard.

Proprietà dei cursori

- **READ-ONLY**: non è possibile modificare il contenuto di un cursore.
- **Non-scrollable**: è possibile **leggere** i dati nel cursore **solo in avanti** e **senza salti** in maniera sequenziale. (si può andare avanti ma non indietro)
- **Asensitive**: evitare di aggiornare le tabelle sulle quali sono stati aperti dei cursori: in questi casi è possibile ottenere dalla lettura dei cursori risultati sbagliati.
 - Se mentre uso un cursore aggiorni la tabella, il **cursore non si aggiorna**. Si evita di fare modifiche mentre si usa un cursore su di essa

```
DECLARE cursor_name CURSOR FOR SELECT_statement;
```

Dopo la dichiarazione di un cursore, esso va aperto con il comando `OPEN`. Prima di leggerne le righe, occorre aprire il cursore.

```
OPEN cursor_name;
```

Successivamente recuperiamo le righe con il comando `FETCH`.

```
FETCH cursor_name INTO variable list;
```

- Per chiudere e rilasciare la memoria occupata, usare il comando CLOSE
 - `CLOSE cursor_name;`

Comportamento del database quando il cursore non trova più record.

Si definisce:

```
DECLARE CONTINUE HANDLER FOR
    NOT FOUND SET no_more_products = 1;
```

e vuol dire che se non trova un record, deve eseguire quell'istruzione.

Esempio completo:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS CursorProc$$
CREATE PROCEDURE CursorProc()
BEGIN
    DECLARE no_more_products, quantity_in_stock INT
        DEFAULT 0;
    DECLARE prd_code VARCHAR(255);
    DECLARE cur_product CURSOR FOR
        SELECT productCode FROM products;
    DECLARE CONTINUE HANDLER FOR
        NOT FOUND SET no_more_products = 1;

    ...
```

```
    ...
OPEN cur_product;
REPEAT
    FETCH cur_product INTO prd_code;
    SELECT quantityInStock INTO quantity_in_stock
    FROM products
    WHERE productCode = prd_code;
UNTIL no_more_products = 1
END REPEAT;
CLOSE cur_product;

    ...
```

Window Function

Essa esegue delle operazioni **simili** a quelle delle **funzioni di aggregazione** quindi il principio è simile.

Nelle funzioni di aggregazione (`GROUP BY`) raggruppa una sola riga. La *Window Function* produce un risultato per ogni riga della query (riga corrente). Le righe sulla quale viene eseguita l'operazione si chiama **FINESTRA**.

Esempio 1:

Con raggruppamento

```
mysql> SELECT SUM(profit) AS total_profit
FROM sales;

+-----+
| total_profit |
+-----+
|          7535 |
+-----+

mysql> SELECT country, SUM(profit) AS country_profit
FROM sales
GROUP BY country
ORDER BY country;

+-----+-----+
| country | country_profit |
+-----+-----+
| Finland |          1610 |
| India   |          1350 |
| USA     |          4575 |
+-----+-----+
```

- "Data 1 riga, vai a prendere tutti gli elementi che sono nella finestra di quella riga, quella finestra com'è definita? partiziona per paese.

Esempio 1:

Con window function

```
mysql> SELECT
    year, country, product, profit,
    SUM(profit) OVER() AS total_profit,
    SUM(profit) OVER(PARTITION BY country) AS country_profit
FROM sales
ORDER BY country, year, product, profit;

+-----+-----+-----+-----+-----+-----+
| year | country | product | profit | total_profit | country_profit |
+-----+-----+-----+-----+-----+-----+
| 2000 | Finland | Computer | 1500 | 7535 | 1610 |
| 2000 | Finland | Phone   | 100  | 7535 | 1610 |
| 2001 | Finland | Phone   | 10   | 7535 | 1610 |
| 2000 | India   | Calculator | 75  | 7535 | 1350 |
| 2000 | India   | Calculator | 75  | 7535 | 1350 |
| 2000 | India   | Computer | 1200 | 7535 | 1350 |
| 2000 | USA     | Calculator | 75  | 7535 | 4575 |
| 2000 | USA     | Computer | 1500 | 7535 | 4575 |
| 2001 | USA     | Calculator | 50   | 7535 | 4575 |
| 2001 | USA     | Computer | 1200 | 7535 | 4575 |
| 2001 | USA     | Computer | 1500 | 7535 | 4575 |
| 2001 | USA     | TV       | 100  | 7535 | 4575 |
| 2001 | USA     | TV       | 150  | 7535 | 4575 |
+-----+-----+-----+-----+-----+-----+
```

dove:

- `SUM(profit) OVER()` è una finestra senza criteri = "prendi tutti i record e fai la somma su tutti i record, cioè somma globale di tutte le righe".
- `SUM(profit) OVER(PARTITION BY country)` -> "prendi tutte le righe del paese e fai la somma di tutte le righe di quel paese".

- Criteri con cui prendere le righe per fare la somma *QUINDI non si fa un raggruppamento*.

Caratteristiche Window Function

- La finestra è definita dall'inclusione della clausola `OVER` che specifica come partizionare le righe del risultato per processarle.
- Una clausola `OVER` **vuota** tratta l'intero insieme di righe come una singola partizione (producendo quindi la somma globale)
- Le window function sono permesse **SOLO** nella **target list della `SELECT`** e nella **`ORDER BY`**.
- Il partizionamento delle righe avviene **dopo l'esecuzione** di `FROM`, `WHERE`, `GROUP BY`, e `HAVING` e **prima** di `ORDER BY`, `LIMIT` e `DISTINCT`. (E' l'ultima operazione dopo la `HAVING`)
- Le funzioni di aggregazione possono anche essere usate come window function purchè sia presente la clausola `OVER`.

Funzioni delle Window

`CUME_DIST()`: Distribuzione cumulativa

`DENSE_RANK()`: Rango della riga corrente all'interno della finestra, senza gap.

`FIRST_VALUE(expr)`: valore di `expr` calcolato sulla prima riga della finestra

`LAST_VALUE(expr)`: valore di `expr` calcolato sull'ultima riga della finestra

`NTH_VALUE(expr, n)`: valore di `expr` calcolato sulla n-esima riga della finestra

`RANK()`: Rango della riga corrente all'interno della finestra, con gap.

`ROW_NUMBER()`: Numero della riga corrente all'interno della finestra

- **Somma cumulativa** = ogni elemento è somma di se stesso + il precedente.
- `dense_rank` e `rank()` = vedere rango elementi
 - **rango**: quanti elementi ho prima dell'elemento corrente. Elementi uguali hanno lo stesso rango
 - il **rango denso** è **incrementale**. Numero di elementi **univoci** secondo l'ordinamento della finestra.

Il rango denso non ha questi salti di valori. Non tiene in conto i duplicato.
- `FIRST_VALUE` prende la **prima riga** di una finestra
- `ROW_NUMBER` numero di riga all'interno della finestra

Esempio:

```
mysql> SELECT
      val,
      ROW_NUMBER() OVER w AS 'row_number',
      RANK() OVER w AS 'rank',
      DENSE_RANK() OVER w AS 'dense_rank'
FROM numbers
WINDOW w AS (ORDER BY val);
```

val	row_number	rank	dense_rank
1	1	1	1
1	2	1	1
2	3	3	2
3	4	4	3
3	5	4	3
3	6	4	3
4	7	7	4
4	8	7	4
5	9	9	5

Definizione della finestra

La finestra si può definire direttamente sulla clausola `OVER` :

```
``SELECT ..., FUNZIONE() OVER(...), ... FROM ...
```

Oppure in una clausola `WINDOW` separata:

```
SELECT ..., FUNZIONE() OVER nome_finestra, ...
FROM ...
WINDOW nome1 AS (...)[, nome2 AS (...), ...]
```

- Se non voglio ripetere la definizione della finestra uso **WINDOW** che permette di assegnare un nome alla definizione della finestra.
- Nella target list della `SELECT` uso quindi il nome della finestra piuttosto che ripetere più volte la definizione della finestra.

Come si partizionano le righe nella finestra

[PARTITION BY field] [ORDER BY field] [frame_extent]

- `frame_extent` *"quanti elementi voglio includere nella finestra".*
- `frame_start`, *prendi tutte le righe da `frame_start` fino alla riga corrente.*

Questa definizione equivale a dire → *"quante righe devono essere incluse all'interno della finestra".*

- `UNBOUNDED PRECEDING` sono tutte le righe precedenti a quella riga.

Esempi:

PARTITION BY subject ORDER BY time
ROWS UNBOUNDED PRECEDING

PARTITION BY subject ORDER BY time
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING