Bazy danych

9. Procedury składowane i kursory

Wyzwalacze

P. F. Góra

http://th-www.if.uj.edu.pl/zfs/gora/

2018

Procedury składowane (*stored procedures*)

Procedury składowane stanowią część schematu bazy danych. Stosuje się je do wykonowania powtarzających się, logicznie takich samych operacji na (bazie) danych, nie wymagających ingerencji ze strony użytkownika.

Zalety używania procedur składowanych:

- Różne aplikacje korzystające z tej samej bazy danych korzystają z tej samej procedury — mniejsze ryzyko popełnienia błędu.
- Zwiększone bezpieczeństwo: Nie udostępniamy użytkownikom bezpośrednio tabel (hacker mógłby dobrać się do tabeli z hasłami!), zezwalamy *tylko* na wykonywanie predefiniowanych procedur.
- Mniejsze koszty uruchomienia i konserwacji.
- Automatyzacja obsługi błędów
- Zmniejszenie kosztu przesyłu danych.

Najprostszy przykład

```
mysql> DELIMITER//
```

Przed zdefiniowaniem procedury należy “redefiniować średnik”, żeby średniki w ciele procedury nie były interpretowane jako koniec zapytania definiującego procedurę.

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE proc01 ()
    -> /* To jest komentarz */
    -> SELECT * FROM arabic;   /* "arabic" jest nazwą tabeli */
    -> //
Query OK, 0 rows affected (0.01 sec)

mysql> DELIMITER ;
```

Wywołanie procedury — instrukcja `CALL`

```
mysql> CALL proc01 ();
+----+-----------+------+
| i  | b         | x    |
+----+-----------+------+
| 1  | jeden     | X    |
| 2  | dwa       | X    |
| 4  | cztery    | X    |
| 3  | trzy      | X    |
| 5  | pięć      | X    |
| 6  | sześć     | X    |
| 7  | siedem    | X    |
| 8  | osiem     | X    |
| 9  | dziewięć  | X    |
| 10 | dziesięć  | X    |
| 12 | dwanaście | X    |
+----+-----------+------+
11 rows in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)
```

Klauzule definicyjne

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE proc02()
    -> LANGUAGE SQL
    -> NOT DETERMINISTIC
    -> SQL SECURITY INVOKER
    -> COMMENT 'Przykład'
    -> SELECT RAND();
    -> //
Query OK, 0 rows affected (1.14 sec)
mysql> DELIMITER ;
mysql> CALL proc02 ();
+------------------+
| RAND()           |
+------------------+
| 0.16568405103468 |
+------------------+
1 row in set (0.02 sec)
Query OK, 0 rows affected (0.02 sec)
```

`LANGUAGE SQL` wymagane ze względu na kompatybilność

`NOT DETERMINISTIC` bo przy takich samych parametrach może dać różne wyniki
`DETERMINISTIC` jeśli parametry wywołania jednoznacznie determinują wynik

`SQL SECURITY INVOKER` przy wywołaniu sprawdzaj przywileje wywołującego
`SQL SECURITY DEFINER` przy wywołaniu sprawdzaj przywileje użytkownika, który stworzył procedurę

Instrukcje złożone. Parametry wywołania.

```
mysql> CREATE PROCEDURE proc03 (IN i INT)
    -> LANGUAGE SQL
    -> DETERMINISTIC
    -> BEGIN
    ->   SELECT i + 2;
    ->   SELECT i - 4;
    -> END; //
Query OK, 0 rows affected (1.71 sec)
mysql> CALL proc03 (2)//
+-------+
| i + 2 |
+-------+
| 4     |
+-------+
1 row in set (0.10 sec)
+-------+
| i - 4 |
+-------+
| -2    |
+-------+
1 row in set (0.10 sec)
Query OK, 0 rows affected (0.10 sec)
```

```
mysql> CREATE PROCEDURE proc04 (INOUT j FLOAT)
    -> LANGUAGE SQL
    -> DETERMINISTIC
    -> BEGIN
    ->   DECLARE z FLOAT;  /* zmienna lokalna */
    ->   SET z = SIN(j);
    ->   SELECT z AS 'zet';
    ->   SET j = j*z;
    -> END; //
Query OK, 0 rows affected (0.00 sec)
```

Zmienne lokalne mają zakres ograniczoney do swojego bloku `BEGIN`–`END`.

```
mysql> SET @x = 2.0//
Query OK, 0 rows affected (0.00 sec)

mysql> CALL proc04 (@x) //
+----------+
| zet      |
+----------+
| 0.909297 |
+----------+
1 row in set (1.12 sec)

Query OK, 0 rows affected (1.13 sec)

mysql> SELECT @x //
+-----------------+
| @x              |
+-----------------+
| 1.8185948133469 |
+-----------------+
1 row in set (0.00 sec)
```

Procedury mogą uzyskać dostęp do tabel

```
mysql> CREATE PROCEDURE proc05 (IN z CHAR(1))
    -> UPDATE arabic SET x=z WHERE MOD(i,2)=0 LIMIT 4;//
Query OK, 0 rows affected (1.12 sec)
mysql> CALL proc05 ('P') //
Query OK, 4 rows affected (0.08 sec)
mysql> SELECT * FROM arabic //
+----+-----------+------+
| i  | b         | x    |
+----+-----------+------+
| 1  | jeden     | X    |
| 2  | dwa       | P    |
| 4  | cztery    | P    |
| 3  | trzy      | X    |
| 5  | pięć      | X    |
| 6  | sześć     | P    |
| 7  | siedem    | X    |
| 8  | osiem     | P    |
| 9  | dziewięć  | X    |
| 10 | dziesięć  | X    |
| 12 | dwanaście | X    |
+----+-----------+------+
11 rows in set (0.00 sec)
```

Czego procedurom *nie wolno* robić?

Procedury nie mogą zmieniać innych procedur. W ciele procedury w MySQL nielegalne są instrukcje `CREATE | ALTER | DROP PROCEDURE`, `CREATE | ALTER | DROP FUNCTION`, `CREATE | ALTER | DROP TRIGGER`.

W MySQL wewnątrz procedury nielegalna jest instrukcja `USE`, można jednak odwoływać się do tabel innej bazy danych niż baza bieżąca (oczywiście o ile mamy do tego uprawnienia).

Procedury mogą natomiast tworzyć, usuwać i modyfikować definicje tabel, widoków i baz danych.

Zakres zmiennych

```
mysql> CREATE PROCEDURE proc06 ()
    -> BEGIN
    ->   DECLARE napis CHAR(4) DEFAULT 'zewn';
    ->   BEGIN
    ->     DECLARE napis CHAR(4) DEFAULT 'wewn';
    ->     SELECT napis;
    ->   END;
    ->   SELECT napis;
    ->   SET napis = 'pqrs';
    ->   SELECT napis;
    -> END;//
Query OK, 0 rows affected (0.62 sec)
```

```
mysql> CALL proc06 () //
+-------+
| napis |
+-------+
| wewn  |
+-------+
1 row in set (0.00 sec)

+-------+
| napis |
+-------+
| zewn  |
+-------+
1 row in set (0.00 sec)

+-------+
| napis |
+-------+
| pqrs  |
+-------+
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.16 sec)
```

Instrukcja warunkowa

```
mysql> CREATE PROCEDURE proc07 (IN j INT)
    -> BEGIN
    ->   DECLARE m INT;
    ->   SET m = (SELECT MAX(i) FROM arabic);
    ->   IF j > m THEN
    ->      INSERT INTO arabic (i,b) VALUES (j,'tekst');
    ->   ELSE
    ->      UPDATE arabic SET i = i + m WHERE i = j;
    ->   END IF;
    -> END;//
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CALL proc07 (6) //
Query OK, 1 row affected (1.75 sec)
mysql> SELECT * FROM arabic //
+----+-----------+------+
| i  | b         | x    |
+----+-----------+------+
| 1  | jeden     | X    |
| 2  | dwa       | P    |
| 4  | cztery    | P    |
| 3  | trzy      | X    |
| 5  | pięć      | X    |
| 18 | sześć     | P    |
| 7  | siedem    | X    |
| 8  | osiem     | P    |
| 9  | dziewięć  | X    |
| 10 | dziesięć  | X    |
| 12 | dwanaście | X    |
+----+-----------+------+
11 rows in set (0.02 sec)
```

```
mysql> CALL proc07 (19) //
Query OK, 1 row affected (0.13 sec)
mysql> SELECT * FROM arabic //
+----+-----------+------+
| i  | b         | x    |
+----+-----------+------+
| 1  | jeden     | X    |
| 2  | dwa       | P    |
| 4  | cztery    | P    |
| 3  | trzy      | X    |
| 5  | pięć      | X    |
| 18 | sześć     | P    |
| 7  | siedem    | X    |
| 8  | osiem     | P    |
| 9  | dziewięć  | X    |
| 10 | dziesięć  | X    |
| 12 | dwanaście | X    |
| 19 | tekst     | X    |
+----+-----------+------+
12 rows in set (0.00 sec)
```

Instrukcja CASE

```
mysql> CREATE PROCEDURE proc08 (IN j INT)
    -> CASE j
    ->   WHEN 0 THEN SELECT j AS 'wynik';
    ->   WHEN 1 THEN SELECT 5*j AS 'wynik';
    ->   ELSE SELECT 10*j AS 'wynik';
    -> END CASE; //
Query OK, 0 rows affected (0.00 sec)
mysql> CALL proc08 (0) //
+-------+
| wynik |
+-------+
| 0     |
+-------+
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
mysql> CALL proc08 (2) //
+-------+
| wynik |
+-------+
| 20    |
+-------+
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.01 sec)
```

Pętla `WHILE`

```
mysql> CREATE PROCEDURE proc09 ()
    -> BEGIN
    ->   DECLARE v INT;
    ->   SET v = 1;
    ->   WHILE v < 3 DO
    ->     SELECT b FROM arabic
    ->       WHERE i = v;
    ->     SET v = v + 1;
    ->   END WHILE;
    -> END; //
Query OK, 0 rows affected (1.27 sec)
```

```
mysql> CALL proc09 () //
+-------+
| b     |
+-------+
| jeden |
+-------+
1 row in set (0.45 sec)

+------+
| b    |
+------+
| dwa  |
+------+
1 row in set (0.47 sec)

Query OK, 0 rows affected (0.47 sec)
```

Pętla REPEAT

```
mysql> CREATE PROCEDURE proc10 (OUT s INT)
    -> BEGIN
    ->   DECLARE j INT DEFAULT 0;
    ->   SET s = 1;
    ->   REPEAT
    ->     SET j = j + 1;
    ->     SET s = s * j;
    ->     UNTIL j > 5
    ->   END REPEAT;
    -> END; //
Query OK, 0 rows affected (0.97 sec)

mysql> CALL proc10(@s) //
Query OK, 0 rows affected (0.08 sec)

mysql> SELECT @s //
+------+
| @s   |
+------+
| 720  |
+------+
1 row in set (0.00 sec)
```

Pętla LOOP, instrukcja LEAVE i etykiety

```
mysql> CREATE PROCEDURE proc11 (IN k INT)
    -> BEGIN
    ->   DECLARE s, v INT DEFAULT 1;
    ->   etykieta: LOOP
    ->     SET s = s*v;
    ->     SET v = v+1;
    ->     IF v > k THEN LEAVE etykieta; END IF;
    ->   END LOOP;
    ->   SELECT k, s;
    -> END; //
Query OK, 0 rows affected (0.94 sec)

mysql> CALL proc11 (6) //
+------+------+
| k    | s    |
+------+------+
| 6    | 720  |
+------+------+
1 row in set (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
```

Etykiety można stawiać przed BEGIN, WHILE, REPEAT i LOOP.

Instrukcja `ITERATE`

`ITERATE` — zignoruj resztę ciała pętli i przejdź do następnej iteracji.

```
mysql> CREATE PROCEDURE proc12 ()
    -> BEGIN
    ->   DECLARE j INT DEFAULT 1;
    ->   ett: REPEAT
    ->     SET j = j + 1;
    ->     IF j = 4  THEN ITERATE ett; END IF;
    ->     SELECT i, b FROM arabic WHERE i=j;
    ->     UNTIL j = 6
    ->   END REPEAT ett;
    -> END; //
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> CALL proc12 () //
+---+------+
| i | b    |
+---+------+
| 2 | dwa  |
+---+------+
1 row in set (0.00 sec)

+---+------+
| i | b    |
+---+------+
| 3 | trzy |
+---+------+
1 row in set (0.00 sec)

+---+------+
| i | b    |
+---+------+
| 5 | pięć |
+---+------+
1 row in set (0.00 sec)

Empty set (0.00 sec)

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

Obsługa błędów (*Error handling*)

“Naturalnym” zastosowanie procedur składowanych jest obsługa błędów, jakie mogą pojawić się przy pracy z bazą danych, na przykład na skutek naruszenia więzów. Rozpatrzmy przykład:

```
mysql> CREATE TABLE tabela
    -> (K SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
    ->  W INT UNSIGNED);
Query OK, 0 rows affected (1.14 sec)

mysql> INSERT INTO tabela VALUES (1,10);
Query OK, 1 row affected (0.12 sec)

mysql> INSERT INTO tabela VALUES (2,-1);
ERROR 1264 (22003): Out of range value adjusted for column 'W' at row 1
mysql> INSERT INTO tabela VALUES (1,5);
ERROR 1062 (23000): Duplicate entry '1' for key 1
```

Pojawienie się błędów przy nie-interaktywnym wprowadzaniu danych może spowodować problemy, których chcemy uniknąć. Zarazem chcemy wiedzieć jakie błędy się pojawiły.

```
mysql> CREATE TABLE log_bledow (B VARCHAR(80)) CHARSET cp1250;
Query OK, 0 rows affected (1.31 sec)

mysql> DELIMITER //
mysql> CREATE PROCEDURE safeinsert (IN k INT, IN w INT)
    -> BEGIN
    ->   /* Pierwszy handler */
    ->   DECLARE EXIT HANDLER FOR 1062
    ->   BEGIN
    ->     SET CHARSET cp1250;
    ->     INSERT INTO log_bledow VALUES
    ->       (CONCAT('Godzina: ',current_time,' powtórzony klucz ',k));
    ->   END;
    ->   /* Drugi handler */
    ->   DECLARE EXIT HANDLER FOR 1264
    ->   BEGIN
    ->     SET CHARSET cp1250;
    ->     INSERT INTO log_bledow VALUES
    ->       (CONCAT('Godzina: ',current_time,' ujemna wartość ',w));
    ->   END;
    ->   /* Ciało procedury - wstawianie */
    ->   INSERT INTO tabela VALUES (k, w);
    -> END; //
Query OK, 0 rows affected (0.49 sec)
```



```
mysql> CALL safeinsert (1,5);
Query OK, 1 row affected (0.09 sec)
mysql> CALL safeinsert (1,6);
Query OK, 1 row affected (0.11 sec)
mysql> CALL safeinsert (2,7);
Query OK, 1 row affected (0.04 sec)
mysql> CALL safeinsert (3,-1);
Query OK, 1 row affected (0.03 sec)

mysql> SELECT * FROM tabela;
+---+------+
| K | W    |
+---+------+
| 1 | 5    |
| 2 | 7    |
+---+------+
2 rows in set (0.00 sec)
mysql> SELECT * FROM log_bledow;
+--------------------------------------+
| B                                    |
+--------------------------------------+
| Godzina: 09:44:06 powtórzony klucz 1 |
| Godzina: 09:44:36 ujemna wartość -1  |
+--------------------------------------+
2 rows in set (0.00 sec)
```

Polecenia `PREPARE` i `EXECUTE`

Przypuśmy, że mamy kilka tabel o *takiej samej* strukturze, na przykład

```
mysql> CREATE TABLE Kowalski
    -> (Data DATE NOT NULL,
    ->  Miasto VARCHAR(16) NOT NULL,
    ->  Kwota DECIMAL(8,2) UNSIGNED NOT NULL,
    ->  PRIMARY KEY(Data, Miasto));
Query OK, 0 rows affected (1.10 sec)

mysql> CREATE TABLE Nowak LIKE Kowalski;
Query OK, 0 rows affected (0.16 sec)

mysql> INSERT INTO Nowak VALUES (...);

mysql> INSERT INTO Kowalski VALUES (...);
```

Tabele `Kowalski` i `Nowak` mają taką samą strukturę, a więc, logicznie rzecz biorąc, można ich użyć w “takim samym” zapytaniu — w zapytaniu różniącym się tylko nazwą tabeli. Po stronie aplikacji zapytanie takie jest bardzo łatwo zbudować jako odpowiedni łańcuch, a następnie przesłać je do serwera SQL. Jednak traci się w ten sposób możliwość korzystania z procedur składowanych, a zatem wszystkie korzyści wynikające z używania takich procedur.

Pytanie:

Czy w SQL nazwy tabel mogą być parametrami procedur składowanych?

Tak — należy jednak wykorzystać polecenia `PREPARE` i `EXECUTE`.

Procedura mająca nazwę tabeli jako argument

```
CREATE PROCEDURE monthsales (IN nazwisko VARCHAR(16), IN miesiac TINYINT UNSIGNED)
LANGUAGE SQL
NOT DETERMINISTIC
SQL SECURITY DEFINER
BEGIN
  DECLARE EXIT HANDLER FOR 1146
  BEGIN
    SELECT nazwisko AS "Nazwisko", "Nie ma takiej tabeli" AS "Brak tabeli";
  END;
  /* tekst tworzonej komendy przechowuję _w_zmiennej_tymczasowej_! */
  SET @tmp = CONCAT('SELECT "',nazwisko,'" AS Nazwisko, SUM(KWOTA) FROM ',nazwisko,
             ' WHERE MONTH(Data)=',miesiac,';');
  SELECT @tmp AS 'Wykonam następującą komendę';
  /* utworzenie zapytania */
  /* "komenda" jest >>handlerem<< utworzonego zapytania */
  PREPARE komenda FROM @tmp;
  /* wykonanie utworzonego zapytania */
  EXECUTE komenda;
  /* usunięcie >>handlera<< */
  DEALLOCATE PREPARE komenda;
END;
```

```
mysql> CALL monthsales ('Kowalski',5);

+-----------------------------------------------------------------------------+
| Wykonam następującą komendę                                                 |
+-----------------------------------------------------------------------------+
| SELECT "Kowalski" AS Nazwisko, SUM(KWOTA) FROM Kowalski WHERE MONTH(Data)=5; |
+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)

+----------+------------+
| Nazwisko | SUM(KWOTA) |
+----------+------------+
| Kowalski | 1741.30    |
+----------+------------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)
```

```
mysql> CALL monthsales ('Nowak', 5);

+-----------------------------------------------------------------------+
| Wykonam następującą komendę                                           |
+-----------------------------------------------------------------------+
| SELECT "Nowak" AS Nazwisko, SUM(KWOTA) FROM Nowak WHERE MONTH(Data)=5; |
+-----------------------------------------------------------------------+
1 row in set (0.00 sec)

+----------+------------+
| Nazwisko | SUM(KWOTA) |
+----------+------------+
| Nowak    | 4404.51    |
+----------+------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)
```

```
mysql> CALL monthsales ('XYZ',4);

+-------------------------------------------------------------------+
| Wykonam następującą komendę                                       |
+-------------------------------------------------------------------+
| SELECT "XYZ" AS Nazwisko, SUM(KWOTA) FROM XYZ WHERE MONTH(Data)=4; |
+-------------------------------------------------------------------+
1 row in set (0.00 sec)

+----------+----------------------+
| Nazwisko | Brak tabeli          |
+----------+----------------------+
| XYZ      | Nie ma takiej tabeli |
+----------+----------------------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)
```

Funkcje składowane

Ważne ograniczenie w MySQL: *funkcje składowane nie mają dostępu do danych przechowywanych w bazie* ☹

```
mysql> CREATE FUNCTION sumasinusow (n INT)
    ->   RETURNS FLOAT
    ->   DETERMINISTIC
    -> BEGIN
    ->   DECLARE sumasinusow FLOAT DEFAULT 0;
    ->   DECLARE i INT DEFAULT 1;
    ->   WHILE i < n DO
    ->     SET sumasinusow = sumasinusow + SIN(i);
    ->     SET i = i + 1;
    ->   END WHILE;
    ->   RETURN sumasinusow;
    -> END; //
Query OK, 0 rows affected (1.01 sec)
```

Ważne: Zmienna zwracana musi być zainicjalizowana (przez `DEFAULT` lub `SET`), gdyż w przeciwnym razie funkcja zwróci `NULL`.

```
mysql> SELECT sumasinusow(15) //
+------------------+
| sumasinusow(15)  |
+------------------+
|  1.09421646595   |
+------------------+
1 row in set (0.43 sec)
```

Kursory

SQL jest językiem deklaratywnym: Określamy *co* chcemy zrobić, nie zaś *jak* to zrobić.

Kursory pozwalają na obsługę tabel wiersz po wierszu, a więc stanowią odejście od paradygmatu deklaratywnego w stronę paradygmatu imperatywnego. Obsługa wiersz po wierszu jest typowa dla aplikacji pisanych w językach wysokiego poziomu. Dlaczego więc robi się to w SQL? Żeby zmniejszyć koszt związany z przesyłaniem znacznej ilości danych.

W MySQL kursory realizowane są za pomocą procedur składowanych.

Co można zrobić z kursorem?

- Zadeklarować
- Otworzyć
- Pobrać dane, następnie zaś przejść do następnego wiersza
- Zamknąć

Kursorów warto używać tylko wtedy, gdy “przesuwają się” po kolumnie indeksowanej z unikalnymi wartościami — najlepiej jeśli jest kluczem głównym.

Przykład: Suma narastająca poprzez kursor

```
mysql> CREATE PROCEDURE kurs1 ()
    -> BEGIN
    ->   /* Najpierw deklarujemy zmienne */
    ->   DECLARE koniec, j SMALLINT UNSIGNED;  /* zmienna koniec ma wartość NULL */
    ->   DECLARE suma, z FLOAT DEFAULT 0.0;
    ->   /* Potem deklarujemy kursor */
    ->   /* Zapytanie SELECT może być bardzo skomplikowane */
    ->   DECLARE k1 CURSOR FOR SELECT I, X FROM przebieg ORDER BY I;
    ->   /* Co zrobić gdy dojdziemy do ostatniego wiersza */
    ->   DECLARE CONTINUE HANDLER FOR NOT FOUND
    ->     SET koniec = 1;
    ->   /* Zakładamy tabelę tymczasową */
    ->   DROP TEMPORARY TABLE IF EXISTS RunningTotal;
    ->   CREATE TEMPORARY TABLE RunningTotal
    ->     (I SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
    ->      X FLOAT, SumaNarastajaca FLOAT);
```

To nie koniec!

```
->   /* Otwieramy kursor */
->   OPEN k1;
->   /* Pętla wiersz po wierszu */
->   petla: LOOP
->     /* pobranie danych do kursora */
->     FETCH k1 INTO j, z;
->     /* wyjdź jeśli już skończyły się wiersze */
->     IF koniec = 1 THEN
->        LEAVE petla;
->     END IF;
->     /* oblicz i wstaw dane do tabeli tymczasowej */
->     SET suma = suma + z;
->     INSERT INTO RunningTotal VALUES (j,z,suma);
->   END LOOP petla;
->   /* koniec pętli po wierszach */
-> END; //
```

```
mysql> CALL kurs1 () ;
Query OK, 1 row affected (2.17 sec)

mysql> SELECT * FROM RunningTotal;
+----+-------+-----------------+
| I  | X     | SumaNarastajaca |
+----+-------+-----------------+
| 1  | 1     | 1               |
| 2  | 1.25  | 2.25            |
| 3  | 1.5   | 3.75            |
| 4  | 1.75  | 5.5             |
| 5  | 2     | 7.5             |
| 6  | -1.1  | 6.4             |
| 7  | -0.85 | 5.55            |
| 8  | -0.6  | 4.95            |
| 9  | -0.35 | 4.6             |
| 10 | -0.3  | 4.3             |
| 11 | -0.5  | 3.8             |
| 12 | -0.75 | 3.05            |
+----+-------+-----------------+
12 rows in set (0.09 sec)
```

Odstępstwa od standardu SQL

Kursory w MySQL (na razie?) *nie* są zgodne ze standardem SQL. Najważniejsze rzeczy, których brakuje, to

1. Zgodnie ze standardem SQL, kursory mogą być *wrażliwe* (*asensitive*) lub *niewrażliwe* (*insensitive*). Kursor niewrażliwy operuje na tymczasowej kopii danych — jest przez to wolniejszy, ale nie raguje na zmiany wprowadzane w danych wczytanych do kursora przez inne wątki. Kursor niewrażliwy jest szybszy, ale jeśli po otwarciu kursora inny wątek zmieni dane wczytane do kursora, kursor *może*, ale *nie musi*, zauważyć wprowadzone zmiany. W MySQL kursory są wrażliwe.

   Zgodnie ze standardem SQL, kursor *niewrażliwy* deklarujemy w sposób:
   `DECLARE` nazwa_kursora `INSENSITIVE CURSOR FOR ...`

2. Kursory w MySQL są *nieprzewijalne*. Kursor po wczytaniu krotki posuwa się o jedną krotkę do przodu. Zgodnie ze standardem SQL, raz otwarty kursor może dowolnie przesuwać się po swoim zakresie.
   `DECLARE` nazwa_kursora `SCROLL CURSOR FOR ...`
   Wówczas legalne są polecenia `FETCH NEXT`, `FETCH PRIOR`, `FETCH LAST`, `FETCH FIRST`, `FETCH RELATIVE` liczba, `FETCH ABSOLUTE` liczba.
3. Kursory są tylko do odczytu — za pomocą kursorów nie można zmieniać wartości otwartej przez kursor tabeli. Zgodnie ze zstandardem SQL, legalne są polecenia typu
   `UPDATE` tabela `SET ... WHERE CURRENT OF CURSOR` nazwa_kursora;

Wyzwalacze (triggers)

Wyzwalacze są procedurami wykonywanymi automatycznie po zmianie zawartości wskazanej tabeli. Wyzwalacze zostały wprowadzone do standardu SQL dość późno. W rezultacie składnia wyzwalaczy wyraźnie różni się dla różnych implementacji. W MySQL składnia ma postać:

```
CREATE TRIGGER nazwa_wyzwalacza
{BEFORE | AFTER }
{INSERT | UPDATE | DELETE }
ON nazwa_tabeli
FOR EACH ROW
wywoływane wyrażenie SQL;
```

```
mysql> DELIMITER //
mysql> CREATE TRIGGER tabela_trig
    -> BEFORE UPDATE ON tabela
    -> FOR EACH ROW
    -> BEGIN
    ->   SET @s = OLD.w;
    ->   SET @n = NEW.w;
    -> END //
Query OK, 0 rows affected (1.41 sec)
mysql> DELIMITER ;
```

```
mysql> UPDATE tabela SET W = W + 8
    -> WHERE K = 1;
Query OK, 1 row affected (1.40 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT @s, @n;
+------+------+
| @s   | @n   |
+------+------+
| 5    | 13   |
+------+------+
1 row in set (0.06 sec)
```

Kwalifikatory `OLD`, `NEW` odnoszą się do zawartości tabeli przed zmianą i po zmianie. Przy `INSERT` legalny jest tylko `NEW`, przy `DELETE` tylko `OLD`, przy `UPDATE` oba. Uwaga: Polecenie `DELIMITER` "przedefiniowuje średnik": ponieważ w ciele wyzwalacza wystepują znaki kończące zapytania, trzeba je odróżnić od znacznika końca zapytania DDL definiującego sam wyzwalacz.

Wyzwalacze mogą odwoływać się tylko do trwałych tabel podstawowych (nie do widoków ani tabel tymczasowych). Każda tabela może mieć co najwyżej jeden wyzwalacz z określoną kombinacją `BEFORE | AFTER` + `INSERT | UPDATE | DELETE`, ale może mieć wyzwalacze każdego z tych rodzajów (zatem łącznie maksimum sześć wyzwalaczy).

Wyzwalacz `INSERT` jest uruchamiany także przez `LOAD DATA`. Wyzwalacze `UPDATE, DELETE` nie są uruchamiane przez zapytania usuwające tabele lub modyfikujące jej schemat. Wyzwalacze nie są uruchamiane przez zdarzenia kaskadowe wywoływane przez klucze obce.

Wyzwalacz nie może ani rozpocząć, ani zakończyć (`COMMIT`), ani odwołać (`ROLLBACK`) transakcji. Każdy błąd spowodowany działaniem wyzwalacza powoduje automatyczne odwołanie transkacji, w obrębie której nastąpiło zdarzenie, które spowodowało uruchomienie tego wyzwalacza.

Co mogą robić wyzwalacze?

1. Wyzwalacze mogą powodować zapisanie w jakimś logu danych dotyczących zmiany, takich jak identyfikator użytkownika, który dokonał zmiany, czasu zmiany, poprzednich wartości itp. Może to służyć do monitorowania (audytu) zmian, a także do wycofania błędnych, niepotrzebnych zmian, jeśli zajdzie taka potrzeba. Jest to chyba najczęstsze zastosowanie wyzwalaczy.

2. Za pomocą wyzwalaczy można uruchamiać kaskadę zdarzeń typu `DELETE` lub `UPDATE`: Jeśli usuniemy (zmodyfikujemy) jakiś wiersz, usuwamy (modyfikujemy) odpowiadające mu wiersze w innych tabelach. **Uwaga!** Jeśli decydujemy się realizować operacje kaskadowe za pomocą wyzwalaczy, *nie ma sensu* definiować ich jednocześnie za pomocą kluczy obcych!

3. Za pomocą wyzwalaczy możemy zautomatyzować pewne operacje, które *powinny* być wykonane po zmianie wiersza w jakiejś tabeli. Na przykład, jeśli wprowadzamy nowy wiersz do tabeli `Pracownicy`, czyli gdy informujemy system o zatrudnieniu nowego pracownika, powinniśmy automatycznie wstawić odpowiedni wiersz do tabeli `Wynagrodzenia`.

4. Wyzwalacze mogą aktualizować w innych tabelach wielkości, które zależą od wartości zapisanych w tabeli, dla której zdefiniowano wyzwalacz. Gdy więc do tej tabeli wprowadzimy nowy wiersz lub gdy usuniemy lub zmodyfikujemy wiersz istniejący, wyzwalacz może zmienić pewne zapisane wielkości, tak, aby odpowiadały one aktualnej zawartości zmienianej tabeli.

Przykład

Pewna organizacja przechowuje informacje o transakcjach ze swoimi klientami. Organizacja chce też mieć podsumowane wielkości obrotów dla poszczególnych klientów. Aby nie musieć ich obliczać za każdym razem, gdy okaże się to potrzebne, tabela zawierająca podsumowane obroty jest aktualizowana przez wyzwalacz.

Tabela `Sprzedaz` zawiera informacje o poszczególnych transakcjach (w znaczeniu biznesowym, nie SQLowym), natomiast tabela `Totals` zawiera podsumowane obroty. Szczegółowa struktura tych tabel ma postać

```
mysql> DESCRIBE Sprzedaz;
+--------+------------------+------+-----+---------+-------+
| Field  | Type             | Null | Key | Default | Extra |
+--------+------------------+------+-----+---------+-------+
| CustId | int(10) unsigned | NO   | PRI | NULL    |       |
| Data   | datetime         | NO   | PRI | NULL    |       |
| Kwota  | float            | NO   |     | NULL    |       |
+--------+------------------+------+-----+---------+-------+
3 rows in set (0.11 sec)

mysql> DESCRIBE Totals;
+--------+------------------+------+-----+---------+-------+
| Field  | Type             | Null | Key | Default | Extra |
+--------+------------------+------+-----+---------+-------+
| CustId | int(10) unsigned | NO   | PRI | NULL    |       |
| Total  | float            | NO   |     | NULL    |       |
+--------+------------------+------+-----+---------+-------+
2 rows in set (0.02 sec)
```

Dla tabeli `Sprzedaz` zdefiniowano nastepujący wyzwalacz:

```
mysql> DELIMITER //
mysql> CREATE TRIGGER UpdateTotals
    -> AFTER INSERT ON Sprzedaz FOR EACH ROW
    -> BEGIN
    ->   UPDATE Totals
    ->   SET Total=Total + NEW.Kwota
    ->   WHERE CustId=NEW.CustId;
    -> END //
Query OK, 0 rows affected (0.86 sec)

mysql> DELIMITER ;
```

Aktualną zawartością tabeli `Totals` jest

```
mysql> SELECT * FROM Totals WHERE CustId=58;
+--------+-------+
| CustId | Total |
+--------+-------+
|     58 |   3.5 |
+--------+-------+
1 row in set (0.00 sec)
```

Wstawiamy teraz nowe wiersze do tabeli `Sprzedaz`

```
mysql> INSERT INTO Sprzedaz VALUES
    -> (58,'2017-11-26 20:22:20',7.20),
    -> (58,'2017-11-26 20:22:28',4.40);
Query OK, 2 rows affected (0.14 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

. . . i zaraz potem sprawdzamy zawartość tabeli `Totals`:

```
mysql> SELECT * FROM Totals WHERE CustId=58;
+--------+-------+
| CustId | Total |
+--------+-------+
|     58 |  15.1 |
+--------+-------+
1 row in set (0.00 sec)
```

$15.1 = 3.5 + 7.20 + 4.40$. Wyzwalacz zadziałał po każdym wstawieniu nowego wiersza.

Oczywiście dla kompletu należałoby także zdefiniować analogiczne wyzwalacze odpowiadające zdarzeniom `DELETE` i `UPDATE` w tabeli `Sprzedaz`.

Wyzwalacze — co może pójść źle?

- Błędy spowodowane działaniem wyzwalaczy mogą być bardzo trudne do wytropienia.
- Użytkownicy nieświadomi istnienia wyzwalaczy mogą być zaskoczeni zmianami, które “same” dokonały się w bazie danych.
- Zdarza się, że wyzwalacz modyfikuje jakąś inną tabelę, dla której zdefiniowano wyzwalacz modyfikujący kolejną tabelę, dla której. . . itd. Może to spowodować prawdziwą *nawałnicę wyzwalaczy* ☺ (ang. *trigger storm*), mogącą znacznie spowolnić pracę systemu. Nie można nawet wykluczyć sytuacji, w której wyzwalacz dla tabeli `T1` modyfikuje tabelę `T2`, dla której zdefiniowano wyzwalacz modyfikujący tabelę `T1`, co, rzecz jasna, powoduje kolejne uruchomienie zdefiniowanego dla niej wyzwalacza, w wyniku czego. . . etc — system wpada w nieskończoną pętlę.

Z tych powodów wiele osób odradza używania wyzwalaczy.

Wyzwalacze — podsumowanie

Dobrze zaprojektowane wyzwalacze mogą znacznie usprawnić działanie bazy. Co prawda wszystkie zadania obsługiwane przez wyzwalacze można powierzyć aplikacji bazodanowej korzystającej z danej bazy danych, ale tego rozwiązania *nie polecam*, bo prędzej lub później okaże sie, że projektant-programista aplikacji zapomni o jakiejś akcji, która powinna być wykonana w bazie po zmienie zawartości pewnej tabeli. Jest to zgodne z zaleceniem, że baza danych jak najwięcej rzeczy powinna robić "sama", nie polegając na komunikujących się z nią aplikacjach.

Alternatywnie, zamiast wyzwalaczy można używać procedur składowanych lub, do zdarzeń kaskadowych, kluczy obcych. Realizowanie zdarzeń kaskadowych w jednej bazie danych przez wyzwalacze **oraz** przez klucze obce oznacza zły styl

programowania i jak uczy praktyka, prędzej lub później prowadzi do katastrofy. Dla czytelności i spójności kodu projektant bazy powinien się zdecydować, czy posługuje się wyzwalaczami, czy w ich miejsce stosuje procedury składowane.

Błędem jest posługiwanie się wyzwalaczami i kluczami obcymi do inicjowania zdarzeń kaskadowych na tych samych tabelach. Używanie wyzwalaczy i kluczy obcych do inicjowania zdarzeń kaskadowych na *różnych* tabelach świadczy o chaotycznym stylu programowania.

Aby odnosić korzyści z wyzwalaczy, ich struktura musi być dobrze przemyślana, one same zaś dobrze udokumentowane.

Klauzula `FOR EACH ROW`

Klauzula `FOR EACH ROW` oznacza, że wyzwalacz działa dla każdego wiersza, który uległ zmianie. Są to wyzwalacze *row level*.

Zgodnie ze standardem SQL, możliwe jest definiowanie wyzwalaczy bez klauzuli `FOR EACH ROW` — taki wyzwalacz wywoływany jest raz w wyniku każdego zapytania wstawiającym/modyfikującym/usuwającym dane, nie zaś dla każdego wstawianego/modyfikowanego/usuwanego wiersza. Są to wyzwalacze *statement level*. Ta cecha nie jest zaimplementowana w MySQL.

Uwaga! Wyzwalacz *row level* nie zadziała, jeśli żadne wiersze nie zostaną zmienione. Wyzwalacz *statement level* zadziała dokładnie raz niezależnie od tego, czy jakieś wiersze zostaną zmienione i ile zostanie zmienionych.

Przykład

Przypuśćmy, że na tabeli `Foobar` założono wyzwalacz w jednej z dwu wersji (i że jest to możliwe):

```
CREATE TRIGGER StatementPopolupo
AFTER UPDATE ON Foobar
...;
```

```
CREATE TRIGGER RowPopolupo
AFTER UPDATE ON Foobar
FOR EACH ROW
...;
```

i wykonano zapytanie

```
UPDATE Foobar
SET Foobar.X = 5
WHERE Foobar.Foo = 'Bar';
```

Jeżeli warunek w klauzuli `WHERE` nie będzie spełniony dla żadnego wiersza, wyzwalacz `RowPopolupo` nie zadziała ani razu, natomiast wyzwalacz`StatementPopolupo` zadziała dokładnie raz. Jeżeli warunek w klauzuli

`WHERE` będzie spełniony dla 15 wierszy, wyzwalacz `RowPopolupo` zadziała 15 razy, po jednym razie dla każdego modyfikowanego wiersza, natomiast wyzwalacz `StatementPopolupo` zadziała także dokładnie raz.

Wyzwalacze `INSTEAD OF`

W niektórych systemach — na przykład w Microsoft SQL Server (MS SQL) — możliwe jest definiowane wyzwalaczy nie tylko z klauzulą `BEFORE` lub `AFTER`, ale także z klauzulą `INSTEAD OF`. Jest to przydatne na przykład dla wyzwalaczy na widokach (co jest niezgodne ze standardem!) opartych na kilku tabelach bazowych: Wyzwalacz typu `INSTEAD OF` może wówczas odpowiednio zmodyfikować wiersze wszystkich potrzebnych tabel bazowych. Bez wyzwalacza tego typu taki widok byłby niemodyfikowalny.

Więcej na ten temat można znaleźć w dokumentacji odpowiednich systemów.