

Stored Procedures

Antonella Poggi, Claudio Corona

Dipartimento di informatica e Sistemistica

SAPIENZA Università di Roma

Progetto di Applicazioni Software

Anno accademico 2008-2009

Questi lucidi sono stati prodotti sulla base del materiale preparato per il corso di Progetto di Basi di Dati da D. Lembo e M. Ruzzi.

Stored Procedures / 1

- Spesso è importante eseguire alcune parti della logica dell'applicazione direttamente nello spazio del processo del sistema di basi di dati
- Questo viene reso possibile “estendendo” SQL (di natura **dichiarativo**) con delle caratteristiche **procedurali** (ad es., PL/SQL di Oracle)
- Le **SQL-invoked routines**, comunemente chiamate **stored procedures** (quando si tratta di funzioni, viene anche usato anche la denominazione **User Defined Functions**) sono l’“estensione procedurale” di SQL di maggiore efficacia che viene messa a disposizione dai DBMS

Stored Procedures / 2

- Inizialmente, le stored procedure erano semplici programmi contenenti un singolo statement SQL, che risiedevano sul DBMS e che erano invocabili dai client
- SQL/PSM (**Persistent Stored Modules**) è il linguaggio standardizzato di SQL 99 per la definizione di stored procedure
- Secondo lo standard, i “moduli persistenti” (**funzioni o procedure**) possono essere definiti completamente in SQL, con comandi SQL/PSM, o utilizzando un diverso linguaggio di programmazione
- Molti DBMS forniscono estensioni proprietarie analoghe a SQL/PSM

Stored Procedures / 3

- Indipendentemente da come sono definite, la principale peculiarità delle stored procedure è quella di poter essere **invocate tramite comandi SQL** oppure direttamente **dalle applicazioni** che interagiscono con il database (as es., tramite **JDBC**)
- L'uso di stored procedure porta diversi **vantaggi**
 - Migliorano lo sviluppo (costituiscono di fatto una libreria di funzioni che favorisce il riuso del codice)
 - Semplificano la comunicazione client-server
 - Favoriscono il rispetto dell'integrità dei dati
 - Agevolano la definizione di politiche di controllo dell'accesso (aumentando così la sicurezza)

Stored Procedures / 4

- Migliorano quindi (in una certa misura)
 - estendibilità (del linguaggio SQL)
 - l'interfacciamento esplicito fra applicazione e DB
 - riusabilità
 - astrazione (chi invoca la procedura ne ignora i dettagli implementativi)

Stored Procedures / 5

- E possono essere utili:
 - per implementare dei vincoli di integrità in maniera procedurale, ovvero per garantirne il soddisfacimento (lo vedremo più avanti)
 - quando diversi client del DBMS (eventualmente scritti in linguaggi diversi ed eseguiti su diverse piattaforme) necessitano delle stesse operazioni sul DB (che risultano opportunamente parametrizzate)
 - per questioni di sicurezza: l'utente non vede la struttura del database, ma può interagire con esso mediante le stored procedure (gestione dell'accesso al DB)

Stored Procedures / 6

Non si deve però cedere alla tentazione di **inglobare tutta la logica dell'applicazione nel DBMS** tramite stored procedure: per applicazioni complesse questa strada non è percorribile!

IMPORTANTE: le stored procedure non devono mai implementare funzionalità che non hanno direttamente a che fare con i dati

Stored Procedures / 7

Le stored procedure possono essere invocate

- Tramite l'interprete dei comandi (ad es., `mysql` per MySQL, `MSQL` per Oracle)
- Da applicazione (ad es., tramite ODBC, o JDBC)
- Da strumenti di supporto del DBMS
- Da un evento specifico (mediante **trigger**)

Definire Stored Procedures

- Le stored procedure possono avere **parametri** di tre tipi differenti:
 - IN: sono argomenti di ingresso per la procedura; al parametro deve essere assegnato un valore al momento dell'invocazione della procedura che successivamente non viene cambiato;
 - OUT: rappresenta l'uscita della procedura che assegna loro i valori che l'utente può utilizzare
 - IN OUT: combinano le caratteristiche precedenti
- Consideriamo due semplici casi di stored procedure scritte in SQL ed in Java

Esempio

Stored Procedure in SQL

```
CREATE PROCEDURE AggiungiInventario(  
    IN isbn CHAR(9), IN aggiungiQta INTEGER)  
UPDATE Libri SET qta_in_stock = qta_in_stock  
                + aggiungiQta  
WHERE libro_isbn=isbn
```

Stored Procedure in Java

```
CREATE PROCEDURE AggiungiInventarioJava(  
    IN isbn CHAR(9), IN aggiungiQta INTEGER)  
LANGUAGE Java  
EXTERNAL NAME  
    'file:///c:/procedureInterne/aggiungiInv.jar'
```

Invocare Stored Procedures

- Le stored procedure possono essere invocate con SQL interattivo tramite il comando `CALL`

```
CALL AggiungiInventario('isbn_1528', 35)
```

- Vedremo pi avanti come possono essere invocate da un'applicazione Java

Funzioni (User defined Functions)

- Le funzioni sono un particolare tipo di stored procedure
- Secondo lo standard, sono definite tramite il comando `CREATE FUNCTION`, con una sintassi analoga al comando `CREATE PROCEDURE`
- Le funzioni però restituiscono un valore, tramite la clausola `RETURNS <tipoDatoSQL>`
- Secondo lo standard, non possono ricevere parametri di tipo `OUT` e `IN OUT`
- Possono essere usate in espressioni SQL

Stored Procedures in MySQL

RIFERIMENTO: MySQL 5.0 manual (capitolo 17)

- MySQL consente esclusivamente la definizione di **stored procedure scritte in SQL**.
- Una stored procedure può essere invocata:
 1. direttamente tramite l'interprete dei comandi per MySQL
 2. con JDBC (o ODBC)
 3. da un evento specifico (mediante **trigger**, che vedremo più avanti)

Definire Stored Procedure in MySQL / 1

Nota: In maiuscolo si indicano i comandi SQL, in minuscolo le categorie sintattiche che vanno opportunamente istanziate, fra parentesi [] si indicano comandi opzionali, mentre {A | B} indica la possibilità di scegliere fra A o B.

```
CREATE
```

```
  [DEFINER = { user | CURRENT_USER }]
  PROCEDURE sp_name ([proc_parameter[,...]])
  [characteristic ...] routine_body
```

```
CREATE
```

```
  [DEFINER = { user | CURRENT_USER }]
  FUNCTION sp_name ([func_parameter[,...]])
  RETURNS type
  [characteristic ...] routine_body
```

Definire Stored Procedure in MySQL / 2

- l'opzione `DEFINER` (non obbligatoria) permette di specificare l'utente creatore e perciò "proprietario" della stored procedure. La scelta di default è l'utente corrente.
- una procedura o una funzione possono essere caratterizzate da una lista di parametri:

```
proc_parameter:  
    [ IN | OUT | INOUT ] param_name type
```

```
func_parameter:  
    param_name type
```

Definire Stored Procedure in MySQL / 3

- una procedura o una funzione possono essere caratterizzate da diverse opzioni che sono:

characteristic:

```
LANGUAGE SQL | [NOT] DETERMINISTIC
| {CONTAINS SQL | NO SQL | READS SQL DATA
  | MODIFIES SQL DATA}
| SQL SECURITY { DEFINER | INVOKER }
| COMMENT 'string'
```

Definire Stored Procedure in MySQL / 4

- `LANGUAGE SQL` indica che la procedura è scritta nel linguaggio SQL; ad oggi MySQL non permette di usare altri linguaggi (quindi SQL è il default).
- `DETERMINISTIC` indica se, dato un certo input, una procedura restituisce sempre lo stesso risultato; di default, una procedura è definita come `NOT DETERMINISTIC`.
- `SQL SECURITY` indica se la procedura deve essere eseguita usando i privilegi dell'utente che definisce o invoca la procedura.

Definire Stored Procedure in MySQL / 5

- `CONTAINS SQL` indica che la procedura contiene comandi SQL (e.g. `SET`, `RELEASE-LOCK`) , ma non contiene istruzioni che scrivono o leggono dati sul/dal db – questo è il default
- `NO SQL` indica che la procedura non contiene comandi SQL
- `READS SQL DATA` indica che la procedura contiene comandi SQL che leggono dati (e.g. `SELECT`)
- `MODIFIES SQL DATA` indica che la procedura contiene comandi SQL che scrivono dati (e.g. `INSERT`)

Definire Stored Procedure in MySQL / 6

`routine_body` consiste di un valido statement di procedura SQL, ovvero

- un semplice statement SQL, come `SELECT` o `INSERT`
- un compound statement:

```
[begin_label:] BEGIN
    [statement_list]
END [end_label]
```

Un compound statement può contenere declaration, loop e altri tipi di statement per strutture di controllo. Più avanti li vedremo uno per uno.

Modificare Stored Procedure in MySQL

```
ALTER {PROCEDURE | FUNCTION} sp_name [characteristic ...]
```

characteristic:

```
{ CONTAINS SQL | NO SQL | READS SQL DATA  
  | MODIFIES SQL DATA}  
| SQL SECURITY { DEFINER | INVOKER }  
| COMMENT 'string'
```

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

Invocare Stored Procedure in MySQL

- le stored procedure vengono invocate mediante il comando `CALL spname([parameter[, ...]])`
 - di default, una stored procedure in MySQL è invocata sul db in uso (preventivamente invocato tramite un comando `use db-name`)
 - per invocare procedure memorizzate in altri database è sufficiente qualificare il nome della procedura col nome del database `db-name`
- viene invocato implicitamente il comando `use db-name`, e appena l'esecuzione della stored procedure è terminata il comando `use` viene annullato

Stored Procedure in MySQL: un esempio / 1

```
CREATE PROCEDURE ex_proc (OUT numero_utenti VARCHAR(25),
                          INOUT incr_param INTEGER)
BEGIN
    # Imposto il valore del parametro OUT
    SELECT count(*) INTO numero_utenti from mysql.user;
    # incremento il valore del parametro INOUT
    SET incr_param = incr_param + 1;
END;
```

Cosa fa questa procedura?

Stored Procedure in MySQL: un esempio / 2

```
mysql> SET @increment = 10;
mysql> CALL ex_proc(@users, @increment);
mysql> SELECT @users, @increment;
+-----+-----+
| @version | @increment |
+-----+-----+
| #utenti  | 11         |
+-----+-----+
```

Gestione del delimitatore di fine istruzione

Per poter definire stored procedure da riga di comando, occorre cambiare il delimitatore di fine istruzione

```
mysql> delimiter //
```

```
mysql> CREATE PROCEDURE procedura(p1 INTEGER)
-> BEGIN
->     istruzione;
->     istruzione;
-> END
-> //
```

```
mysql> delimiter ;
```

Compound Statement

Gli statement che appaiono all'interno di un compound statement possono essere di vario tipo:

1. normali statement SQL
2. DECLARE statement
 - (a) per variabili
 - (b) per condizioni e handler
 - (c) per cursori
3. statement di apertura/chiusura/fetch di un cursore
4. flow control statement
 - IF, CASE, REPEAT, WHILE, LOOP, LEAVE, ITERATE

Usò delle variabili nelle stored procedure / 1

E' possibile dichiarare delle variabili all'interno delle stored procedure

```
DECLARE var_name[,...] type [DEFAULT value]
```

Le variabili così dichiarate sono "locali" alla procedura. Per impostare il valore di una variabile si utilizza il comando SET all'interno della procedura

```
SET var_name = expr [, var_name = expr] ...
```

N.B.: il comando SET può essere usato anche esternamente alle procedure, o invocato da JDBC.

Uso delle variabili nelle stored procedure / 2

Per prelevare il risultato di una query ed inserirlo in una variabile si usa il comando `SELECT INTO`

```
SELECT col_name[,...] INTO var_name[,...] table_expr
```

N.B.: i nomi delle variabili devono essere diversi da quelli delle colonne della tabella. Usando lo stesso nome, MySQL lo interpretera' sempre come il riferimento alla variabile.

Uso delle variabili: esempio

```
CREATE PROCEDURE sp1 (xname VARCHAR(5))
  BEGIN
    DECLARE newname VARCHAR(5) DEFAULT 'bob';
    DECLARE xid INTEGER;

    SELECT xname,id INTO newname,xid
      FROM table1 WHERE name = xname;
    SELECT newname;
  END;
```

Condizioni

Alcune condizioni possono richiedere uno speciale “trattamento”:

- errore che occorre
- stato del flusso di controllo interno alla procedura

Può pertanto essere utile assegnare un nome ad una condizione.

```
DECLARE condition_name CONDITION FOR condition_value
```

```
condition_value:
```

```
    SQLSTATE [VALUE] sqlstate_value  
    | mysql_error_code
```

Handlers

Gli handlers permettono di definire il “trattamento” da eseguire a fronte del realizzarsi di una (o più condizioni).

```
DECLARE handler_type HANDLER FOR
    condition_value[,...] statement
handler_type:
    CONTINUE
    | EXIT
    | UNDO
condition_value:
    SQLSTATE [VALUE] sqlstate_value
    | condition_name | SQLWARNING
    | NOT FOUND | SQLEXCEPTION
    | mysql_error_code
```

Uso dei cursori

Un cursore e' un particolare tipo di variabile che permette di gestire le tuple restituite da una query.

Dichiarazione di un cursore:

```
DECLARE cursor_name CURSOR FOR select_statement
```

Apertura del cursore:

```
OPEN cursor_name
```

Prelievo delle tuple successive:

```
FETCH cursor_name INTO var_name [, var_name] ...
```

Chiusura del cursore:

```
CLOSE cursor_name
```

DECLARE statement: ordine di scrittura

- i `DECLARE` statement si possono trovare solo dentro un compound statement, e devono essere indicati prima di qualsiasi altro statement all'interno del compound statement
- i cursori devono essere dichiarati prima degli handler
- le variabili e le condizioni devono essere dichiarate prima di cursori e handler

Uso dei cursori: esempio

```
CREATE PROCEDURE curdemo()  
BEGIN  
    DECLARE done INT DEFAULT 0;  
    DECLARE a CHAR(16);  
    DECLARE b,c INT;  
    DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;  
    DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;  
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;  
  
    OPEN cur1;  
    OPEN cur2;  
  
    REPEAT
```

```
    FETCH cur1 INTO a, b;
    FETCH cur2 INTO c;
    IF NOT done THEN
        IF b < c THEN
            INSERT INTO test.t3 VALUES (a,b);
        ELSE
            INSERT INTO test.t3 VALUES (a,c);
        END IF;
    END IF;
UNTIL done END REPEAT;

CLOSE cur1;
CLOSE cur2;
END
```

Controllo del flusso: IF-THEN-ELSE

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list] ...
    [ELSE statement_list]
END IF
```

Controllo del flusso: CASE / 1

Ci sono due modi di utilizzare il costrutto CASE.

Nel modo classico, mediante un valore:

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

Controllo del flusso: CASE / 2

Oppure mediante una serie di condizioni:

```
CASE
```

```
    WHEN search_condition THEN statement_list  
    [WHEN search_condition THEN statement_list] ...  
    [ELSE statement_list]
```

```
END CASE
```

(questo secondo metodo e' utile per rimpiazzare lunghe sequenze di IF-THEN-ELSE)

Controllo del flusso: LOOP, LEAVE, ITERATE

Sintassi del comando LOOP:

```
[begin_label:] LOOP
    statement_list
END LOOP [end_label]
```

E' possibile dichiarare delle etichette (label) per identificare una certa istruzione di LOOP

Per eseguire nuovamente il blocco di istruzioni del ciclo LOOP si utilizza il comando ITERATE

```
ITERATE label
```

Per uscire del ciclo, si usa il comando LEAVE

```
LEAVE label
```

Esempio del controllo di flusso con LOOP

Esempio di utilizzo del comando LOOP:

```
CREATE PROCEDURE iterazione(p1 INT)
BEGIN
    label1: LOOP
        SET p1 = p1 + 1;
        IF p1 < 10 THEN ITERATE label1; END IF;
        LEAVE label1;
    END LOOP label1;
    SET @x = p1;
END
```

Controllo del flusso: REPEAT e WHILE

Sintassi del comando REPEAT

```
[begin_label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [end_label]
```

Sintassi del comando WHILE

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

Esempio del controllo di flusso con REPEAT

```
mysql> delimiter //
```

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

```
  -> BEGIN
```

```
  ->   SET @x = 0;
```

```
  ->   REPEAT SET @x = @x + 1; UNTIL @x > p1 END REPEAT;
```

```
  -> END
```

```
  -> //
```

```
Query OK, 0 rows affected (0.00 sec)
```

Esempio del controllo di flusso con WHILE

```
CREATE PROCEDURE ciclo-while()  
BEGIN  
    DECLARE v1 INT DEFAULT 5;  
  
    WHILE v1 > 0 DO  
        ...  
        SET v1 = v1 - 1;  
    END WHILE;  
END
```