

Lekcja

Temat: Usystematyzowanie materiału dla procedur, funkcji i wyzwalaczy (triggery) w MySQL

Składnia dla procedury:

DELIMITER //

```
CREATE PROCEDURE nazwa_procedury(  
    [IN nazwa_parametru typ_danych],  
    [OUT nazwa_parametru typ_danych],  
    [INOUT nazwa_parametru typ_danych]  
)
```

[MODIFIER]

BEGIN

-- Deklaracje zmiennych (opcjonalne)

DECLARE zmienna1 typ_danych;

-- instrukcje SQL

END//

DELIMITER ;

👉 Parametry w procedurach:

- **IN** – tylko do przekazania wartości wejściowej
- **OUT** – do zwrócenia wartości na zewnątrz
- **INOUT** – można przekazać i zmienić wartość w trakcie działania

Składnia dla funkcji:

```
CREATE FUNCTION nazwa_funkcji([parametry])
```

```
RETURNS typ_danych
```

[MODIFIER]

[SQL SECURITY {DEFINER | INVOKER}]

[DETERMINISTIC | NOT DETERMINISTIC]

[READS SQL DATA | MODIFIES SQL DATA]

BEGIN

-- Deklaracje zmiennych (opcjonalne)

DECLARE zmienna1 typ_danych;

-- Logika programu

-- Instrukcje SQL, obliczenia

RETURN wartość;

END //

DELIMITER ;

Elementy składni:

1. **DELIMITER //** mówi: „kończ polecenie dopiero przy //, nie przy ;”
DELIMITER; przywraca normalne zachowanie po zakończeniu tworzenia funkcji.

Przykład:

```
BEGIN
  SET x = 10;
  RETURN x;
END;
```

W MySQL **średnik (;)** jest domyślnym **znakiem końca polecenia SQL**.

MySQL bez zmiany delimitera **pomyśli, że SET x = 10; kończy całe polecenie** i wyświetli błąd składni:

#1064 - Something is wrong in your syntax obok 'SET x = 10' w linii 2

♦ Rozwiązanie — tymczasowa zmiana delimitera

Zmieniasz delimiter na coś innego (np. //, \$\$, ###), żeby MySQL wiedział, że **cała funkcja kończy się dopiero tam**, gdzie Ty wskażesz.

DELIMITER //

```
CREATE FUNCTION oblicz_vat(cena DECIMAL(10,2))
  RETURNS DECIMAL(10,2)
  DETERMINISTIC
  BEGIN
    DECLARE wynik DECIMAL(10,2);
    SET wynik = cena * 0.23;
    RETURN wynik;
  END//
```

DELIMITER ;

2. **CREATE FUNCTION nazwa_funkcji:** Definiuje nazwę funkcji, unikalną w schemacie.
3. **[parametry]** (opcjonalne): Parametry wejściowe (tylko IN, bez OUT czy INOUT).
4. **RETURNS typ_danych:** Określa typ zwracanej wartości (np. INT, VARCHAR, DECIMAL).
5. **[MODIFIER]** (opcjonalne): Opcje, takie jak:
 - o **DETERMINISTIC:** Procedura zwraca ten sam wynik dla tych samych danych wejściowych. Przykład: Funkcja obliczająca kwadrat liczby (liczba * liczba)

jest deterministyczna, ponieważ dla tej samej wartości wejściowej (np. 5) zawsze zwróci ten sam wynik (25).

- **NOT DETERMINISTIC:** Wynik może się różnić dla tych samych danych. Przykład: Funkcja zwracająca aktualny czas (NOW()) lub losową wartość (RAND()) jest niedeterministyczna, ponieważ wynik zależy od zewnętrznych czynników (czasu lub losowości).
- **CONTAINS SQL, NO SQL, READS SQL DATA, MODIFIES SQL DATA:** Określają, czy funkcja używa lub modyfikuje dane.

CONTAINS SQL:

- i. Oznacza, że procedura lub funkcja **zawiera instrukcje SQL, ale nie określa, czy odczytuje, czy modyfikuje dane.**
- ii. Jest to domyślny modyfikator, jeśli żaden inny nie zostanie wybrany.

NO SQL:

- ☐ Oznacza, że procedura lub funkcja **nie zawiera żadnych poleceń SQL** ani nie wykonuje operacji na danych w bazie.
- ☐ Używane dla procedur/funkcji, które wykonują tylko operacje na zmiennych lokalnych lub parametrach, bez odwoływania się do bazy danych.

READS SQL DATA:

- ☐ Oznacza, że procedura lub funkcja **odczytuje dane z tabel** (np. za pomocą SELECT), ale ich nie modyfikuje.

MODIFIES SQL DATA:

- ☐ Oznacza, że procedura lub funkcja **modyfikuje dane w tabelach** (np. za pomocą INSERT, UPDATE, DELETE)

❖ **SQL SECURITY DEFINER**

- Określa, że kod ma działać z uprawnieniami twórcy (definera), a nie użytkownika, który wywołuje.

❖ **SQL SECURITY INVOKER**

- Określa, że kod ma działać z uprawnieniami użytkownika, który wywołuje procedurę/funkcję

6. **BEGIN ... END:** Zawiera logikę funkcji, w tym:

- Deklaracje zmiennych (DECLARE).
- Instrukcje SQL i obliczenia.
- Obowiązkowe RETURN wartość zwracającą pojedynczą wartość.

Wywołanie

Procedura **CALL** nazwa_procedury()

Funkcja **SELECT** nazwa_funkcji()

Wyzwalacz (trigger) w MySQL to specjalny rodzaj procedury składowanej, która jest **automatycznie wywoływana w odpowiedzi na określone zdarzenia w tabeli**, takie jak wstawianie (**INSERT**), aktualizacja (**UPDATE**) lub usuwanie (**DELETE**) danych. Wyzwalacze służą do automatycznego wykonywania operacji w bazie danych, np. do zapewnienia spójności danych, logowania zmian czy automatycznego wypełniania pól.

Rodzaje wyzwalaczy w MySQL

Wyzwalacze w MySQL można podzielić na podstawie dwóch kryteriów: **czasu wywołania** i **zdarzenia**, na które reagują.

1. Czas wywołania:

- **BEFORE:** Wyzwalacz jest uruchamiany przed wykonaniem operacji (np. przed wstawieniem rekordu).
- **AFTER:** Wyzwalacz jest uruchamiany po wykonaniu operacji (np. po wstawieniu rekordu).

2. Zdarzenia:

- **INSERT:** Wyzwalacz **reaguje na wstawienie nowego rekordu do tabeli**.
- **UPDATE:** Wyzwalacz **reaguje na aktualizację istniejącego rekordu**.
- **DELETE:** Wyzwalacz **reaguje na usunięcie rekordu z tabeli**.

W efekcie można stworzyć sześć kombinacji wyzwalaczy:

- BEFORE INSERT
- AFTER INSERT
- BEFORE UPDATE
- AFTER UPDATE
- BEFORE DELETE
- AFTER DELETE

Składnia wyzwalacza w MySQL

```
CREATE TRIGGER nazwa_wyzwalacza
[BEFORE | AFTER] [INSERT | UPDATE | DELETE]
ON nazwa_tabeli
FOR EACH ROW
BEGIN
    -- Kod wyzwalacza (operacje do wykonania)
END;
```

- **nazwa_wyzwalacza:** Unikalna nazwa wyzwalacza.
- **nazwa_tabeli:** Tabela, do której wyzwalacz jest przypisany.
- **FOR EACH ROW:** Wyzwalacz jest wykonywany dla każdego wiersza, który podlega operacji.
- Wewnątrz wyzwalacza można używać słów kluczowych NEW (dla nowych danych w INSERT i UPDATE) oraz OLD (dla starych danych w UPDATE i DELETE).

NEW – dane nowego wiersza (dla **INSERT**, **UPDATE**)

OLD – dane starego wiersza (dla **UPDATE**, **DELETE**)

Lekcja

Temat: Sesja w MySQL. Transakcje w MySQL

Sesja to połączenie klienta z serwerem MySQL, które trwa od momentu zalogowania się do bazy (np. przez `mysql -u root -p` lub przez aplikację)

👉 aż do momentu, gdy to połączenie zostanie **zamknięte**.

W jednej sesji użytkownik może uruchamiać wiele transakcji, jedna po drugiej — ale tylko jedną naraz.

✖ Przykład — poprawny przebieg w jednej sesji:

```
-- sesja 1
START TRANSACTION;

UPDATE konto SET saldo = saldo - 100 WHERE id = 1;
UPDATE konto SET saldo = saldo + 100 WHERE id = 2;
COMMIT; -- kończymy pierwszą transakcję

-- teraz możemy rozpocząć drugą
START TRANSACTION;
DELETE FROM historia WHERE data < '2024-01-01';
COMMIT;
```

● Tutaj wszystko jest OK — transakcje wykonywane jedna po drugiej.

Transakcja to zestaw kilku poleceń SQL (np. INSERT, UPDATE, DELETE), które są wykonywane jako jedna całość.

Czyli:

albo wszystkie operacje się udają (zostają zapisane w bazie),
albo żadna z nich — jeśli coś pójdzie nie tak (wszystko się cofa).

♦ Podstawowe polecenia transakcyjne:

START TRANSACTION;	- rozpoczyna transakcję
COMMIT;	- zatwierdza wszystkie zmiany
ROLLBACK;	- cofnięcie wszystkich zmian do początku transakcji
SAVEPOINT nazwa;	- tworzy punkt przywracania transakcji
ROLLBACK TO nazwa;	- cofnięcie zmian tylko do danego punktu
RELEASE SAVEPOINT nazwa;	- usuwa punkt przywracania

✖ Przykład podstawowej transakcji

```
CREATE TABLE konto (  
  id INT PRIMARY KEY,  
  imie VARCHAR(50),  
  saldo DECIMAL(10,2)  
);
```

```
INSERT INTO konto VALUES  
(1, 'Adam', 1000.00),  
(2, 'Beata', 2000.00);
```

♦ Przykład 1 – przelew między kontami:

```
START TRANSACTION;
```

```
UPDATE konto SET saldo = saldo - 200 WHERE id = 1; -- Adam traci 200 zł  
UPDATE konto SET saldo = saldo + 200 WHERE id = 2; -- Beata dostaje 200 zł
```

```
COMMIT; -- Zatwierdzenie zmian
```

➡ Jeśli **wszystko się uda, zmiany zostaną na stałe zapisane** w bazie.

♦ Przykład 2 – błąd w trakcie transakcji

```
START TRANSACTION;
```

```
UPDATE konto SET saldo = saldo - 200 WHERE id = 1; -- Adam traci 200 zł  
UPDATE konto SET saldo = saldo + 200 WHERE id = 99; -- ❌ konto 99 nie istnieje
```

```
ROLLBACK; -- cofnięcie wszystkich zmian
```

➡ W efekcie **Adam nie traci 200 zł**, bo cała transakcja zostaje cofnięta.
To jest **bezpieczeństwo danych** – nic się nie "rozjedzie".

✂ SAVEPOINT — punkt przywracania

Czasem chcesz cofnąć **tylko część transakcji**, a nie całość.

♦ Przykład 3 – użycie SAVEPOINT:

```
START TRANSACTION;
```

```
UPDATE konto SET saldo = saldo - 100 WHERE id = 1;  
SAVEPOINT po_pierwszej_operacji;
```

```
UPDATE konto SET saldo = saldo - 500 WHERE id = 2;  
ROLLBACK TO po_pierwszej_operacji; -- Cofamy tylko drugą zmianę
```

```
COMMIT; -- Zatwierdzamy pierwszą zmianę
```

➡ W efekcie:

- Adamowi zabrano 100 zł ✓
- Druga operacja (z konta 2) została cofnięta ✗

✂ **RELEASE usunięcie SAVEPOINT ;**

RELEASE SAVEPOINT po_pierwszej_operacji;

Lekcja

Temat: Tabela tymczasowa w MySQL

Tabela tymczasowa (temporary table) w MySQL to specjalny rodzaj tabeli, która istnieje tylko w ramach bieżącej sesji połączenia z bazą danych. **Jest ona automatycznie usuwana po zakończeniu sesji** (np. po rozłączeniu się z serwerem MySQL).

Tabele tymczasowe są przydatne do przechowywania pośrednich wyników zapytań, przetwarzania danych tymczasowo lub unikania konfliktów z trwałymi tabelami. Są widoczne tylko dla użytkownika, który je utworzył, i nie wpływają na inne sesje.

Polecenie tworzące tabele tymczasową:

CREATE TEMPORARY TABLE

♦ Tworzenie tabeli tymczasowej z wyniku zapytania

```
CREATE TEMPORARY TABLE nazwa_tabeli_tymczasowej AS  
SELECT * FROM nazwa_tabeli;
```

♦ Tworzenie tabeli tymczasowej

Składnia jest prawie taka sama jak dla zwykłej tabeli, z dodatkiem słowa kluczowego

TEMPORARY

```
CREATE TEMPORARY TABLE nazwa_tabeli_tymczasowej (  
    kolumna1 typ_danych [opcje],  
    kolumna2 typ_danych [opcje],  
    -- itd.  
);
```

Polecenie usuwające tabele tymczasową:

```
DROP TEMPORARY TABLE IF EXISTS temp_tab;
```

Kiedy używać tabel tymczasowych?

- ☐ Do przetwarzania dużych zbiorów danych w złożonych zapytaniach (np. w procedurach składowanych).
- ☐ Do tymczasowego przechowywania wyników podzapytań.
- ☐ W raportach lub analizach, gdzie nie chcesz modyfikować stałych tabel.
- ☐ Aby uniknąć blokad w wieloużytkownikowych środowiskach.

Ograniczenia

- Nie można tworzyć indeksów pełnotekstowych ani widoków na tabelach tymczasowych.
- W niektórych silnikach (np. InnoDB) mogą być wolniejsze dla bardzo dużych zbiorów.
- Jeśli sesja się zakończy nieoczekiwanie, tabela zniknie.

Uwaga: jeśli utworzysz tymczasową tabelę o takiej samej nazwie jak istniejąca stała tabela, MySQL **będzie używać wersji tymczasowej** w danej sesji. Po jej usunięciu znów zobaczysz oryginalną tabelę.

♦ Widoczność i izolacja

- Tabela tymczasowa jest **widoczna tylko w ramach bieżącego połączenia**.
- **Inne sesje** (nawet ten sam użytkownik) **nie mają do niej dostępu**.
- Dzięki temu nie musisz martwić się o kolizje nazw między użytkownikami lub zapytaniami.

♦ Wydajność i miejsce przechowywania

- MySQL tworzy tymczasowe tabele w **pamięci RAM (MEMORY)** lub **na dysku (InnoDB / MyISAM)** – zależnie od ich rozmiaru i typu danych.
- Dla małych zbiorów danych (bez kolumn typu **TEXT** czy **BLOB**) tabela będzie w pamięci.
- Gdy przekroczy limit **tmp_table_size** lub **max_heap_table_size**, MySQL **przeniesie ją automatycznie na dysk**.

♦ Indeksy i klucze

Możesz w tabelach tymczasowych:

definiować **PRIMARY KEY**, **UNIQUE**, **INDEX** itp.

```
CREATE TEMPORARY TABLE produkty_tmp (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  nazwa VARCHAR(100),  
  cena DECIMAL(10,2),  
  INDEX (cena)  
);
```

- Jednak nie możesz tworzyć **indeksów FULLTEXT** ani **SPATIAL** w tabelach tymczasowych.

♦ Typowe zastosowania

- ✓ **Przechowywanie wyników pośrednich** — np. podczas tworzenia raportów lub obliczeń.
- ✓ **Łączenie dużych danych etapami** (np. przez JOIN-y z danymi wstępnie przefiltrowanymi).
- ✓ **Przyspieszanie złożonych zapytań** (zamiast tworzyć tymczasowe widoki).
- ✓ **Izolacja danych dla konkretnego użytkownika lub procesu** — szczególnie przy analizie danych sesyjnych.
- ✓ **Wielokrotne użycie danych w obrębie jednej transakcji** bez konieczności ponownego zapytania do głównej tabeli.

♦ Ograniczenia

- ⚠ **Brak replikacji:** dane z tabel tymczasowych nie są replikowane między serwerami Master–Slave.
- ⚠ **Brak trwałości:** po restarcie serwera MySQL tabele tymczasowe znikają.
- ⚠ **Nie można używać ALTER TABLE** do zmiany struktury w niektórych wersjach MySQL.
- ⚠ **Uważaj na nazwy:** jeśli zapomnisz o **TEMPORARY**, możesz nadpisać istniejącą tabelę trwałą o tej samej nazwie.

Lekcja

Temat: BETWEEN, IN, LIKE

```
CREATE TABLE Klienci (  
  id_klienta INT PRIMARY KEY,  
  imie VARCHAR(50),  
  nazwisko VARCHAR(50),  
  miasto VARCHAR(50)  
);
```

```
CREATE TABLE Zamowienia (  
    id_zamowienia INT PRIMARY KEY,  
    id_klienta INT,  
    data_zamowienia DATE,  
    kwota DECIMAL(10,2),  
    FOREIGN KEY (id_klienta) REFERENCES Klienci(id_klienta)  
);
```

```
INSERT INTO Klienci (id_klienta, imie, nazwisko, miasto) VALUES  
(1, 'Jan', 'Kowalski', 'Warszawa'),  
(2, 'Anna', 'Nowak', 'Kraków'),  
(3, 'Piotr', 'Zieliński', 'Gdańsk'),  
(4, 'Ewa', 'Wiśniewska', 'Poznań');
```

```
INSERT INTO Zamowienia (id_zamowienia, id_klienta, data_zamowienia, kwota) VALUES  
(101, 1, '2024-01-15', 350.00),  
(102, 2, '2024-02-10', 150.00),  
(103, 3, '2024-03-05', 900.00),  
(104, 1, '2024-03-20', 120.00),  
(105, 2, '2024-05-02', 450.00);
```

Przykłady warunków

BETWEEN

```
SELECT  
    K.imie,  
    K.nazwisko,  
    Z.kwota  
FROM Klienci K  
JOIN Zamowienia Z ON K.id_klienta = Z.id_klienta  
WHERE Z.kwota BETWEEN 200 AND 500;
```

➡ Pokazuje zamówienia o kwocie **od 200 do 500 zł** (włącznie).

IN

```
SELECT  
    K.imie,  
    K.miasto,  
    Z.data_zamowienia  
FROM Klienci K  
JOIN Zamowienia Z ON K.id_klienta = Z.id_klienta  
WHERE K.miasto IN ('Warszawa', 'Kraków');
```

LIKE

```
SELECT  
    K.imie,  
    K.nazwisko,
```

```

Z.kwota
FROM Klienci K
JOIN Zamowienia Z ON K.id_klienta = Z.id_klienta
WHERE K.nazwisko LIKE 'Kow%';

```

✚ LIKE z wyszukiwaniem w środku tekstu

```

SELECT * FROM Klienci
WHERE miasto LIKE '%a%';

```

✚ LIKE wyszukuje każdy dowolny znak _

```

SELECT * FROM Klienci
WHERE nazwisko LIKE '____i';

```

Zakresy

Oznaczenie	Definicja
$(a; b)$	$x \in (a; b) \Leftrightarrow a < x < b$
$\langle a; b \rangle$ lub $[a, b]$	$x \in \langle a; b \rangle \Leftrightarrow a \leq x \leq b$
$\langle a; b \rangle$ lub $[a, b)$	$x \in \langle a; b \rangle \Leftrightarrow a \leq x < b$
$(a; b)$ lub $(a, b]$	$x \in (a; b) \Leftrightarrow a < x \leq b$
$(-\infty; a)$	$x \in (-\infty; a) \Leftrightarrow x < a$
$(a; \infty)$	$x \in (a; \infty) \Leftrightarrow x > a$

Podsumowując

Zapis spotykany	Znaczenie	W kodzie (np. JS / C / SQL)
$\langle 200; 400 \rangle$	od 200 (włącznie) do 400 (bez 400)	<code>x >= 200 && x < 400</code>
$\langle 200; 400 \rangle$	od 200 do 400 (włącznie)	<code>x >= 200 && x <= 400</code>
$(200; 400 \rangle$	większe niż 200, do 400 włącznie	<code>x > 200 && x <= 400</code>
$(200; 400)$	między 200 a 400, bez obu końców	<code>x > 200 && x < 400</code>

SELECT

TRUE	AND	NULL	AS wynik1,	-- NULL
FALSE	AND	NULL	AS wynik2,	-- 0
FALSE	AND	FALSE	AS wynik3,	-- 0
FALSE	AND	TRUE	AS wynik4,	-- 0
TRUE	AND	FALSE	AS wynik5,	-- 0
TRUE	AND	TRUE	AS wynik6,	-- 1
TRUE	OR	NULL	AS wynik7,	-- 1
FALSE	OR	NULL	AS wynik8,	-- NULL
FALSE	OR	FALSE	AS wynik9,	-- 0
FALSE	OR	TRUE	AS wynik10,	-- 1
TRUE	OR	FALSE	AS wynik11,	-- 1
TRUE	OR	TRUE	AS wynik12,	-- 1
NOT NULL			AS wynik13;	