Uniwersytet Wrocławski

Bazy danych 2022

Wykład 6

SQL: wyzwalacze, elementy proceduralne

Bartosz Brzostowski

Wydział Fizyki i Astronomii UWr
semestr letni r. akad. 2021/22

7 kwietnia 2022

Jak zrealizować np. logowanie zdarzeń?

- Przypuśćmy, że w bazie z zajęć chcemy logować zmiany w zamówieniach
- Tworzymy tabelę...

```
CREATE TABLE log (
  idl  int UNSIGNED NOT NULL AUTO_INCREMENT,
  z_id int UNSIGNED NOT NULL,
  p_id int UNSIGNED NOT NULL,
  bylo int UNSIGNED DEFAULT NULL,
  jest int UNSIGNED DEFAULT NULL,
  kiedy TIMESTAMP,
  PRIMARY KEY (idl)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

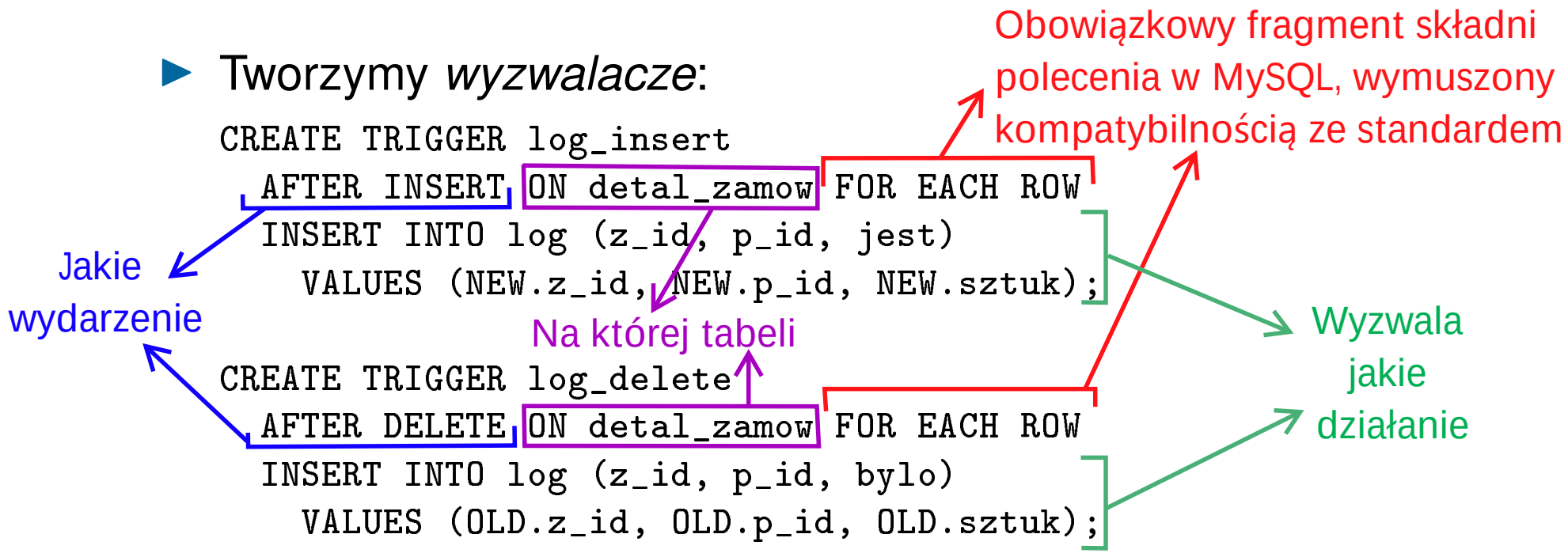
- Jak utrzymać jej zawartość? W warstwie aplikacji trudno będzie zachować integralność: tabela `detal_zamow` może być modyfikowana w wielu miejscach kodu

Kontrola po stronie DBMS

- Tworzymy *wyzwalacze*:

```
CREATE TRIGGER log_insert
  AFTER INSERT ON detal_zamow FOR EACH ROW
  INSERT INTO log (z_id, p_id, jest)
    VALUES (NEW.z_id, NEW.p_id, NEW.sztuk);

CREATE TRIGGER log_delete
  AFTER DELETE ON detal_zamow FOR EACH ROW
  INSERT INTO log (z_id, p_id, bylo)
    VALUES (OLD.z_id, OLD.p_id, OLD.sztuk);
```



- ... i sprawdzamy, czy działają:

```
mysql> INSERT INTO detal_zamow (z_id, p_id, sztuk)
    -> VALUES (2, 3, 5), (49, 18, 100);
Query OK, 2 rows affected (0,04 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> DELETE FROM detal_zamow WHERE idd >= 36;
Query OK, 2 rows affected (0,02 sec)
```

Efekty

Przy okazji widać domyślne zachowanie kolumny typu TIMESTAMP: zachowuje datę operacji INSERT/UPDATE

- Po `INSERT`

```
mysql> SELECT * FROM log;
+-----+------+------+------+------+---------------------+
| idl | z_id | p_id | bylo | jest | kiedy               |
+-----+------+------+------+------+---------------------+
|   3 |    2 |    3 | NULL |    5 | 2019-05-06 19:24:50 |
|   4 |   49 |   18 | NULL |  100 | 2019-05-06 19:24:50 |
+-----+------+------+------+------+---------------------+
```

- Po `DELETE`

```
mysql> SELECT * FROM log;
+-----+------+------+------+------+---------------------+
| idl | z_id | p_id | bylo | jest | kiedy               |
+-----+------+------+------+------+---------------------+
|   3 |    2 |    3 | NULL |    5 | 2019-05-06 19:24:50 |
|   4 |   49 |   18 | NULL |  100 | 2019-05-06 19:24:50 |
|   5 |    2 |    3 |    5 | NULL | 2019-05-06 19:25:45 |
|   6 |   49 |   18 |  100 | NULL | 2019-05-06 19:25:45 |
+-----+------+------+------+------+---------------------+
```

Przypadek UPDATE

- Wyzwalacz:

```
CREATE TRIGGER log_update
  AFTER UPDATE ON detal_zamow FOR EACH ROW
  INSERT INTO log (z_id, p_id, bylo, jest)
    VALUES (NEW.z_id, NEW.p_id, OLD.sztuk, NEW.sztuk);
```

- Efekt:

dostępne oba na raz!

```
mysql> UPDATE detal_zamow
    -> SET sztuk = sztuk + 1 WHERE idd = 35;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM log;
+-----+------+------+------+------+---------------------+
| idl | z_id | p_id | bylo | jest | kiedy               |
+-----+------+------+------+------+---------------------+
...
|   6 |   49 |   18 |  100 | NULL | 2019-05-06 19:25:45 |
|   8 |   41 |   18 |    4 |    5 | 2019-05-06 19:53:22 |
+-----+------+------+------+------+---------------------+
```

Składnia

- `CREATE TRIGGER` *[nazwa_wyzwalacza]*
  `{ BEFORE | AFTER } { INSERT | UPDATE | DELETE }`
  `ON` *[nazwa_tabeli]* `FOR EACH ROW`
  `[ { PRECEDES | FOLLOWS }` *[nazwa_innego_wyzw]* `]`
  [wyrażenia_SQL]
- Jeśli brak `PRECEDES` / `FOLLOWS`: wykonywane według kolejności definicji (do wersji 5.6: dla każdej tabeli dozwolony jeden wyzwalacz `BEFORE UPDATE` itd.)
- Tradycyjne usuwanie: `DROP TRIGGER ...`
- ... oraz sprawdzanie istniejących: `SHOW TRIGGERS ...`
- „Alias tabeli” `OLD` (poza `INSERT`): tylko do odczytu
- `NEW` (poza `DELETE`): modyfikowalny (`SET NEW.foo = ...`); jeśli chcemy zmienić wartość, która zostanie wpisana / zaktualizowana — musi to być wyzwalacz `BEFORE INSERT` / `UPDATE`

Wyzwalacz AFTER nic nie zmieni - w trakcie jego wykonywania operacja INSERT/UPDATE została już zrealizowwana

Bloki instrukcji

- Możemy wywoływać więcej niż jedną instrukcję w bloku `BEGIN ... END`
- Oddzielane średnikami, ale średnik również konczy wyrażenie SQL. Rozwiązanie:

```
mysql> delimiter //
mysql> CREATE TRIGGER ... FOR EACH ROW BEGIN ... ;
    -> ... ; END //
mysql> delimiter ;
```

Ten średnik nie jest delimiterem zapytania SQL, tylko definiuje instrukcję złożoną (tak jak w wielu językach imperatywnych).

- Instrukcje warunkowe:
  - `IF` *[warunek1]* `THEN` *[instrukcje1]*
    `[ ELSEIF` *[warunek2]* `THEN` *[instrukcje2]* `] ...`
    `[ ELSE` *[instrukcjeE]* `] END IF`
  - `CASE` *[wyrażenie]*
    `WHEN` *[wyrażenie1]* `THEN` *[instrukcje1]* `[ ... ]`
    `[ ELSE` *[instrukcjeE]* `] END CASE`
  - `CASE`
    `WHEN` *[warunek1]* `THEN` *[instrukcje1]* `[ ... ]`
    `[ ELSE` *[instrukcjeE]* `] END CASE`

Przykład: integralność danych

- Nie wszystkie więzy łatwo wyrazić, np. w tabeli `produkty` nie można (przed wersją 8.0.16) zdefiniować
  `CHECK (cena >= 0)` (można zmienić typ danych na `UNSIGNED` ale *to co innego*)
- Wyzwalacz, który „poprawia” wstawianą wartość:

```
CREATE TRIGGER dodatnia_cena
BEFORE INSERT ON produkty FOR EACH ROW
IF NEW.cena < 0 THEN SET NEW.cena = 0; END IF //
```

- Albo tak: `... SET NEW.cena = -NEW.cena; ...`
- Albo tak (bez instrukcji warunkowej):

```
CREATE TRIGGER dodatnia_cena
BEFORE INSERT ON produkty FOR EACH ROW
SET NEW.cena = ABS(NEW.cena);
```

Przy czym te wersje mają inny efekt niż pierwszy wariant. Ale oczywiście 'poprawność' danych zapewniają wszystkie trzy.

- Plus odpowiednik `BEFORE UPDATE` — z *identycznym* kodem

Spójność danych nie wystarczy sprawdzać przy INSERT - wyzwalacze często będą występować w zestawach, po jednym dla INSERT i UPDATE oraz ewentualnie DELETE

Subtelności

Można myśleć tak, że dla DROP TABLE usunięcie wierszy jest "niejawne".

- Wyzwalacze nie zawsze aktywowane wtedy, kiedy by się oczekiwało: `DROP TABLE` nie wywoła triggera dla `DELETE`, ale `REPLACE` wywoła taki dla `INSERT`
- *Cascaded foreign key actions do not activate triggers* — trzeba jawnie definiować akcje dla tabel podrzędnych w wyzwalaczach dla tabel nadrzędnych

Zależne od DBMS: np. w PostgreSQL tak nie ma

- Można zdefiniować zapętlający się wyzwalacz:

```
mysql> CREATE TRIGGER foo BEFORE INSERT ON log FOR EACH ROW
    -> INSERT INTO log (z_id, p_id) VALUES (0, 0);
Query OK, 0 rows affected (0,00 sec)
```

- ... ale (oczywiście?) dana operacja nie jest wykonywana:

```
mysql> INSERT INTO log (z_id, p_id) VALUES (1, 1);
ERROR 1442 (HY000): Can't update table 'log' in stored
function/trigger because it is already used by statement
which invoked this stored function/trigger.
```

Ograniczenia MySQL

- PostgreSQL dopuszcza wyzwalacze wywoływane przez kilka różnych zdarzeń (np. `INSERT` lub `UPDATE` — patrz ostatni przykład)
- PostgreSQL dopuszcza wyzwalacze wywoływane rekurencyjnie, uniknięcie zapętlenia jest po naszej stronie

Brak tego w MySQL to istotne ograniczenie

- PostgreSQL dopuszcza wyzwalacze wywoływane raz na (wyzwalające) wyrażenie, zamiast raz na modyfikowany wiersz (`FOR EACH STATEMENT` — domyślne, i dlatego w MySQL `FOR EACH ROW` jest obowiązkowe)
- Dla odmiany MS SQL nie ma jawnych wyzwalaczy `FOR EACH ROW`, ale zasadniczo można je uzyskać iteracją wewnątrz wyzwalacza `FOR EACH STATEMENT`
- PostgreSQL dopuszcza wyzwalacze dotyczące perspektyw — `INSTEAD OF` zamiast `BEFORE` / `AFTER`
- Oracle ma wyzwalacze „globalne” dla `CREATE` / `ALTER`

Przykład / subtelności / ograniczenia cd.

- A gdybyśmy chcieli zablokować „nielegalny” `INSERT` zamiast go poprawiać?
- PostgreSQL: funkcja wywoływana przez wyzwalacz `BEFORE` ma zwracać `NULL` (bo normalnie zwraca potencjalnie zmienioną krotkę NEW)
- MySQL: trzeba zrobić w wyzwalaczu coś „nielegalnego”:

```
mysql> CREATE TRIGGER dodatnia_cena
    -> BEFORE INSERT ON produkty FOR EACH ROW
    -> IF NEW.cena < 0 THEN SET NEW.nazwa = NULL; END IF //
Query OK, 0 rows affected (0,00 sec)

mysql> INSERT INTO produkty (nazwa, cena, ilosc)
    -> VALUES ("a", -10, 10) //
ERROR 1048 (23000): Column 'nazwa' cannot be null
```

a użytkownik na to, że "a" to nie NULL i o co w ogóle chodzi

(choć takie rzeczy i tak powinna obsłużyć aplikacja)

- Albo: `... SIGNAL SQLSTATE '45000'; ...`, różnica:

```
ERROR 1644 (45000): Unhandled user-defined exception
condition
```

tu oznacza to administratora BD

Bloki, zmienne, pętle

Dokumentacja MySQL m.in.
13.6 Compound Statements a zwł.
13.6.5 Flow Control Statements

- Blok instrukcji: `[ [etykieta] : ] BEGIN ... END`
- Zmienne użytkownika (sesji): nazwa poprzedzona `@`
- Zmienne lokalne: instrukcja `DECLARE` na początku bloku `BEGIN ... END` — zasięg zmiennej to ten blok
- `DECLARE [zmienna1], ... [typ] [ DEFAULT [wart] ]`
- Przypisanie do zmiennej (dowolnego rodzaju): `SET`
- Pętle:
  - *for* — brak
  - *while* — `WHILE [warunek] DO [instrukcje] END WHILE`
  - *do-while* — `REPEAT [instr] UNTIL [war] END REPEAT`
  - *forever* — `LOOP [instrukcje] END LOOP`
- Jeśli przed pętlą lub blokiem umieści się etykietę:
  - *break* — `LEAVE [etykieta]`
  - *continue* (tylko dla pętli) — `ITERATE [etykieta]`

Procedury i funkcje

13.1.16 CREATE PROCEDURE and CREATE FUNCTION Statements

- `CREATE PROCEDURE` *`[nazwa]`* `(` *`[lista_argumentów]`* `)` *`[instrukcja]`* — nic nie zwraca
- Instrukcja: albo pojedyncza, albo blok
- Argumenty: *`[tryb] [nazwa] [typ]`*
  - Typ: dowolny MySQLowy, tryb: `IN` (domyślny) / `OUT` / `INOUT`
  - Argumenty w trybie `IN`: można wywołać procedurę dla stałych wartości w ich miejscu, procedura może je modyfikować, ale zmiany nie są „trwałe”
  - Argumenty w trybie `[IN]OUT` muszą być zmiennymi, trwale modyfikowane; różnica: argument `OUT` wewnątrz procedury na początku jest NULLem
- Wywołanie procedury: `CALL` *`[nazwa]`* `...`
- `CREATE FUNCTION` *`[nazwa]`* `(` *`[lista_argumentów]`* `)` `RETURNS` *`[typ] [instrukcja]`*
- Argumenty *bez żadnego trybu*, zachowanie tak jak `IN`
- Musi pojawiać się instrukcja `RETURN` *`[wyrażenie]`*
- `SELECT`, nie `CALL`