w wykładzie wykorzystano:

1. dr inż. Piotr Macioł, wykłady
2. dr Maciej Bobrowski, *Wykład z programowania baz danych*, http://gryf.mif.pg.gda.pl/~mate/wyklady/bazy/

Procedury i funkcje. Triggery. Widoki.

Krzysztof Regulski
WIMiIP, KISiM,
regulski@agh.edu.pl
B5, pok. 408

MySQL

- MySQL jest szybkim, wielowątkowym serwerem relacyjnych baz danych obsługującym język zapytań baz danych – SQL.
- Pracuje w systemie wielodostępowym
- Jest bardzo szybki i wydajny
- Najnowsze wersje posiadają wbudowane wydajne i bezpieczne mechanizmy transakcji i widoków.
- Jest jednym z najbardziej popularnych serwerów baz danych.

Uruchamianie

```
c:\mysql\bin> mysql -u root -p
Enter password: *******
Welcome to the MySQL monitor. Commands end
with ; or \g.
Your MySQL connection id is 456 to server
version 4.1.2

Type 'help' for help.

mysql>
```

Opcje uruchamiania - I

-u *nazwa użytkownika;*

-p *hasło użytkownika*;

-h *adres internetowy* komputera, na którym znajduje się serwer baz danych MySQL, domyślnie jest to komputer lokalny localhost czyli IP 127.0.0.1 ;

-P *numer portu* warstwy transportowej modelu TCP/IP, na którym nasłuchuje serwer baz danych komputera. Najczęściej jest to numer 3306 ;

Opcje uruchamiania - II

-e *polecenie SQL,* wykonanie polecenia SQL, podanego w cudzysłowach po opcji -e, przekierowanie informacji do okna terminala i powrót do powłoki użytkownika;

-B, wypisywanie każdego rekordu w nowej linii, a kolumn rozdzielonych tabulatorem. Przydatna opcja w przypadku wysyłania wyników zapytania na okno terminala (razem z opcją -e);

-V, wypisanie wersji serwera bazy danych MySQL i powrót do powłoki; nie ma trybu interaktywnego;

Podstawowe polecenia

- **help** wyświetla listę podstawowych poleceń wraz z krótkim opisem;
- **use** *nazwa_bazy_danych* służy do wyboru bazy danych;
- **source** *nazwa pliku* wykonuje polecenia zebrane w pliku;
- **pager** *nazwa_programu* przekierowuje wyjście z zapytań do odpowiedniego programu;
- **nopager** wyłącza przekierowanie wyjścia z zapytań;
- **tee** *nazwa_pliku* zapisuje sesję pracy z bazą danych; wszystkie wydane polecenia będą logowane w pliku; aby zakończy zapisywanie do pliku wystarczy wydać polecenie **notee**
- **show databases** wyświetla nazwy wszystkich baz danych na serwerze baz danych, do użytkownik ma dostęp;
- **show tables** wyświetla listę wszystkich tabel wybranej bazy danych;
- **describe|desc** *nazwa_tabeli* wyświetla informacje o strukturze tabeli;
- **select now()** wyświetla aktualną datę i godzinę;
- **select user()** wyświetla nazwę użytkownika i nazwę komputera, z którego nastąpiło połączenie z bazą danych;

Typy danych - BLOB

MySQL obsługuje cztery typy BLOB i cztery TEXT:

- tinyblob - od 0 do 255 znaków
- blob - od 0 do 65 535 znaków
- mediumblob - od 0 do 16 777 216 znaków
- longblob - od 0 do 2 147 483 648 znaków
- (tiny-, medium-, long-) TEXT - od 0 do 2 147 483 648 znaków (nie rozróżnia małych i wielkich znaków)

  BLOB przechowuje znaki w formie numerycznej, porównywanie i sortowanie odbywa się poprzez wartości numeryczne. TEXT – na podstawie *character set*.

- ‘a’ oraz ‘a ‘ w przypadku TEXT będą sobie równe, dla BLOB - nie

Operatory i funkcje

```
:=
|| OR, XOR, && AND
BETWEEN, CASE, WHEN, THEN, ELSE
=, <=> (NULL-safe equal), >=, >, <=, <, <>, !=, IS, LIKE,
  REGEXP, IN
-, +, *, /, DIV, %, MOD
<<, >>, ^,|,&, - (unary minus), ~ (unary bit inversion)
 NOT, !
 BINARY, COLLATE
```

Operatory i funkcje

- Jeśli po obydwu stronach operatora znajduje się wartość NULL to wynik operacji porównania jest NULL, poza przypadkiem operatora <=>
- Jeśli po obydwu stronach operatora są stringi, to są one porównywane jak stringi.
- Jeśli po obydwu stronach operatora są liczby całkowite, to są one porównywane jak liczby całkowite.
- Wartości szesnastkowe są traktowane tak jak "binarne" stringi, poza przypadkiem, kiedy z jednej strony operatora mamy wartość szesnastkową a z drugiej liczbę.
- Jeśli jednym z argumentów jest zmienna typu TIMESTAMP lub DATETIME a drugim jest stała, to podczas operacji porównania stała jest zamieniana do typu TIMESTAMP i dopiero wtedy następuje porównanie.
- We wszystkich innych przypadkach argumenty są porównywane jak liczby zmiennoprzecinkowe (rzeczywiste).
- Domyślnie operacje porównywania stringów nie uwzględniają wielkości liter. Dlatego 'string1' jest taki sam jak 'StRinG1'.
- Za pomocą funkcji CAST() można zamienić argument do odpowiedniego typu danej. Ponadto kodowanie stringów można zamienić na inne kodowanie przy pomocy funkcji CONVERT().

Funkcje

- COALESCE(*wart1,wart2,...*) zwraca pierwszą wartość z listy nierówną NULL
- LEAST(*wart1,wart2,...*) zwraca najmniejszą wartość z listy. Argumenty są porównywane według następujących zasad:
  - » jeśli wszystkie wartości są liczbami całkowitymi to są porównywane jak liczby całkowite
  - » jeśli wszystkie wartości są liczbami rzeczywistymi to są porównywane jak liczby rzeczywiste
  - » jeśli jakiś argument jest stringiem ze znaczącą wielkością znaków, to argumenty są porównywane tak, jak stringi z uwzględnieniem rozróżniania wielkości znaków
  - » we wszystkich innych wypadkach, argumenty są porównywane tak jak stringi, przy czym nie rozróżnia się wielkości znaków.
- GREATEST(*wart1,wart2,...*) zwraca największą wartość z listy. Argumenty są porównywane na takich samych zasadach jak w przypadku funkcji LEAST()
- ISNULL(*wyr*) zwraca 1 jeśli wyr jest równe NULL, w przeciwnym wypadku zwraca 0
- INTERVAL(*N,N1,N2,N3,...*) zwraca indeks pierwszego mniejszego od N elementu z listy, albo -1 jeśli N jest równe NULL.

Wyrażenia regularne

- *wyr* REGEXP *wzorzec*
- *wyr* RLIKE *wzor*

  Dopasowywanie wzorca *wzor* do wyrażenia *wyr*. *wzor* może być tzw. rozszerzonym wyrażeniem regularnym. Wyrażenie zwraca wartość 1 jeśli dopasowanie się powiodło i 0 jeśli się nie powiodło. Jeśli *wzor* lub *wyr* jest NULL, wtedy wynik też jest NULL. RLIKE jest synonimem REGEXP dla kompatybilności z mSQL.
- STRCMP(*wyr1,wyr2*) zwraca 0 jeśli paramery są takie same, -1 jeśli pierwszy argument jest mniejszy od drugiego, oraz 1 w innych wypadkach.
- BINARY(*wyr*) zamienia *wyr* występujący z prawej strony operatora do stringu "binarnego„
- LIKE jest szybsze, ale porównuje cały ciąg znaków, REGEXP wyszukuje wyrażenia w dowolnym miejscu ciągu znaków

Polecenia dotyczące tabel

- ANALYZE TABLE sprawdza tylko klucze (główne i obce) w tabeli.
- CHECK TABLE sprawdza tabelę pod względem jakichkolwiek błędów jak również spójność samych danych w tabeli.
- OPTIMIZE TABLE. Polecenie to jest używane po wykonaniu większej ilości zmian w tabeli (usunięcie sporej ilości rekordów, update większej ilości danych). Usuwane lub zmieniane rekordy są zapisywane na specjalnej liście i kolejna operacja INSERT powoduje przesunięcie odpowiednich indeksów.
- REPAIR TABLE. Naprawia prawdopodobnie zepsutą tabelę. Może się to zdarzyć na przykład po gwałtownym wyłączeniu zasilania.
- BACKUP TABLE. Polecenie to powoduje skopiowanie minimalnej ilości plików z tablicami potrzebnych do odtworzenia tabeli z backupu.
- RESTORE TABLE. Polecenie odtwarza tabelę lub tabele z backupu wykonanego poleceniem BACKUP TABLE

Procedury składowane

- Procedura jest umiejscowiona bezpośrednio w systemie bazy danych, a nie po stronie klienta.
- Pozwala to na zmniejszenie liczby kroków wymiany danych pomiędzy klientem a systemem zarządzania bazą danych, co może przyczynić się do wzrostu wydajności systemu.
- Zastosowanie procedur składowanych pozwala również wprowadzić bardziej przejrzysty interfejs pomiędzy bazą danych a aplikacjami z niej korzystającymi.
- **Zalety:**
  - » uporządkowanie/centralizacja operacji na bazie danych
  - » wprowadzanie reguł bezpieczeństwa (klient ma prawo wykonać procedurę, a nie wykonać dowolne zapytanie)
  - » zmniejszenie liczby interakcji z bazą danych

Procedury użytkownika

- Aby móc zdefiniować procedurę w mysql trzeba posiadać odpowiednie przywileje (CREATE ROUTINE, ALTER ROUTINE, EXECUTE).
- Można to sprawdzić poleceniem SHOW GRANTS FOR 'user'@'localhost';).
- Może się zdarzyć, że konieczne będzie uruchomienie skryptu *mysql_fix_privilege_tables*. Może go uruchomić zwykły użytkownik bazy danych, nie potrzebne są do wykonania tego skryptu uprawnienia administratora.

Procedury użytkownika

- Podczas definiowania procedury przeszkadzać może znak średnika ';'. Wystarczy, że w konsoli mysql wpiszemy np "DELIMITER //", co pozwoli nam na zapisanie kodu.
- Natomiast znak średnika zastąpi '//', a na końcu warto wpisać "DELIMITER ;//" tak aby średnik miał swoje dawne "znaczenie".
- Poza tym warto ustawić zmienna globalną *log_bin_trust_function_creators* na **1**.
  Pozowli to nam na zapisywanie funkcji. Możemy się do tego celu posłużyć poleceniem *SET GLOBAL log_bin_trust_function_creators = 1;*, które wpisujemy w konsoli mysql.
- wygodnie jest pisać wszelkie funkcje w osobnym pliku *.sql, a później korzystać z polecenia "source *nazwa_pliku*" w konsoli mysql.

Procedury użytkownika

- Procedurę wywołuje się poleceniem CALL *procedura*()
- Nie można używać wewnątrz procedur takich poleceń jak: CREATE PROCEDURE, ALTER PROCEDURE, DROP PROCEDURE, CREATE FUNCTION, DROP FUNCTION, CREATE TRIGGER, DROP TRIGGER.

Parametry

- Po nazwie procedury, podczas jej definicji, można zdefiniować listę parametrów w nawiasach okrągłych.
- CREATE PROCEDURE p5 () ... pusta lista argumentów
- CREATE PROCEDURE p5 ([IN] *nazwa typ_danych*) ... jeden parametr wejściowy. Słowo IN jest opcjonalne, gdyż parametry domyślnie są wejściowe.
- CREATE PROCEDURE p5 (OUT *nazwa typ_danych*) ... jeden parametr wyjściowy. Słowo OUT jest obligatoryjne.
- CREATE PROCEDURE p5 ([INOUT] *nazwa typ_danych*) ... jeden parametr wejściowy i wyjściowy jednocześnie.

Procedury użytkownika

```
mysql> CREATE PROCEDURE p5 (p INT) SET @x = p //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL p5(12345)//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x//
+-------+
| @x    |
+-------+
| 12345 |
+-------+
1 row in set (0.00 sec)
```

Procedury użytkownika

```
mysql> CREATE PROCEDURE p6 (OUT p INT) SET p = -5 //
Query OK, 0 rows affected (0.00 sec)

mysql> CALL p6(@y)//
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @y//
+------+
| @y   |
+------+
| -5   |
+------+
1 row in set (0.00 sec)
```

Deklarowanie zmiennych

Wewnątrz procedur można deklarować zmienne widoczne wewnątrz bloku (zmienne lokalne).

```
mysql> CREATE PROCEDURE p8 ()
BEGIN
  DECLARE a INT;
  DECLARE b INT;
  SET a = 5;
  SET b = 5;
  INSERT INTO tab VALUES (a);
  SELECT s1 * a FROM tab WHERE s1 >= b;
END; //
```

Zmienne deklarowane są zaraz po słowie BEGIN.

Klauzula DEFAULT

```
mysql> CREATE PROCEDURE p10 ()
BEGIN
  DECLARE a, b INT DEFAULT 5;
  INSERT INTO t VALUES (a);
  SELECT s1 * a FROM t WHERE s1 >= b;
END; //
```

Słowo DEFAULT umożliwia nadanie wartości początkowych zmiennym.

Zasięg zmiennych

```
mysql> CREATE PROCEDURE p11 ()
BEGIN
  DECLARE x1 CHAR(5) DEFAULT 'outer';
  BEGIN
    DECLARE x1 CHAR(5) DEFAULT 'inner';
    SELECT x1;
  END;
  SELECT x1;
END; //


mysql> CALL p11(); //
| x1    |
| inner |
1 row in set (0.02 sec)
+-------+
| x1    |
| outer |
1 row in set (0.02 sec)

Query OK, 0 rows affected (0.02 sec)
```

IF-THEN-ELSE

Wewnątrz procedur składowanych można używać wyrażeń warunkowych

```
mysql> CREATE PROCEDURE p12 (IN par1 INT)
  BEGIN
  DECLARE v1 INT;
  SET v1 = par1 + 1;
  IF v1 = 0 THEN
    INSERT INTO tab VALUES (17);
  END IF;
  IF par1 = 0 THEN
    UPDATE tab SET s1 = s1 + 1;
  ELSE
    UPDATE tab SET s1 = s1 + 2;
  END IF;
  END; //
```

Wyrażenie CASE

```
mysql> CREATE PROCEDURE p13 (IN par1 INT)
BEGIN
  DECLARE v1 INT;
  SET v1 = par1 + 1;
  CASE v1
    WHEN 0 THEN INSERT INTO tab VALUES (17);
    WHEN 1 THEN INSERT INTO tab VALUES (18);
    ELSE INSERT INTO tab VALUES (19);
  END CASE;
END; //
```

Pętle I

- WHILE ... END WHILE
- LOOP ... END LOOP
- REPEAT ... END REPEAT
- GOTO

WHILE ... END WHILE

```
mysql> CREATE PROCEDURE p14 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  WHILE v < 5 DO
    INSERT INTO tab VALUES (v);
    SET v = v + 1;
  END WHILE;
END; //
```

Pętle II

```
mysql> CALL p14()//
Query OK, 1 row affected (0.00 sec)
```

Nie ma się co martwić, że dostaliśmy `1 row affected` bo ta informacja dotyczy tylko ostatniego polecenia `INSERT`.

```
mysql> SELECT * FROM tab; //
+------+
| s1   |
+------+
....
|    0 |
|    1 |
|    2 |
|    3 |
|    4 |
+------+
9 rows in set (0.00 sec)
```

Pętle III

REPEAT ... END REPEAT

```
mysql> CREATE PROCEDURE p15 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  REPEAT
    INSERT INTO tab VALUES (v);
    SET v = v + 1;
    UNTIL v >= 5
  END REPEAT;
END; //
mysql> CALL p15()//
Query OK, 1 row affected (0.00 sec)
mysql> SELECT COUNT(*) FROM tab//
+----------+
| COUNT(*) |
+----------+
|       14 |
+----------+
1 row in set (0.00 sec)
```

Pętle IV

LOOP ... END LOOP

```
mysql> CREATE PROCEDURE p16 ()
BEGIN
  DECLARE v INT;
  SET v = 0;
  znacznik: LOOP
    INSERT INTO tab VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE znacznik;
    END IF;
  END LOOP;
END; //
```

Pętlę LOOP opuszcza się słowem `LEAVE`.

Znaczniki

```
mysql> CREATE PROCEDURE p17 ()
   zn1: BEGIN
   zn2: WHILE 0 = 1 DO LEAVE zn2; END WHILE;
   zn3: REPEAT LEAVE zn3; UNTIL 0 =0 END REPEAT;
   zn4: LOOP LEAVE zn4; END LOOP;
END; //
```

Znaczników można używać przed BEGIN, WHILE, REPEAT, LOOP. Są to jedyne miejsca w procedurach, w których rozpoczynanie wyrażeń znacznikami jest dozwolone.

Można też używać znaczników na końcach tych wyrażeń. Mają wtedy znaczenie organizacyjne kodu; pełnią funkcję komentarzy.

```
mysql> CREATE PROCEDURE p18 ()
   zn1: BEGIN
   zn2: WHILE 0 = 1 DO LEAVE zn2; END WHILE zn2;
   zn3: REPEAT LEAVE zn3; UNTIL 0 =0 END REPEAT zn3 ;
   zn4: LOOP LEAVE zn4; END LOOP zn4;
   END zn1 ; //
```

ITERATE

```
mysql> CREATE PROCEDURE p20 ()
  BEGIN
  DECLARE v INT;
  SET v = 0;
  znacznik: LOOP
    IF v = 3 THEN
      SET v = v + 1;
      ITERATE znacznik;
    END IF;
    INSERT INTO tab VALUES (v);
    SET v = v + 1;
    IF v >= 5 THEN
      LEAVE znacznik;
    END IF;
  END LOOP;
  END; //
```

GOTO

GOTO nie jest elementem standardu SQL.

```
mysql> CREATE PROCEDURE p... ()
  BEGIN
  ...
  LABEL label1;
  ...
  GOTO label1;
  ...
  END; //
```

Obsługa błędów I

Składowane procedury umożliwiają logowanie błędów

Przykład: Załóżmy, że chcemy logować informacje o błędach, jakie mogą się pojawić podczas umieszczania nowych danych w tablicach. Załóżmy, że chcemy aby w specjalnej tablicy pojawiały się informacje o błędach związanych z więzami między parą klucza głównego i obcego.

```
mysql> CREATE TABLE tab1
s1 INT, PRIMARY KEY (s1))
ENGINE=INNODB;//
mysql> CREATE TABLE tab2 (s1 INT, KEY (s1),
FOREIGN KEY (s1) REFERENCES tab1 (s1))
ENGINE=INNODB;//
mysql> INSERT INTO tab2 VALUES (5);//
...

ERROR 1216 (23000): Cannot add or update a child row: a
foreign key constraint fails
```

Obsługa błędów II

– Rozpoczęliśmy utworzeniem tablicy rodzic, następnie utworzyliśmy tablicę dziecko.

– Używamy tablic typu INNODB więc spodziewamy się, że podczas wstawiania rekordów do tablicy dziecko będą automatycznie sprawdzane relacje między odpowiednimi rekordami.

– Następnie próbujemy wstawić rekord do tablicy dziecko z wartością w polu klucz obcy, która nie istnieje w polu klucz główny tablicy rodzic.

– Objawił się kod błędu związany z takim zachowaniem - kod 1216. Ten kod błędu wykorzystamy do napisania naszej procedury.

Obsługa błędów III

Utwórzmy tablicę, w której będziemy logować informacje o błędach.

```
mysql> CREATE TABLE error_log (error CHAR(80)) //
```

Następnie budujemy naszą procedurę.

```
mysql> CREATE PROCEDURE p22 (param INT)
  BEGIN
    DECLARE EXIT HANDLER FOR 1216
    INSERT INTO error_log VALUES
     (CONCAT('Time: ',current_date,
     '. Foreign Key Reference Failure For
      Value = ',param));
    INSERT INTO tab2 VALUES (param);
  END;//
```

Obsługa błędów IV

– Wyrażenie DECLARE EXIT HANDLER zapewnia obsługę błędów.

– Znaczy ono dosłownie, że jeśli pojawi się błąd nr 1216, wówczas procedura wstawi nowy rekord do tablicy error_log z taką treścią jaką zdefiniowano w procedurze.

– Słowo EXIT oznacza, że procedura zakończy działanie gdy zostanie wykryty błąd 1216 (można też wymusić dalsze działanie bez przerywania kolejnych działań).

– Po wywołaniu procedury p22

```
mysql> CALL p22(5) //
```

– wstawienie wartości 5 do tab2 zakończy się błędem. Jednak tym razem nie pokaże sie nam na ekranie żaden błąd. Do tablicy error_log zostanie wstawiony jeden nowy rekord.

Składnia DECLARE HANDLER

```
DECLARE
{ EXIT | CONTINUE }
HANDLER FOR
{ kod_błędu | { SQLSTATE znak_błędu } | warunek }
wyrażenie SQL
```

Jak widać, obsługa błędu polega na napisaniu kodu procedury, który uruchomi się w odpowiednim momencie. Można wyróżnić dwa przypadki: pierwszy - EXIT - który już widzieliśmy a który powoduje opuszczenie procedury tuż po próbie wykonania polecenia, która właśnie zwróci nam odpowiedni błąd, i drugi - CONTINUE - który nie spowoduje przerwania działania procedury, lecz cały dalszy kod jest wykonywany jak gdyby nic się nie stało.

Przykład DECLARE CONTINUE HANDLER

```
mysql> CREATE TABLE tab (s1 int, primary
key(s1));//
CREATE PROCEDURE p23 ()
  BEGIN
    DECLARE CONTINUE HANDLER
    FOR SQLSTATE '23000' SET @x2 = 1;
    SET @x = 1;
    INSERT INTO tab VALUES (1);
    SET @x = 2;
    INSERT INTO tab VALUES (1);
    SET @x = 3;
  END;//
```

W przykładzie powyżej użyto dodatkowo SQLSTATE (23000), który zapewnia trochę ogólniejszą obsługę błędu niż po prostu kod błędu. W tym wypadku jest sprawdzana poprawność nie tylko dla klucza obcego ale też dla klucza głównego.

- Pierwszym wyrażeniem, które jest w tej procedurze wykonywane jest SET @x = 1;.
- Następnie wartość 1 jest wstawiana do tablicy tab w pole, które jest kluczem głównym.
- Następnie wartością x a właściwie (@x) staje się 2 (SET @x = 2;).
- Następnie procedura próbuje wstawić wartość 1 w pole, które jest kluczem głównym w tablicy tab.
- Oczywiście taki INSERT się nie powiedzie, gdyż pole s1 jest unikalne. Zatem, ponieważ polecenie INSERT się nie powiedzie i nie powiedzie się, gdyż jest to błąd '23000', uruchomi się obsługa błędu i wyrażeniem, które się kolejno wykona będzie wyrażenie SET @x2 = 1;.
- Ale to nie koniec ponieważ zadeklarowaliśmy CONTINUE HANDLER i kolejnym wyrażeniem, które się wykona będzie SET @x = 3; a więc wyrażenie, które stoi bezpośrednio po wyrażeniu SQL, które spowodowało uruchomienie obsługi błędu.

– Zatem po wywołaniu procedury p23:

```
mysql> CALL p23()//
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @x, @x2//
+------+------+
| @x   | @x2  |
+------+------+
|3     |1     |
+------+------+
1 row in set (0.00 sec)
```

DECLARE

- DECLARE warunek
- Przy pomocy warunku możemy jakby nadać błędowi dodatkowy identyfikator, tutaj ciąg znaków.
- Na początek jeszcze raz definicja tablicy tab
- mysql> CREATE TABLE tab
  s1 INT, PRIMARY KEY (s1))
  engine=innodb;//
  Następnie już odpowiednia procedura.
- mysql> CREATE PROCEDURE p24 ()

```
  BEGIN
    DECLARE `Błąd relacji klucz główny-obcy`
    CONDITION FOR SQLSTATE '23000';
    DECLARE EXIT HANDLER FOR
    `Błąd relacji klucz główny-obcy` ROLLBACK;
    START TRANSACTION;
      INSERT INTO tab VALUES (1);
      INSERT INTO tab VALUES (1);
    COMMIT;
  END; //
```

- W przykładzie najpierw definiujemy inny indetyfikator dla błędu 23000. Następnie (drugi DECLARE używamy tego identyfikatora do zdefiniowania właściwej reakcji na błąd 23000 (teraz tylko inaczej nazywany w ciele procedury). Widzimy, że tak naprawdę reakcją na błąd jest zakończenie transakcji ze skutkiem ROLLBACK.
- Spróbujmy uruchomić tą procedurę:

```
mysql> CALL p24()//
Query OK, 0 rows affected (0.28 sec)

mysql> SELECT * FROM tab//
Empty set (0.00 sec)
```

Jak widać, do tablicy `tab` nic nie zostało wstawione. Inaczej mówiąc, zadziałał mechanizm `ROLLBACK` transakcji (obydwa polecenia INSERT znajdowały się przecież wewnątrz transakcji.

Kursory

```
– mysql> CREATE PROCEDURE p25 (OUT w1 INT)
    BEGIN
      DECLARE a,b INT;
      DECLARE cur1 CURSOR FOR SELECT s1 FROM tab;
      DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET b = 1;
      OPEN cur1;
      REPEAT
        FETCH cur1 INTO a;
      UNTIL b = 1
      END REPEAT;
      CLOSE cur1;
      SET w1 = a;
    END;//
```

- Procedura rozpoczyna się od deklaracji. Akurat w tym miejscu kolejność deklaracji jest istotna.
- Najpierw deklaracja zmiennych, następnie deklaracja kursora (DECLARE cur1 CURSOR FOR, następnie deklaracja obsługi błędu (w tym przypadku zamiast kodu błędu lub odpowiedniego SQLSTATE podajemy NOT FOUND co jest tym samym co użycie kodu błędu 13160 lub SQLSTATE '02000'). Jeśli zamienilibyśmy kolejność, wówczas zostałby zgłoszony błąd.

- Pierwszym wykonanym poleceniem jest otworzenie kursora OPEN cur1;. A ponieważ cur1 jest skojarzony z poleceniem SELECT s1 FROM tab więc w tym miejscu uruchomi się właśnie to polecenie SELECT.
- Następnie jakby wyłapujemy to co zwraca polecenie SELECT. Zakładamy bowiem, że w tabeli tab jest wiele rekordów z wartościami w polu s1. Polecenie FETCH cur1 INTO a; powoduje umieszczenie danych rekordu w zmiennej a. Dzieje się tak dopóty, dopóki w tablicy tab są rekordy. Jeśli ich zabraknie, wówczas uruchomi się obsługa błędu (braku rekordów) i do zmiennej b zostanie wstawiona wartość 1 zgodnie z tym, jak zadeklarowano obsługę takiego błędu. A ponieważ b = 1 jest warunkiem zakończenia pętli UNTIL, więc w tym momencie nastąpi zakończenie pętli.
- Dalej zamykamy kursor (nie musimy tego robić, gdyż procedura w momencie zakończenia działania sama zrobi to za nas) a dalej do zmiennej wyjściowej w1 wstawiamy wartość zmiennej a a zatem dane ostatniego rekordu w tablicy tab (właściwie z pola s1).

– Spróbujmy wykonać tą procedurę.

–
```
mysql> CALL p25(@w1)//
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @w1//
+------+
| @w1  |
+------+
|5     |
+------+
1 row in set (0.00 sec)
```

– Ponieważ ostatnim rekordem w tablicy tab był rekord z wartością 5 w polu s1, więc w zmiennej w1 jest też 5.

Ograniczenia kursorów

- Kursorów nie można zmieniać (wykonywać na nich UPDATE). Można powiedzieć, że są READ ONLY. Nie można napisać:
- ```
  FETCH cursor1 INTO var1;
  UPDATE tab SET col1 = 'var' WHERE CURRENT OF cursor1;
  ```
- Kursory są też NOT SCROLLABLE, co oznacza, że możemy za pomocą kursorów pobierać tylko następne rekordy. Nie możemy poruszać się w tył i do przodu na liście rekordów. Nie można zrobić czegoś takiego:
- ```
  FETCH PRIOR cursor1 INTO var1;
  FETCH ABSOLUTE 55 cursor1 INTO var1;
  ```
- Właściwość ASENSITIVE oznacza, że od momentu otworzenia kursora aż do momentu jego zamknięcia nie powinniśmy wykonywać żadnych zmian w tablicach, które używamy przy deklaracji kursorów. W przeciwnym wypadku może się okazać, że dostaniemy zupełnie inne wyniki niż oczekiwaliśmy.

Jak definiujemy składowaną funkcję?

```
mysql> CREATE FUNCTION fun (n DECIMAL(3,0))
    RETURNS DECIMAL(20,0)
  BEGIN
    DECLARE wsp DECIMAL(20,0) DEFAULT 1;
    DECLARE licznik DECIMAL(3,0);
    SET licznik = n;
    wsp_petla: REPEAT
      SET wsp = wsp * licznik;
      SET licznik = licznik - 1;
    UNTIL licznik = 1
    END REPEAT;
    RETURN wsp;
  END //
```

Definicja podobna jest, jak widać, do definicji procedury, przy czym dodatkowo musimy dodać klauzulę RETURN gdyż funkcja zawsze coś zwraca. Musimy też zdefiniować jakiego typu dane zwraca funkcja.

– Jednak funkcje mają pewną "wadę". Otóż nie można w definicji funkcji operować na tablicach, nie można wykonywać transakcji, co można było robić przy pomocy procedur. Zatem, w ciele funkcji niedozwolone są następujące komendy SQL:

–
```
ALTER 'CACHE INDEX' CALL COMMIT CREATE DELETE
DROP 'FLUSH PRIVILEGES' GRANT INSERT KILL
LOCK OPTIMIZE REPAIR REPLACE REVOKE
ROLLBACK SAVEPOINT
'SET zmienna_systemowa' 'SET TRANSACTION'
SHOW 'START TRANSACTION' TRUNCATE UPDATE
```

– W przyszłości ma się to zmienić w MySQL. Jak widać, lepszym rozwiązaniem jest zatem używanie procedur ze zmienną OUT

Metadata

- Metadata to dane o danych. Inaczej mówi się na takie dane: słowniki danych, albo katalog systemowy.
- Zobaczmy co MySQL może nam powiedzieć na temat utworzonych przez nas danych - procedur i funkcji. Wszystkie one są przechowywane w odpowiednich tablicach w bazie mysql. Są to więc dane w tablicy.
- Dwa sposoby dowiedzenia się czegoś o procedurach i funkcjach składowanych w tablicach bazy mysql wykorzystują polecenia SHOW, zaś dwa następne wykorzystują polecenia SELECT.

– SHOW CREATE PROCEDURE - To polecenie nie zwróci nam dokładnie tego wszystkiego co napisaliśmy kiedy definiowaliśmy procedurę, ale zwraca dość sporo informacji. Jest to dość podobne polecenie do SHOW CREATE TABLE. Przykład:

– mysql> show create procedure p6 //

```
+-----------+----------+--------------------+
| Procedure | sql_mode | Create Procedure   |
+-----------+----------+--------------------+
| p6        |          | CREATE PROCEDURE   |
|           |          | `db5`.`p6`(out p   |
|           |          | int) set p = -5    |
+-----------+----------+--------------------+
```

– SHOW PROCEDURE STATUS To polecenie zwróci nam trochę więcej i trochę innych informacji.

– mysql> SHOW PROCEDURE STATUS LIKE 'p6'//

```
+------+------+-----------+-----------------+
| Db   | Name | Type      | Definer         | ...
+------+------+-----------+-----------------+
| db5 | p6    | PROCEDURE | root@localhost | ...
+------+------+-----------+-----------------+
1 row in set (0.01 sec)
```

- SELECT FROM mysql.proc
- Dane o procedurze są zawarte głównie w tablicy mysql.proc. Jest to jednak na tyle duża tablica, że nie będę przytaczał pełnego wyniku.
- Przykład:
-
```
mysql> SELECT * FROM mysql.proc WHERE name = 'p6'//
+------+------+-----------+----------------+
| db   | name | type      | specific_name | ...
+------+------+-----------+----------------+
| db5 | p6    | PROCEDURE | p6             | ...
+------+------+-----------+----------------+
1 row in set (0.00 sec)
```

- SELECT FROM INFORMATION_SCHEMA Jest to chyba najlepszy sposób na to, aby dowiedzieć się jak najwięcej o zdefiniowanych procedurach. Jednak jest dość złożony.
- INFORMATION_SCHEMA jest bazą danych informacyjną. Przechowywane są tam informacje o innych bazach danych serwera MySQL. Wewnątrz INFORMATION_SCHEMA znajduje się kilka tablic "tylko do czytania". Są to tzw. widoki, nie właściwe tablice, zatem nie są z nimi skojarzone żadne pliki.
-
```
mysql> SELECT COUNT(*) FROM
INFORMATION_SCHEMA.ROUTINES
WHERE ROUTINE_SCHEMA = 'baza1';//
+-----------+
| COUNT(*)  |
+-----------+
|        28 |
+-----------+
1 row in set (0.02 sec)
```

DROP PROCEDURE

- Do usunięcia procedury służy komenda DROP PROCEDURE,
- `mysql> DROP PROCEDURE p6 //`
- Istnieje też komenda umożliwiająca zmianę definicji procedury (`ALTER PROCEDURE`), jednak łatwiej jest chyba definicję całej procedury zapisywać w pliku, a jeśli zachodzi potrzeba zmiany procedury wystarczy usunąć starą wersję i w pliku z definicja procedury wstawić nową definicję, a następnie zawartość pliku wlać do odpowiedniej bazy danych.

Porównanie z innymi systemami baz danych

porównanie z bazą Oracle (i językiem PL/SQL).

- W systemie Oracle język procedur jest bardziej zaawansowany. Na przykład możemy manipulować tablicami i używać transakcji w funkcjach, czego nie możemy zrobić za pomocą MySQL.
- Trochę inna jest składnia niektórych wyrażeń, na przykład: a:=b zamiast SET a=b.
- W bazach Oracle możemy użyć RETURN w definicji procedury, natomiast w MySQL używamy LEAVE etykieta. Czyli, zamiast CREATE PROCEDURE ... RETURN; ... w MySQL używamy CREATE PROCEDURE nazwa() etykieta: BEGIN ... LEAVE etykieta; END.
- W przypadku kursorów w bazie Oracle możemy deklarować kursor po otworzeniu go. MySQL wymaga, aby najpierw była deklaracja a potem otworzenie kursora.
- W przypadku kursorów w bazie Oracle możemy napisać CURSOR nazwa IS zaś w MySQL piszemy DECLARE nazwa CURSOR.

- W bazie Oracle można by więc napisać:
- ```
  CREATE PROCEDURE
  nazwa
  AS
  var1 INTEGER
  var1 := 55
  END
  ```
- Zaś w MySQL wyglądało by to tak:
- ```
  CREATE PROCEDURE
  nazwa
  BEGIN
  DECLARE var1 INTEGER;
  SET var1 = 55;
  END
  ```
- Jak widać, w Oracle nie ma średników za poleceniami SQL w procedurach.
- Oczywiście różnic jest znacznie, znacznie więcej. Zostały wymienione tylko te, które dotyczyły materiału przerobionego wyżej. Istnieje oprogramowanie http://www.ispirer.com, za pomocą którego można przerobić kod procerdury z PL/SQL na kod procedury działający w MySQL.

- porównanie z bazą SQL Server
  - Micosoft SQL Server umożliwia dostęp do tablic z poziomu funkcji, podczas gdy MySQL nie ma takich możliwości.
  - zmienne w procedurach w SQL Server rozpoczynają się znakiem et (@), podczas gdy zmienne w MySQL są po prostu znakami alfanumerycznymi (nie zaczynają się znakiem et). Oczywiście mogą takie zmienne wystąpić, ale co innego wtedy znaczą.
  - Za pomocą SQL Server możemy zadeklarować zmienne w następujący sposób DECLARE v1 [typ], v2 [typ], zaś za pomocą MySQL możemy napisać tylko: DECLARE v1 [typ]; DECLARE v2 [typ].
  - SQL Server nie ma bloków kodu otoczonych BEGIN ... END zaś MySQL wymaga używania bloków BEGIN ... END.
  - SQL Server nie wymaga aby wyrażenia w procedurach kończyły się średnikami, zaś w MySQL wyrażenia wewnątrz prodcedury lub funkcji muszą się kończyć średnikami.
  - SQL serwer posiada specjalne zmienne wskazujące w danej chwili na numer rekordu z tablicy. MySQL ma natomiast funkcję FOUND_ROWS().

- » SQL Server posiada inną składnię pętli WHILE...BEGIN W MySQL zaś jest pętla WHILE...DO.
- » Jeśli w SQL Server mamy w procedurze zmienną o nazwie @x to w MySQL oznacza to zmienną sesji, a nie zmienną procedury. Zatem trzeba nazwę tej zmiennej zamienić na inną. Ale trzeba przy tym uważać, aby nie obciąć po prostu znaczka @ gdyż wtedy może się okazać, że jest x to nazwa kolumny w tablicy.

– Oczywiście różnić jest dużo, dużo więcej.

- W bazie Micosoft SQL SERVER można więc napisać:
-
```
CREATE PROCEDURE
nazwa
AS
DECLARE @x VARCHAR(100)
EXECUTE jakas_proced @x
DECLARE c CURSOR FOR
SELECT * FROM tab
END
```

- Zaś w MySQL wyglądało by to tak:
-
```
CREATE PROCEDURE
nazwa
()
BEGIN
DECLARE v_x VARCHAR(100);
CALL sp_procedure2(v__x);
DECLARE c CURSOR FOR
SELECT * FROM t;
END
```

AGH

- porównanie z bazą DB2
  - » Baza DB2 (IBM) posiada więcej możliwości niż MySQL w zakresie pisania procedur i funkcji. Między innymi umożliwia dostęp do tablic z poziomu funkcji.
  - » W DB2 mamy wyrażenia ze zmiennymi PATH, SIGNAL, można przeładowywać nazwy funkcji (różnią się one wtedy parametrami).
  - » w DB2 mamy składnię etykieta: ... GOTO etykieta zaś w MySQL mamy label etykieta; ... GOTO etykieta".
  - » Należy jednak podkreślić bardzo duże podobieństwo między procedurami pisanymi w DB2 i w MySQL.

– W bazie IBM DB2 można więc napisać:

–
```
CREATE PROCEDURE
nazwa
(param1 INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE v INTEGER;
  IF param1 >=5 THEN
  &mnsp; CALL p26();
  &mnsp; SET v = 2;
  END IF;
  INSERT INTO tab VALUES (v);
END
@
```

– Zaś w MySQL wyglądało by to tak:

```
– CREATE PROCEDURE
  sp_name
  (parameter1 INTEGER)
  LANGUAGE SQL
  BEGIN
    DECLARE v INTEGER;
    IF param1 >=5 THEN
      CALL p26();
      SET v = 2;
    END IF;
    INSERT INTO tab VALUES (v);
  END
  //
```

Wyzwalacze (triggery)

- Trigger jest obiektem związanym z tablicą, który aktywuje się gdy do tablicy następuje odpowiednie zapytanie.
- W poniższym przykładzie definiujemy tablicę a następnie trigger związany z tą tablicą, który będzie uruchomiony w momencie wykonania polecenia INSERT (a właściwie tuż przed nim) wstawiającego dane do tej tablicy.
- Trigger oblicza sumę elementów jednego z pól tablicy we wszystkich rekordach.
- Przykład
-
```
mysql> CREATE TABLE konto (numer INT, ile DECIMAL(10,2));
mysql> CREATE TRIGGER konto_bi BEFORE INSERT ON konto
       FOR EACH ROW SET @sum = @sum + NEW.ile;
```

- Aby sprawdzić jak to działa, można wstawić do tablicy konto odpowiednie wartości i sprawdzić, jaka jest wartość zmiennej @sum. A zatem
- mysql> SET @sum = 0;
  mysql> INSERT INTO konto
  VALUES(137,14.98),(141,1937.50),(97,-100.00);
  mysql> SELECT @sum AS 'Suma';

```
+----------+
| Suma     |
+----------+
| 1852.48  |
+----------+
1 row in set (0.00 sec)
```

- Widzimy, że trigger policzył sumę w polu ile z wszystkich wierszy 14.98 + 1937.50 - 100
- Aby zniszczyć trigger, należy się posłużyć komendą DROP TRIGGER i użyć nazwy triggera razem z nazwą tablicy w notacji kropkowej.
- mysql> DROP TRIGGER konto.konto_bi;

Po co są triggery

- Pozwalają sprawdzać i zapobiegać dostawaniu się do bazy danych "nieodpowiednich danych".
- Możemy zmieniać polecenia INSERT, UPDATE i DELETE lub w ogóle ich zaniechać w zależności od tego, jakie informacje dostarczy nam trigger.
- Możemy monitorować aktywność działań na danych w bazie danych w czasie sesji z bazą danych.
- Podczas definicji triggerów możemy używać tych samych wyrażeń, których używaliśmy przy definicji składowanych procedur i funkcji a w szczególności:
  - słów BEGIN oraz END a zatem możemy używać struktur blokowych w triggerach.
  - słów IF, CASE, WHILE, LOOP, REPEAT, LEAVE, ITERATE a zatem możemy sterować w sposób kontrolowany przepływem danych i dokonywać decyzji co dalej zrobi trigger.
  - słów DECLARE i SET, a zatem możemy deklarować i definiować zmienne, tak jak w procedurach i funkcjach.

Po co są triggery

- możemy deklarować uchwyty.
- Jak widzimy, w triggerach możemy prawie to samo co w funkcjach i procedurach.
- Triggery różnią się jednak od nich tym, że automatycznie są uruchamiane podczas wykonywania konkretnych zapytań do konkretnych tablic, umożliwiają zatem dość specyficzne ograniczanie dostępu do danych a jednocześnie normują sposób dostępu do danych w taki sposób jak tylko to zdefinuijemy.
- Tak samo, jak w przypadku funkcji, tak i w przypadku triggerów nie możemy używać podczas ich definicji następujących wyrażeń:
- START TRANSACTION, COMMIT, ROLLBACK

Jak definiujemy trigger?

- ```
  mysql> CREATE TRIGGER
            { BEFORE | AFTER }
            { INSERT | UPDATE | DELETE }
            ON < tablica >
            FOR EACH ROW
            < specjalne_wyrażenie_SQL >
  ```

- Triggery muszą mieć nazwę. Ich nazwa nie może być dłuższa niż 64 znaki, zatem przypominają pod tym względem inne obiekty w bazie danych (tablice, zmienne, procedury, itp).
- Triggery są nierozerwalnie związane z tablicami. Zatem w dalszej części wykładu o triggerach przyjęto konwencję, według której w nazwie triggera jest nazwa tablicy.
  ```

- Triggery uruchamiane są przed (ang. BEFORE (w skrócie B)) lub po (ang. AFTER (w skrócie A)) zadanym poleceniu, zatem dla szybkiej identyfikacji triggera do ich nazwy będzie zawsze dodawana literka B lub A. Ponad to triggery uruchamiane są w momencie wykonywania na tablicy konkretnego polecenie SQL (INSERT (w skrócie I), UPDATE (w skrócie U) lub DELETE (w skrócie D)) i w dalszej części wykładu do nazwy triggera będzie też dodawana odpowiednia litera w celu identyfikacji triggera.
- Przykłady
  - tab1_bu - nazwa triggera, który związany jest na stałe z tablicą tab1, i uruchamiany jest przed (b, before) wykonywaniem poleceń UPDATE (u).
  - studenci_ad - nazwa triggera, który związany jest na stałe z tablicą studenci i który uruchamiany jest po (a, after) wykonaniu na danych w tej tablicy poleceń DELETE (d, delete).
  - towar_ai - nazwa triggera, który związany jest na stałe z tablicą towar, i który uruchamiany jest po (a, after) wstawieniu danych do tej tablicy (i, INSERT).

Odnoszenie się do wartości OLD i NEW

- Odnoszenie się do wartości pól poprzez zwyczajną nazwę pola może być w przypadku triggerów dwuznaczne. Nie wiadomo bowiem o którą wartość chodzi w przypadku, kiedy nie wykonano jeszcze polecenia INSERT (bi - BEFORE INSERT) a trigger już się uruchomił i może pracować na wszystkich wierszach (FOR EACH ROW) z tabeli.
- Analogicznie jest w przypadku usuwania rekordów i uruchamiania triggerów po wykonaniu polecenia DELETE. Zdefiniowano więc dwa słowa: OLD oraz NEW, które jasno definiują, o które dane nam chodzi. W przypadku poleceń INSERT możemy używać jedynie słowa NEW, w przypadku poleceń DELETE jedynie słowa OLD, natomiast w przypadku komend UPDATE, zarówno OLD jak i NEW.

– Przykład użycia obydwu słów w poleceniu UPDATE

```
– mysql> CREATE TRIGGER tab_au
           BEFORE UPDATE ON tab;
           FOR EACH ROW
           BEGIN
             SET @old = OLD.s1;
             SET @new = NEW.s1;
           END;//
```

– Jeśli tablica tab zawiera jeden wiersz z wartością 10 w polu s1 i wykonam polecenie UPDATE tab SET s1 = s1 + 1; wówczas, kiedy wykonywanie tego polecenia zostanie zakończone, zmienna @old zawierać będzie wartość 10, natomiast zmienna @new zawierać będzie wartość 11.

Przykład

- Utwórzmy prostą tabelę i zdefiniujmy związany z nią trigger. Tutaj, podobnie jak w przypadku procedur i funkcji należy najpierw zdefiniować znak końca polecenia, gdyż w innym wypadku, w przypadku złożonych triggerów, znak końca polecenia wewnątrz triggera może zostać zinterpretowany jako znak końca definicji triggera. Tutaj, we wszystkich przykładach poniżej, znakiem końca polecenia będzie podwójny ukośnik //.
- mysql> DELIMITER //;

```
mysql> CREATE TABLE tab (s1 INTEGER) //

mysql> CREATE TRIGGER tab_bi
         BEFORE INSERT ON tab
         FOR EACH ROW
         BEGIN
           SET @a = 'trigger włączony';
           SET NEW.s1 = 105;
         END;//
mysql> DELIMITER ;//
```

Następny przykład

- Zdefiniujemy trigger, który będzie sprawdzał wprowadzane zmiany w tablicy. Jeśli ktoś wykona polecenie UPDATE na danych w tablicy zdefiniowanej poniżej, wprowadzając dowolne zmiany, wówczas za każdym razem (dla każdego wiersza FOR EACH ROW) nowa wartość będzie sprawdzana i dopiero kiedy pomyślnie przejdzie testy zostanie zaakceptowana, w przeciwnym wypadku zostanie wprowadzona wartość z zakresu od 0 do 100.
- Użyjemy tablicy konto zdefiniowanej wcześniej, z wartościami wcześniej już wprowadzonymi.

```
mysql> SELECT * from konto;
+-------+---------+
| numer | ile     |
+-------+---------+
|   137 |   14.98 |
|   141 | 1937.50 |
|    97 | -100.00 |
+-------+---------+
3 rows in set (0.00 sec)
```

```
mysql> DELIMITER //;

mysql> CREATE TRIGGER konto_bu
         BEFORE UPDATE ON konto
         FOR EACH ROW
         BEGIN
           IF NEW.ile < 0 THEN
             SET NEW.ile = 0;
           ELSEIF NEW.ile > 100 THEN
             SET NEW.ile = 100;
           END IF;
         END;//
mysql> DELIMITER ;//
```

```
mysql> UPDATE konto SET ile=-20;
Query OK, 3 rows affected (0.13 sec)
Rows matched: 3 Changed: 3 Warnings: 0

mysql> SELECT * from konto;
+-------+------+
| numer | ile  |
+-------+------+
|   137 | 0.00 |
|   141 | 0.00 |
|    97 | 0.00 |
+-------+------+
3 rows in set (0.00 sec)
```

```
mysql> UPDATE konto SET ile=105;
Query OK, 3 rows affected (0.00 sec)
Rows matched: 3 Changed: 3 Warnings: 0

mysql> SELECT * from konto;
+-------+---------+
| numer | ile     |
+-------+---------+
|   137 | 100.00 |
|   141 | 100.00 |
|    97 | 100.00 |
+-------+---------+
3 rows in set (0.00 sec)
```

Widoki

- Widoki (ang. views) są dostępne od wersji 5.1 MySQL
- `mysql> CREATE VIEW v1 AS SELECT col1 FROM tab;`
  `Query OK, 0 rows affected (0.01 sec)`
- Z widoków możemy zawsze korzystać za pomocą poleceń SELECT. Niektóre widoki są też zmienialne, tzn. można na nich wykonywać polecenia UPDATE, DELETE a nawet INSERT.

– Nie można definiować widoków dla tablic tymczasowych lub tymczasowych widoków. Jest to standard SQL i tak samo jest w MySQL.

```
mysql> CREATE TEMPORARY TABLE tab1 (col1 INT);
Query OK, 0 rows affected (0.27 sec)

mysql> CREATE VIEW v_tab1 AS SELECT * FROM tab1;
ERROR 1352 (HY000): View's SELECT contains a
temporary table 'tab1'

mysql> CREATE TEMPORARY VIEW vvv AS SELECT 'a';
ERROR 1064 (42000): You have an error in your SQL
syntax; ...
```

– Widok nie może mieć takiej samej nazwy jak jakakolwiek istniejąca w bazie danych tablica lub widok.

```
mysql> CREATE TABLE tab (s1 INT);
Query OK, 0 rows affected (0.30 sec)

mysql> CREATE VIEW tab AS SELECT 'a';
ERROR 1050 (42S01): Table 'tab' already exists
```

- Nie można utworzyć widoku dla tablicy, która aktualnie nie istnieje w wybranej bazie danych.
- `mysql> CREATE VIEW v AS SELECT * FROM tablica_nieistniejąca;`
  `ERROR 1146 (42S02): Table 'db5.tablica_nieistniejąca' doesn't exist`
- System Oracle posiada opcję FORCE do zdefiniowania widoku pomimo braku tablicy.
- Nie można utworzyć widoków jeśli nie mamy odpowiednich przywilejów:
- `mysql> CREATE VIEW v AS SELECT 'a' FROM tab;`
  `ERROR 1142 (42000): create view command denied to user 'user'@'localhost' for table 'v`
- Od wersji 5.* trzeba posiadać przywilej CREATE VIEW albo przywilej SUPER czyli przywilej administratora.

– Nie mozna definiować widoków i w czasie definicji odwoływać się do zmiennych sesji lub do innych zmiennych.

–
```
mysql> CREATE VIEW v1 AS SELECT * FROM tab
  WHERE s1 = @var1 AND s2 = ?;
ERROR 1351 (HY000): View's SELECT contains a variable or
parameter

mysql> CREATE PROCEDURE p1 (param INT)
  CREATE VIEW v2 AS SELECT param FROM tab//
Query OK, 0 rows affected (0.01 sec)

mysql> CALL p1(5)//
ERROR 1351 (HY000): View's SELECT contains a variable or
  parameter
```

- Na razie nie można używać podzapytań w widokach. Jest to wada MySQL.
-
```
mysql> CREATE VIEW v3 AS SELECT * FROM
    (SELECT s1 FROM t1) AS x;
ERROR 1349 (HY000): View's SELECT contains a subquery in the
FROM clause
```

- Podczas definicji widoku nie możemy nazywać dwóch kolumn w taki sam sposób.
-
```
mysql> CREATE TABLE t1 (kol1 INT);
Query OK, 0 rows affected (0.29 sec)

mysql> SELECT kol1, kol1 FROM t1;
Empty set (0.00 sec)

mysql> CREATE VIEW vt1 AS SELECT kol1, kol1 FROM t1;
ERROR 1060 (42S21): Duplicate column name 'kol1'
```

- Chociaż przy pojedynczym poleceniu SELECT jest to dozwolone, to jednak w przypadku widoków nie można tak zrobić. W przypadku widoków (tablic) każda kolumna musi być unikalna tak jak unikalne muszą być nazwy kolumn (atrybutów) w zwykłych tablicach.
- Nie można używać innej liczby kolumn w definicji widoku niz w definicji tablicy.
-
```
mysql> CREATE VIEW v (s1) AS SELECT s1, s2 FROM tab;
ERROR 1353 (HY000): View's SELECT and view's field list
have different column counts
```

Metadane

- Tabela CHARACTER_SETS: W tej tabeli są dostępne informacje analogiczne jak po wykonaniu polecenia SHOW CHARACTER SET. Czyli możemy dostać informacje o dostępnych sposobach kodowania znaków w bazie MySQL. Przykłady kolumn:
- CHARACTER_SET_NAME - zawiera informacje o sposobach kodowania dostępnych dla bieżącego użytkownika.
- DEFAULT_COLLATE_NAME - zawiera informacje o bieżącym sposobie porównywania znaków
- MAXLEN - zawiera liczbę bajtów potrzebnych do przechowywania każdego znaku w danym systemie kodowania.

- Tabela COLLATIONS: W tej tabeli przechowywane są informacje o dostępnych sposobach porównywania znaków w ciągach znaków dostępnych dla bieżącego użytkownika.
- Przykłady kolumn:
- COLLATION_NAME - zawiera informacje o dostepnych sposobach porównywania znaków międzynarodowych dla bierzącego użytkownika. Jest to ta sama informacja, która dostalibyśmy kolumnie COLLATION wykonując polecenie SHOW COLLATION.
- CHARACTER_SET_NAME - domyślny sposób kodowania znaków, którego dotyczy porównywanie COLLATION_NAME. Jest to ta sama informacja, jaką dostalibyśmy w kolumnie CHARSET wykonując polecenie SHOW COLLATION.
- SORTLEN - ilość pamięci potrzebnej do sortowania ciągów znaków zakodowanych sposobem takim jak w kolumnie CHARACTER_SET_NAME i porównywanych tak jak to okreiślono w kolumnie COLLATION_NAME. Jest to taka sama informacja jak w kolumnie SORTLEN po wydaniu komendy SHOW COLLATION.

- Tabela COLLATION_CHARACTER_SET_APPLICABILITY: Zawiera w dwóch swoich kolumnach informacje o dostępnych sposobach kodowania dla każdego sposobu porównywania dostępnego dla bierzącego użytkownika.
- Przykłady kolumn:
- COLLATION_NAME - zawiera informacje o możliwych sposobach porównywania dla bierzącego użytkownika.
- CHARACTER_SET_NAME - zawiera informacje o możliwych sposobach kodowania znaków dla bierzącego użytkownika, dla których stosuje się sposób porównywania.

- Tabela COLUMN_PRIVILEGES: W tej tabeli znajdują się wszystkie informacje dotyczące przywilejów kolumnowych nadanych dla bieżącej bazy danych.
- Nie ma odpowiednika SHOW. Zamiast niego można zebrać te informacje korzystając z wyników poleceń SHOW GRANTS oraz SHOW FULL COLUMNS.

- Tabela COLUMNS: Zawiera informacje o dostępnych kolumnach w bazie danych dla bieżącego użytkownika. Ekwiwalentem tych informacji są informacje uzyskane poleceniem SHOW COLUMNS.
- Tabela KEY_COLUMN_USAGE: Zawiera informacje o kolumnach dostępnych dla bieżącego użytkownika, które w jakiś sposób są związane parami kluczy główny-obcy lub są częścią klucza głównego (pamiętamy, że klucz główny może się składać z więcej niż jednej kolumny).
- Nie ma odpowiednika w postaci polecenia SHOW dla takich informacji.

- Tabela ROUTINES Tabela ROUTINES zawiera informacje o dostępnych dla aktualnego użytkownika składowanych procedurach i funkcjach.
- SPECIFIC_NAME - zawiera informacje o nazwach procedur i funkcji składowanych dostępnych do wykonywania dla aktualnego użytkownika.
- ROUTINE_CATALOG - zawsze NULL, gdyż MySQL nie wspiera katalogowania w bazach danych. Ale ROUTINE_CATALOG jest w standardzie SQL.
- ROUTINE_SCHEMA - zawiera informacje o bazie danych, z którą związana jest składowana procedura lub funkcja.
- ROUTINE_TYPE - procedura lub funkcja.
- DTD_IDENTIFIER - w przypadku funkcji jest tu informacja o typie zwracanych danych.
- ROUTINE_DEFINITION - tak dużo definicji ciała procedury lub funkcji, na ile pozwala na to dlugość pola.
- SECURITY_TYPE - twórca lub odtwórca ('DEFINER' or 'INVOKER').
- i jeszcze wiele innych. Są to dość cenne informacje, szczególnie dla administratora, gdyż większość z tych informacji jest niedostępna w tablicy mysql.proc.

- Tabela SCHEMA_PRIVILEGES: Ta tabela zawiera informacje o przywilejach nadanych na poziomie globalnym. Dość łądnie sformatowane dane o przywilejach, aczkolwiek podobne informacje dostaniemy wykonując polecenie SHOW GRANTS.
- Tabela SCHEMATA: Zawiera informacje o bazach danych, do których aktualny użytkownik ma dostęp. Analogiczne dane dostaniemy wykonując SHOW DATABASES, aczkolwiek w przypadku schamtów informacyjnych mamy jeszzce dane np. o systemie kodowania w tych bazach danych.
- Tabela STATISTICS: Zawiera informacje o indeksach w tablicach, do których ma dostęp aktualny użytkownik. Taka tablica nie jest w standardzie SQL. Analogiczne dane uzyskamy (w okrojonej wersji) wydając polecenie SHOW INDEXES.

- Tabela TABLE_CONSTRAINTS: Zawiera dość cenne informacje o tablicach, do których ma dostęp aktualny użytkownik, w których występują klucze główne i obce. Takie informacje są dośc cenne, gdyż właściwie nie ma prostej metdoy dowiedzenia się takich rzeczy poza poleceniem SHOW CREATE TABLE.
- Tabela TABLE_PRIVILEGES: dostarcza informacje o przywilejach na poziomie tablicowym dla aktualnej bazy danych. Nie ma odpowiednika. Można ię natomiast posłużyć poleceniami: SHOW FULL COLUMNS oraz SHOW GRANTS.
- Tabela TABLES: Zawiera informacje o tablicach dostępnych w aktualnej bazie danych aktualnie pracującemu użytkownikowi. Podobne dane dostaniemy też wykonując SHOW TABLE STATUS.

- Czyli na przykład:
- `mysql> SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME='tab1';`

- podobnie:
- `mysql> SHOW TABLE STATUS LIKE 't1%';`

- Tabela USER_PRIVILEGES: Tabela zawiera informacje o wszystkich użytkownikach i ich przywilejach w aktualnej bazie danych. Trochę więcej informacji jest zebrane w tabeli mysql.user.
- Czyli na przykład:
- `mysql> SELECT * FROM INFORMATION_SCHEMA.USER_PRIVILEGES;`
  i podobnie:
- `mysql> SELECT * FROM mysql.user;`

- Tabela VIEWS Pokazuje widoki, które są dostępne dla aktualnego użytkownika. Nie ma odpowiednika SHOW.
- TABLE_CATALOG - zawsze NULL, gdyż MySQL nie wspiera idei katalogowania baz danych.
- TABLE_SCHEMA - pokazuje informacje o bazie danych, z która związane są widoki.
- TABLE_NAME - nazwa widoku, do którego użytkownik ma dostęp.
- VIEW_DEFINITION - definicja widoku, czyli odpowiedniego polecenia SELECT.
- CHECK_OPTION - zawiera informacje o wartości tej klauzuli w definicji widoku. Może więc to być: 'NONE', 'LOCAL', lub 'CASCADED'.
- IS_UPDATABLE - informacja o tym czy widok jest zmienialny czy też nie.