# Język SQL
# (na przykładzie serwera MySQL)

**Materiały pomocnicze do wykładu**
**Wersja 2.2.4**
**Ostatnia aktualizacja: 16-05-2006**

ARTUR GRAMACKI

**Uniwersytet Zielonogórski**
**Instytut Informatyki i Elektroniki**

# Spis treści

**1 Uwagi wstępne** **4**

1.1 Na początek . . . . . 4

1.2 Podstawowe grupy poleceń . . . . . 5

1.3 Przechowywanie danych w MySQL . . . . . 6

**2 Polecenie SELECT** **8**

2.1 Składnia polecenia SELECT . . . . . 8

2.2 Najprostsze przykłady . . . . . 8

2.3 Klauzula ORDER BY . . . . . 12

2.4 Klauzula WHERE . . . . . 14

2.5 Ograniczanie wyników wyszukiwania za pomocą klauzuli LIMIT . . . . . 17

2.6 Operatory i funkcje porównywania . . . . . 18

2.7 Aliasy . . . . . 19

2.8 Wyrażenia . . . . . 21

2.9 Wartości puste (NULL) . . . . . 22

2.10 Eliminowanie duplikatów . . . . . 24

2.11 Funkcje agregujące . . . . . 24

2.12 Klauzula GROUP BY . . . . . 26

2.13 Klauzula HAVING . . . . . 28

2.14 Złączania tabel . . . . . 29

2.14.1 Iloczyn kartezjański (ang. *Cross Join*) . . . . . 29

2.14.2 Złączenia równościowe (ang. *Equi Join* lub *Inner Join*) . . . . . 30

2.14.3 Złączenia nierównościowe (ang. *Theta Join*) . . . . . 42

2.14.4 Złączenia zwrotne (ang. *Self Join*) . . . . . 44

2.14.5 Złączenia zewnętrzne (ang. *Outer Joins*) . . . . . 47

2.15 Operatory UNION, UNION ALL . . . . . 55

2.16 Podzapytania . . . . . 60

2.16.1 Podzapytania zwracające jeden rekord . . . . . 60
2.16.2 Podzapytania zwracające więcej niż jeden rekord . . . . . 61
2.16.3 Operatory ANY oraz ALL . . . . . 63
2.16.4 Podzapytania skorelowane, operatory EXISTS oraz NOT EXISTS . 65
2.16.5 Przykłady podzapytań, które można zastąpić złączeniami . . . . . 68
2.16.6 Podzapytania w klauzuli FROM . . . . . 71

**3 Funkcje formatujące** **74**
3.1 Uwagi wstępne . . . . . 74
3.2 Funkcje operujące na łańcuchach . . . . . 74
3.3 Funkcje operujące na liczbach . . . . . 79
3.4 Funkcje operujące na dacie i czasie . . . . . 80

**4 Polecenie INSERT** **83**

**5 Polecenie UPDATE** **88**

**6 Polecenie DELETE** **93**

**7 Polecenie CREATE** **96**
7.1 Tworzenie tabel . . . . . 96
7.2 Tworzenie i wykorzystywanie widoków (ang. *view*) . . . . . 100
7.3 Tworzenie ograniczeń integralnościowych (ang. *constraints*) . . . . . 101
7.4 Obsługa ograniczeń w MySQL . . . . . 110
7.5 Indeksy . . . . . 114

**8 Polecenie ALTER** **117**

**9 Polecenie DROP** **121**

**10 Model demonstracyjny** **125**

# Rozdział 1

# Uwagi wstępne

## 1.1 Na początek

Opracowanie omawia podstawowe elementy języka SQL na przykładzie serwera MySQL. W zamierzeniu autora ma ono stanowić materiał pomocniczy do prowadzenia wykładu oraz ćwiczeń laboratoryjnych z przedmiotu *Bazy danych* (oraz pokrewne). Opracowanie może być materiałem do samodzielnego studiowania, jednak należy mieć świadomość, że brak jest w nim bardziej systematycznego omówienia języka SQL. Język ten omówiono posługując się dużą liczbą (ponad 100) przykładów, ograniczając natomiast do minimum komentarz słowny. Starano się przedstawić jedynie jego najważniejsze elementy, najbardziej przydatne w praktyce. Wiele pomniejszych kwestii jest pominiętych lub omówionych tylko pobieżnie.

Wszystkie przykłady testowane były w systemie MySQL, w wersji 5.0.16, jednak powinny bez żadnych zmian działać również w nowszych wersjach. Wcześniejsze wersje mogą natomiast nie wspierać pewnych elementów języka SQL (np. podzapytania). Większość przykładów można wykonać również w innych niż MySQL systemach bazodanowych (np. ORACLE). Niektóre przykłady wykorzystują jednak specyficzne cechy MySQL-a oraz wprowadzane przez niego rozszerzenia i odstępstwa od standardu SQL i dlatego uruchomienie ich na innej bazie danych może wymagać wprowadzenia niewielkich zmian.

Szczegółowy opis wszystkich poleceń języka SQL w systemie MySQL można znaleźć w dokumentacji. Szczególnie polecamy bardzo obszerne i dokładne opracownanie [3].

Zdecydowania większość przykładów operuje na demonstracyjnym modelu, którego krótki opis zamieszczono na końcu opracowania (rozdział 10). Dlatego też należy upewnić się, że skrypt tworzący ten model wykonał się bezbłędnie.

Wszystkie przykłady pokazane w opracowaniu zostały wykonane w konsoli tekstowej serwera MySQL. Poniżej pokazano wygląd ekranu po prawidłowy uruchomieniu programu oraz po prawidłowym połączeniu się z jego pomocą do serwera MySQL (nazwa: użytkownika, jego hasło oraz nazwa bazy danych są przykładowe. Oczywiście hasło zwykle nie powinno być wpisywane „otwartym" testem. Lepiej po przełączniku `-p` nie wpisywać hasła — zostaniemy o nie poproszeni w trakcie uruchamiania programu).

W zależności od konfiguracji systemu napisy mogą być w języku innym niż angielski — domyślnie jest to oczywiście język angielski.

```
shell> mysql -u ulab -phlab blab
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 5.0.16-nt-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

W opracowaniu przyjęto generalną zasadę (poza bardzo nielicznymi wyjątkami), że:

- wszystkie polecenia języka SQL umieszczone są w ramkach i pisane są czcionką `maszynową`,
- wyniki poleceń języka SQL, komunikaty ostrzeżeń oraz komunikaty o błędach nie są umieszczane w ramkach i również pisane są czcionką `maszynową`,
- wszelkie słowa kluczowe języka SQL pisane są `DUŻYMI LITERAMI` w kolorze NIEBIESKIM,
- ciągi znakowe ujmowane w apostrofy lub cydzysłowy pisane są w kolorze czerwonym,
- nazwy funkcji pisane są `DUŻYMI LITERAMI` w kolorze OLIWKOWYM.

## 1.2 Podstawowe grupy poleceń

| Grupa | Polecenie | Opis |
|---|---|---|
| DML | SELECT | wyświetlanie rekordów |
| DML | INSERT | wstawianie rekordów |
| DML | UPDATE | modyfikacja rekordów |
| DML | DELETE | kasowanie rekordów |
| DDL | CREATE | tworzenie obiektów (np. tabel) |
| DDL | ALTER | modyfikowanie obiektów (np. tabel) |
| DDL | DROP | kasowanie obiektów (np. tabel) |
| DCL | COMMIT | zatwierdza zmiany wprowadzone za pomocą poleceń DML |
| DCL | ROLLBACK | wycofuje (anuluje) zmiany wprowadzone za pomocą poleceń DML |

Poszczególne polecenia SQL, w zależności od wykonywanych czynności, można zebrać w trzy podstawowe grupy. Są to:

- DML — Data Manipluation Language,
- DDL — Data Definition Language,
- DCL — Data Control Language.

## 1.3 Przechowywanie danych w MySQL

Pomimo, że opracowanie niniejsze w założeniach ma omawiać język SQL w oderwaniu od konkretnej implementacji, musimy jednak w bardzo wielkim skrócie omówić zagadnienie przechowywania danych w serwerze MySQL. Jest to niezbędne między innymi po to, aby zrozumieć uwagi podane w przykładzie 2.2.

Wszystkie czynności opisane poniżej wykonujemy po prawidłowym zalogowaniu się do serwera MySQL (musimy podać właściwą nazwę użytkownika oraz jego hasło). Użytkowników tworzy zwykle administrator serwera. Muszą oni posiadać odpowiednie uprawnienia, które też zwykle nadawane są im przez administratora[1].

Tabele (oraz inne obiekty, jak np. indeksy) przechowane są w tzw. *bazach danych*. W serwerze MySQL utworzenie nowej bazy danych skutkuje *powstaniem katalogu* o nazwie identycznej z nazwą tworzonej bazy danych[2]. Nie jest więc możliwe utworzenie jakiejś tabeli gdzieś poza bazą danych. Wniosek z powyższego jest taki, że aby móc rozpocząć pracę z serwerem MySQL musimy najpierw utworzyć nową bazę danych za pomocą polecenia `CREATE DATABASE` (ta czynność jest też zwykle zarezerwowana tylko dla administratora serwera MySQL) lub też pracować na jakieś istniejącej już bazie danych, do której mamy odpowiednie uprawnienia.

Następnie za pomocą polecenia `USE` musimy połączyć się z tą bazą danych, a dopiero po tych czynnościach możemy utworzyć naszą tabelę (lub inny obiekt, jednak najczęściej będzie to właśnie tabela). Utworzenie tabeli powoduje, że w katalogu z bazą danych tworzone są pliki o nazwach odpowiadających nazwom tworzonych tabel (mają one zwykle rozszerzenia *MYD*, *MYI*, *frm*).

Poniżej pokazano polecenia, które:

- kasują bazę danych (bo za chwilę będziemy tworzyli nową bazę danych a nie da się utworzyć dwóch baz danych o tych samych nazwach),
- kasujemy konto użytkownika `ulab` (a w zasadzie dwa konta — jedno do połączeń lokalnych a drugie do połączeń globalnych. Szczegóły patrz [3], rozdział *The MySQL Access Privilege System*),
- tworzymy nową bazę danych o nazwie `blab`,
- tworzymy konto użytkownika `ulab` i jednocześnie nadajemy mu pewne uprawnienia (patrz uwagi w drugim punkcie),
- łączymy się z nowoutworzoną bazą `blab`,
- w bazie danych `blab` tworzymy przykładową tabelę.

```
DROP DATABASE blab;
DROP USER 'ulab'@'localhost';
DROP USER 'ulab'@'%';
CREATE DATABASE blab;
```

[1]Uprawnienia nadajemy za pomocą polecenia `GRANT` a odbieramy za pomocą polecenia `REVOKE`. Szczegóły patrz [3].

[2]W wersji 5.x bazy danych powstają w podkatalogu *data*

```
GRANT ALL PRIVILEGES ON blab.* TO 'ulab'@'localhost' IDENTIFIED BY 'hlab';
GRANT ALL PRIVILEGES ON blab.* TO 'ulab'@'%' IDENTIFIED BY 'hlab';
USE blab;
CREATE TABLE test (id INT PRIMARY KEY);
```

Możemy łatwo przekonać się, że w podkatalogu *data* powstał kolejny podkatalog o nazwie *blab* a w nim plik o nazwie *test.frm*.

# Rozdział 2

# Polecenie SELECT

## 2.1 Składnia polecenia SELECT

Poniższy diagram zaczerpnięto wprost z [3]. Nie omawiamy go w tym miejscu a zamieszczamy tylko w celach ilustracyjnych. Większość z zamieszczonych opcji zostanie omówiona w dalszej części opracowania. Niektóre (te mniej istotne) nie będą natomiast wcale omawiane.

```
SELECT
[ALL | DISTINCT | DISTINCTROW ]
[HIGH_PRIORITY]
[STRAIGHT_JOIN]
[SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
702
[SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
select_expr, ...
[INTO OUTFILE 'file_name' export_options
| INTO DUMPFILE 'file_name']
[FROM table_references
[WHERE where_definition]
[GROUP BY {col_name | expr | position}
[ASC | DESC], ... [WITH ROLLUP]]
[HAVING where_definition]
[ORDER BY {col_name | expr | position}
[ASC | DESC] , ...]
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
[PROCEDURE procedure_name(argument_list)]
[FOR UPDATE | LOCK IN SHARE MODE]]
```

## 2.2 Najprostsze przykłady

### Przykład 1

```
SELECT * FROM region;
```

```
+----+----------------------+
```

```
| id | name                 |
+----+----------------------+
|  3 | Africa / Middle East |
|  4 | Asia                 |
|  5 | Europe               |
|  1 | North America        |
|  2 | South America        |
+----+----------------------+
```

**Komentarz:**

Słowa kluczowe języka SQL nie są czułe na wielkość liter. We wszystkich przykładach będziemy jednak stosować konsekwentnie zasadę, iż wszelkie słowa kluczowe oraz tzw. *aliasy* (o aliasach będzie mowa w rodziale 2.7) pisane będą DUŻYMI LITERAMI.

Nazwy baz danych oraz nazwy tabel są czułe na wielkość liter tylko wówczas, gdy serwer MySQL jest zainstalowany w systemie operacyjnym, który jest czuły na wielkość liter w nazwach plików oraz katalogów (chodzi tu głównie o systemy z rodziny UNIX oraz LINUX. Systemy z rodziny Windows oraz DOS „od zawsze" były nieczułe na wielkości liter w nazwach plików i katalogów). Zalecamy jednak, aby niezależnie od systemu operacyjnego, *tworzyć bazy danych oraz tabele posługując się wyłącznie małymi literami*. Dzięki temu unikniemy w przyszłości problemów z przenoszeniem baz danych MySQL między różnymi systemami operacyjnymi.

Wielkości liter nie trzeba przestrzegać w odwoływaniu się do atrybutów obiektów (np. nazwy kolumn). Jednak we wszystkich przykładach będziemy konsekwentnie używać w tych przypadkach małych liter. Będziemy więc zawsze pisać `SELECT * FROM region` zamiast na przykład `select * from region`, czy też `SELECT * from REGION` (choć wszystkie te trzy formy są w gruncie rzeczy poprawne).

Dla zwiększenia czytelności można stosować dowolną ilość spacji, tabulatorów, znaków przejścia do nowej linii. Uwaga ta staje się bardziej istotna, gdy polecenia SQL są długie i skomplikowane. Wówczas pisanie ich „ciurkiem" bardzo zmniejsza ich czytelność.

Każde polecenie SQL **musi być zakończone średnikiem**.

Gwiazdka zastępuje nazwy wszystkich kolumn. Zostaną one wyświetlone w dokładnie takiej kolejności, w jakiej występują w definicji tabeli. Nic nie stoi jednak na przeszkodzie, aby jawnie wymienić wszystkie kolumny - choćby po to, aby zmienić domyślną kolejność wyświetlania.

W MySQL-u jest możliwe jednoczesne używanie gwiazdki oraz specyfikowanie jawne nazw kolumn (inna sprawa, czy ma to jakiś sens praktyczny).

```
SELECT *, id FROM region;
```

```
+----+----------------------+----+
| id | name                 | id |
+----+----------------------+----+
|  3 | Africa / Middle East |  3 |
|  4 | Asia                 |  4 |
|  5 | Europe               |  5 |
|  1 | North America        |  1 |
|  2 | South America        |  2 |
+----+----------------------+----+
```

**Komentarz:**

Gwiazdkę można łączyć z bezpośrednim wymienianiem nazw kolumn. Inna sprawa, czy ma to jakiś sens praktyczny.

### Przykład 2

```
SELECT first_name, last_name FROM emp; # To jest komentarz jednolinijkowy
-- To też jest komentarz jednolinijkowy.
-- Po drugim minusie jest obowiązkowy biały znak (spacja, tabulator).

SELECT first_name /* komentarz 1 */, last_name  /* komentarz 2 */ FROM emp;

SELECT first_name
/*
To jest
komentarz
wielolinikowy.
*/
FROM emp;
```

Pokazano możliwe do stosowania rodzaje komentarzy. Zwróćmy tylko uwagę, że komentarz w stylu -- (dwa znaki minus) wymaga, aby *po drugim minusie był przynajmniej jeden biały znak* (spacja, tabulator).

### Przykład 3

```
SELECT name, id FROM region;
```

```
+----------------------+----+
| name                 | id |
+----------------------+----+
| Africa / Middle East |  3 |
| Asia                 |  4 |
| Europe               |  5 |
| North America        |  1 |
| South America        |  2 |
+----------------------+----+
```

**Komentarz:**

Kolumny wyświetlane są od lewej do prawej w takiej kolejności, w jakiej były wymienione w wyrażeniu SELECT (chyba, że użyto znaku gwiazdki). Kolejność wyświetlania może więc być zupełnie inna niż rzeczywisty układ kolumn w tabeli. Trzeba być tego w pełni świadomym, gdyż wynik wyświetlany na ekranie zwykle nie odzwierciedla w jednoznaczny sposób budowy poszczególnych tabel.

### Przykład 4

```
SELECT id, id, id FROM region;
```

```
+----+----+----+
| id | id | id |
+----+----+----+
|  1 |  1 |  1 |
|  2 |  2 |  2 |
|  3 |  3 |  3 |
|  4 |  4 |  4 |
|  5 |  5 |  5 |
+----+----+----+
```

**Komentarz:**

Dowolne kolumny można wyświetlać dowolną ilość razy. Inna sprawa, czy ma to jakiś sens.

### Przykład 5

```
DESCRIBE emp;
```

```
+----------------+---------------+------+-----+---------+----------------+
| Field          | Type          | Null | Key | Default | Extra          |
+----------------+---------------+------+-----+---------+----------------+
| id             | int(11)       | NO   | PRI | NULL    | auto_increment |
| last_name      | varchar(25)   | NO   |     |         |                |
| first_name     | varchar(25)   | YES  |     | NULL    |                |
| userid         | varchar(8)    | YES  | MUL | NULL    |                |
| start_date     | datetime      | YES  |     | NULL    |                |
| comments       | varchar(255)  | YES  |     | NULL    |                |
| manager_id     | int(11)       | YES  | MUL | NULL    |                |
| title          | varchar(25)   | YES  | MUL | NULL    |                |
| dept_id        | int(11)       | YES  | MUL | NULL    |                |
| salary         | decimal(11,2) | YES  |     | NULL    |                |
| commission_pct | decimal(4,2)  | YES  |     | NULL    |                |
+----------------+---------------+------+-----+---------+----------------+
11 rows in set (0.01 sec)
```

**Komentarz:**

Polecenie DESCRIBE pozwala szybko wyświetlić szczegóły budowy tabeli, takie jak:

- nazwy kolumn,
- typy kolumn,

- rodzaje ograniczeń założone na poszczególnych kolumnach,
- inne dane (np., czy kolumna ma włączoną opcję `auto_increment`).

Więcej informacji na temat tego (w sumie bardzo prostego w użyciu) polecenia można znaleźć w dokumentacji.

## 2.3 Klauzula ORDER BY

### Przykład 6

```
SELECT
  last_name, salary
FROM
  emp
ORDER BY
  salary ASC;
```

```
+--------------+---------+
| last_name    | salary  |
+--------------+---------+
| Newman       |  750.00 |
| Patel        |  795.00 |
| Patel        |  795.00 |
| Chang        |  800.00 |
| Markarian    |  850.00 |
| Dancs        |  860.00 |
| Smith        |  940.00 |
| Schwartz     | 1100.00 |
| Biri         | 1100.00 |
| Urguhart     | 1200.00 |
| Nozaki       | 1200.00 |
| Menchu       | 1250.00 |
| Catchpole    | 1300.00 |
| Havel        | 1307.00 |
| Nagayama     | 1400.00 |
| Maduro       | 1400.00 |
| Magee        | 1400.00 |
| Quick-To-See | 1450.00 |
| Ngao         | 1450.00 |
| Dumas        | 1450.00 |
| Giljum       | 1490.00 |
| Sedeghi      | 1515.00 |
| Nguyen       | 1525.00 |
| Ropeburn     | 1550.00 |
| Velasquez    | 2500.00 |
+--------------+---------+
25 rows in set (0.07 sec)
```

### Komentarz:

Klauzula `ORDER BY` służy do sortowania wyników według jedenego lub kilku wybranych kolumn. Domyślnie dane sortowane są w porządku wzrastającym (`1,2,3...` oraz `a,b,c...`). Dlatego też słowo kluczowe `ASC` można pominąć — choć, gdy je jawnie wyspecyfikujemy nic złego się nie stanie.

### Przykład 7

```
SELECT last_name, salary FROM emp ORDER BY salary DESC;
```

```
+--------------+---------+
| last_name    | salary  |
+--------------+---------+
| Velasquez    | 2500.00 |
| Ropeburn     | 1550.00 |
| Nguyen       | 1525.00 |
| Sedeghi      | 1515.00 |
| Giljum       | 1490.00 |
| Quick-To-See | 1450.00 |
| Dumas        | 1450.00 |
| Ngao         | 1450.00 |
| Magee        | 1400.00 |
| Nagayama     | 1400.00 |
| Maduro       | 1400.00 |
| Havel        | 1307.00 |
| Catchpole    | 1300.00 |
| Menchu       | 1250.00 |
| Nozaki       | 1200.00 |
| Urguhart     | 1200.00 |
| Schwartz     | 1100.00 |
| Biri         | 1100.00 |
| Smith        |  940.00 |
| Dancs        |  860.00 |
| Markarian    |  850.00 |
| Chang        |  800.00 |
| Patel        |  795.00 |
| Patel        |  795.00 |
| Newman       |  750.00 |
+--------------+---------+
25 rows in set (0.00 sec)
```

**Komentarz:**

Chcą otrzymać dane posortowane „od największego do najmniejszego” musimy użyć słowa kluczowego `DESC`.

### Przykład 8

```
SELECT
  name, region_id
FROM
  dept
ORDER BY
```

```
  region_id DESC,
  name ASC;
```

```
+----------------+-----------+
| name           | region_id |
+----------------+-----------+
| Operations     |         5 |
| Sales          |         5 |
| Operations     |         4 |
| Sales          |         4 |
| Operations     |         3 |
| Sales          |         3 |
| Operations     |         2 |
| Sales          |         2 |
| Administration |         1 |
| Finance        |         1 |
| Operations     |         1 |
| Sales          |         1 |
+----------------+-----------+
12 rows in set (0.00 sec)
```

**Komentarz:**

Słowo kluczowe `ASC` można pominąć, bo jest to opcja domyślna. Słowa kluczowego `DESC` pominąć nie można. Sortowanie odbywa się najpierw według pierwszej wymienionej kolumnie, a następnie według drugiej.

## 2.4 Klauzula WHERE

### Przykład 9

```
SELECT
  last_name, salary
FROM
  emp
WHERE
  salary > 1500
ORDER BY
  salary DESC;
```

```
+-----------+---------+
| last_name | salary  |
+-----------+---------+
| Velasquez | 2500.00 |
| Ropeburn  | 1550.00 |
| Nguyen    | 1525.00 |
| Sedeghi   | 1515.00 |
+-----------+---------+
```

**Komentarz:**

Klauzula `WHERE` służy do ograniczania ilości wyświetlanych rekordów. Podany warunek logiczny może być w zasadzie dowolnie złożony. Można używać wszystkich dostępnych operatorów (będzie jeszcze o tym mowa poniżej).

W miarę zwiększania się wielkości zapytania SQL, warto stosować wcięcia i przejścia do nowej linii. Zdecydowanie zwiększa to czytelność kodu! W niniejszym opracowaniu zastosowane wcięcia i przejścia do nowej linii są kompromisem pomiędzy czytelnością a długością zapisu.

### Przykład 10

```
SELECT
  name, credit_rating
FROM
  customer
WHERE
  credit_rating LIKE 'EXCELLENT';
```

```
+----------------------------+---------------+
| name                       | credit_rating |
+----------------------------+---------------+
| Unisports                  | EXCELLENT     |
| Womansport                 | EXCELLENT     |
| Kam's Sporting Goods       | EXCELLENT     |
| Sportique                  | EXCELLENT     |
| Beisbol Si!                | EXCELLENT     |
| Futbol Sonora              | EXCELLENT     |
| Kuhn's Sports              | EXCELLENT     |
| Hamada Sport               | EXCELLENT     |
| Big John's Sports Emporium | EXCELLENT     |
+----------------------------+---------------+
9 rows in set (0.00 sec)
```

### Komentarz:

Gdy w klauzuli `WHERE` odnosimy się do ciągów znaków, musimy ujmować je w apostrofy (nie cudzysłowy!). W MySQL-u domyślnie nie są uwzględniane wielkości liter, więc gdy w klauzuli `WHERE` przykładowo wpiszemy `'ExceLLent'` otrzymamy identyczny wynik.

Aby porównywanie ciągów uwzględniało wielkość liter, musimy dodać do polecenia słowo kluczowe `BINARY`. Możemy również w trakcie tworzenia tabel od razu zdefiniować daną kolumnę w taki sposób, aby zawsze porównywanie uwzględniało wielkość liter. Przykładowo zamiast definicji `VARCHAR(30)` możemy podać `VARCHAR(30) BINARY`. Wówczas nie trzeba w poleceniach SQL używać słowa kluczowego `BINARY`. Porównajmy:

```
SELECT
  name, credit_rating
FROM
  customer
WHERE
  credit_rating LIKE BINARY 'ExCeLLeNT';
```

```
+----------------------------+---------------+
```

```
| name                        | credit_rating |
+-----------------------------+---------------+
| Unisports                   | EXCELLENT     |
| Womansport                  | EXCELLENT     |
| Kam's Sporting Goods        | EXCELLENT     |
| Sportique                   | EXCELLENT     |
| Beisbol Si!                 | EXCELLENT     |
| Futbol Sonora               | EXCELLENT     |
| Kuhn's Sports               | EXCELLENT     |
| Hamada Sport                | EXCELLENT     |
| Big John's Sports Emporium  | EXCELLENT     |
+-----------------------------+---------------+
9 rows in set (0.00 sec)
```

Jak widać zapytanie i tym razem zwróciło ten sam zestaw rekordów.

### Przykład 11

```sql
SELECT
  first_name, last_name, start_date
FROM
  emp
WHERE
  start_date > '1992-01-01';
```

```
+------------+-----------+---------------------+
| first_name | last_name | start_date          |
+------------+-----------+---------------------+
| Antoinette | Catchpole | 1992-02-09 00:00:00 |
| Henry      | Giljum    | 1992-01-18 00:00:00 |
| Mai        | Nguyen    | 1992-01-22 00:00:00 |
| Elena      | Maduro    | 1992-02-07 00:00:00 |
+------------+-----------+---------------------+
```

### Komentarz:

W MySQL daty powinny być zawsze podawane w formacie `RRRR-MM-DD`. Gdy użyjemy innego formatu, wynik może być błędny a serwer MySQL nie zasygnalizuje tego. Porównajmy:

```sql
SELECT
  first_name, last_name, start_date
FROM
  emp
WHERE
  start_date > '01-01-1992';
```

```
+------------+--------------+---------------------+
| first_name | last_name    | start_date          |
+------------+--------------+---------------------+
| Carmen     | Velasquez    | 1990-03-03 00:00:00 |
| LaDoris    | Ngao         | 1990-03-08 00:00:00 |
| Midori     | Nagayama     | 1991-06-17 00:00:00 |
```

```
| Mark        | Quick-To-See | 1990-04-07 00:00:00 |
| Audry       | Ropeburn     | 1990-03-04 00:00:00 |
| Molly       | Urguhart     | 1991-01-18 00:00:00 |

...

| Vikram      | Patel        | 1991-08-06 00:00:00 |
| Chad        | Newman       | 1991-07-21 00:00:00 |
| Alexander   | Markarian    | 1991-05-26 00:00:00 |
| Eddie       | Chang        | 1990-11-30 00:00:00 |
| Radha       | Patel        | 1990-10-17 00:00:00 |
| Bela        | Dancs        | 1991-03-17 00:00:00 |
| Sylvie      | Schwartz     | 1991-05-09 00:00:00 |
+-------------+--------------+---------------------+
25 rows in set, 1 warning (0.00 sec)
```

**Przykład 12**

```
SELECT
  CURRENT_TIMESTAMP,
  DATE_FORMAT(CURRENT_TIMESTAMP, '%W :: %M :: %d :: %Y :: %T')
  AS "biezaca data i godzina";
```

```
+---------------------+--------------------------------------------+
| CURRENT_TIMESTAMP   | biezaca data i godzina                     |
+---------------------+--------------------------------------------+
| 2006-03-28 11:57:20 | Tuesday :: March :: 28 :: 2006 :: 11:57:20 |
+---------------------+--------------------------------------------+
```

**Komentarz:**

Używając funkcji `DATE_FORMAT` można wyświetlić datę oraz godzinę w praktycznie dowolnym formacie. Decyduje o tym postać tzw. maski, czyli drugi parametr funkcji `DATE_FORMAT`.

## 2.5 Ograniczanie wyników wyszukiwania za pomocą klauzuli LIMIT

**Przykład 13**

```
SELECT last_name FROM emp LIMIT 5;
```

```
+--------------+
| last_name    |
+--------------+
| Velasquez    |
| Ngao         |
| Nagayama     |
| Quick-To-See |
| Ropeburn     |
+--------------+
```

**Komentarz:**

Otrzymaliśmy pięć pierwszych wierszy zwracanych przez polecenie SELECT.

**Przykład 14**

```
SELECT last_name FROM emp LIMIT 5,4;
```

```
+-----------+
| last_name |
+-----------+
| Urguhart  |
| Menchu    |
| Biri      |
| Catchpole |
+-----------+
```

**Komentarz:**

Tym razem w klauzuli LIMIT użyliśmy 2. parametrów. Wyświetlamy więc 4 rekordy pominąwszy 5 pierwszych rekordów. Opcja ta bardzo przydaje się przy tworzeniu aplikacji internetowych. Zwykle jest ona używana łącznie z klauzulą GROUP BY, aby kolejność w jakiej zwracane są wiersze, miała jakiś sens. Bez GROUP BY rekordy są zwracane w takiej kolejności, w jakiej występują fizycznie w tabelach.

## 2.6 Operatory i funkcje porównywania

**Przykład 15**

Poniżej zamieszczono listę najczęściej wykorzystywanych w MySQL operatorów i funkcji służących do porównywania wartości oraz kilkanaście podstawowych przykładów ich użycia.

**operatory arytmetyczne:**

```
+, -, *, /, DIV, %, MOD
```

**operatory i funkcje porównania:**

```
=, <=>, >=, >, <=, <, <>, !=
IS boolean_value
IS NOT boolean_value
IS NULL
IS NOT NULL
expr BETWEEN minval AND maxval
expr NOT BETWEEN minval AND maxval
COALESCE(val,...)
GREATEST(value1,value2,...)
expr IN (val,...)
expr NOT IN (val,...)
IFNULL(expr)
INTERVAL(N,N1,N2,N3,...)
LEAST(value1,value2,...)
```

**operatory logiczne:**

```
NOT !
AND &&
OR ||
XOR
```

**operatory i funkcje sterujące przepływem:**

```
CASE value WHEN [compare-val] THEN result [WHEN
[compare-val] THEN result ...] [ELSE result] END CASE WHEN
[condition] THEN result [WHEN [condition] THEN result ...] [ELSE
result] END

IF(expr1,expr2,expr3)
IFNULL(expr1,expr2)
NULLIF(expr1,expr2)
```

**przykłady użycia:**

```
SELECT last_name FROM emp WHERE salary >= 1500;

SELECT last_name FROM emp WHERE salary BETWEEN 1000 AND 2000;

SELECT last_name FROM emp WHERE salary >= 1000 AND salary <= 2000;

SELECT name FROM dept WHERE name IN ('sales', 'finance');

SELECT last_name FROM emp WHERE id IN (1, 10, 20);

SELECT last_name FROM emp WHERE last_name LIKE 'N%';

SELECT last_name FROM emp WHERE last_name LIKE '____'; -- 4 znaki podkreślenia

SELECT last_name FROM emp WHERE last_name LIKE 'N___'; -- 3 znaki podkreślenia

SELECT last_name FROM emp WHERE last_name LIKE 'N__a'; -- 2 znaki podkreślenia

SELECT last_name FROM emp WHERE manager_id = 3 AND salary > 1000;

-- Zwróćmy uwagę, że nie ma spacji pomiędzy nazwą funkcji a nawiasem.
SELECT IFNULL(salary,0) FROM emp;
```

**Komentarz:**

Operatory w klauzuli `WHERE` stosujemy do zawężania ilości wyświetlanych rekordów. Funkcja `IFNULL` jest bardzo często wykorzystywana w praktyce.

## 2.7 Aliasy

**Przykład 16**

```sql
SELECT
  first_name AS Imie,
  last_name AS "Nazwisko",
  start_date AS "Data zatrudnienia"
FROM
  emp
WHERE
  dept_id = 41;
```

```
+---------+----------+---------------------+
| Imie    | Nazwisko | Data zatrudnienia   |
+---------+----------+---------------------+
| LaDoris | Ngao     | 1990-03-08 00:00:00 |
| Molly   | Urguhart | 1991-01-18 00:00:00 |
| Elena   | Maduro   | 1992-02-07 00:00:00 |
| George  | Smith    | 1990-03-08 00:00:00 |
+---------+----------+---------------------+
```

**Komentarz:**

Alias pozwala nam zmieniać nagłówki kolumn. Aliasy możemy poprzedzić słowem kluczowym `AS` (choć nie jest to konieczne). Wydaje się, że warto jednak je stosować, gdyż wówczas unikamy sytuacji, gdy omyłkowe pominięcie przecinka rozdzielającego kolumny w klauzuli `SELECT` może prowadzić do tworzenia „niechcianych” aliasów. Porównajmy:

```sql
SELECT first_name, last_name FROM emp;  -- jest przecinek
SELECT first_name last_name FROM emp;   -- brak przecinka
```

O ile w drugim aliasie użycie cudzysłowów jest zbędne (choć, gdy pojawią się nic złego się nie stanie), o tyle w aliasie trzecim jest to konieczne. Jest tak dlatego, że definiowany alias zawiera białe znaki i bez cudzysłowów polecenie SQL byłoby błędne. Porównajmy:

```sql
SELECT
  first_name AS Imie,
  last_name AS "Nazwisko",
  start_date AS Data zatrudnienia
FROM
  emp
WHERE
  dept_id = 41;
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near 'zatru
dnienia
FROM
  emp
WHERE dept_id = 41' at line 4
mysql>
```

### Przykład 17

```sql
SELECT R.name FROM region AS R;
```

```
+-----------------------+
| name                  |
+-----------------------+
| Africa / Middle East |
| Asia                  |
| Europe                |
| North America         |
| South America         |
+-----------------------+
```

**Komentarz:**

Aliasy można również stosować do nazw tabel. Praktyczną przydatność tej możliwości poznamy, gdy będziemy omawiać złączenia tabel (patrz rozdział 2.14). W powyższym przykładzie użyty alias praktycznie nie wprowadza nic nowego a tylko niepotrzebnie wydłuża zapis.

## 2.8 Wyrażenia

### Przykład 18

```
SELECT
  last_name "Nazwisko", salary*12+100
FROM
  emp
WHERE
  dept_id = 41;
```

```
+----------+---------------+
| Nazwisko | salary*12+100 |
+----------+---------------+
| Ngao     |      17500.00 |
| Urguhart |      14500.00 |
| Maduro   |      16900.00 |
| Smith    |      11380.00 |
+----------+---------------+
```

**Komentarz:**

W poleceniu `SELECT` można używać w zasadzie dowolnych wyrażeń — powyżej pole `SALARY` pomnożono przez 12 i dodano 100. W tym przykładzie aż prosi się o zastosowanie aliasu dla tego wyrażenia.

### Przykład 19

```
SELECT id, name*100 FROM region;
```

```
+----+----------+
| id | name*100 |
+----+----------+
|  3 |        0 |
|  4 |        0 |
```

```
|  5 |         0 |
|  1 |         0 |
|  2 |         0 |
+----+----------+
```

**Komentarz:**

Próba pomnożenia pola znakowego przez liczbę jest działaniem niewątpliwie błędnym logicznie. Serwer MySQL jest jednak wyrozumiały i wyświetla „jakiś” wynik.

## 2.9 Wartości puste (NULL)

### Przykład 20

```
SELECT name
FROM customer
WHERE country IS NULL;
```

```
+----------------------+
| name                 |
+----------------------+
| Kam's Sporting Goods |
+----------------------+
```

**Komentarz:**

Wartość `NULL` oznacza, że w danej komórce nie ma żadnych danych. Najczęściej ma to znaczenie mniej więcej zbliżone do: „wartość nieznana w tym momencie”.

Wpisanie do komórki np. jednej spacji, mimo tego, że też jej nie widać na wydruku, nie jest tym samym co pozostawienie w niej wartości `NULL`. Testowania wartości `NULL` nie można wykonać w taki sposób jak poniżej. Trzeba użyć zwrotu `IS NULL`:

```
SELECT name
FROM customer
WHERE country = '';
```

```
Empty set (0.00 sec)

mysql>
```

### Przykład 21

```
SELECT name FROM customer WHERE state IS NOT NULL;
```

```
+---------------------------+
| name                      |
+---------------------------+
| Womansport                |
| Big John's Sports Emporium |
| Ojibway Retail            |
+---------------------------+
```

**Komentarz:**

Można też zrobić tak jak poniżej (tylko po co, skoro `IS NOT NULL` brzmi bardziej elegancko i jest chyba bardziej czytelnie):

```
SELECT name FROM customer WHERE state LIKE '%';
```

```
+----------------------------+
| name                       |
+----------------------------+
| Womansport                 |
| Big John's Sports Emporium |
| Ojibway Retail             |
+----------------------------+
```

### Przykład 22

```
SELECT IFNULL(state, '-'), IFNULL(country, '?')
FROM warehouse;
```

```
+-------------------+---------------------+
| IFNULL(state,'-') | IFNULL(country,'?') |
+-------------------+---------------------+
| WA                | USA                 |
| -                 | Brazil              |
| -                 | Nigeria             |
| -                 | ?                   |
| -                 | Czechozlovakia      |
+-------------------+---------------------+
```

**Komentarz:**

Funkcja `IFNULL` jest bardzo przydatna, gdy chcemy, aby wartość `NULL` była jakoś wyróżniona w wyświetlanym wyniku. Jest ona również bardzo przydatna, gdy odsługiwane są wartości numeryczne — wówczas jej użycie pozwala uniknąć wielu niespodziewanych błędów (wszelkie operacje arytmetyczne na wartościach `NULL` dają w wyniku też wartość `NULL`). Porównajmy:

```
SELECT commission_pct*10 FROM emp WHERE salary > 1500;
```

```
+-------------------+
| commission_pct*10 |
+-------------------+
|              NULL |
|              NULL |
|            100.00 |
|            150.00 |
+-------------------+
```

```
SELECT IFNULL(commission_pct*10, 0) FROM emp WHERE salary > 1500;
```

```
+------------------------------+
| IFNULL(commission_pct*10, 0) |
```

```
+------------------------------+
|                         0.00 |
|                         0.00 |
|                       100.00 |
|                       150.00 |
+------------------------------+
```

## 2.10 Eliminowanie duplikatów

**Przykład 23**

```sql
SELECT DISTINCT name FROM dept;
```

```
+----------------+
| name           |
+----------------+
| Administration |
| Finance        |
| Operations     |
| Sales          |
+----------------+
```

**Komentarz:**

Słowo kluczowe `DISTINCT` (zamiennie można używać słowa kluczowego `UNIQUE` — są to równorzędne synonimy) pozwala usunąć z wyświetlanego wyniku duplikaty. W naszym zapytaniu interesowały nas wszystkie nazwy stanowisk a to, że pewne z nich są przypisane do więcej niż jednego pracownika nie jest dla nas w tym momencie istotne. W wyniku użycia operatora `DISTINCT` wynikowe rekordy są dodatkowo sortowane. Można też używać słowa kluczowego `ALL`, które powoduje wyświetlenie wszystkich rekordów w tabeli. Jest ono przyjmowane domyślnie i z tego powodu nie ma sensu wpisywać go jawnie.

## 2.11 Funkcje agregujące

**Przykład 24**

```sql
SELECT MAX(salary) "Zarobki MAX" FROM emp;
```

```
+-------------+
| Zarobki MAX |
+-------------+
|     2500.00 |
+-------------+
```

**Komentarz:**

Używamy funkcji `MAX`, aby wypisać najwyższe zarobki.

**Przykład 25**

```
SELECT MAX(salary), MIN(salary), AVG(salary), SUM(salary), COUNT(salary)
FROM emp;
```

```
+-------------+-------------+-------------+-------------+---------------+
| MAX(salary) | MIN(salary) | AVG(salary) | SUM(salary) | COUNT(salary) |
+-------------+-------------+-------------+-------------+---------------+
|     2500.00 |      750.00 | 1255.080000 |    31377.00 |            25 |
+-------------+-------------+-------------+-------------+---------------+
```

**Komentarz:**

Polecenie `COUNT(SALARY)` podaje całkowitą liczbę rekordów spełniających warunki zapytania. Pozostałe funkcje są samodokumentujące się.

### Przykład 26

```
SELECT COUNT(*)
FROM emp
WHERE salary > 1500;
```

```
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
```

**Komentarz:**

Uzyskujemy informację o tym, ile rekordów spełnia warunek `WHERE`. Zamiast gwiazdki można użyć nazwy dowolnej kolumny z tabeli `EMP` a nawet dowolnego wyrażenia stałego. Porównajmy:

```
SELECT COUNT('cokolwiek')
FROM emp
WHERE salary > 1500;
```

```
+--------------------+
| COUNT('cokolwiek') |
+--------------------+
|                  4 |
+--------------------+
```

```
SELECT COUNT(id)
FROM emp
WHERE salary > 1500;
```

```
+-----------+
| COUNT(id) |
+-----------+
|         4 |
+-----------+
```

## 2.12 Klauzula GROUP BY

### Przykład 27

```
SELECT
  dept_id, SUM(salary), "wyrazenie stale"
FROM
  emp
GROUP BY
  dept_id;
```

```
+---------+-------------+-----------------+
| dept_id | SUM(salary) | wyrazenie stale |
+---------+-------------+-----------------+
|      10 |     1450.00 | wyrazenie stale |
|      31 |     2800.00 | wyrazenie stale |
|      32 |     1490.00 | wyrazenie stale |
|      33 |     1515.00 | wyrazenie stale |
|      34 |     2320.00 | wyrazenie stale |
|      35 |     1450.00 | wyrazenie stale |
|      41 |     4990.00 | wyrazenie stale |
|      42 |     3245.00 | wyrazenie stale |
|      43 |     2700.00 | wyrazenie stale |
|      44 |     2100.00 | wyrazenie stale |
|      45 |     3267.00 | wyrazenie stale |
|      50 |     4050.00 | wyrazenie stale |
+---------+-------------+-----------------+
12 rows in set (0.00 sec)
```

**Komentarz:**

Gdy używamy funkcji grupującej `GROUP BY` musi ona wystąpić po klauzuli `WHERE`. W klauzuli `SELECT` mogą wówczas wystąpić tylko (porównaj przykład powyżej):

- funkcje agregujące (np. `SUM, MIN, MAX, AVG, COUNT`),
- nazwy kolumn występujące w funkcji grupującej `GROUP BY`,
- wyrażenia stałe.

**Uwaga:** MySQL zmienia zdefiniowaną normą SQL funkcjonalność klauzuli `GROUP BY`. Można mianowicie w klauzuli `SELECT` wpisywać kolumny, które nie są wymienione w `GROUP BY`. Może to jednak prowadzić do „dziwnych” (trudnych w interpretacji i mylących zarazem) wyników. Porównajmy:

```
SELECT
  first_name, dept_id, SUM(salary)
FROM
  emp
GROUP BY
  dept_id;
```

```
+------------+---------+-------------+
| first_name | dept_id | SUM(salary) |
+------------+---------+-------------+
| Mark       |      10 |     1450.00 |
| Midori     |      31 |     2800.00 |
| Henry      |      32 |     1490.00 |
| Yasmin     |      33 |     1515.00 |
| Mai        |      34 |     2320.00 |
| Andre      |      35 |     1450.00 |
| LaDoris    |      41 |     4990.00 |
| Roberta    |      42 |     3245.00 |
| Ben        |      43 |     2700.00 |
| Antoinette |      44 |     2100.00 |
| Marta      |      45 |     3267.00 |
| Carmen     |      50 |     4050.00 |
+------------+---------+-------------+
12 rows in set (0.01 sec)
```

Jak interpretować pierwszą kolumnę w powyższym wyniku? Prawda, że dziwne?

### Przykład 28

```
SELECT title, SUM(salary), COUNT(*)
FROM emp
GROUP BY title;
```

```
+----------------------+-------------+----------+
| title                | SUM(salary) | COUNT(*) |
+----------------------+-------------+----------+
| President            |     2500.00 |        1 |
| Sales Representative |     7380.00 |        5 |
| Stock Clerk          |     9490.00 |       10 |
| VP, Administration   |     1550.00 |        1 |
| VP, Finance          |     1450.00 |        1 |
| VP, Operations       |     1450.00 |        1 |
| VP, Sales            |     1400.00 |        1 |
| Warehouse Manager    |     6157.00 |        5 |
+----------------------+-------------+----------+
8 rows in set (0.00 sec)
```

### Komentarz:

W powyższym przykładzie wyświetliliśmy sumę zarobków wszystkich pracowników pracujących na poszczególnych stanowiskach. Dodatkowo wyświetliliśmy informację o tym, ilu pracowników pracuje na każdym ze stanowisk.

### Przykład 29

```
SELECT title, COUNT(*)
FROM emp
GROUP BY title DESC;
```

```
+----------------------+----------+
| title                | COUNT(*) |
+----------------------+----------+
| Warehouse Manager    |        5 |
| VP, Sales            |        1 |
| VP, Operations       |        1 |
| VP, Finance          |        1 |
| VP, Administration   |        1 |
| Stock Clerk          |       10 |
| Sales Representative |        5 |
| President            |        1 |
+----------------------+----------+
8 rows in set (0.00 sec)
```

**Komentarz:**

Kolejną zmianą w stosunku do standardowego zachowania się `GROUP BY` jest możliwość sortowania kolejności grup. Domyślna kolejność jest rosnąca. W powyższym przykładzie zmieniliśmy kierunek sortowania na malejący (od 'Z' do 'A').

### Przykład 30

```sql
SELECT title, SUM(salary)
FROM emp
WHERE SUM(salary) > 2000
GROUP BY title;
```

```
ERROR 1111 (HY000): Invalid use of group function
```

**Komentarz:**

W tym przykładzie (niestety błędnym!) staraliśmy się wyświetlić tylko sumę zarobków pracowników z tych stanowisk, gdzie ta suma jest większa niż 2000. Okazuje się, że aby wykonać takie zadanie należy użyć klauzuli `HAVING` (patrz niżej).

## 2.13 Klauzula HAVING

### Przykład 31

```sql
SELECT title, SUM(salary)
FROM emp
GROUP BY title
HAVING SUM(salary) > 2000;
```

```
+----------------------+-------------+
| title                | SUM(salary) |
+----------------------+-------------+
| President            |     2500.00 |
| Sales Representative |     7380.00 |
| Stock Clerk          |     9490.00 |
| Warehouse Manager    |     6157.00 |
+----------------------+-------------+
```

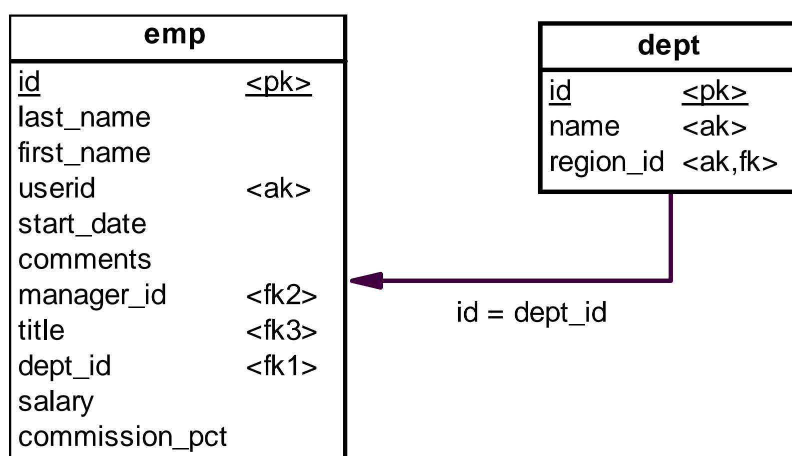
**Komentarz:**

Zasada działania klauzuli `HAVING` jest następująca: podczas tworzenia każdej grupy (za pomocą `GROUP BY`) obliczana jest suma zarobków (`SUM(salary)`) dla tej grupy. Kiedy warunek logiczny sprawdzany z pomocą klauzuli `HAVING` jest spełniony, taka grupa jest uwzględniana w wyniku i wyświetlana.

## 2.14 Złączania tabel

### 2.14.1 Iloczyn kartezjański (ang. *Cross Join*)

**Przykład 32**




```sql
SELECT
  E.first_name, E.last_name, D.name
FROM
  emp E, dept D;
```

```
+------------+-------------+----------------+
| first_name | last_name   | name           |
+------------+-------------+----------------+
| Carmen     | Velasquez   | Administration |
| Carmen     | Velasquez   | Finance        |
| Carmen     | Velasquez   | Operations     |

...

| Carmen     | Velasquez   | Operations     |
| Sylvie     | Schwartz    | Sales          |
| Sylvie     | Schwartz    | Sales          |
| Sylvie     | Schwartz    | Sales          |
+------------+-------------+----------------+
300 rows in set (0.00 sec)
```

**Komentarz:**

W tym przykładzie zapomniano o warunku złączeniowym. Efektem jest iloczyn kartezjański relacji `emp` oraz `dept`. Wynik liczy 300 rekordów (25 rekordów w tabeli `emp` razy 12 rekordów w tabeli `dept`). Gdyby złączanych tabel było więcej, należy liczyć się z ogromną liczbą (bezsensownych!) wynikowych rekordów. Może to czasami prowadzić do zawieszenia się całego systemu bazodanowego! Zapytanie zwracające iloczyn kartezjański nazywa się w terminologii bazodanowej *cross–join*.

### 2.14.2 Złączenia równościowe (ang. *Equi Join* lub *Inner Join*)

**Przykład 33**

```
SELECT
  emp.first_name, emp.last_name, dept.name
FROM
  emp, dept
WHERE
  emp.dept_id = dept.id;
```

```
+------------+--------------+----------------+
| first_name | last_name    | name           |
+------------+--------------+----------------+
| Carmen     | Velasquez    | Administration |
| Audry      | Ropeburn     | Administration |
| Mark       | Quick-To-See | Finance        |
| LaDoris    | Ngao         | Operations     |
| Molly      | Urguhart     | Operations     |
| Elena      | Maduro       | Operations     |
| George     | Smith        | Operations     |
| Roberta    | Menchu       | Operations     |
| Akira      | Nozaki       | Operations     |
| Vikram     | Patel        | Operations     |
| Ben        | Biri         | Operations     |
| Chad       | Newman       | Operations     |
| Alexander  | Markarian    | Operations     |
| Antoinette | Catchpole    | Operations     |
| Eddie      | Chang        | Operations     |
| Marta      | Havel        | Operations     |
| Bela       | Dancs        | Operations     |
| Sylvie     | Schwartz     | Operations     |
| Midori     | Nagayama     | Sales          |
| Colin      | Magee        | Sales          |
| Henry      | Giljum       | Sales          |
| Yasmin     | Sedeghi      | Sales          |
| Mai        | Nguyen       | Sales          |
| Radha      | Patel        | Sales          |
| Andre      | Dumas        | Sales          |
+------------+--------------+----------------+
25 rows in set (0.02 sec)
```

**Komentarz:**

Dwie tabele wiążemy ze sobą używając tzw. kolumny wiążącej (najczęściej są to klucz główny w jednej tabeli i klucz obcy w drugiej). W naszym przykładzie wiążemy ze sobą każdy wiersz z tabeli `emp` z odpowiadającym mu wierszem z tabeli `dept`. Jak łatwo zauważyć w zapytaniu nie uwzględniliśmy faktu, że poszczególne oddziały firmy (tabela `dept`) są jeszcze pogrupowane w regiony (tabela `region`). Stąd mamy np. wiele rekordów z danymi pracowników pracujących w dziale o nazwie *Operations* ale nie jesteśmy w stanie odczytać w jakich regionach te oddziały się znajdują.

W poleceniu `SELECT` nazwy kolumn poprzedziliśmy nazwą tabeli. Gdy nazwy kolumn są unikalne, to nie jest to konieczne, aczkolwiek zalecane.

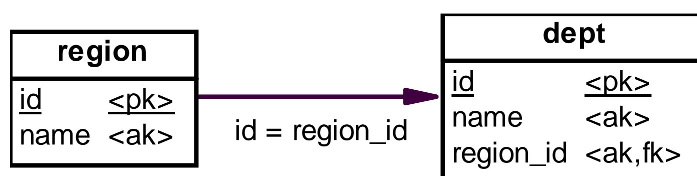
W klauzuli `WHERE` nie możemy już opuścić nazw kolumn (chyba, że nazwy kolumn są unikalne).

Dla zwartości zapisu można zamiast całych nazw tabel zdefiniować aliasy.

Problem właściwego użycia złączeń jest niezwykle istotny w zapytaniach SQL. Gdy popełnimy choćby najmniejszy błąd (np. złączając 10 tabel zapomnimy o jednym tylko złączeniu) system wygeneruje błędne wyniki i niejednokrotnie będzie bardzo trudno te błędy zauważyć. Zapytanie SQL będzie bowiem *poprawne składniowo ale błędne merytorycznie*. Serwer SQL nie wygeneruje więc żadnego błędu lub ostrzeżenia, tylko po prostu zwróci niewłaściwe dane. W kolejnych przykładach wrócimy jeszcze do tego problemu.

Tego typu złączenie nazywane jest złączeniem równościowym (ang. *natural–join*).

### Przykład 34




```
SELECT
  name, name
FROM
  region, dept
WHERE
  id = region_id;
```

```
ERROR 1052 (23000): Column 'name' in field list is ambiguous
```

```
SELECT
  name, name
FROM
  region, dept
WHERE
  region.id = dept.region_id;
```

```
ERROR 1052 (23000): Column 'name' in field list is ambiguous
```

**Komentarz:**

Tutaj nie można opuścić nazw tabel. System nie jest w stanie rozstrzygnąć, o którą kolumnę o nazwie `name` chodzi — w obu użytych tabelach jest kolumna o takiej nazwie. Zapytanie, aby wykonało się, musi wyglądać następująco:

```
SELECT
  region.name, dept.name
FROM
  region, dept
WHERE
  region.id = dept.region_id
ORDER BY
  region.name;
```

```
+----------------------+----------------+
| name                 | name           |
+----------------------+----------------+
| Africa / Middle East | Sales          |
| Africa / Middle East | Operations     |
| Asia                 | Sales          |
| Asia                 | Operations     |
| Europe               | Sales          |
| Europe               | Operations     |
| North America        | Finance        |
| North America        | Sales          |
| North America        | Operations     |
| North America        | Administration |
| South America        | Sales          |
| South America        | Operations     |
+----------------------+----------------+
12 rows in set (0.01 sec)
```

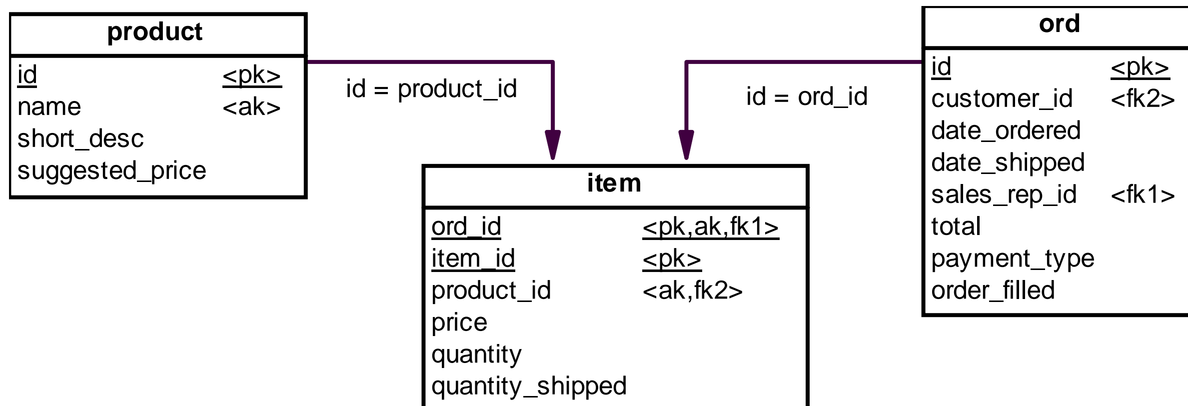
Wersja z aliasami będzie natomiast wyglądała następująco (pominięto słowa kluczowe `AS`):

```
SELECT
  R.name "Nazwa regionu", D.name "Nazwa działu"
FROM
  region R, dept D
WHERE
  R.id = D.region_id
ORDER BY
  R.name;
```

**Komentarz:**

Zapytanie udało się wreszcie wykonać, gdyż we właściwy sposób odwołano się do poszczególnych kolumn w tabelach. Aby zapis zapytania był bardziej zwarty w drugim przypadku użyto aliasów dla nazw tabel oraz utworzono aliasy dla nazw kolumn. Dzięki temu jesteśmy w stanie zorientować się która kolumna co oznacza.

### Przykład 35




```sql
SELECT
  O.id, O.total, I.price, I.quantity, I.price * I.quantity, P.name
FROM
  ord O, item I, product P
WHERE
  O.id = I.ord_id       AND
  P.id = I.product_id   AND
  O.id = 100;
```

```
+-----+-----------+--------+----------+--------------------+------------------------+
| id  | total     | price  | quantity | I.price * I.quantity | name                 |
+-----+-----------+--------+----------+--------------------+------------------------+
| 100 | 601100.00 | 135.00 |      500 |           67500.00 | Bunny Boot             |
| 100 | 601100.00 | 380.00 |      400 |          152000.00 | Pro Ski Boot           |
| 100 | 601100.00 |  14.00 |      500 |            7000.00 | Bunny Ski Pole         |
| 100 | 601100.00 |  36.00 |      400 |           14400.00 | Pro Ski Pole           |
| 100 | 601100.00 | 582.00 |      600 |          349200.00 | Himalaya Bicycle       |
| 100 | 601100.00 |  20.00 |      450 |            9000.00 | New Air Pump           |
| 100 | 601100.00 |   8.00 |      250 |            2000.00 | Prostar 10 Pound Weight |
+-----+-----------+--------+----------+--------------------+------------------------+
7 rows in set (0.00 sec)
```

### Komentarz:

Bardzo częsty w praktyce przypadek, gdy złączamy dane z tabel, które są ze sobą w relacji `N:N` (tutaj tabele `ord` oraz `product`). Dla sprawdzenia, czy dane w tabelach `ord` oraz `item` są spójne możemy wykonać poniższe zapytanie (czy domyślasz się o jaką spójność danych chodzi?):

```sql
SELECT  SUM(price * quantity)
FROM item
WHERE ord_id = 100;
```

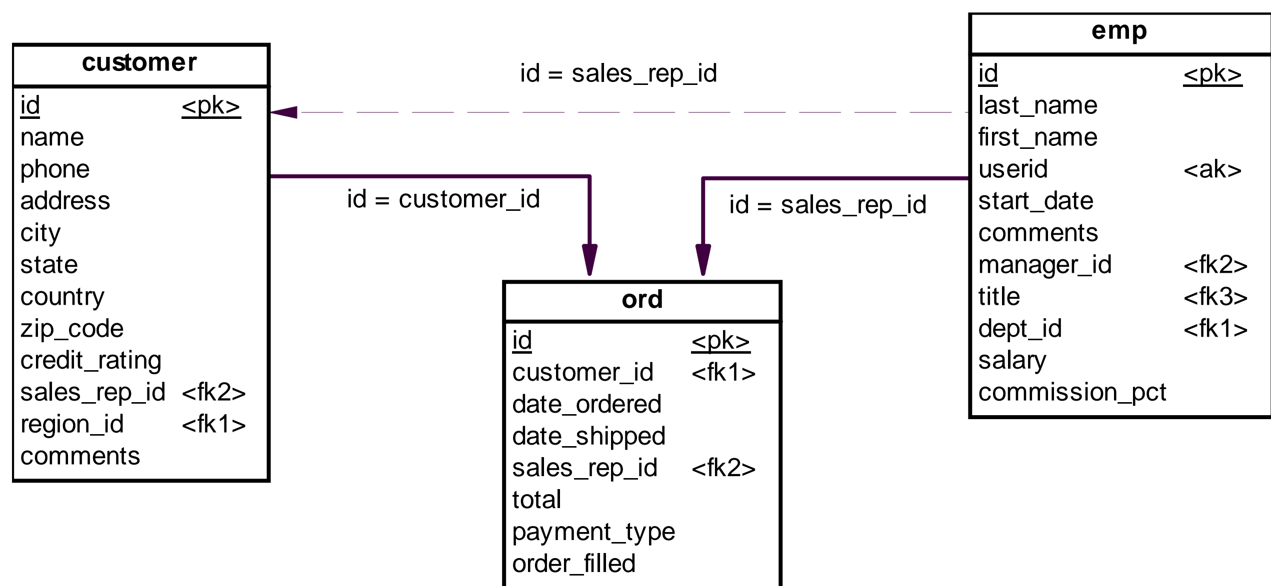
```
+-----------------------+
| SUM(price * quantity) |
+-----------------------+
|             601100.00 |
+-----------------------+
```

Powyższe zapytanie można również zapisać w nieco innej (ale w pełni równoważnej) postaci. Nowa składnia jest zdefiniowana w najnowszej normie ANSI/SQL i pozwala w bardziej

wyraźny sposób oddzielić warunki złączeniowe od innych warunków logicznych (tu: `O.id = 100`).

```
SELECT
  O.id, O.total, I.price, I.quantity, I.price * I.quantity, P.name
FROM
  item I
  INNER JOIN ord O     ON (I.ord_id = O.id)
  INNER JOIN product P ON (I.product_id = P.id)
WHERE
  O.id = 100;
```

### Przykład 36




```
SELECT
  O.id "Nr zam.", C.name "Klient", C.phone, E.first_name, E.last_name
FROM
  ord O, customer C, emp E
WHERE
  O.customer_id = C.id  AND
  O.sales_rep_id = E.id AND
  O.id = 100;
```

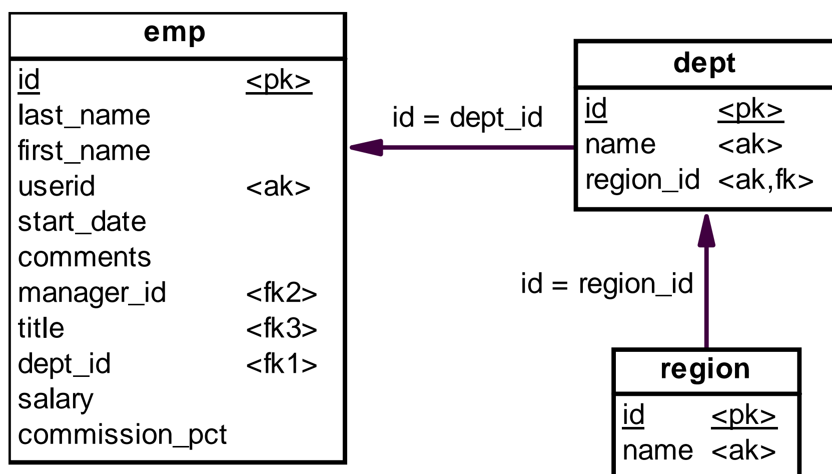
```
+---------+------------+----------------+------------+-----------+
| Nr zam. | Klient     | phone          | first_name | last_name |
+---------+------------+----------------+------------+-----------+
|     100 | Womansport | 1-206-104-0103 | Colin      | Magee     |
+---------+------------+----------------+------------+-----------+
```

**Komentarz:**

Wyświetlamy dane o zamówieniu o numerze 100. Dane pobieramy z trzech tabel. Z tabeli `ord` numer zamówienia, z tabeli `customer` nazwę klienta i jego numer telefonu i wreszcie z tabeli `emp` imię i nazwisko pracownika odpowiedzialnego za dane zamówienie.

Tabela `customer` oraz `emp` też są ze sobą połączone relacją (pola `id` oraz `sales_rep_id`, jednak w kontekście tego przykładu to połączenie jest nieistotne, więc na rysunku zaznaczono je linią przerywaną.

### Przykład 37




```
SELECT
  R.name "Region", D.name "Wydzial", SUM(salary) "Koszty placowe"
FROM
  emp E, dept D, region R
WHERE
  E.dept_id = D.id AND
  R.id = D.region_id
GROUP BY
  R.name, D.name
ORDER BY
  D.name;
```

```
+----------------------+----------------+----------------+
| Region               | Wydzial        | Koszty placowe |
+----------------------+----------------+----------------+
| North America        | Administration |        4050.00 |
| North America        | Finance        |        1450.00 |
| Europe               | Operations     |        3267.00 |
| Asia                 | Operations     |        2100.00 |
| Africa / Middle East | Operations     |        2700.00 |
| South America        | Operations     |        3245.00 |
| North America        | Operations     |        4990.00 |
| North America        | Sales          |        2800.00 |
| Europe               | Sales          |        1450.00 |
| Asia                 | Sales          |        2320.00 |
| Africa / Middle East | Sales          |        1515.00 |
| South America        | Sales          |        1490.00 |
+----------------------+----------------+----------------+
12 rows in set (0.05 sec)
```

### Komentarz:

Wyświetlamy listę wydziałów (pamiętajmy, że w różnych regionach występują wydziały o tych samych nazwach) wraz z sumą wszystkich ich kosztów płacowych (zarobki pracowników w danym wydziale). Aby wykonać to zadanie wymagane jest odpowiednie pogrupowanie danych.

Gdy w zapytaniu nieco zmienimy klauzulę `GROUP BY` (grupowanie następować będzie tylko według regionów ) otrzymamy inny wynik. Porównajmy:

```
SELECT
  R.name "Region", SUM(salary) "Koszty placowe"
FROM
  emp E, dept D, region R
WHERE
  E.dept_id = D.id AND
  R.id = D.region_id
GROUP BY
  R.name          -- teraz grupowanie tylko według jednej kolumny
ORDER BY
  R.name;
```

```
+----------------------+----------------+
| Region               | Koszty placowe |
+----------------------+----------------+
| Africa / Middle East |        4215.00 |
| Asia                 |        4420.00 |
| Europe               |        4717.00 |
| North America        |       13290.00 |
| South America        |        4735.00 |
+----------------------+----------------+
5 rows in set (0.00 sec)
```

Na koniec zobaczmy jaki wynik otrzymamy, gdy *omyłkowo usuniemy* raz jeden a raz drugi z warunków złączeniowych. Widzimy, że serwer za każdym razem wyświetla jakieś dane, z tym że są one za każdym razem błędne merytorycznie. W tym konkretnym przypadku błędy stosunkowo łatwo jest zauważyć (podejrzanie wyglądają zdublowane dane w trzeciej kolumnie). W praktyce jednak, gdy danych w tabelach jest dużo więcej niż w naszym modelu demonstracyjnym, błędy tego typu mogą być trudne lub bardzo trudne do dostrzeżenia na pierwszy rzut oka. Często jest po prostu tak, że błędy zauważamy dopiero w czasie działania aplikacji, gdy „coś się nie zgadza" w zestawieniach.

Podstawowym problemem związanym ze złączeniami bazodanowymi jest to, że serwer SQL *w żaden sposób nie informuje nas o potencjalnej możliwości wystąpienia błędu*. Zapytania są bowiem poprawne od strony formalnej, jednak błędne merytorycznie.

```
SELECT
  R.name "Region", D.name "Wydzial", SUM(salary) "Koszty placowe"
FROM
  emp E, dept D, region R
WHERE
  E.dept_id = D.id     -- usunięto  R.id = D.region_id
GROUP BY
  R.name, D.name
ORDER BY
```

```
  D.name;
```

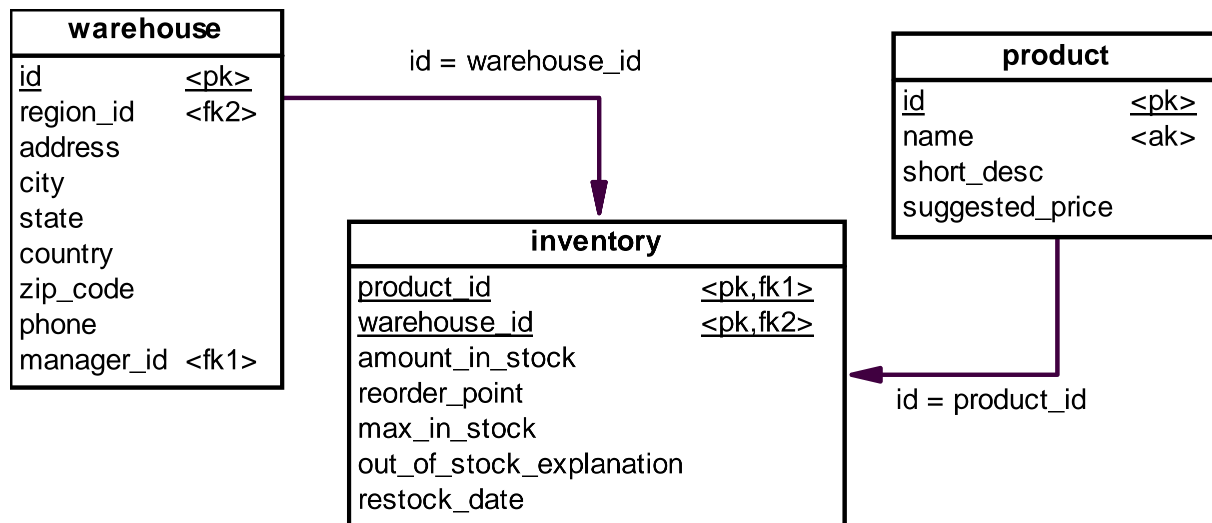
```
+----------------------+----------------+----------------+
| Region               | Wydzial        | Koszty placowe |
+----------------------+----------------+----------------+
| South America        | Administration |        4050.00 |
| North America        | Administration |        4050.00 |
| Europe               | Administration |        4050.00 |
| Asia                 | Administration |        4050.00 |
| Africa / Middle East | Administration |        4050.00 |
| South America        | Finance        |        1450.00 |
| North America        | Finance        |        1450.00 |
| Europe               | Finance        |        1450.00 |
| Asia                 | Finance        |        1450.00 |
| Africa / Middle East | Finance        |        1450.00 |
| Asia                 | Operations     |       16302.00 |
| Africa / Middle East | Operations     |       16302.00 |
| South America        | Operations     |       16302.00 |
| North America        | Operations     |       16302.00 |
| Europe               | Operations     |       16302.00 |
| South America        | Sales          |        9575.00 |
| North America        | Sales          |        9575.00 |
| Europe               | Sales          |        9575.00 |
| Asia                 | Sales          |        9575.00 |
| Africa / Middle East | Sales          |        9575.00 |
+----------------------+----------------+----------------+
20 rows in set (0.01 sec)
```

```sql
SELECT
  R.name "Region", D.name "Wydzial", SUM(salary) "Koszty placowe"
FROM
  emp E, dept D, region R
WHERE
  R.id = D.region_id      -- usunięto E.dept_id = D.id
GROUP BY
  R.name, D.name
ORDER BY
  D.name;
```

```
+----------------------+----------------+----------------+
| Region               | Wydzial        | Koszty placowe |
+----------------------+----------------+----------------+
| North America        | Administration |       31377.00 |
| North America        | Finance        |       31377.00 |
| Europe               | Operations     |       31377.00 |
| Asia                 | Operations     |       31377.00 |
| Africa / Middle East | Operations     |       31377.00 |
| South America        | Operations     |       31377.00 |
| North America        | Operations     |       31377.00 |
| Europe               | Sales          |       31377.00 |
| Asia                 | Sales          |       31377.00 |
| Africa / Middle East | Sales          |       31377.00 |
```

```
| South America          | Sales          |       31377.00 |
| North America          | Sales          |       31377.00 |
+------------------------+----------------+----------------+
12 rows in set (0.01 sec)
```

## Przykład 38




```
SELECT
  CONCAT(
    IFNULL(W.country,'?'),
    ', ',
    IFNULL(W.state,'?'),
    ', ',
    W.city)
    AS "Hurtownia (kraj, stan, miasto)",
  I.amount_in_stock AS "Stan biezacy",
  I.max_in_stock AS "Stan max"
FROM
  warehouse W, product P, inventory I
WHERE
  P.id = I.product_id   AND
  W.id = I.warehouse_id AND
  I.max_in_stock - I.amount_in_stock < 10
ORDER BY
  W.id, P.name;
```

```
+--------------------------------+--------------+----------+
| Hurtownia (kraj, stan, miasto) | Stan biezacy | Stan max |
+--------------------------------+--------------+----------+
| USA, WA, Seattle               |          993 |     1000 |
| USA, WA, Seattle               |          173 |      175 |
| Brazil, ?, Sao Paolo           |           98 |      100 |
| Brazil, ?, Sao Paolo           |          175 |      175 |
| Brazil, ?, Sao Paolo           |          132 |      140 |
```

```
| Brazil, ?, Sao Paolo            |           97 |      100 |
| Nigeria, ?, Lagos               |           70 |       70 |
| Nigeria, ?, Lagos               |           35 |       35 |
| Nigeria, ?, Lagos               |           65 |       70 |
| Nigeria, ?, Lagos               |           61 |       70 |
| ?, ?, Hong Kong                 |          135 |      140 |
| ?, ?, Hong Kong                 |          250 |      250 |
+---------------------------------+--------------+----------+
12 rows in set (0.17 sec)
```

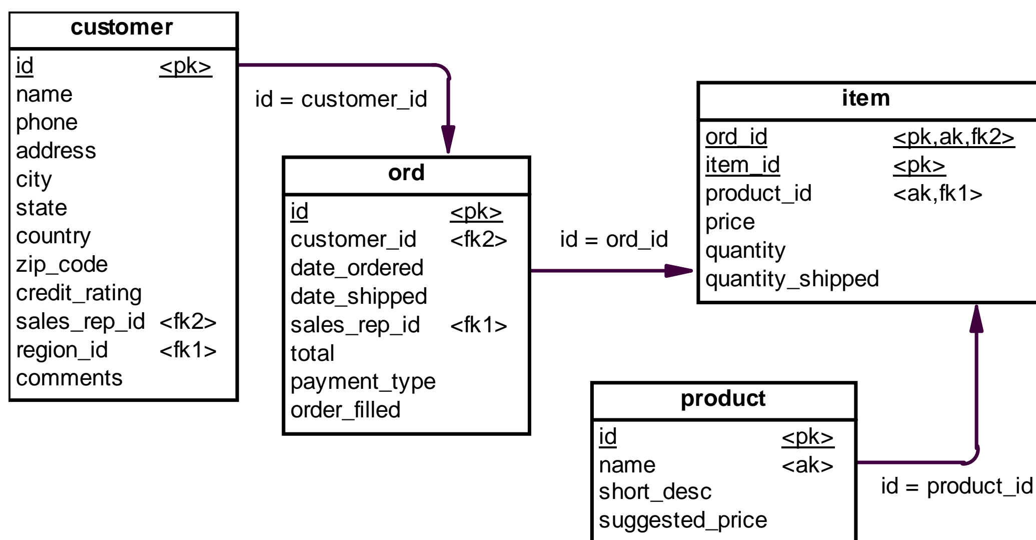
**Komentarz:**

Wyświetlamy stan magazynowy wszystkich produktów z rozbiciem na poszczególne hurtownie (tabele `warehouse`, `product`, `inventory`). Ograniczamy się tylko do tych produktów, których sprzedaż, czyli różnica wartości pól:

`inventory.max_in_stock - inventory.amount_in_stock`

jest mniejsza niż 10.

## Przykład 39




```
SELECT
  O.id "Nr zam.",
  LEFT(C.name,10) AS "Klient",
  LEFT(P.name,10) AS "Produkt",
  O.payment_type AS "Platnosc",
  DATE_FORMAT(O.date_ordered, '%d-%m-%Y')  AS "Data",
  I.price AS "Cena",
  I.quantity AS "Ilosc"
FROM
  customer C,
  ord O,
  item I,
  product P
WHERE
```

```
  C.id = O.customer_id AND
  O.id = I.ord_id AND
  P.id = I.product_id AND
  O.payment_type = 'CREDIT' AND
  O.date_ordered BETWEEN '1992-08-01' AND '1992-08-31'
ORDER BY
 O.id, P.name;
```

```
+---------+------------+------------+----------+------------+---------+-------+
| Nr zam. | Klient     | Produkt    | Platnosc | Data       | Cena    | Ilosc |
+---------+------------+------------+----------+------------+---------+-------+
|      97 | Unisports  | Grand Prix | CREDIT   | 28-08-1992 | 1500.00 |    50 |
|      97 | Unisports  | Junior Soc | CREDIT   | 28-08-1992 |    9.00 |  1000 |
|      99 | Delhi Spor | Black Hawk | CREDIT   | 31-08-1992 |    8.00 |    25 |
|      99 | Delhi Spor | Black Hawk | CREDIT   | 31-08-1992 |    9.00 |    18 |
|      99 | Delhi Spor | Cabrera Ba | CREDIT   | 31-08-1992 |   45.00 |    69 |
|      99 | Delhi Spor | Griffey Gl | CREDIT   | 31-08-1992 |   80.00 |    53 |
|     100 | Womansport | Bunny Boot | CREDIT   | 31-08-1992 |  135.00 |   500 |
|     100 | Womansport | Bunny Ski  | CREDIT   | 31-08-1992 |   14.00 |   500 |
|     100 | Womansport | Himalaya B | CREDIT   | 31-08-1992 |  582.00 |   600 |
|     100 | Womansport | New Air Pu | CREDIT   | 31-08-1992 |   20.00 |   450 |
|     100 | Womansport | Pro Ski Bo | CREDIT   | 31-08-1992 |  380.00 |   400 |
|     100 | Womansport | Pro Ski Po | CREDIT   | 31-08-1992 |   36.00 |   400 |
|     100 | Womansport | Prostar 10 | CREDIT   | 31-08-1992 |    8.00 |   250 |
|     101 | Kam's Spor | Cabrera Ba | CREDIT   | 31-08-1992 |   45.00 |    50 |
|     101 | Kam's Spor | Grand Prix | CREDIT   | 31-08-1992 |   16.00 |    15 |
|     101 | Kam's Spor | Griffey Gl | CREDIT   | 31-08-1992 |   80.00 |    27 |
|     101 | Kam's Spor | Major Leag | CREDIT   | 31-08-1992 |    4.29 |    40 |
|     101 | Kam's Spor | Pro Curlin | CREDIT   | 31-08-1992 |   50.00 |    30 |
|     101 | Kam's Spor | Prostar 10 | CREDIT   | 31-08-1992 |    8.00 |    20 |
|     101 | Kam's Spor | Prostar 10 | CREDIT   | 31-08-1992 |   45.00 |    35 |
|     112 | Futbol Son | Junior Soc | CREDIT   | 31-08-1992 |   11.00 |    50 |
+---------+------------+------------+----------+------------+---------+-------+
21 rows in set (0.00 sec)
```

**Komentarz:**

Wyświetlamy szczegóły zamówień, które regulowane były karta kredytową i które złożone zostały w sierpniu 1992. Z wyniku zapytania mamy możliwość odczytu:

- danych klienta, który składał zamówienie,
- nazwy produktu, na który wystawiono zamówienie,
- ceny, za którą sprzedano produkt,
- ilość sprzedanych produktów każdego rodzaju.

Zapytanie pobiera dane z czterech tabel. Zwróćmy uwagę również na sposób formatowania polecenia SQL. Dzięki wprowadzeniu dużej ilości znaków końca linii zapytanie zyskało na

czytelności. Przy dużych zapytaniach (łączących dane z więcej niż 4–5 tabel) nie powinniśmy zbyt mocno oszczędzać na ilościach linii zajętych przez zapytanie, gdyż w efekcie otrzymamy bardzo mało czytelny tekst[1].

Do odpowiedniego sformatowania wyników użyto funkcji `LEFT` oraz `DATE_FORMAT`.

### Przykład 40

```
SELECT
  O.id "Nr zam.",
  LEFT(C.name, 20) "Klient",
  LEFT(C.city, 20) "Miasto",
  SUM(I.price * I.quantity) "Suma"
FROM
  customer C,
  ord O,
  item I,
  product P
WHERE
  C.id = O.customer_id AND
  O.id = I.ord_id AND
  P.id = I.product_id AND
  O.payment_type = 'CREDIT' AND
  O.date_ordered BETWEEN '1992-08-01' AND '1992-08-31'
GROUP BY
  O.id, C.name, C.city
ORDER BY
  O.id;
```

```
+---------+----------------------+-----------+-----------+
| Nr zam. | Klient               | Miasto    | Suma      |
+---------+----------------------+-----------+-----------+
|      97 | Unisports            | Sao Paolo |  84000.00 |
|      99 | Delhi Sports         | New Delhi |   7707.00 |
|     100 | Womansport           | Seattle   | 601100.00 |
|     101 | Kam's Sporting Goods | Hong Kong |   8056.60 |
|     112 | Futbol Sonora        | Nogales   |    550.00 |
+---------+----------------------+-----------+-----------+
```

### Komentarz:

Na bazie zapytania z poprzedniego przykładu dokonaliśmy pogrupowania wyniku wg. zamówień. W wyniku umieściliśmy całkowitą sumę pieniędzy, na którą opiewało zamówienie. (przykładowo dla zamówienia o `id=97` suma ta wynosi 84000. Łatwo sprawdzić poprawność tego wyniku korzystając z danych uzyskanych w poprzednik przykładzie —

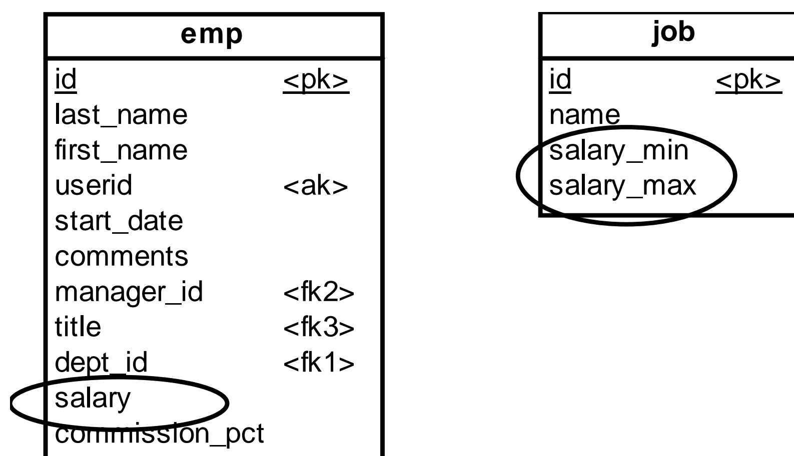
[1] Spróbuj szybko odczytać zapytanie. Jest to to samo zapytanie co w przykładzie powyżej, jednak po usunięciu wszystkich znaków nowej linii oraz wcięć!

```
SELECT O.id "Nr zam.",LEFT(C.name,10) AS "Klient",LEFT(P.name,10) AS "Produkt",
O.payment_type AS "Platnosc",DATE_FORMAT(O.date_ordered, '%d-%m-%Y') AS "Data",
I.price AS "Cena", I.quantity AS "Ilosc" FROM customer C,ord O,item I, product P
WHERE C.id=O.customer_id AND O.id=I.ord_id AND P.id=I.product_id AND O.payment_type
= 'CREDIT' AND O.date_ordered BETWEEN '1992-08-01' AND '1992-08-31' ORDER BY O.id,
P.name;
```

50 x 1500 + 9 x 1000 = 84000).

### 2.14.3 Złączenia nierównościowe (ang. *Theta Join*)

**Przykład 41**




```
-- Tworzymy poniższą tabelę, aby móc zaprezentować
-- tzw. złączenia nierównościowe.

DROP TABLE IF EXISTS job;

CREATE TABLE job(
id          INTEGER  PRIMARY KEY,
name        VARCHAR(20),
salary_min DECIMAL(11,2),
salary_max DECIMAL(11,2));

INSERT INTO job VALUES(1, 'President', 2000, 4000);
INSERT INTO job VALUES(2, 'Stock Clerk', 700, 1200);
INSERT INTO job VALUES(3, 'Sales Representative', 1200, 1400);
INSERT INTO job VALUES(4, 'Warehouse Manager', 1000, 1500);
```

```
SELECT * FROM job;
+----+----------------------+------------+------------+
| id | name                 | salary_min | salary_max |
+----+----------------------+------------+------------+
|  1 | President            |    2000.00 |    4000.00 |
|  2 | Stock Clerk          |     700.00 |    1200.00 |
|  3 | Sales Representative |    1200.00 |    1400.00 |
|  4 | Warehouse Manager    |    1000.00 |    1500.00 |
+----+----------------------+------------+------------+
```

```
SELECT
  LEFT(E.title, 11) AS "title",
  LEFT(J.name, 11) AS "name",
  LEFT(E.first_name, 10) AS "first_name",
  LEFT(E.last_name, 10) AS "last_name",
```

```
  CONCAT(J.salary_min, '-', J.salary_max) AS „Przedział”,
  E.salary
FROM
  emp E, job J
WHERE
  E.salary BETWEEN J.salary_min AND J.salary_max
ORDER BY
  E.last_name;
```

```
+-------------+-------------+------------+------------+-----------------+---------+
| title       | name        | first_name | last_name  | Przedział       | salary  |
+-------------+-------------+------------+------------+-----------------+---------+
| Warehouse M | Warehouse M | Ben        | Biri       | 1000.00-1500.00 | 1100.00 |
| Warehouse M | Stock Clerk | Ben        | Biri       | 700.00-1200.00  | 1100.00 |
| Warehouse M | Warehouse M | Antoinette | Catchpole  | 1000.00-1500.00 | 1300.00 |
| Warehouse M | Sales Repre | Antoinette | Catchpole  | 1200.00-1400.00 | 1300.00 |
| Stock Clerk | Stock Clerk | Eddie      | Chang      | 700.00-1200.00  |  800.00 |
| Stock Clerk | Stock Clerk | Bela       | Dancs      | 700.00-1200.00  |  860.00 |
| Sales Repre | Warehouse M | Andre      | Dumas      | 1000.00-1500.00 | 1450.00 |
| Sales Repre | Warehouse M | Henry      | Giljum     | 1000.00-1500.00 | 1490.00 |
| Warehouse M | Warehouse M | Marta      | Havel      | 1000.00-1500.00 | 1307.00 |
| Warehouse M | Sales Repre | Marta      | Havel      | 1200.00-1400.00 | 1307.00 |
| Stock Clerk | Warehouse M | Elena      | Maduro     | 1000.00-1500.00 | 1400.00 |
| Stock Clerk | Sales Repre | Elena      | Maduro     | 1200.00-1400.00 | 1400.00 |
| Sales Repre | Warehouse M | Colin      | Magee      | 1000.00-1500.00 | 1400.00 |
| Sales Repre | Sales Repre | Colin      | Magee      | 1200.00-1400.00 | 1400.00 |
| Stock Clerk | Stock Clerk | Alexander  | Markarian  | 700.00-1200.00  |  850.00 |
| Warehouse M | Warehouse M | Roberta    | Menchu     | 1000.00-1500.00 | 1250.00 |
| Warehouse M | Sales Repre | Roberta    | Menchu     | 1200.00-1400.00 | 1250.00 |
| VP, Sales   | Warehouse M | Midori     | Nagayama   | 1000.00-1500.00 | 1400.00 |
| VP, Sales   | Sales Repre | Midori     | Nagayama   | 1200.00-1400.00 | 1400.00 |
| Stock Clerk | Stock Clerk | Chad       | Newman     | 700.00-1200.00  |  750.00 |
| VP, Operati | Warehouse M | LaDoris    | Ngao       | 1000.00-1500.00 | 1450.00 |
| Stock Clerk | Sales Repre | Akira      | Nozaki     | 1200.00-1400.00 | 1200.00 |
| Stock Clerk | Stock Clerk | Akira      | Nozaki     | 700.00-1200.00  | 1200.00 |
| Stock Clerk | Warehouse M | Akira      | Nozaki     | 1000.00-1500.00 | 1200.00 |
| Stock Clerk | Stock Clerk | Radha      | Patel      | 700.00-1200.00  |  795.00 |
| Stock Clerk | Stock Clerk | Vikram     | Patel      | 700.00-1200.00  |  795.00 |
| VP, Finance | Warehouse M | Mark       | Quick-To-S | 1000.00-1500.00 | 1450.00 |
| Stock Clerk | Warehouse M | Sylvie     | Schwartz   | 1000.00-1500.00 | 1100.00 |
| Stock Clerk | Stock Clerk | Sylvie     | Schwartz   | 700.00-1200.00  | 1100.00 |
| Stock Clerk | Stock Clerk | George     | Smith      | 700.00-1200.00  |  940.00 |
| Warehouse M | Warehouse M | Molly      | Urguhart   | 1000.00-1500.00 | 1200.00 |
| Warehouse M | Sales Repre | Molly      | Urguhart   | 1200.00-1400.00 | 1200.00 |
| Warehouse M | Stock Clerk | Molly      | Urguhart   | 700.00-1200.00  | 1200.00 |
| President   | President   | Carmen     | Velasquez  | 2000.00-4000.00 | 2500.00 |
+-------------+-------------+------------+------------+-----------------+---------+
34 rows in set (0.00 sec)
```

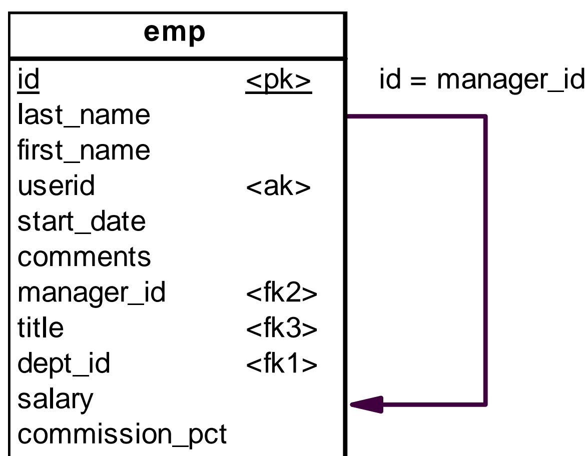
**Komentarz:**

Powyższy przykład to tzw. połączenie nierównościowe (ang. *theta-join*). Złączamy ze sobą relacje, które nie są powiązane ze sobą więzami integralnościowymi (parą kluczy — obcy i główny). W powyższym przykładzie wyświetlono listę pracowników oraz na bazie tabeli `job` sprawdzono, czy zarobki poszczególnych pracowników mieszczą się w „widełkach”. Można przykładowo zauważyć, że pracownik o nazwisku *Urguhart*, pracujący na stanowisku *Warehouse Manager* zarabia kwotę 1200, która to kwota mieści się w przedziale

dla stanowisk *Stock Clerk* oraz *Sales Representative*. Z kolei pracownik o nazwisku *Biri* „podchodzi" pod zarobki dla stanowisk *Stock Clerk* oraz *Warehouse Manager*.

### 2.14.4 Złączenia zwrotne (ang. *Self Join*)

**Przykład 42**




```
SELECT
  E2.last_name, E2.title
FROM
  emp AS E1, emp AS E2
WHERE
  E1.last_name = 'Biri' AND
  E2.title = E1.title;
```

```
+-----------+-------------------+
| last_name | title             |
+-----------+-------------------+
| Urguhart  | Warehouse Manager |
| Menchu    | Warehouse Manager |
| Biri      | Warehouse Manager |
| Catchpole | Warehouse Manager |
| Havel     | Warehouse Manager |
+-----------+-------------------+
```

**Komentarz:**

Tak jak łączy się tabelę z inną tabelą, tak samo można złączyć tabelę z samą sobą. W przykładzie jak powyżej szukamy zależności między wierszami tabeli. Chcemy mianowicie poznać nazwiska wszystkich pracowników, które pracują w tym samym dziale co pracownik o nazwisku *Biri*. Aby to zrobić, musimy znaleźć w tabeli `emp` nazwę działu, w którym pracuje *Biri* a następnie wyszukać w tej samej tabeli tych pracowników, którzy są zatrudnieni w tym samym dziale.

Zadeklarowaliśmy 2 aliasy dla tabeli `emp`. Można więc powiedzieć, że symulujemy sytuacje, jakbyśmy posiadali dwie oddzielne tabele `E1` oraz `E2`, które mają dokładnie tą samą zawartość. Możemy je wówczas złączyć tak, jakbyśmy to robili z dwiema oddzielnymi tabelami.

W tabeli `E1` najpierw odnajdujemy poszukiwane nazwisko (`WHERE E1.last_name = 'Biri'`). Następnie w tabeli `E2` odnajdujemy te wiersze, które mają w polu `title` tą samą nazwę wydziału (`E2.title = E1.title`).

Zwróćmy również uwagę, że gdy pomylimy się i napiszemy `SELECT E1.last_name` (zamiast `e2` jest `E1`) otrzymamy błędny wynik. Trzeba więc bardzo uważać!

```
+-----------+-------------------+
| last_name | title             |
+-----------+-------------------+
| Biri      | Warehouse Manager |
| Biri      | Warehouse Manager |
| Biri      | Warehouse Manager |
| Biri      | Warehouse Manager |
| Biri      | Warehouse Manager |
+-----------+-------------------+
```

### Przykład 43

```sql
SELECT last_name, title, id, manager_id
FROM emp;
```

```
+--------------+----------------------+----+------------+
| last_name    | title                | id | manager_id |
+--------------+----------------------+----+------------+
| Velasquez    | President            |  1 |       NULL |
| Ngao         | VP, Operations       |  2 |          1 |
| Nagayama     | VP, Sales            |  3 |          1 |
| Quick-To-See | VP, Finance          |  4 |          1 |
| Ropeburn     | VP, Administration   |  5 |          1 |
| Urguhart     | Warehouse Manager    |  6 |          2 |
| Menchu       | Warehouse Manager    |  7 |          2 |
| Biri         | Warehouse Manager    |  8 |          2 |
| Catchpole    | Warehouse Manager    |  9 |          2 |
| Havel        | Warehouse Manager    | 10 |          2 |
| Magee        | Sales Representative | 11 |          3 |
| Giljum       | Sales Representative | 12 |          3 |
| Sedeghi      | Sales Representative | 13 |          3 |
| Nguyen       | Sales Representative | 14 |          3 |
| Dumas        | Sales Representative | 15 |          3 |
| Maduro       | Stock Clerk          | 16 |          6 |
| Smith        | Stock Clerk          | 17 |          6 |
| Nozaki       | Stock Clerk          | 18 |          7 |
| Patel        | Stock Clerk          | 19 |          7 |
| Newman       | Stock Clerk          | 20 |          8 |
| Markarian    | Stock Clerk          | 21 |          8 |
| Chang        | Stock Clerk          | 22 |          9 |
| Patel        | Stock Clerk          | 23 |          9 |
| Dancs        | Stock Clerk          | 24 |         10 |
| Schwartz     | Stock Clerk          | 25 |         10 |
+--------------+----------------------+----+------------+
25 rows in set (0.00 sec)
```

**Komentarz:**

W kolumnie `manager_id` wpisany jest identyfikator „szefa” danego pracownika. Zwróćmy uwagę, że pole `manager_id` u pracownika na stanowisku *President* ma wartość `NULL`, co oznacza, że nie ma on swojego zwierzchnika.

### Przykład 44

```
SELECT
  E1.last_name "Pracownik",
  E1.title "Pracownik stanowisko",
  E2.last_name "Szef",
  E2.title "Szef stanowisko"
FROM
  emp E1, emp E2
WHERE
  E1.manager_id = E2.id;
```

```
+--------------+----------------------+-----------+-------------------+
| Pracownik    | Pracownik stanowisko | Szef      | Szef stanowisko   |
+--------------+----------------------+-----------+-------------------+
| Ngao         | VP, Operations       | Velasquez | President         |
| Nagayama     | VP, Sales            | Velasquez | President         |
| Quick-To-See | VP, Finance          | Velasquez | President         |
| Ropeburn     | VP, Administration   | Velasquez | President         |
| Urguhart     | Warehouse Manager    | Ngao      | VP, Operations    |
| Menchu       | Warehouse Manager    | Ngao      | VP, Operations    |
| Biri         | Warehouse Manager    | Ngao      | VP, Operations    |
| Catchpole    | Warehouse Manager    | Ngao      | VP, Operations    |
| Havel        | Warehouse Manager    | Ngao      | VP, Operations    |
| Magee        | Sales Representative | Nagayama  | VP, Sales         |
| Giljum       | Sales Representative | Nagayama  | VP, Sales         |
| Sedeghi      | Sales Representative | Nagayama  | VP, Sales         |
| Nguyen       | Sales Representative | Nagayama  | VP, Sales         |
| Dumas        | Sales Representative | Nagayama  | VP, Sales         |
| Maduro       | Stock Clerk          | Urguhart  | Warehouse Manager |
| Smith        | Stock Clerk          | Urguhart  | Warehouse Manager |
| Nozaki       | Stock Clerk          | Menchu    | Warehouse Manager |
| Patel        | Stock Clerk          | Menchu    | Warehouse Manager |
| Newman       | Stock Clerk          | Biri      | Warehouse Manager |
| Markarian    | Stock Clerk          | Biri      | Warehouse Manager |
| Chang        | Stock Clerk          | Catchpole | Warehouse Manager |
| Patel        | Stock Clerk          | Catchpole | Warehouse Manager |
| Dancs        | Stock Clerk          | Havel     | Warehouse Manager |
| Schwartz     | Stock Clerk          | Havel     | Warehouse Manager |
+--------------+----------------------+-----------+-------------------+
24 rows in set (0.00 sec)
```

**Komentarz:**

W pierwszej kolumnie wyświetlamy nazwisko pracownika, w drugiej stanowisko na jakim pracuje, w trzeciej nazwisko „szefa” a w czwartej stanowisko, na jakim pracuje „szef”. Użycie aliasów jest tutaj obowiązkowe.

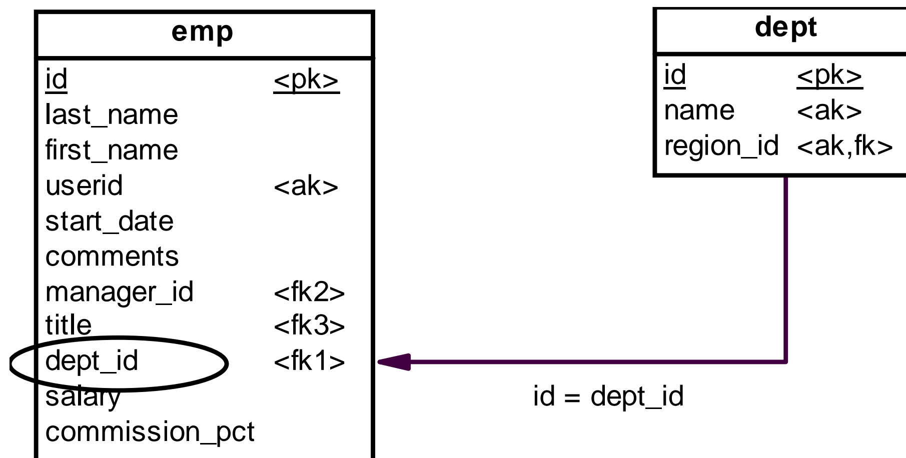
Zwróćmy uwagę, że gdy w klauzuli `WHERE` zmienimy warunek na `E2.manager_id = E1.id` otrzymamy w wyniku inaczej posortowane rekordy. Ponadto w pierwszej oraz drugiej kolumnie są w tej chwili dane „szefa" a w trzeciej i czwartej dane pracownika (ponieważ nie zmieniono aliasów dla kolumn, uzyskany wynik jest bardzo mylący). Trzeba o tym pamiętać, aby uniknąć trudnych do zdiagnozowania błędów.

```
+-----------+---------------------+--------------+----------------------+
| Pracownik | Pracownik stanowisko | Szef         | Szef stanowisko      |
+-----------+---------------------+--------------+----------------------+
| Velasquez | President           | Ngao         | VP, Operations       |
| Velasquez | President           | Nagayama     | VP, Sales            |
| Velasquez | President           | Quick-To-See | VP, Finance          |
| Velasquez | President           | Ropeburn     | VP, Administration   |
| Ngao      | VP, Operations      | Urguhart     | Warehouse Manager    |
| Ngao      | VP, Operations      | Menchu       | Warehouse Manager    |
| Ngao      | VP, Operations      | Biri         | Warehouse Manager    |
| Ngao      | VP, Operations      | Catchpole    | Warehouse Manager    |
| Ngao      | VP, Operations      | Havel        | Warehouse Manager    |
| Nagayama  | VP, Sales           | Magee        | Sales Representative |
| Nagayama  | VP, Sales           | Giljum       | Sales Representative |
| Nagayama  | VP, Sales           | Sedeghi      | Sales Representative |
| Nagayama  | VP, Sales           | Nguyen       | Sales Representative |
| Nagayama  | VP, Sales           | Dumas        | Sales Representative |
| Urguhart  | Warehouse Manager   | Maduro       | Stock Clerk          |
| Urguhart  | Warehouse Manager   | Smith        | Stock Clerk          |
| Menchu    | Warehouse Manager   | Nozaki       | Stock Clerk          |
| Menchu    | Warehouse Manager   | Patel        | Stock Clerk          |
| Biri      | Warehouse Manager   | Newman       | Stock Clerk          |
| Biri      | Warehouse Manager   | Markarian    | Stock Clerk          |
| Catchpole | Warehouse Manager   | Chang        | Stock Clerk          |
| Catchpole | Warehouse Manager   | Patel        | Stock Clerk          |
| Havel     | Warehouse Manager   | Dancs        | Stock Clerk          |
| Havel     | Warehouse Manager   | Schwartz     | Stock Clerk          |
+-----------+---------------------+--------------+----------------------+
24 rows in set (0.01 sec)
```

### 2.14.5 Złączenia zewnętrzne (ang. *Outer Joins*)

**Przykład 45**

Aby omówić problem tzw. *złączeń zewnętrznych* wprowadzimy pewną modyfikację w oryginalnych danych naszego modelu demonstracyjnego.




W tabeli `emp` w danych wybranych pracowników usuniemy informację o numerach działów, w których pracują. Wykonamy więc następujące polecenia:

```
UPDATE emp
SET dept_id = NULL
WHERE dept_id IN (41, 42, 43, 44, 45);
```

```
Query OK, 15 rows affected (0.07 sec)
Rows matched: 15  Changed: 15  Warnings: 0
```

```
SELECT first_name, last_name, dept_id
FROM emp
WHERE dept_id IS NULL;
```

```
+------------+-----------+---------+
| first_name | last_name | dept_id |
+------------+-----------+---------+
| LaDoris    | Ngao      |    NULL |
| Molly      | Urguhart  |    NULL |
| Roberta    | Menchu    |    NULL |
| Ben        | Biri      |    NULL |
| Antoinette | Catchpole |    NULL |
| Marta      | Havel     |    NULL |
| Elena      | Maduro    |    NULL |
| George     | Smith     |    NULL |
| Akira      | Nozaki    |    NULL |
| Vikram     | Patel     |    NULL |
| Chad       | Newman    |    NULL |
| Alexander  | Markarian |    NULL |
| Eddie      | Chang     |    NULL |
| Bela       | Dancs     |    NULL |
| Sylvie     | Schwartz  |    NULL |
+------------+-----------+---------+
15 rows in set (0.00 sec)
```

## Przykład 46

```
SELECT D.id, D.name, E.first_name, E.last_name
```

```
FROM  dept D, emp E
WHERE  D.id = E.dept_id
ORDER BY D.id;
```

```
+----+----------------+------------+--------------+
| id | name           | first_name | last_name    |
+----+----------------+------------+--------------+
| 10 | Finance        | Mark       | Quick-To-See |
| 31 | Sales          | Midori     | Nagayama     |
| 31 | Sales          | Colin      | Magee        |
| 32 | Sales          | Henry      | Giljum       |
| 33 | Sales          | Yasmin     | Sedeghi      |
| 34 | Sales          | Mai        | Nguyen       |
| 34 | Sales          | Radha      | Patel        |
| 35 | Sales          | Andre      | Dumas        |
| 50 | Administration | Carmen     | Velasquez    |
| 50 | Administration | Audry      | Ropeburn     |
+----+----------------+------------+--------------+
10 rows in set (0.01 sec)
```

**Komentarz:**

Wiersze z obu relacji nie posiadające odpowiedników spełniających warunek połączenia nie są wyświetlane. W efekcie „gubimy" informację o pracownikach, którzy nie są przypisani do żadnego działu. Jest to z pewnością bardzo niekorzystne zjawisko. Gdy obecność pól z wartością `NULL` nie zostanie prawidłowo „obsłużona", można spodziewać się wielu trudnych do zdiagnozowania błędów!

### Przykład 47

```
SELECT
  D.id, D.name, E.first_name, E.last_name
FROM
  emp E LEFT OUTER JOIN dept D
ON
  D.id = E.dept_id
ORDER
  BY D.name;
```

```
+------+----------------+------------+--------------+
| id   | name           | first_name | last_name    |
+------+----------------+------------+--------------+
| NULL | NULL           | LaDoris    | Ngao         |
| NULL | NULL           | Sylvie     | Schwartz     |
| NULL | NULL           | Bela       | Dancs        |
| NULL | NULL           | Marta      | Havel        |
| NULL | NULL           | Eddie      | Chang        |
| NULL | NULL           | Antoinette | Catchpole    |
| NULL | NULL           | Alexander  | Markarian    |
| NULL | NULL           | Ben        | Biri         |
| NULL | NULL           | Chad       | Newman       |
| NULL | NULL           | Roberta    | Menchu       |
```

```
| NULL | NULL           | Vikram     | Patel        |
| NULL | NULL           | Molly      | Urguhart     |
| NULL | NULL           | Akira      | Nozaki       |
| NULL | NULL           | George     | Smith        |
| NULL | NULL           | Elena      | Maduro       |
|   50 | Administration | Carmen     | Velasquez    |
|   50 | Administration | Audry      | Ropeburn     |
|   10 | Finance        | Mark       | Quick-To-See |
|   31 | Sales          | Midori     | Nagayama     |
|   35 | Sales          | Andre      | Dumas        |
|   34 | Sales          | Mai        | Nguyen       |
|   33 | Sales          | Yasmin     | Sedeghi      |
|   32 | Sales          | Henry      | Giljum       |
|   31 | Sales          | Colin      | Magee        |
|   34 | Sales          | Radha      | Patel        |
+------+----------------+------------+--------------+
25 rows in set (0.00 sec)
```

**Komentarz:**

Użycie operatora `LEFT OUTER JOIN` spowodowało, że pojawiła się informacja o brakujących pracownikach (tych, którzy nie są przypisani do żadnego działu).

Operacja złączenia zewnętrzego **rozszerza możliwości** zwykłego złączenia. Zwraca ona te same rekordy co złączenie zwykłe **plus** wszystkie te rekordy z tabeli `emp`, które nie pasują do żadnego wiersza z tabeli `dept`.

Dokładnie taki sam wynik otrzymamy, gdy użyjemy tzw. *prawego złączenia zewnętrznego*, czyli gdy napiszemy `dept D RIGHT OUTER JOIN emp E`.

Zwróćmy również uwagę, że używając operatorów złączeń zewnętrznych zamiast klauzuli `WHERE` pojawia się klauzula `ON`.

Standard SQL definiuje jeszcze polecenia `FULL OUTER JOIN`, które jest jakby połączeniem poleceń `LEFT OUTER JOIN` oraz `RIGHT OUTER JOIN`. Jednak obecna wersja serwera MySQL (5.0.16) nie ma jaszcze tej funkcji zaimplementowanej.

## Przykład 48

```sql
SELECT
  D.id, D.name, E.first_name, E.last_name
FROM
  dept D LEFT OUTER JOIN emp E
ON
  D.id = E.dept_id
ORDER BY
  D.name;
```

```
+----+----------------+------------+--------------+
| id | name           | first_name | last_name    |
+----+----------------+------------+--------------+
| 50 | Administration | Carmen     | Velasquez    |
| 50 | Administration | Audry      | Ropeburn     |
| 10 | Finance        | Mark       | Quick-To-See |
```

```
| 41 | Operations      | NULL       | NULL         |
| 42 | Operations      | NULL       | NULL         |
| 43 | Operations      | NULL       | NULL         |
| 44 | Operations      | NULL       | NULL         |
| 45 | Operations      | NULL       | NULL         |
| 31 | Sales           | Midori     | Nagayama     |
| 31 | Sales           | Colin      | Magee        |
| 32 | Sales           | Henry      | Giljum       |
| 33 | Sales           | Yasmin     | Sedeghi      |
| 34 | Sales           | Mai        | Nguyen       |
| 34 | Sales           | Radha      | Patel        |
| 35 | Sales           | Andre      | Dumas        |
+----+-----------------+------------+--------------+
15 rows in set (0.00 sec)
```

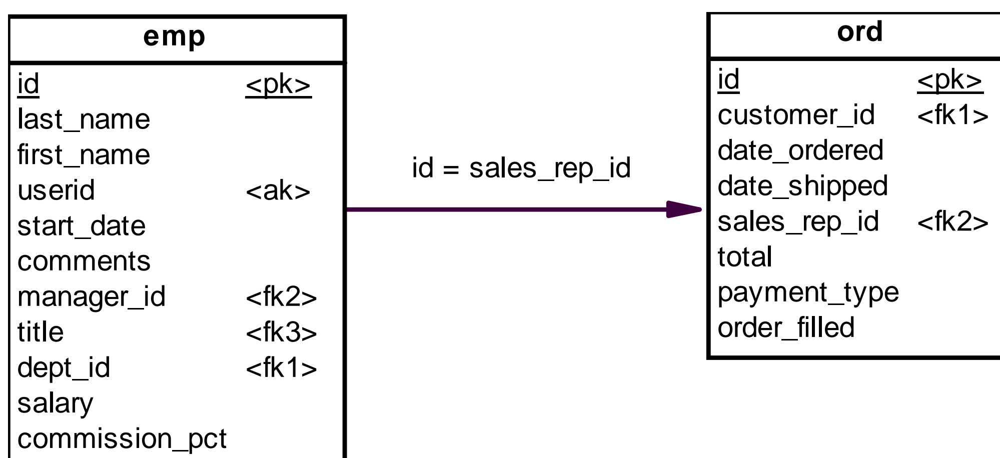
**Komentarz:**

Użycie operatora złączenia zewnętrznego spowodowało, że pojawiła się informacja o brakujących działach (tych o numerach od 41 do 45 — nie są one przypisane do żadnego pracownika).

Operacja złączenia zewnętrznego rozszerza możliwości zwykłego złączenia. Zwracane są te same rekordy co złączenie zwykłe plus wszystkie te rekordy z tabeli `dept`, które nie pasują do żadnego wiersza z tabeli `emp`.

Uzyskany wynik również nie jest zadawalający, gdyż nie o to nam chodziło. Poprzedni przykład pokazuje wynik zgodny z oczekiwaniami. Okazuje się więc, że tabele pojawiające się z lewej i prawej strony operatora złączenia zewnętrznego muszą być umieszczone we właściwy sposób.

**Przykład 49**




```
SELECT
  D.id, SUM(E.salary) "Suma zar."
FROM
  dept D, emp E
WHERE
  D.id = E.dept_id
```

```
GROUP BY
  D.id;
```

```
+----+-----------+
| id | Suma zar. |
+----+-----------+
| 10 |   1450.00 |
| 31 |   2800.00 |
| 32 |   1490.00 |
| 33 |   1515.00 |
| 34 |   2320.00 |
| 35 |   1450.00 |
| 50 |   4050.00 |
+----+-----------+
7 rows in set (0.01 sec)
```

**Komentarz:**

Tracimy informację o tym, że istnieją działy o numerach od 41 do 45. Jeżeli w tych działach nikt nie pracuje, to koszty płacowe będą tam zerowe. Taka informacja też jest istotna!

### Przykład 50

```
SELECT
  D.id, SUM(E.salary) "Suma zar."
FROM
  dept D LEFT OUTER JOIN emp E
ON
  D.id = E.dept_id
GROUP BY
  D.id;
```

```
+----+-----------+
| id | Suma zar. |
+----+-----------+
| 10 |   1450.00 |
| 31 |   2800.00 |
| 32 |   1490.00 |
| 33 |   1515.00 |
| 34 |   2320.00 |
| 35 |   1450.00 |
| 41 |      NULL |
| 42 |      NULL |
| 43 |      NULL |
| 44 |      NULL |
| 45 |      NULL |
| 50 |   4050.00 |
+----+-----------+
12 rows in set (0.01 sec)
```

**Komentarz:**

Pojawia się informacja o działach od 41 do 45. Mimo tego, że nie ma tam żadnych kosztów płacowych, to jednak informacja, że działy takie istnieją jest bardzo istotna.

Za pomocą funkcji IFNULL można zamiast wartości NULL wyświetlić np. liczbę zero, czyli zamiast SUM(E.salary) możemy napisać IFNULL(SUM(E.salary), 0).

### Przykład 51

```
SELECT
  D.id, SUM(E.salary) "Suma zar."
FROM
  emp E LEFT OUTER JOIN dept D
ON
  D.id = E.dept_id
GROUP BY
  D.id;
```

```
+------+-----------+
| id   | Suma zar. |
+------+-----------+
| NULL |  16302.00 |
|   10 |   1450.00 |
|   31 |   2800.00 |
|   32 |   1490.00 |
|   33 |   1515.00 |
|   34 |   2320.00 |
|   35 |   1450.00 |
|   50 |   4050.00 |
+------+-----------+
8 rows in set (0.00 sec)
```

**Komentarz:**

Tym razem operator z lewej strony występuje tabela emp. Liczba w pierwszym wierszu jest po prostu sumą zarobków wszystkich pracowników, którzy nie są przypisani do żadnego działu (gdyż dokonujemy grupowania danych wg. pola dept.id). Aby sprawdzić ten wynik wykonajmy następujące zapytanie:

```
SELECT SUM(salary)
FROM emp
WHERE dept_id IS NULL;
```

```
+-------------+
| SUM(salary) |
+-------------+
|    16302.00 |
+-------------+
```

### Przykład 52

```
SELECT D.id, SUM(E.salary) "Suma zar."
FROM  dept D, emp E
WHERE D.id = E.dept_id
GROUP BY D.id            -- brak średnika!
```

```
UNION

SELECT D.id, NULL          -- ta wartość NULL pojawi się w~wyniku
FROM dept D
WHERE D.id NOT IN
  (SELECT dept_id
   FROM emp
   WHERE dept_id IS NOT NULL); -- podzapytanie
```

```
+----+-----------+
| id | Suma zar. |
+----+-----------+
| 10 |   1450.00 |
| 31 |   2800.00 |
| 32 |   1490.00 |
| 33 |   1515.00 |
| 34 |   2320.00 |
| 35 |   1450.00 |
| 50 |   4050.00 |
| 41 |      NULL |
| 42 |      NULL |
| 43 |      NULL |
| 44 |      NULL |
| 45 |      NULL |
+----+-----------+
12 rows in set (0.00 sec)
```

**Komentarz:**

Gdy operator złączenia zewnętrzego jest niedostępny (W MySQL-u jest, w innych systemach niekoniecznie) należy użyć operatora `UNION`. Aby ułatwić zrozumienie istoty działania tego zapytania poniżej pokazano wyniki, jakie zwracają poszczególne jego fragmenty.

Jeżeli w wyniku działania zapytania nie zostaną znalezione wartości pasujące do `D.dept_id`, to wierszowi zostanie przypisana wartość `NULL` w kolumnie `SUM(E.salary)`.

```
SELECT
  D.id, SUM(E.salary)
FROM
  dept D, emp E
WHERE
  D.id = E.dept_id
GROUP BY
  D.id
```

```
+----+---------------+
| id | SUM(E.salary) |
+----+---------------+
| 10 |       1450.00 |
| 31 |       2800.00 |
| 32 |       1490.00 |
| 33 |       1515.00 |
| 34 |       2320.00 |
```

```
| 35 |       1450.00 |
| 50 |       4050.00 |
+----+---------------+
7 rows in set (0.00 sec)
```

```
SELECT
  D.id, 0
FROM
  dept D
WHERE
  D.id NOT IN
   (SELECT dept_id
    FROM emp
    WHERE dept_id IS NOT NULL);
```

```
+----+---+
| id | 0 |
+----+---+
| 41 | 0 |
| 42 | 0 |
| 43 | 0 |
| 44 | 0 |
| 45 | 0 |
+----+---+
5 rows in set (0.00 sec)
```

```
SELECT dept_id
FROM emp
WHERE dept_id IS NOT NULL;
```

```
+---------+
| dept_id |
+---------+
|      10 |
|      31 |
|      31 |
|      32 |
|      33 |
|      34 |
|      34 |
|      35 |
|      50 |
|      50 |
+---------+
10 rows in set (0.00 sec)
```

## 2.15 Operatory UNION, UNION ALL

**Przykład 53**

```
SELECT name FROM region
UNION
SELECT name FROM dept;
```

```
+----------------------+
| name                 |
+----------------------+
| Africa / Middle East |
| Asia                 |
| Europe               |
| North America        |
| South America        |
| Administration       |
| Finance              |
| Operations           |
| Sales                |
+----------------------+
9 rows in set (0.02 sec)
```

**Komentarz:**

Operator `UNION` umożliwia połączenie dwóch lub więcej instrukcji `SELECT` z jednoczesnym sumowaniem ich wyników. Wiersze wynikowe każdej instrukcji `SELECT` zostają obliczone i ustawione jeden pod drugim, po czym zostają posortowane w celu wyeliminowania powtarzających się wyników.

Zwykłe złączenie wierszy z dwóch lub więcej tabel powoduje, że wiersze układają się obok siebie. Operator `UNION` sprawia, że wiersze z różnych instrukcji `SELECT` układają się w wyniku jeden pod drugim.

Istnieje też następujący wariant: `UNION ALL` — w wyniku jego działania nie dochodzi do posortowania wierszy. Pozostają więc w nich powtarzające się wyniki.

Składnia UNION jest następująca:

```
SELECT ...
UNION [ALL | DISTINCT]
SELECT ...
[UNION [ALL | DISTINCT]
SELECT ...]
```

Przykład:

```
SELECT A, B FROM tabela1 WHERE ...
UNION
SELECT X, Y FROM tabela2 WHERE ...
ORDER BY 2, 1;
```

Uwagi do powyższego:

- liczba wyrażeń w każdym zapytaniu musi być taka sama (u nas: `A`, `B` oraz `X`, `Y`),
- typy danych odpowiadających sobie wyrażeń w każdym zapytaniu muszą być zgodne (u nas: `X` musi mieć być tego samego typu co `A`, a `Y` tego samego typu co `B`),

- nazwy odpowiadających sobie kolumn w dwóch zapytaniach nie muszą być takie same (u nas: `A` i `X` oraz `B` i `Y`),

- ponieważ istnieje taka możliwość, że nazwy odpowiadających sobie kolumn w zapytaniach będą różne, sortowanie wartości za pomocą opcjonalnej klauzuli `ORDER BY` musi się odbywać według pozycji, a nie nazwy (u nas: `ORDER BY 2, 1`). Jedynym wyjątkiem jest sytuacja, gdy nazwy odpowiadających sobie kolumn, po których odbywa się sortowanie, są takie same (jak w bieżącym przykładzie),

- polecenie `UNION` może być używane do przeprowadzania włamań do systemów bazodanowych!

- Standard SQL definiuje jeszcze polecenia `MINUS` oraz `INTERSECT`. Pierwsze znajduje wszystkie rekordy, które *występują w pierwszej relacji, ale nie występują w drugiej*. Polecenie `INTERSECT` natomiast znajduje wszystkie rekordy, które *występują zarówno w jednej jak i w drugiej relacji*. Obecna wersja serwera MySQL (5.0.16) nie ma jaszcze tych funkcji zaimplementowanych.

Gdy funkcje `MINUS` oraz `INTERSECT` zostaną zaimplementowane wyniki powinny być następujące:

```
SELECT id FROM customer
INTERSECT
SELECT customer_id FROM ord;
```

```
+-----+
| id  |
+-----+
| 201 |
| 202 |
| 203 |
| 204 |
| 205 |
| 206 |
| 208 |
| 209 |
| 210 |
| 211 |
| 212 |
| 213 |
| 214 |
+-----+
15 rows in set (0.29 sec)
```

**Komentarz:**

Operator `INTERSECT` zwraca te wiersze, które występują w oby zapytaniach `SELECT`. W naszym przypadku zwrócony wynik należy odczytać następująco: wyświetlone są numery `id` tych klientów, którzy złożyli *choć jedno zamówienie*.

```
SELECT id FROM customer
MINUS
SELECT customer_id FROM ord;
```

```
+-----+
| id  |
+-----+
| 207 |
| 215 |
+-----+
```

### Komentarz:

Operator `MINUS` zwraca te wiersze, które występują w pierwszym zapytaniu `SELECT` a nie występują w drugim. W naszym przypadku zwrócony wynik należy odczytać następująco: wyświetlone są numery `id` tych klientów, którzy nie złożyli *ani jednego zamówienia*.

## Przykład 54

```sql
SELECT name FROM region
UNION ALL
SELECT name FROM dept;
```

```
+----------------------+
| name                 |
+----------------------+
| Africa / Middle East |
| Asia                 |
| Europe               |
| North America        |
| South America        |
| Administration       |
| Finance              |
| Operations           |
| Operations           |
| Operations           |
| Operations           |
| Operations           |
| Sales                |
| Sales                |
| Sales                |
| Sales                |
| Sales                |
+----------------------+
17 rows in set (0.00 sec)
```

### Komentarz:

Operator `UNION ALL` nie usuwa powtarzających się wartości, nie dochodzi też do sortowania wyników. Pierwsze 5 wierszy pochodzi z tabeli `region` a ostatnie 12 z tabeli `dept`.

## Przykład 55

```sql
SELECT DISTINCT name FROM region
UNION ALL
SELECT DISTINCT name FROM dept
ORDER BY 1 DESC;
```

```
+----------------------+
| name                 |
+----------------------+
| South America        |
| Sales                |
| Operations           |
| North America        |
| Finance              |
| Europe               |
| Asia                 |
| Africa / Middle East |
| Administration       |
+----------------------+
9 rows in set (0.00 sec)
```

**Komentarz:**

Tym razem użyliśmy dodatkowo operatora DISTINCT, który spowodował, że usunięte zostały duplikaty a wynik końcowy jest posortowany malejąco — jawnie przez użytkownika a nie automatycznie.

### Przykład 56

```
SELECT last_name FROM emp WHERE last_name LIKE 'N%'
UNION
SELECT name FROM region
ORDER BY last_name;
```

```
+----------------------+
| last_name            |
+----------------------+
| Africa / Middle East |
| Asia                 |
| Europe               |
| Nagayama             |
| Newman               |
| Ngao                 |
| Nguyen               |
| North America        |
| Nozaki               |
| South America        |
+----------------------+
10 rows in set (0.00 sec)
```

**Komentarz:**

Tym razem kolumny w oby tabelach mają inne nazwy. Jest to dopuszczalne. Trzeba tylko pamiętać, aby odpowiadające sobie kolumny były tego samego typu. Nagłówek wynikowej tabeli jest taki, jak nazwa kolumny w pierwszym poleceniu SELECT.

Sortowanie musi odbywać się podług nazwy kolumny z pierwszego polecenia SELECT. Gdy będzie inaczej serwer nie wykona zapytania. Porównajmy:

```
SELECT last_name FROM emp WHERE last_name LIKE 'N%'
```

```
UNION
SELECT name FROM region
ORDER BY name;
```

```
ERROR 1054 (42S22): Unknown column 'name' in 'order clause'
```

Sortowanie może odbywać się również również według numeru kolumny a nie jej nazwy. Czyli w powyższym zapytaniu możemy napisać `ORDER BY 1`.

**Uwaga:**

Standard języka SQL definiuje również operatory `INTERSECT` oraz `MINUS`, które są swego rodzaju uzupełnieniem funkcjonalności oferowanej przez `UNION`. Jednak w obecnej wersji serwera MySQL (5.0.16) nie są one jeszcze zaimplementowane.

## 2.16 Podzapytania

### 2.16.1 Podzapytania zwracające jeden rekord

**Przykład 57**

```
SELECT first_name, last_name, salary
FROM emp
WHERE salary =
  (SELECT MIN(salary) FROM emp);
```

```
+------------+-----------+--------+
| first_name | last_name | salary |
+------------+-----------+--------+
| Chad       | Newman    | 750.00 |
+------------+-----------+--------+
```

**Komentarz:**

Wyświetlamy dane pracownika zarabiającego najmniej.

Zapytanie z podzapytaniem działa w taki sposób, że jako pierwszy wyznaczany jest wynik podzapytania i jest on zapamiętywany w buforze tymczasowym. Następnie warunek w głównym zapytaniu jest sprawdzany z wynikiem podzapytania. Jeżeli wynik jest dodatni dane zostają zaliczone do wyniku ostatecznego. W przeciwnym wypadku są one odrzucane.

**Przykład 58**

```
SELECT first_name, last_name, salary
FROM emp
WHERE (title, salary) =
  (SELECT 'Warehouse Manager', MIN(salary)
   FROM emp
   WHERE title = 'Warehouse Manager');
```

```
+------------+-----------+---------+
| first_name | last_name | salary  |
+------------+-----------+---------+
| Ben        | Biri      | 1100.00 |
+------------+-----------+---------+
```

**Komentarz:**

Wyświetlamy dane pracowników zarabiających najmniej spośród tych, którzy pracują na stanowisku *Warehouse Manager*. Zwróćmy uwagę, na to, jak sformułowano warunek `WHERE`.

### Przykład 59

```
SELECT first_name, last_name, salary
FROM emp
WHERE salary =
  (SELECT MIN(salary), dept_id
   FROM emp
   WHERE title = 'Warehouse Manager');
```

```
ERROR 1140 (42000): Mixing of GROUP columns (MIN(),MAX(),COUNT(),...)
with no GROUP columns is illegal if there is no GROUP BY clause
```

```
SELECT first_name, last_name, salary
FROM emp
WHERE salary =
  (SELECT salary
   FROM emp
   WHERE title = 'Warehouse Manager');
```

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

**Komentarz:**

Musimy pamiętać o tym, aby liczba i typy wartości w klauzuli `SELECT` podzapytania była zgodna z tym, czego oczekuje klauzula `WHERE`.

## 2.16.2 Podzapytania zwracające więcej niż jeden rekord

### Przykład 60

```
SELECT first_name, last_name, salary
FROM emp
WHERE salary <
  (SELECT AVG(salary) FROM emp);
```

```
+------------+-----------+---------+
| first_name | last_name | salary  |
+------------+-----------+---------+
| Molly      | Urguhart  | 1200.00 |
| Roberta    | Menchu    | 1250.00 |
```

```
| Ben          | Biri      | 1100.00 |
| George       | Smith     |  940.00 |
| Akira        | Nozaki    | 1200.00 |
| Vikram       | Patel     |  795.00 |
| Chad         | Newman    |  750.00 |
| Alexander    | Markarian |  850.00 |
| Eddie        | Chang     |  800.00 |
| Radha        | Patel     |  795.00 |
| Bela         | Dancs     |  860.00 |
| Sylvie       | Schwartz  | 1100.00 |
+------------+-----------+---------+
12 rows in set (0.00 sec)
```

**Komentarz:**

Wyświetlamy dane o pracownikach, którzy zarabiają mniej niż wynosi średnia dla wszystkich pracowników.

Istnieją sytuacje, gdy wymaganego wyniku nie uzyskamy inaczej, jak z użyciem podzapytań. Taki przypadek jest przedstawiony w bieżącym przykładzie. Porównajmy:

```sql
SELECT first_name, salary FROM emp WHERE salary < AVG(salary);
```

```
ERROR 1111 (HY000): Invalid use of group function
```

Morzemy próbować uzyskać podobny efekt „na piechotę":

```sql
SELECT AVG(salary) FROM emp;
```

```
+-------------+
| AVG(salary) |
+-------------+
| 1255.080000 |
+-------------+
```

```sql
SELECT first_name, last_name
FROM emp
WHERE salary < 1255.08;
```

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Molly      | Urguhart  |
| Roberta    | Menchu    |
| Ben        | Biri      |
| George     | Smith     |
| Akira      | Nozaki    |
| Vikram     | Patel     |
| Chad       | Newman    |
| Alexander  | Markarian |
| Eddie      | Chang     |
| Radha      | Patel     |
| Bela       | Dancs     |
| Sylvie     | Schwartz  |
```

```
+------------+-----------+
12 rows in set (0.00 sec)
```

**Przykład 61**

```
SELECT first_name, last_name, salary
FROM emp
WHERE
  salary < (SELECT AVG(salary) FROM emp)
  AND
  dept_id IN (SELECT id FROM dept WHERE name = 'Sales');
```

```
+------------+-----------+--------+
| first_name | last_name | salary |
+------------+-----------+--------+
| Radha      | Patel     | 795.00 |
+------------+-----------+--------+
```

**Komentarz:**

Wyświetlamy dane o pracownikach, którzy zarabiają mniej niż wynosi średnia dla wszystkich pracowników oraz (logiczne `AND`) pracują w dziale o nazwie *Sales*.

Podzapytania można zagłębiać, jednak należy robić to bardzo ostrożnie. Stają się one wówczas mało czytelne a ponadto bardzo wzrasta czas ich wykonywania.

### 2.16.3 Operatory ANY oraz ALL

**Przykład 62**

```
SELECT first_name, last_name, salary
FROM emp
WHERE salary >
  (SELECT salary FROM emp WHERE last_name = 'Patel');
```

```
ERROR 1242 (21000): Subquery returns more than 1 row
```

```
SELECT first_name, last_name, salary
FROM emp
WHERE salary > ANY
  (SELECT salary FROM emp WHERE last_name = 'Patel');
```

```
+------------+--------------+---------+
| first_name | last_name    | salary  |
+------------+--------------+---------+
| Carmen     | Gramacki     | 2500.00 |
| LaDoris    | Ngao         | 1450.00 |
| Midori     | Nagayama     | 1400.00 |

...

| Eddie      | Chang        |  800.00 |
```

```
| Bela         | Dancs         |  860.00 |
| Sylvie       | Schwartz      | 1100.00 |
+------------+-------------+---------+
22 rows in set (0.02 sec)
```

**Komentarz:**

W zapytaniu chcieliśmy wyświetlić listę pracowników, którzy zarabiają więcej niż pracownik o nazwisku *Patel*. Niestety pojawił się błąd, gdyż istnieje dwóch pracowników o tym nazwisku.

W takich sytuacjach bardzo przydaje się operator `ANY`. W powyższym przykładzie warunek w `WHERE` jest prawdziwy, gdy liczba po lewej stronie jest większa niż jedna z liczb na liście. Zapis „`> ANY`” należy tłumaczyć jako „większe niż przynajmniej jeden element listy”.

### Przykład 63

```sql
SELECT first_name, last_name, salary
FROM emp
WHERE salary > ALL
  (SELECT salary FROM emp WHERE last_name LIKE 'S%');
```

```
+------------+-----------+---------+
| first_name | last_name | salary  |
+------------+-----------+---------+
| Carmen     | Velasquez | 2500.00 |
| Audry      | Ropeburn  | 1550.00 |
| Mai        | Nguyen    | 1525.00 |
+------------+-----------+---------+
```

**Komentarz:**

Istnieje też podobny do operatora `ANY` operator `ALL`, który służy do porównywania wartości ze wszystkimi wartościami zwracanymi przez podzapytanie. Zapis `"> ALL"` należy więc tłumaczyć jako „większe niż każdy element listy”.

Aby przekonać się, że wynik jest rzeczywiście poprawny wykonajmy polecenie `SELECT` z podzapytania i stwierdzamy, że rzeczywiście powyższe zapytanie wyświetliło dane tylko tych pracowników, którzy zarabiają więcej niż 1515 **oraz** więcej niż 940 **oraz** więcej niż 1100.

```sql
SELECT salary FROM emp WHERE last_name LIKE 'S%';
```

```
+---------+
| salary  |
+---------+
| 1515.00 |
|  940.00 |
| 1100.00 |
+---------+
```

Używając operatora `ALL` należy ostrożnie używać go z operatorem równa się (np. `salary = ALL`), gdyż w przypadku, gdy lista zwracana w podzapytaniu zawiera różne wartości (np.

1515, 940, 1100 jak w powyższym przykładzie) całe zapytanie nigdy nie zwróci żadnego rekordu. Dzieje się tak dlatego, że liczba po lewej stronie nie może być **jednocześnie** równa 1515, 940 oraz 1100.

### 2.16.4 Podzapytania skorelowane, operatory EXISTS oraz NOT EXISTS

**Przykład 64**

```
-- Tabela E1 z zewnętrznej instr. SELECT jest używana w instrukcji wewnętrznej.
-- Z tego powodu zapytania wewnętrzne oraz zewnętrzne są nazywane skorelowanymi.

SELECT first_name, last_name, salary, title
FROM emp E1
WHERE E1.salary <
  (SELECT AVG(salary)
   FROM emp E2
   WHERE E2.title = E1.title)
ORDER BY title, salary;
```

```
+------------+-----------+---------+----------------------+
| first_name | last_name | salary  | title                |
+------------+-----------+---------+----------------------+
| Colin      | Magee     | 1400.00 | Sales Representative |
| Andre      | Dumas     | 1450.00 | Sales Representative |
| Chad       | Newman    |  750.00 | Stock Clerk          |
| Vikram     | Patel     |  795.00 | Stock Clerk          |
| Radha      | Patel     |  795.00 | Stock Clerk          |
| Eddie      | Chang     |  800.00 | Stock Clerk          |
| Alexander  | Markarian |  850.00 | Stock Clerk          |
| Bela       | Dancs     |  860.00 | Stock Clerk          |
| George     | Smith     |  940.00 | Stock Clerk          |
| Ben        | Biri      | 1100.00 | Warehouse Manager    |
| Molly      | Urguhart  | 1200.00 | Warehouse Manager    |
+------------+-----------+---------+----------------------+
11 rows in set (0.02 sec)
```

Dla sprawdzenia wykonajmy:

```
SELECT AVG(salary), title
FROM emp
GROUP BY title;
```

```
+-------------+----------------------+
| AVG(salary) | title                |
+-------------+----------------------+
| 2500.000000 | President            |
| 1476.000000 | Sales Representative |
|  949.000000 | Stock Clerk          |
| 1550.000000 | VP, Administration   |
| 1450.000000 | VP, Finance          |
| 1450.000000 | VP, Operations       |
```

```
| 1400.000000 | VP, Sales             |
| 1231.400000 | Warehouse Manager     |
+-------------+-----------------------+
8 rows in set (0.00 sec)
```

**Komentarz:**

Wyświetlany dane pracowników, którzy zarabiają poniżej średniej zarobków dla swojego stanowiska (pole `title`). Zwróćmy uwagę, że gdyby zamienić warunek `WHERE salary <` na warunek `WHERE salary <=` otrzymalibyśmy również dane pracowników, którzy są jedynymi pracownikami na danym stanowisku, czyli:

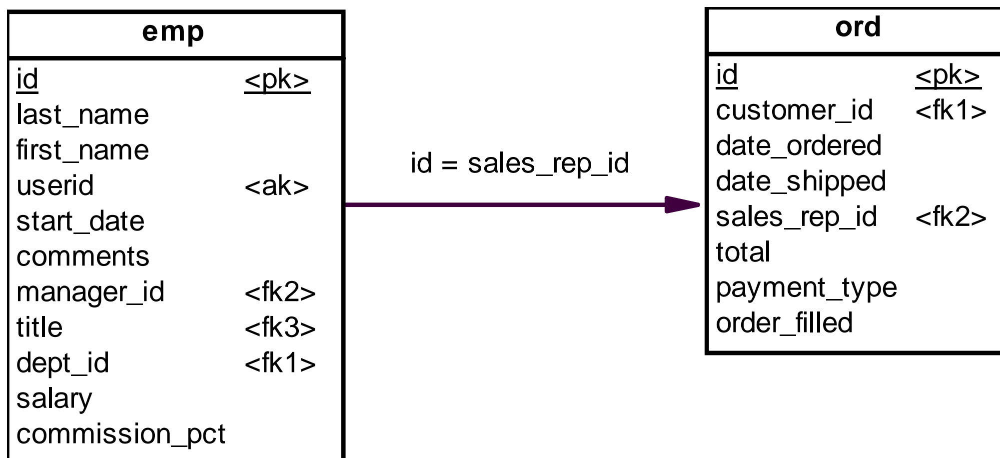
```
+------------+--------------+---------+----------------------+
| first_name | last_name    | salary  | title                |
+------------+--------------+---------+----------------------+
| Carmen     | Velasquez    | 2500.00 | President            |
| Colin      | Magee        | 1400.00 | Sales Representative |
| Andre      | Dumas        | 1450.00 | Sales Representative |
| Chad       | Newman       |  750.00 | Stock Clerk          |
| Radha      | Patel        |  795.00 | Stock Clerk          |
| Vikram     | Patel        |  795.00 | Stock Clerk          |
| Eddie      | Chang        |  800.00 | Stock Clerk          |
| Alexander  | Markarian    |  850.00 | Stock Clerk          |
| Bela       | Dancs        |  860.00 | Stock Clerk          |
| George     | Smith        |  940.00 | Stock Clerk          |
| Audry      | Ropeburn     | 1550.00 | VP, Administration   |
| Mark       | Quick-To-See | 1450.00 | VP, Finance          |
| LaDoris    | Ngao         | 1450.00 | VP, Operations       |
| Midori     | Nagayama     | 1400.00 | VP, Sales            |
| Ben        | Biri         | 1100.00 | Warehouse Manager    |
| Molly      | Urguhart     | 1200.00 | Warehouse Manager    |
+------------+--------------+---------+----------------------+
16 rows in set (0.01 sec)
```

Używając zwykłego podzapytania, jest ono obliczane jako pierwsze, a wyniki tymczasowe są przechowywane w buforach tymczasowych. Warunek w głównym zapytaniu jest sprawdzany z wynikiem podzapytania. Wiersze zostają zaliczone do wyniku zapytania, jeżeli spełniają warunek podzapytania.

Innym rodzajem podzapytań są tzw. **podzapytania skorelowane**. W takim podzapytaniu wartość z głównego zapytania (kolumna `title`) jest **przekazywana** do podzapytania, aby mogło być ono wykonane.

W naszym przykładzie najpierw zostanie pobrany pierwszy rekord przez główne zapytanie. W kolejnym kroku zostanie wykonane podzapytanie na bazie danych zwróconych przez główne zapytanie. W zależności od otrzymanego wyniku, dane zostaną zaliczone bądź też odrzucone. Powyższa procedura zostanie powtórzona dla każdego wiersza zwracanego przez główne zapytanie.

**Przykład 65**




```
SELECT id, first_name, last_name
FROM emp E
WHERE EXISTS
  (SELECT 1 FROM ord O
   WHERE O.sales_rep_id = E.id);
```

```
+----+------------+-----------+
| id | first_name | last_name |
+----+------------+-----------+
| 11 | Colin      | Magee     |
| 12 | Henry      | Giljum    |
| 13 | Yasmin     | Sedeghi   |
| 14 | Mai        | Nguyen    |
| 15 | Andre      | Dumas     |
+----+------------+-----------+
```

**Komentarz:**

Wyrażenie z operatorem logicznym `EXISTS` jest prawdziwe, gdy w wyniku działania podzapytania zostanie zwrócony co najmniej jeden rekord. W przeciwnym wypadku jest ono fałszywe. Operator `NOT EXISTS` działa przeciwnie do `EXISTS`.

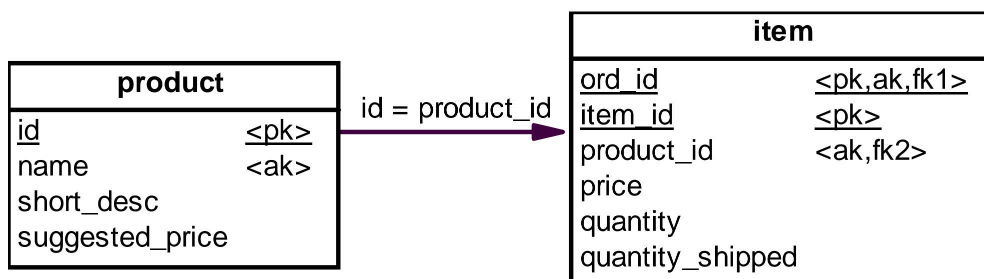
W przykładzie wyświetlamy dane pracowników, którzy choć raz „opiekowali" się złożonym przez klienta zamówieniem (czyli ich numer `id` występuje choć raz w tabeli `ord`). Dla sprawdzenia poprawności możemy wykonać poniższe zapytanie:

```
SELECT sales_rep_id, COUNT(*)
FROM ord
GROUP BY sales_rep_id;
```

```
+--------------+----------+
| sales_rep_id | COUNT(*) |
+--------------+----------+
|           11 |        5 |
|           12 |        3 |
|           13 |        1 |
|           14 |        3 |
|           15 |        4 |
+--------------+----------+
5 rows in set (0.02 sec)
```

Z powyższego wynika, że tylko pięciu pracowników (z ich całkowitej liczby 25) brało udział w realizowaniu zamówień.

**Przykład 66**




```
SELECT name
FROM product P
WHERE NOT EXISTS
  (SELECT *
   FROM item I
   WHERE I.product_id = P.id);
```

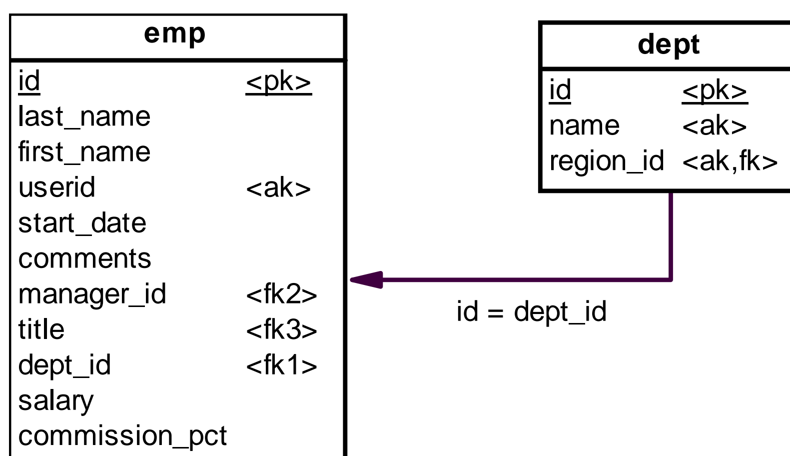
```
+-------------------------+
| name                    |
+-------------------------+
| Prostar 20 Pound Weight |
| Prostar 50 Pound Weight |
+-------------------------+
```

**Komentarz:**

Przykład analogiczny do poprzedniego. Wyświetlamy nazwy produktów, które nie pojawiły się w żadnym zamówieniu. Tutaj używamy operatora `NOT EXISTS`.

### 2.16.5 Przykłady podzapytań, które można zastąpić złączeniami

**Przykład 67**



```
SELECT first_name, last_name, dept_id
FROM emp
WHERE dept_id IN
  (SELECT id FROM dept WHERE name = 'Sales');
```

```
+------------+-----------+---------+
| first_name | last_name | dept_id |
+------------+-----------+---------+
| Midori     | Nagayama  |      31 |
| Colin      | Magee     |      31 |
| Henry      | Giljum    |      32 |
| Yasmin     | Sedeghi   |      33 |
| Mai        | Nguyen    |      34 |
| Andre      | Dumas     |      35 |
| Radha      | Patel     |      34 |
+------------+-----------+---------+
7 rows in set (0.06 sec)
```

**Komentarz:**

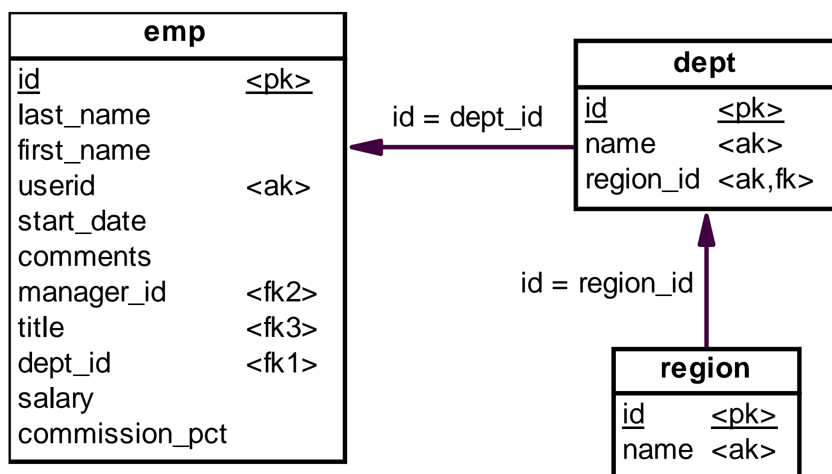
Wyświetlamy dane o pracownikach, którzy pracują w dziale o nazwie *Sales* (pamiętajmy, że w tabeli `dept` jest pięć działów o nazwie *Sales*, każdy zlokalizowany w innym regionie świata).

Tego typu zapytania wykonują się stosunkowo wolno. Podzapytań należy więc używać ostrożnie.

Często wymagany wynik można uzyskać bez potrzeby stosowania podzapytań. Jeżeli jest taka możliwość, powinniśmy ją wykorzystać. Po prostu złączenia wykonują się o wiele szybciej niż podzapytania. Porównajmy:

```
SELECT E.first_name, E.last_name, D.id
FROM emp E, dept D
WHERE E.dept_id = D.id AND
      D.name = 'Sales';
```

## Przykład 68



```
SELECT first_name, last_name FROM emp WHERE dept_id IN
  (
  SELECT id FROM dept
  WHERE region_id IN
    (
    SELECT id FROM region
    WHERE name = 'Europe'
    )
  );
```

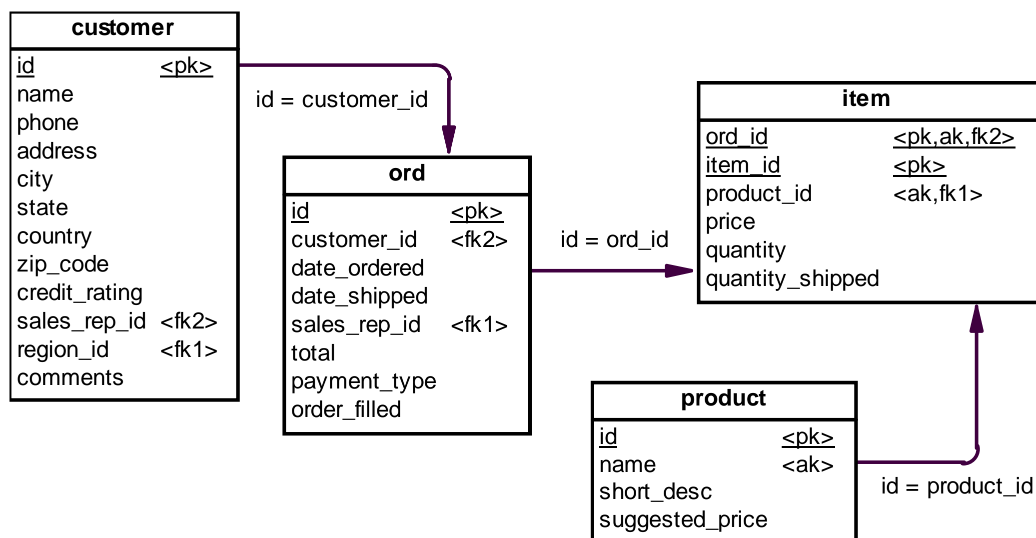
```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Marta      | Havel     |
| Andre      | Dumas     |
| Bela       | Dancs     |
| Sylvie     | Schwartz  |
+------------+-----------+
```

**Komentarz:**

Wyświetlamy dane o pracownikach, którzy pracują w jakimkolwiek dziale zlokalizowanym w Europie. Używamy **zagłębionych podzapytań**. Oczywiście dużo prościej jest to napisać z użyciem złączeń. A dodatkowo uzyskamy to, że takie zapytanie będzie działało zdecydowanie szybciej. Porównajmy:

```
SELECT E.first_name, E.last_name
FROM emp E, dept D, region R
WHERE
  E.dept_id = D.id    AND
  D.region_id = R.id AND
  R.name = 'Europe';
```

## Przykład 69



```
SELECT I.product_id, P.name, C.name
FROM item I, product P, customer C, ord O
WHERE
  I.product_id = P.id  AND
  I.ord_id = O.id      AND
  O.customer_id = C.id AND
  ord_id IN
  (
  SELECT id FROM ord
  WHERE customer_id IN
    (
    SELECT id FROM customer
    WHERE name = 'Unisports'
    )
  );
```

```
+------------+--------------------+-----------+
| product_id | name               | name      |
+------------+--------------------+-----------+
|      20106 | Junior Soccer Ball | Unisports |
|      30321 | Grand Prix Bicycle | Unisports |
+------------+--------------------+-----------+
```

**Komentarz:**

Wyświetlamy nazwy wszystkich produktów (z tabeli PRODUCT), które zostały zamówione przez klienta (tabela CUSTOMER) o nazwie *Unisports*. Użycie zagnieżdżonych podzapytań bardzo zagmatwało zapytanie. Zdecydowanie prościej uzyskamy ten sam wynik z użyciem złączeń. Porównajmy:

```
SELECT I.product_id, P.name, C.name
FROM item I, ord O, customer C, product P
WHERE
  I.product_id = P.id  AND
  I.ord_id = O.id      AND
  O.customer_id = C.id AND
  C.name = 'Unisports';
```

### 2.16.6 Podzapytania w klauzuli FROM

#### Przykład 70

```
SELECT 1
FROM (SELECT 1) AS xyz;
```

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

```
-- Zwróćmy uwagę, że jest "gwiazdka" a wyświetla się tylko jedna kolumna.
-- Jest to kolumna z "widoku dynamicznego" zdefiniowanego w podzapytaniu.

SELECT * FROM
  (SELECT name FROM dept) AS xyz;
```

```
+----------------+
| name           |
+----------------+
| Administration |
| Finance        |
| Operations     |
| Operations     |
| Operations     |
| Operations     |
| Operations     |
| Sales          |
| Sales          |
| Sales          |
| Sales          |
| Sales          |
+----------------+
12 rows in set (0.00 sec)
```

```
SELECT moj_alias.*
FROM
  (SELECT name FROM dept) moj_alias;
```

```
+----------------+
| name           |
+----------------+
| Administration |
| Finance        |
| Operations     |
| Operations     |
| Operations     |
| Operations     |
| Operations     |
| Sales          |
| Sales          |
| Sales          |
| Sales          |
| Sales          |
+----------------+
12 rows in set (0.00 sec)
```

**Komentarz:**

Zwykle podzapytania używane są w klauzuli `WHERE`. Noszą one nazwę podzapytań zagnieżdżonych (ang. *nested subquery*). Istnieje również możliwość tworzenia podzapytań w klauzuli `FROM`. Wówczas noszą one nazwę *inline views*. Polecenie w podzapytaniu można traktować jako swego rodzaju tabela (widok) dynamiczna.

Uwaga: zapytanie w klauzuli WHERE musi mieć zdefiniowany alias. Dzieje się tak dlatego, że w SQL każda tabela, nawet dynamiczna, musi mieć jakąś nazwę. W przykładzie powyżej tą tabeą dynamiczną jest wynik zwracany przez zapytanie SELECT name FROM dept. Gdy aliasu nie będzie serwer wygeneruje błąd:

```
SELECT moj_alias.* FROM
  (SELECT name FROM dept);
```

```
ERROR 1248 (42000): Every derived table must have its own alias
```

# Rozdział 3

# Funkcje formatujące

## 3.1 Uwagi wstępne

W czasie pisania zapytań `SELECT` niejednokrotnie chcielibyśmy wyświetlić dane w nieco innym układzie niż standardowy. Przykładowo standardowym formatem wyświetlania daty jest `YYYY-MM-DD`, podczas gdy my chcielibyśmy wyświetlić go w bardziej rozpowszechnionym w Polsce formacie `DD-MM-YYYY`. W innej sytuacji chcielibyśmy wyświetlić tylko inicjały pracownika a nie imię i nazwisko w pełnym brzmieniu. W takich (oraz wielu innych) sytuacjach możemy skorzystać z bardzo bogatej oferty tzw. *funkcji formatujących*.

W niniejszym opracowaniu omówione zostaną tylko wybrane, najważniejsze według nas, funkcje. W rzeczywistości jest ich dużo więcej. Zachęcamy więc do zapoznania się z nimi odnajdując ich opis w dostępnej literaturze, np. [3].

## 3.2 Funkcje operujące na łańcuchach

**Przykład 71**

```
mysql> SELECT CONCAT('a', 'b');
```

```
+------------------+
| CONCAT('a', 'b') |
+------------------+
| ab               |
+------------------+
```

```
SELECT CONCAT ('a', 'b');
```

```
ERROR 1305 (42000): FUNCTION blab.CONCAT does not exist
```

**Komentarz:**

W serwerze MySQL *nazwa funkcji nie może być oddzielona od nawiasu otwierającego żadną spacją*. Pozwala to parserowi odróżnić wywołanie funkcji od odwołania do tabeli i kolumny o tej samej nazwie, jaką posiada funkcja (w MySQL jest to możliwe!). Dozwolone są za to odstępy po obu stronach argumentów. Można jednak tak uruchomić serwer,

aby spacje, o których tu mowa były dopuszczalne (należy mianowicie ustawić zmienną `-sql-mode=IGNORE_SPACE`). Sugerujemy jednak, aby mimo wszystko nie używać spacji do oddzielania nazwy funkcji od nawiasu otwierającego.

### Przykład 72

```
SELECT last_name, salary
FROM emp
WHERE last_name LIKE 'noZAki';
```

```
+-----------+---------+
| last_name | salary  |
+-----------+---------+
| Nozaki    | 1200.00 |
+-----------+---------+
```

### Komentarz:

W trakcie porównywania łańcuchów MySQL domyślnie nie uwzględnia wielkości liter. Przykładowo powyższe polecenie zwróci jeden rekord, mimo że nazwisko pracownika jest napisane bez dbałości o zachowanie wielkości liter.

Można jednak zmodyfikować powyższe zapytanie SQL tak, aby wielkość liter była honorowana:

```
SELECT last_name, salary
FROM emp
WHERE last_name LIKE BINARY 'noZAki';
```

```
Empty set (0.15 sec)

mysql>
```

Gdy chcemy, na trwałe wymusić uwzględnianie wielkości liter, należy w trakcie tworzenia tabeli (lub modyfikacji poleceniem `ALTER`) nieco inaczej zdefiniować kolumnę łańcuchową (`CHAR, VARCHAR TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT`). Należy mianowicie po nazwie typu umieścić słowo kluczowe `BINARY`. Odpowiedni fragment ze składni polecenia `CREATE TABLE` zamieszczono poniżej:

```
CHAR(length) [BINARY | ASCII | UNICODE]
VARCHAR(length) [BINARY]
TINYTEXT [BINARY]
TEXT [BINARY]
MEDIUMTEXT [BINARY]
LONGTEXT [BINARY]
```

### Przykład 73

Poniżej zamieszczono kilkanaście przykładów z najczęściej wykorzystywanymi funkcjami operującymi na stringach. Serwer MySQL ma ich o wiele więcej — szczegóły patrz dokumentacja.

```
SELECT CONCAT('-->', first_name, '-', last_name, '<--') AS "Pracownik Ngao"
FROM emp
```

```
WHERE last_name = 'NgAo';
```

```
+---------------------+
| Pracownik Ngao      |
+---------------------+
| -->LaDoris-Ngao<--  |
+---------------------+
```

```
SELECT RPAD(last_name,20,'.*') AS "Pracownicy"
FROM emp
WHERE last_name LIKE 'G%' OR last_name LIKE 'D%';
```

```
+----------------------+
| Pracownicy           |
+----------------------+
| Giljum.*.*.*.*.*.*.* |
| Dumas.*.*.*.*.*.*.*. |
| Dancs.*.*.*.*.*.*.*. |
+----------------------+
```

```
SELECT LPAD(RPAD(last_name,20,'* '),40,'# ') AS "Pracownicy"
FROM emp
WHERE last_name LIKE 'D%';
```

```
+------------------------------------------+
| Pracownicy                               |
+------------------------------------------+
| # # # # # # # # # # Dumas* * * * * * * * |
| # # # # # # # # # # Dancs* * * * * * * * |
+------------------------------------------+
```

```
-- 6 znaków podkreślenia (nazwiska 6. literowe)

SELECT last_name, SUBSTR(last_name, 2, 3) AS "3 znaki poczynając od 2."
FROM emp
WHERE last_name LIKE '______';
```

```
+-----------+--------------------------+
| last_name | 3 znaki poczynając od 2. |
+-----------+--------------------------+
| Menchu    | enc                      |
| Giljum    | ilj                      |
| Nguyen    | guy                      |
| Maduro    | adu                      |
| Nozaki    | oza                      |
| Newman    | ewm                      |
+-----------+--------------------------+
```

```
SELECT LTRIM('   ...   xyz   ...     ') AS "LTRIM" ;
SELECT RTRIM('   ...   xyz   ...     ') AS "RTRIM" ;
```

```
+-------------------+
| LTRIM             |
+-------------------+
| ...   xyz  ...    |
+-------------------+

+-------------------+
| RTRIM             |
+-------------------+
|    ...   xyz  ... |
+-------------------+
```

```
-- Pełna składnia:
-- TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str)
--      lub
-- TRIM(remstr FROM] str)

SELECT TRIM(BOTH '*' FROM '***...xyz...***') AS "TRIM";
```

```
+-----------+
| TRIM      |
+-----------+
| ...xyz... |
+-----------+
```

```
SELECT
  first_name, LENGTH(first_name) "Długość imienia",
  last_name, LENGTH(last_name) "Długość nazwiska"
FROM
  emp
WHERE
  LENGTH(last_name) > 6
ORDER BY
  LENGTH(last_name) DESC;
```

```
+------------+-----------------+--------------+------------------+
| first_name | Długość imienia | last_name    | Długość nazwiska |
+------------+-----------------+--------------+------------------+
| Mark       |               4 | Quick-To-See |               12 |
| Carmen     |               6 | Velasquez    |                9 |
| Antoinette |              10 | Catchpole    |                9 |
| Alexander  |               9 | Markarian    |                9 |
| Midori     |               6 | Nagayama     |                8 |
| Audry      |               5 | Ropeburn     |                8 |
| Molly      |               5 | Urguhart     |                8 |
| Sylvie     |               6 | Schwartz     |                8 |
| Yasmin     |               6 | Sedeghi      |                7 |
+------------+-----------------+--------------+------------------+
9 rows in set (0.00 sec)
```

```
SELECT REPLACE('UNIX','UNI','LINU') AS "REPLACE";
```

```
+---------+
| REPLACE |
+---------+
| LINUX   |
+---------+
```

```
SELECT userid, REVERSE(userid) AS "diresu"
FROM emp
WHERE userid LIKE 'a%';
```

```
+----------+----------+
| userid   | diresu   |
+----------+----------+
| acatchpo | ophctaca |
| adumas   | samuda   |
| amarkari | irakrama |
| anozaki  | ikazona  |
| aropebur | rubepora |
+----------+----------+
```

```
SELECT last_name, LOWER(last_name), UPPER(last_name)
FROM emp
WHERE salary > 1500;
```

```
+-----------+------------------+------------------+
| last_name | LOWER(last_name) | UPPER(last_name) |
+-----------+------------------+------------------+
| Velasquez | velasquez        | VELASQUEZ        |
| Ropeburn  | ropeburn         | ROPEBURN         |
| Sedeghi   | sedeghi          | SEDEGHI          |
| Nguyen    | nguyen           | NGUYEN           |
+-----------+------------------+------------------+
```

```
SELECT CHAR(77,121,83,81,76);
```

```
+-------------------------+
| CHAR(77,121,83,81,76)   |
+-------------------------+
| MySQL                   |
+-------------------------+
```

```
SELECT SUBSTRING('MySQL',3) AS "SQL";
```

```
+-----+
| SQL |
+-----+
| SQL |
+-----+
```

```
SELECT name, INSTR(name, 'a') AS "Gdzie litera A?" FROM region;
```

```
+----------------------+-----------------+
| name                 | Gdzie litera A? |
+----------------------+-----------------+
| Africa / Middle East |               1 |
| Asia                 |               1 |
| Europe               |               0 |
| North America        |               7 |
| South America        |               7 |
+----------------------+-----------------+
```

```
SELECT REPEAT('Lubie MySQL-a ', 3) AS "Lubię?";
```

```
+--------------------------------------------+
| Lubię?                                     |
+--------------------------------------------+
| Lubie MySQL-a Lubie MySQL-a Lubie MySQL-a  |
+--------------------------------------------+
```

## 3.3 Funkcje operujące na liczbach

### Przykład 74

Poniżej zamieszczono kilkanaście przykładów z najczęściej wykorzystywanymi funkcjami operującymi na liczbach. Serwer MySQL ma ich o wiele więcej — szczegóły patrz dokumentacja.

```
SELECT
  10 AS "Promień",
  PI() * POW(10, 2) AS "Pole",
  2 * PI() * 10 AS "Obwód";
```

```
+---------+-----------------+-----------+
| Promień | Pole            | Obwód     |
+---------+-----------------+-----------+
|      10 | 314.15926535898 | 62.831853 |
+---------+-----------------+-----------+
```

```
-- Za każdym wywołaniem inna liczba losowa.
SELECT RAND() AS "Liczba losowa";

-- Za każdym wywołaniem taka sama liczba losowa.
SELECT RAND(10) AS "Liczba losowa";
```

```
+-----------------+
| Liczba losowa   |
+-----------------+
| 0.5337429601082 |
+-----------------+

+------------------+
| Liczba losowa    |
```

```
+-------------------+
| 0.65705152196535 |
+-------------------+
```

```
SELECT FLOOR(1.56), ROUND(1.56), FLOOR(-1.56), ROUND(-1.56);
```

```
+-------------+-------------+--------------+--------------+------------+
| FLOOR(1.56) | ROUND(1.56) | FLOOR(-1.56) | ROUND(-1.56) | CEIL(1.56) |
+-------------+-------------+--------------+--------------+------------+
|           1 |           2 |           -2 |           -2 |          2 |
+-------------+-------------+--------------+--------------+------------+
```

```
mysql> SELECT CRC32('Artur Gramacki');
```

```
+-------------------------+
| CRC32('Artur Gramacki') |
+-------------------------+
|              2383994569 |
+-------------------------+
```

## 3.4 Funkcje operujące na dacie i czasie

### Przykład 75

Poniżej zamieszczono kilkanaście przykładów z najczęściej wykorzystywanymi funkcjami operującymi na polach typu data i czas. Serwer MySQL ma ich o wiele więcej — szczegóły patrz dokumentacja.

```
SELECT
  CURRENT_DATE() "A",
  CURRENT_TIME() "B",
  DATE_FORMAT(CURRENT_DATE(), '%W-%M-%Y') "C",
  DATE_FORMAT(CURRENT_DATE(), '%M, %D %Y') "D",
  TIME_FORMAT(CURRENT_TIME(), '%H:%i:%s-%p') "E";
```

```
+------------+----------+----------------------+------------------+-------------+
| A          | B        | C                    | D                | E           |
+------------+----------+----------------------+------------------+-------------+
| 2006-04-26 | 19:23:38 | Wednesday-April-2006 | April, 26th 2006 | 19:23:38-PM |
+------------+----------+----------------------+------------------+-------------+
```

```
SELECT
  CURRENT_DATE(),
  DATEDIFF(CURRENT_DATE(), '1967-01-26') AS "Zyję już tyle dni!";
```

```
+----------------+--------------------+
| CURRENT_DATE() | Zyję już tyle dni! |
+----------------+--------------------+
| 2006-04-26     |              14335 |
+----------------+--------------------+
```

```
SELECT
  CURRENT_TIMESTAMP(),
  DATE_ADD(SYSDATE() ,INTERVAL '5 2:12' DAY_MINUTE)
  AS "Po dadaniu: 5 dni, 2 godzin oraz 12 sek.";

SELECT
  CURRENT_TIMESTAMP(),
  DATE_SUB(SYSDATE() ,INTERVAL '5 2:12' DAY_MINUTE)
  AS "Po odjeciu 5 dni, 2 godzin oraz 12 sek.";
```

```
+---------------------+-----------------------------------------+
| CURRENT_TIMESTAMP() | Po dadaniu 5 dni, 2 godzin oraz 12 sek. |
+---------------------+-----------------------------------------+
| 2006-04-26 18:40:56 | 2006-05-01 20:52:56                     |
+---------------------+-----------------------------------------+

+---------------------+-----------------------------------------+
| CURRENT_TIMESTAMP() | Po odjeciu 5 dni, 2 godzin oraz 12 sek. |
+---------------------+-----------------------------------------+
| 2006-04-26 18:40:56 | 2006-04-21 16:28:56                     |
+---------------------+-----------------------------------------+
```

```
SELECT CONCAT("Dzisiaj mamy ", DAYNAME(CURRENT_DATE()));
```

```
+-------------------------------------------------+
| CONCAT("Dzisiaj mamy ", DAYNAME(CURRENT_DATE())) |
+-------------------------------------------------+
| Dzisiaj mamy Wednesday                          |
+-------------------------------------------------+
```

```
SELECT
  CONCAT(
  'Dzisiaj mamy ',
   DAYOFYEAR(NOW()),
  ' dzień roku, ',
   WEEKOFYEAR(NOW()),
  ' tydzień roku oraz ',
   DAYOFWEEK(NOW()),
  ' dzień tygodnia. ');
```

```
+------------------------------------------------------------------+
| Dzisiaj mamy 116 dzień roku, 17 tydzień roku oraz 4 dzień tygodnia. |
+------------------------------------------------------------------+
```

```
SELECT
  CONCAT(
    'Do końca roku pozostało ',
    DATEDIFF('2006-12-31', NOW()),
    ' dni.');
```

```
+-----------------------------------+
| Do końca roku pozostało 249 dni.  |
+-----------------------------------+
```

```
SELECT
  NOW(),
  YEAR(NOW())   AS "Rok",
  MONTH(NOW())  AS "Miesiac",
  DAY(NOW())    AS "Dzien",
  HOUR(NOW())   AS "Godz.",
  MINUTE(NOW()) AS "Min.",
  SECOND(NOW()) AS "Sek.";
```

```
+---------------------+------+---------+-------+-------+------+------+
| NOW()               | Rok  | Miesiac | Dzien | Godz. | Min. | Sek. |
+---------------------+------+---------+-------+-------+------+------+
| 2006-04-26 19:15:16 | 2006 |       4 |    26 |    19 |   15 |   16 |
+---------------------+------+---------+-------+-------+------+------+
```

```
SELECT
  STR_TO_DATE('26/01/1967 23--05--44', '%d/%m/%Y %H--%k--%s')
  AS "Jakaś data";
```

```
+---------------------+
| Jakaś data          |
+---------------------+
| 1967-01-26 05:00:44 |
+---------------------+
```

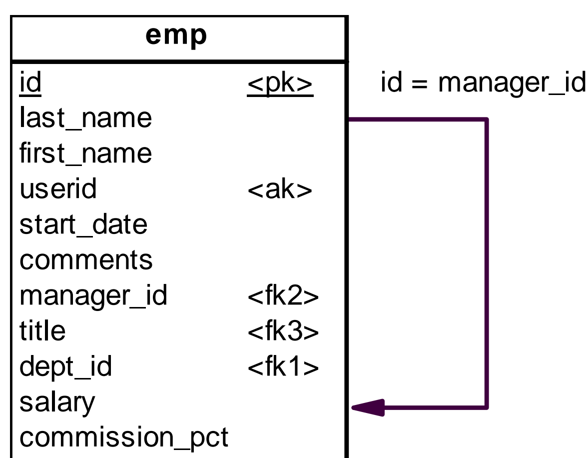
```
SELECT
  TIME_TO_SEC('00:10:01'),
  CURRENT_TIME(),
  TIME_TO_SEC(NOW());
```

```
+-------------------------+----------------+--------------------+
| TIME_TO_SEC('00:10:01') | CURRENT_TIME() | TIME_TO_SEC(NOW()) |
+-------------------------+----------------+--------------------+
|                     601 | 20:29:22       |              73762 |
+-------------------------+----------------+--------------------+
```

# Rozdział 4

# Polecenie INSERT

**Przykład 76**




```
INSERT INTO emp
VALUES
  (100, 'Gramacki', 'Artur', null, null, null, null, null, null, null, null);
```

```
INSERT INTO emp
VALUES
  (100, 'Gramacki', 'Artur', null, null, null, null, null, null, null, null);
```

```
ERROR 1062 (23000): Duplicate entry '100' for key 1
```

```
INSERT INTO emp
VALUES
  (101, 'Gramacki', 'Jarosław', null, null, null, null, null, null, null, null);
```

```
INSERT INTO emp (id, last_name, start_date)
VALUES (102, 'Kowalski', '2004-05-01');
```

```
SELECT first_name, last_name
FROM emp
WHERE last_name LIKE 'Gra%';
```

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| Artur      | Gramacki  |
| Jarosław   | Gramacki  |
+------------+-----------+
```

**Komentarz:**

Pokazano dwie wersje polecenia `INSERT`. Pierwsza wymaga podawania wartości dla wszystkich pól, nawet gdy są tam wartości `NULL`. Ponadto wartości te musimy podawać w ściśle określonej kolejności (tak, jak występują w tabeli).

Druga wersja jest wygodniejsza, gdyż wpisujemy wartości tylko do interesujących nas pól. W obu przypadkach system odmówi wstawienia rekordu, gdy zostanie naruszone chociaż jedno z ograniczeń integralnościowych założonych na tabeli. W przykładzie powyżej próbujemy zarejestrować drugiego pracownika z ta samą wartością klucza głównego.

### Przykład 77

```sql
INSERT INTO region
VALUES
  (100,'Lubuskie'),(101,'Wielkopolskie'),(102,'Dolnoslaskie');
```

```
Query OK, 3 rows affected (0.06 sec) Records: 3  Duplicates: 0  Warnings: 0
```

**Komentarz:**

W systemie MySQL można również stosować nieco bardziej zwartą postać polecenie `INSERT`. Nie jest to co prawda zgodne ze standardem SQL ale bardzo upraszcza zapis i jest w związku z tym chętnie stosowane.

### Przykład 78

```sql
INSERT INTO region (name) VALUES ('Mazowieckie');
INSERT INTO region (name) VALUES ('Dolnoslaskie');
SELECT * FROM region ORDER BY id;
```

```
+-----+----------------------+
| id  | name                 |
+-----+----------------------+
|   1 | North America        |
|   2 | South America        |
|   3 | Africa / Middle East |
|   4 | Asia                 |
|   5 | Europe               |
| 100 | Lubuskie             |
| 101 | Wielkopolskie        |
| 102 | Dolnoslaskie         |
| 103 | Mazowieckie          |
| 104 | Dolnoslaskie         |
+-----+----------------------+
```

**Komentarz:**

Pamiętamy, że przy tworzeniu tabeli *region* jedna z kolumn (`id`) została zdefiniowana jako typ `AUTO_INCREMENT`. Aby wykorzystać możliwość automatycznego generowania unikalnych wartości dla tego typu kolumny musimy użyć innej wersji polecenia `INSERT`, która pozwoli nam na wstawianie danych tylko do wskazanych kolumn.

Zwróćmy również uwagę, że serwer MySQL zadbał o to, aby wygenerowane numery dla pola `id` na pewno były unikalne. Zobaczmy jakie wartości zostaną wygenerowane, gdy wykasujemy dwa wstawione poprzednio rekordy i następnie ponownie je utworzymy. Zauważamy, że MySQL nie wykorzystał „zwolnionego” numeru 104 ale użył nowej wartości. Zachowanie takie jest podyktowane potrzebą zapewnienia lepszej „jakości” wprowadzanych danych. Czy potrafisz wytłumaczyć dlaczego lepiej jest nie wykorzystywać ponownie numeru 104?

```
DELETE FROM region WHERE name LIKE 'Doln%';
INSERT INTO region (name) VALUES ('Dolnoslaskie');
SELECT * FROM region ORDER BY id;
```

```
+-----+----------------------+
| id  | name                 |
+-----+----------------------+
|   1 | North America        |
|   2 | South America        |
|   3 | Africa / Middle East |
|   4 | Asia                 |
|   5 | Europe               |
| 100 | Lubuskie             |
| 101 | Wielkopolskie        |
| 103 | Mazowieckie          |
| 105 | Dolnoslaskie         |
+-----+----------------------+
```

## Przykład 79

```
INSERT INTO emp SET first_name='Tomasz', last_name='Wisniewski';
```

**Komentarz:**

Polecenie `INSERT` można też używać w taki sposób jak pokazano powyżej. Wymieniamy po prostu nazwy kolejnych kolumn i zaraz potem podajemy wartości dla tych kolumn. Taka forma instrukcji `INSERT` jest w pełni równoważna z tymi podanymi we wcześniejszych przykładach. Możesz więc wybrać sobie tą wersję, która Ci najbardziej odpowiada.

## Przykład 80

```
CREATE TABLE emp2 (
  imie VARCHAR(25),
  nazwisko VARCHAR(25),
  zarobki (DECIMAL(11,2)
);

INSERT INTO emp2 (imie, nazwisko, zarobki)
```

```
  SELECT first_name, last_name, salary
  FROM emp
  WHERE salary > 1500;
```

```
SELECT * FROM emp2;
```

```
+--------+-----------+
| imie   | nazwisko  |
+--------+-----------+
| Carmen | Velasquez |
| Audry  | Ropeburn  |
| Yasmin | Sedeghi   |
| Mai    | Nguyen    |
+--------+-----------+
```

**Komentarz:**

W MySQL istnieje bardzo ciekawa opcja polecenia `SELECT`. Można mianowicie wstawić dane do tabeli pobierając je z innej tabeli. W przykładzie powyżej najpierw tworzymy nową tabelę, a następnie zapełniamy ją danymi z innej tabeli.

### Przykład 81

```
-- Tabela testowa.
DROP TABLE IF EXISTS test;

CREATE TABLE test (
  id    INT PRIMARY KEY
);
```

```
-- Wstawiamy kilka przykładowych rekordów. Następuje naruszenie klucza głównego.
-- Serwer wypisuje komunikat o błędzie.
-- Żaden rekord nie jest wstawiany.

INSERT INTO test VALUES (1), (1), (2), (3);
```

```
ERROR 1062 (23000): Duplicate entry '1' for key 1
```

```
-- Wstawiamy kilka przykładowych rekordów. Następuje naruszenie klucza głównego.
-- Używamy opcji IGNORE, więc serwer ignoruje duplikaty.
-- Wstawiane są tylko "dobre" rekordy.

INSERT IGNORE INTO test VALUES (1), (1), (2), (3);
```

```
Query OK, 3 rows affected (0.03 sec)
Records: 4  Duplicates: 1  Warnings: 0
```

```
SELECT * FROM test;
```

```
+----+
| id |
+----+
```

```
|    1 |
|    2 |
|    3 |
+----+
```

**Komentarz:**

W przykładzie pokazujemy efekt użycia opcji `IGNORE`. Można jej używać również przy poleceniu `UPDATE`. Jej działanie jest tam analogiczne.

# Rozdział 5

# Polecenie UPDATE

### Przykład 82

```
UPDATE
  emp
SET
  userid = 'AG',
  salary = 2000,
  start_date = '2004-01-01'
WHERE
  id = 100;
```

```
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

### Komentarz:

Uaktualniamy jeden rekord. Zwróćmy uwagę, na użyty format daty (`RRRR-MM-DD`. Jest to domyślny format dla dat w MySQL-u).

### Przykład 83

```
UPDATE
  emp
SET
  comments =
  CONCAT
    (first_name, ' - ', last_name, ' - ', IFNULL(start_date, 'brak danych'));
```

```
Query OK, 27 rows affected (0.01 sec)
Rows matched: 27  Changed: 27  Warnings: 0
```

```
SELECT first_name, last_name, start_date, comments
FROM emp
WHERE id IN (100, 101);
```

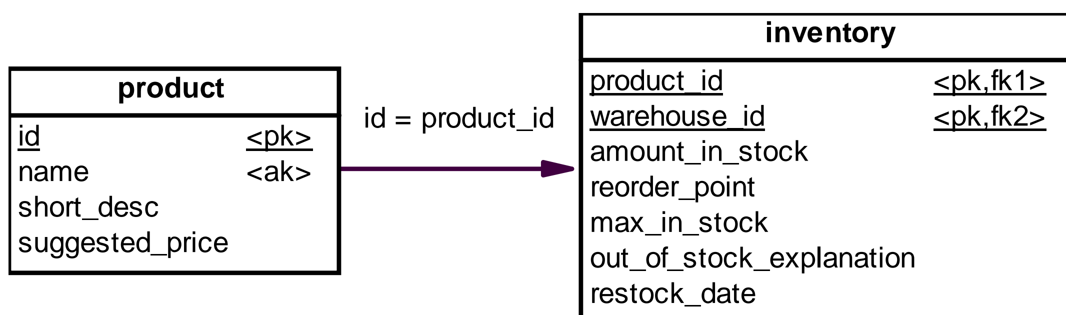
```
+------------+-----------+------------+-----------------------------------+
| first_name | last_name | start_date | comments                          |
+------------+-----------+------------+-----------------------------------+
```

```
| Artur       | Gramacki  | NULL       | Artur - Gramacki - brak danych    |
| Jaroslaw    | Gramacki  | NULL       | Jaroslaw - Gramacki - brak danych |
+------------+-----------+------------+----------------------------------+
2 rows in set (0.00 sec)
```

**Komentarz:**

Uaktualniono wszystkie rekordy w tabeli `emp`. W polu `comments` wpisano, w odpowiedni sposób sformatowane, dane z innych pól. Zwróćmy uwagę, że pole `start_date` zostało „owinięte” w funkcję `IFNULL`, aby otrzymać założony wynik.

### Przykład 84




```
UPDATE
  product
SET
  suggested_price = suggested_price*0.9
WHERE
  id IN (
    SELECT product_id
    FROM inventory
    WHERE max_in_stock - amount_in_stock < 20 AND
    warehouse_id =
      (SELECT id FROM warehouse WHERE city = 'Sao Paolo')
        );
```

**Komentarz:**

Obniżamy (jednym zapytaniem SQL-owym — nie ręcznie) cenę tych produktów w bazie, których sprzedano mniej niż 20 sztuk (różnica wartości w kolumnach `max_in_stock` i `amount_in_stock`). Obniżki cen dokonujemy tylko dla produktów z hurtowni w *Sao Paolo*.

Pisząc polecenie `UPDATE` warto, zanim je wykonamy, sprawdzić które rekordy zostaną uaktualnione. Najwygodniej jest to zrobić pisząc analogiczne polecenie `SELECT`. U nas warunek `UPDATE`-u sprawdzimy w następujący sposób:

```
SELECT
  product_id, max_in_stock, amount_in_stock, max_in_stock - amount_in_stock
FROM
  inventory
WHERE
  max_in_stock - amount_in_stock < 20 AND
```

```
  warehouse_id =
    (SELECT id FROM warehouse WHERE city = 'Sao Paolo');
```

```
+------------+-------------+-----------------+-------------------------------+
| product_id | max_in_stock | amount_in_stock | max_in_stock - amount_in_stock |
+------------+-------------+-----------------+-------------------------------+
|      20510 |          175 |             175 |                              0 |
|      20512 |          175 |             162 |                             13 |
|      30426 |          210 |             200 |                             10 |
|      32861 |          140 |             132 |                              8 |
|      50417 |          100 |              82 |                             18 |
|      50418 |          100 |              98 |                              2 |
|      50536 |          100 |              97 |                              3 |
+------------+-------------+-----------------+-------------------------------+
7 rows in set (0.01 sec)
```

Z kolei wiersze, które zostaną uaktualnione sprawdzimy w następujący sposób:

```
SELECT
  id, name, suggested_price
FROM
  product
WHERE
  id IN (
    SELECT product_id
    FROM  inventory
    WHERE max_in_stock - amount_in_stock < 20 AND
    warehouse_id = (
      SELECT id FROM warehouse WHERE city = 'Sao Paolo')
      );
```

```
+-------+-----------------------+-----------------+
| id    | name                  | suggested_price |
+-------+-----------------------+-----------------+
| 20510 | Black Hawk Knee Pads  |            9.00 |
| 20512 | Black Hawk Elbow Pads |            8.00 |
| 30426 | Himalaya Tires        |           18.25 |
| 32861 | Safe-T Helmet         |           60.00 |
| 50417 | Griffey Glove         |           80.00 |
| 50418 | Alomar Glove          |           75.00 |
| 50536 | Winfield Bat          |           50.00 |
+-------+-----------------------+-----------------+
7 rows in set (0.01 sec)
```

Teraz dopiero można „na spokojnie” uruchomić polecenie `UPDATE`.

### Przykład 85

| ID | NAME | REGION_ID |
|---|---|---|
| 10 | Finance | 1 |
| 31 | Sales | 1 |
| 32 | Sales | 2 |
| 33 | Sales | 3 |
| 34 | Sales | 4 |
| 35 | Sales | 5 |
| 41 | Operations | 1 |
| 42 | Operations | 2 |
| 43 | Operations | 3 |
| 44 | Operations | 4 |
| 45 | Operations | 5 |
| 50 | Administration | 1 |

| ID | NAME |
|---|---|
| 1 | North America |
| 2 | South America |
| 3 | Africa / Middle East |
| 4 | Asia |
| 5 | Europe |

| ID | LAST_NAME | FIRST_NAME | USERID | START_DATE | DEPT_ID | SALARY |
|---|---|---|---|---|---|---|
| 4 | Quick-To-See | Mark | mquickto | 1990-04-07 | 10 | 1450 |
| 3 | Nagayama | Midori | mnagayam | 1991-06-17 | 31 | 1400 |
| 11 | Magee | Colin | cmagee | 1990-05-14 | 31 | 1400 |
| 12 | Giljum | Henry | hgiljum | 1992-01-18 | 32 | 1490 |
| 13 | Sedeghi | Yasmin | ysedeghi | 1991-02-18 | 33 | 1515 |
| | | | | **(2004-03-01)** | **(41)** | **(1666)** |
| 14 | Nguyen | Mai | mnguyen | 1992-01-22 | 34 | 1525 |
| | | | | **(2004-03-01)** | **(41)** | **(1677)** |
| 23 | Patel | Radha | rpatel | 1990-10-17 | 34 | 795 |
| 15 | Dumas | Andre | adumas | 1991-10-09 | 35 | 1450 |
| 2 | Ngao | LaDoris | lngao | 1990-03-08 | 41 | 1450 |
| 16 | Maduro | Elena | emaduro | 1992-02-07 | 41 | 1400 |
| .. | ... | ... | ... | ... | ... | ... |
| 19 | Patel | Vikram | vpatel | 1991-08-06 | 42 | 795 |

```
-- Nieładnie, gdyż trzeba podawać numerki id.
UPDATE
  emp
SET
  dept_id = 41, start_date = CURRENT_DATE, salary = salary * 1.1
WHERE
  salary > 1500 AND
  dept_id IN (31, 32, 33, 34, 35);
```

```
UPDATE
  emp
SET
  -- jedna wartość !!!
  dept_id = (SELECT id FROM dept WHERE name ='Operations' AND region_id = 1),
  start_date = CURRENT_DATE,
  salary = salary * 1.1
WHERE
  -- kilka wartości: (31, 32, 33, 34, 35)
  dept_id IN (SELECT id FROM dept WHERE name = 'Sales') AND
     salary > 1500;
```

**Komentarz:**

W nawiasach podano wartości po zmianach.

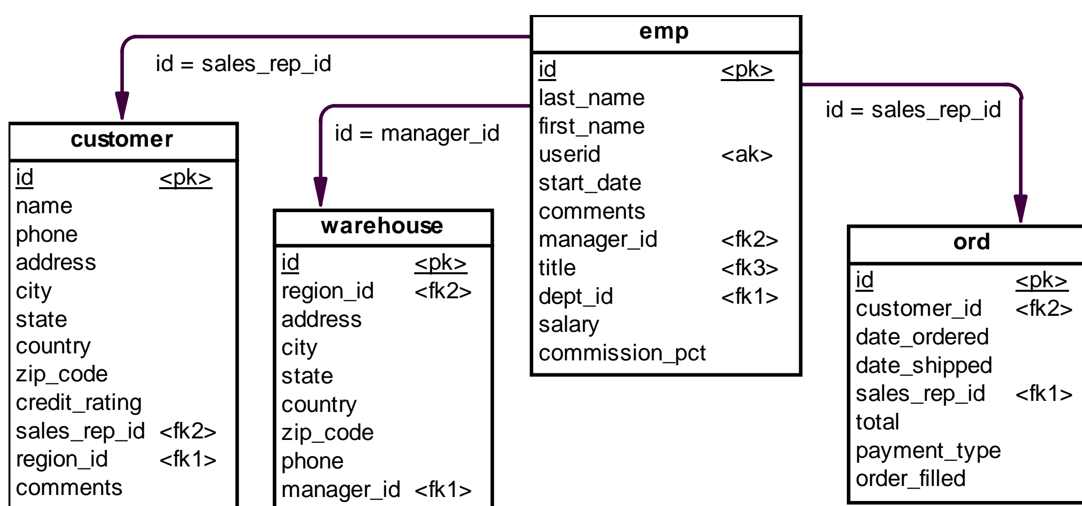
Przepisano (jednym zapytaniem SQL-owym — nie ręcznie) wszystkich pracowników z wydziału *Sales* zarabiających ponad 1500 do wydziału *Operations* zlokalizowanym w regionie

*North America*, zwiększając im jednocześnie płacę o 10% i modyfikując datę zatrudnienia (pole `start_date`) na bieżącą datę.

# Rozdział 6

# Polecenie DELETE

**Przykład 86**




```
DELETE FROM emp;
```

```
ERROR 1451 (23000): Cannot delete or update a~parent row: a~foreign key constrai
nt fails (‘blab/emp‘, CONSTRAINT ‘emp_manager_id_fk‘ FOREIGN KEY (‘manager_id‘)
REFERENCES ‘emp‘ (‘id‘))
```

### Komentarz:

Próba wykasowania wszystkich rekordów w tabeli `emp` nie powiodła się. System wykrył, że w innych tabelach istnieją powiązane rekordy (czyli te, które mają ograniczenie `FOREIGN KEY`. Na powyższym rysunku pokazano te tabele). Ponieważ polecenie (w tym przypadku `DELETE`, ale może to być też `UPDATE` lub `INSERT`) traktowane jest jako niepodzielna transakcja (tzn. musi ona być w całości wykonana lub w całości odrzucona) więc niemożność usunięcia choćby jednego rekordu anuluje całe polecenie. Gdy spróbujemy usunąć tylko „bezpieczne” rekordy, polecenie zakończy się sukcesem.

W systemie MySQL możliwe jest takie zdefiniowanie tabeli, że w trakcie kasowania z niej rekordów zostaną również (w sposób automatyczny) wykasowane wszystkie ewentualnie istniejące rekordy powiązane. Chodzi tutaj o mechanizm tzw. *kasowania kaskadowego*.

Realizowane jest to poprzez użycie klauzuli `ON DELETE CASCADE` w trakcie definiowania klucza obcego w tabeli podrzędnej (u nas chodzi np. o klucz `warehouse_manager_id_fk`).

Należy zauważyć, że tego typu rozwiązanie może być bardzo niebezpieczne w praktyce (można łatwo i nieświadomie utracić bardzo duża liczbę danych). Powinno więc być używane z wielka uwagą. O kasowaniu kaskadowym będzie mowa również w rozdziale omawiającym polecenie `DROP`.

### Przykład 87

```
SELECT COUNT(*)
FROM emp;
```

```
+----------+
| COUNT(*) |
+----------+
|       27 |
+----------+
```

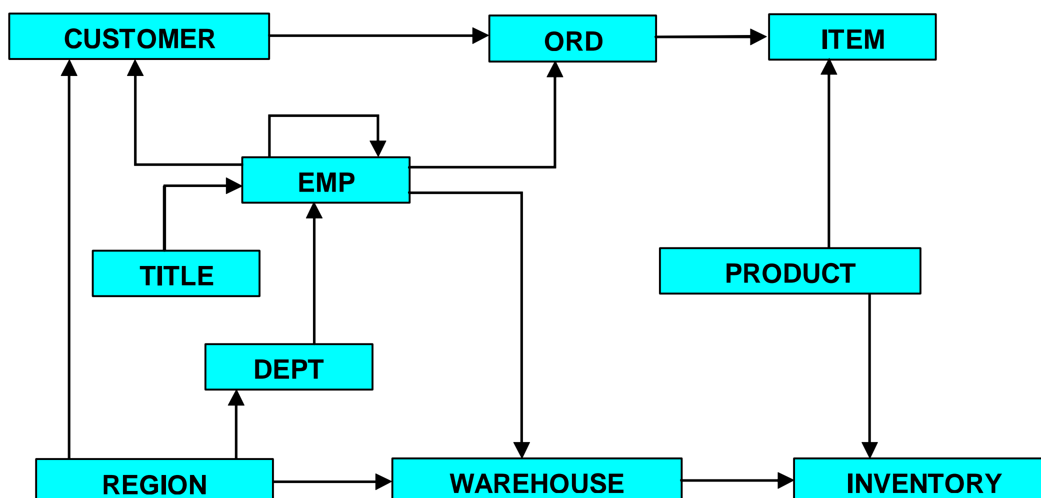
```
DELETE FROM emp
WHERE last_name LIKE 'Gramacki';

SELECT COUNT(*) FROM emp;
```

```
+----------+
| COUNT(*) |
+----------+
|       25 |
+----------+
```

### Komentarz:

Kasowanie rekordów udało się, gdyż nie posiadały one żadnych powiązanych rekordów.

### Przykład 88



```sql
DROP TABLE IF EXISTS item;
DROP TABLE IF EXISTS inventory;
DROP TABLE IF EXISTS ord;
DROP TABLE IF EXISTS product;
DROP TABLE IF EXISTS warehouse;
DROP TABLE IF EXISTS customer;
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
DROP TABLE IF EXISTS region;
DROP TABLE IF EXISTS title;
```

**Komentarz:**

Wykasowanie danych z wszystkich tabel modelu demonstracyjnego. Zwróćmy uwagę, że tabele musiały być kasowane w ściśle określonej kolejności. Kasowanie rozpoczynamy od tabel najbardziej „zagłębionych” w strukturze relacyjnej a kończymy na tych najbardziej „zewnętrznych”.

# Rozdział 7

# Polecenie CREATE

## 7.1 Tworzenie tabel

**Przykład 89**

```
/*==============================================================*/
/* Tabela: studenci                                             */
/*==============================================================*/
-- To też jest znak komentarza (wszystko za znakami '--'
-- jest pomijane przez analizator SQL-a
DROP TABLE IF EXISTS studenci;

CREATE TABLE studenci
(
   stud_id        INT            NOT NULL AUTO_INCREMENT PRIMARY KEY,
   imie           VARCHAR(20)    NOT NULL,
   nazwisko       VARCHAR(30)    NOT NULL,
   typ_uczel_id   CHAR(1)            NULL
);
```

**Komentarz:**

W niniejszym opracowaniu stosujemy konsekwentnie zasadę, że słowa kluczowe języka SQL piszemy wielkimi literami a identyfikatory (np. *imie*, *nazwisko*, *studenci*) małymi. Zachęcamy do konsekwentnego stosowania tej konwencji, gdyż poprawia to czytelność kodów,

Całe polecenie może być wpisywane „ciurkiem”[1], jednak warto konsekwentnie stosować wcięcia oraz przejścia do nowej linii (podobnie jak poprzednio, zabiegi te poprawiają czytelność kodów),

Tekst, który występuje między znakami `/*` oraz `*/` jest traktowany jako komentarz i pomijany przez interpreter języka SQL. Innym rodzajem komentarza są dwa znaki minus, często stosowane, aby skomentować niewielką ilość tekstu (wszystko za tymi znakami jest pomijane przez analizator SQL-a),

[1] Sam przekonaj się, czy takie polecenie będzie czytelne: `CREATE TABLE studenci (stud_id INT NOT NULL,imie VARCHAR(20) NOT NULL,nazwisko VARCHAR(30) NOT NULL,typ_uczel_id CHAR(1))`?

Nasza bardzo prosta pierwsza tabela (inaczej: *relacja*) składa się z czterech kolumn: pierwsza jest typu całkowitego (kolumna `stud_id`), kolejne dwie są typu znakowego o zmiennej szerokości (kolumny `imie` oraz `nazwisko`), ostatnia kolumna (`typ_uczel_id`) jest też typu znakowego, ale o stałej szerokości,

Kolumna `stud_id` jest tzw. kluczem głównym (będzie o tym mowa poniżej) oraz jest zdefiniowana jako `AUTO_INCREMENT`. Wartość tej kolumny przy dodawaniu rekordów może być automatycznie zwiększana[2]. Gdy dodajemy wiersz do tabeli zawierającej kolumnę typu `AUTO_INCREMENT`, nie podajemy wartości dla tej kolumny (patrz opis plecenia `INSERT` w rozdziale 4), bo odpowiedni numer nada baza danych. Pozwala to na proste i efektywne rozwiązanie problemu generowania unikatowych wartości dla kolejnych wierszy tabeli.

System MySQL obsługuje wiele różnych typów danych. Dokumentacja bardzo dokładnie to opisuje, więc nie będziemy w tym miejscu podawać tych informacji. Wszystkie je można jednak podzielić na następujące podstawowe grupy. W ramach każdej z grup wymieniono najważniejsze warianty. Szczegóły doczytaj w dokumentacji:

- typ *logiczny* (`TRUE`, `FALSE`, `NULL`),
- typy *liczbowe* (`INTEGER` (oraz jego odmiany), `FLOAT`, `DOUBLE`, `NUMERIC`),
- związane *z datą i czasem* (`DATE`, `TIME`, `DATETIME`, `TIMESTAMP`, `YEAR`, `DATETIME`),
- typy *łańcuchowe* (`CHAR`, `VARCHAR`, `TEXT` (oraz jego odmiany), `ENUM`, `SET`),
- *duże obiekty binarne* (`BLOB`).

Trzy pierwsze kolumny mają status `NOT NULL`, co oznacza, że dla każdego wiersza w tej tabeli w tych kolumnach *musi być obowiązkowo wpisana jakaś wartość* (nie można pozostawić wartości pustych),

Całość polecenia zakończona jest średnikiem.

W MySQL nie istnieje operacja nadpisania istniejącej już tabeli. Aby utworzyć nową tabelę, starą trzeba najpierw usunąć poleceniem `DROP TABLE`. Próba utworzenia tabeli o nazwie, która już istnieje, kończy się komunikatem o błędzie. Podczas tworzenia można używać słów kluczowych `IF NOT EXISTS`, które powodują, że MySQL nie generuje błędu. Niemniej jednak nowa tabela o tej samej co już istniejąca nazwie, nie powstaje. Porównajmy:

```
CREATE TABLE emp (id INT);
```

```
ERROR 1050 (42S01): Table 'emp' already exists
```

```
CREATE TABLE IF NOT EXISTS emp (id INT);
```

```
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

### Przykład 90

[2] Ale nie musi koniecznie, gdyż to, czy `AUTO_INCREMENT` zostanie wykorzystane zależy od użytej wersji polecenia `INSERT`.

```
DROP TABLE IF EXISTS emp_temp;

CREATE TABLE emp_temp AS
  SELECT
    E.first_name, E.last_name, E.salary, D.name
  FROM
    emp E, dept D
  WHERE
    salary > 1500 AND
    E.dept_id = D.id;
```

```
SELECT * FROM emp_temp;
+------------+-----------+---------+----------------+
| first_name | last_name | salary  | name           |
+------------+-----------+---------+----------------+
| Carmen     | Velasquez | 2500.00 | Administration |
| Audry      | Ropeburn  | 1550.00 | Administration |
| Yasmin     | Sedeghi   | 1515.00 | Sales          |
| Mai        | Nguyen    | 1525.00 | Sales          |
+------------+-----------+---------+----------------+

DESC emp_temp;
+------------+---------------+------+-----+---------+-------+
| Field      | Type          | Null | Key | Default | Extra |
+------------+---------------+------+-----+---------+-------+
| first_name | varchar(25)   | YES  |     | NULL    |       |
| last_name  | varchar(25)   | NO   |     |         |       |
| salary     | decimal(11,2) | YES  |     | NULL    |       |
| name       | varchar(25)   | NO   |     |         |       |
+------------+---------------+------+-----+---------+-------+
```

### Komentarz:

Ciekawą możliwością jest tworzenie tabel w oparciu o zapytanie. W ten sposób możemy bardzo szybko utworzyć tabelę zawierającą dane w interesującym nas układzie. Oczywiście nie należy „bez umiaru” tworzyć w ten sposób tabel. Każda tabela zajmuje bowiem określoną ilość miejsca na dysku i niepotrzebnie komplikuje zarządzanie całością. Lepszym rozwiązanie jest wykorzystywanie do tego celu widoków (ang. *view*). Jest o nich mowa w dalszych rozdziałach opracowania.

## Przykład 91

```
DROP TABLE IF EXISTS emp_temp;

CREATE TABLE emp_temp AS
  SELECT
    UPPER(E.first_name) "Imie",
    UPPER(E.last_name) "Nazwisko",
    CONCAT('Zarobki: ', E.salary) "Zarobki",
    D.name
    FROM emp E, dept D
  WHERE
```

```
    salary > 1500 AND
    E.dept_id = D.id;
```

```
SELECT * FROM emp_temp;
+--------+-----------+-----------------+----------------+
| Imie   | Nazwisko  | Zarobki         | name           |
+--------+-----------+-----------------+----------------+
| CARMEN | VELASQUEZ | Zarobki: 2500.00 | Administration |
| AUDRY  | ROPEBURN  | Zarobki: 1550.00 | Administration |
| YASMIN | SEDEGHI   | Zarobki: 1515.00 | Sales          |
| MAI    | NGUYEN    | Zarobki: 1525.00 | Sales          |
+--------+-----------+-----------------+----------------+

DESC emp_temp;
+----------+---------------+------+-----+---------+-------+
| Field    | Type          | Null | Key | Default | Extra |
+----------+---------------+------+-----+---------+-------+
| Imie     | varchar(25)   | YES  |     | NULL    |       |
| Nazwisko | varchar(25)   | NO   |     |         |       |
| Zarobki  | varbinary(40) | YES  |     | NULL    |       |
| name     | varchar(25)   | NO   |     |         |       |
+----------+---------------+------+-----+---------+-------+
```

**Komentarz:**

Przykład analogiczny do poprzedniego, tylko polecenie `SELECT` jest bardziej złożone. Zwróćmy uwagę, że tym razem w zasadzie trzeba zdefiniować aliasy. Utworzenie tych aliasów niesie za sobą odpowiednie konsekwencje, a mianowicie zmieniają się w stosunku do pierwowzorów typy oraz nazwy kolumn w nowozdefiniowanej tabeli. Porównajmy:

```
DROP TABLE IF EXISTS emp_temp;

CREATE TABLE emp_temp AS
  SELECT
    UPPER(E.first_name),
    UPPER(E.last_name),
    CONCAT('Zarobki: ', E.salary),
    D.name
    FROM emp E, dept D
  WHERE
    salary > 1500 AND
    E.dept_id = D.id;
```

```
DESC emp_temp;
+-------------------------------+---------------+------+-----+---------+-------+
| Field                         | Type          | Null | Key | Default | Extra |
+-------------------------------+---------------+------+-----+---------+-------+
| UPPER(E.first_name)           | varchar(25)   | YES  |     | NULL    |       |
| UPPER(E.last_name)            | varchar(25)   | NO   |     |         |       |
| CONCAT('Zarobki: ', E.salary) | varbinary(40) | YES  |     | NULL    |       |
| name                          | varchar(25)   | NO   |     |         |       |
+-------------------------------+---------------+------+-----+---------+-------+
4 rows in set (0.01 sec)
```

Zwróćmy uwagę jak niepraktyczne są nazwy kolumn (powód: nie używanie aliasów).

## 7.2 Tworzenie i wykorzystywanie widoków (ang. *view*)

### Przykład 92

```
CREATE OR REPLACE VIEW emp_dept_view AS
SELECT
  E.first_name, E.last_name, D.name
FROM
  emp E, dept D
WHERE
  E.dept_id = D.id;
```

```
DESC emp_dept_view;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| first_name | varchar(25) | YES  |     | NULL    |       |
| last_name  | varchar(25) | NO   |     |         |       |
| name       | varchar(25) | NO   |     |         |       |
+------------+-------------+------+-----+---------+-------+

SELECT * FROM emp_dept_view;
+------------+--------------+----------------+
| first_name | last_name    | name           |
+------------+--------------+----------------+
| Carmen     | Velasquez    | Administration |
| Audry      | Ropeburn     | Administration |
| Mark       | Quick-To-See | Finance        |

...

| Mai        | Nguyen       | Sales          |
| Radha      | Patel        | Sales          |
| Andre      | Dumas        | Sales          |
+------------+--------------+----------------+
25 rows in set (0.00 sec)
```

### Komentarz:

Widoki[3] (ang. *view*) zostały zaimplementowane w wersji 5. serwera MySQL. Są to obiekty bardzo przydatne i funkcjonalne. W poprzednim podpunkcie tworzyliśmy nowe tabele w oparciu o tabele już istniejące. Nowotworzone tabele oczywiście zajmują określoną ilość miejsca na dysku i często tworzenie nowych tabel w celach innych niż jako np. kopie bezpieczeństwa jest dość dyskusyjne.

Możliwe jest jednak utworzenie tzw. *widoku*, który może być traktowany jako bardzo wygodny sposób prezentacji danych w wymaganym przez użytkownika układzie. Gdy

[3]Widoki czasami nazywane są też *perspektywami*.

przykładowo bardzo często musimy wyświetlać dane o pracownikach (tabela `emp`) w połączeniu z danymi z innych tabel (np. `dept`), to wygodniej jest utworzyć stosowny widok, niż za każdym razem wpisywać dość długie polecenie SQL. Na rysunku powyżej pokazano szczegóły omawianego widoku.

Utworzyliśmy widok oparty o dane z dwóch różnych tabel. Zauważmy również, że użyliśmy polecenia `CREATE OR REPLACE`, co oznacza, że aby zmienić definicje widoku nie musimy go wcześniej kasować. Mamy tutaj do czynienia ze swego rodzaju nadpisywaniem obiektu. Pamiętamy, że w przypadku innych obiektów (jak np. tabele), ich zmiana wymagała *de facto* wykasowania obiektu i utworzenia go od nowa.

Widać więc że utworzenie widoku nie powoduje skopiowania danych, które używane są w widoku. Serwer MySQL przechowuje tylko definicję widoku a dane pobierane są *on-line* w momencie odwołania do niego. Mamy więc do czynienia z sytuacją zupełnie inną w przypadku polecenia `CREATE TABLE ... AS SELECT`.

**Przykład 93**

```
DELETE FROM emp_demp_view;
```

```
ERROR 1146 (42S02): Table 'blab.emp_demp_view' doesn't exist
```

```
INSERT INTO emp_dept_view VALUES ('a', 'b', 'c');
```

```
ERROR 1394 (HY000): Can not insert into join view 'blab.emp_dept_view'
      without fields list
```

```
INSERT INTO emp_dept_view (first_name, last_name, name)
VALUES ('a', 'b', 'c');
```

```
ERROR 1393 (HY000): Can not modify more than one base table through
      a~join view 'blab.emp_dept_view'
```

**Komentarz:**

Korzystanie z widoku wiąże się jednak z pewnymi ograniczeniami. Przykładowo nie uda się operacja wykasowania danych z widoku, gdy widok ten jest zbudowany w oparciu o *więcej niż jedną* tabelę. Podobne ograniczenia wystąpią przy próbie wstawiania rekordów. Z powyższego widać, że ograniczeń w używaniu widoków jest dużo i w praktyce używane są one raczej tylko w trybie do odczytu. Więcej szczegółów na temat pracy z widokami znajdziesz w literaturze.

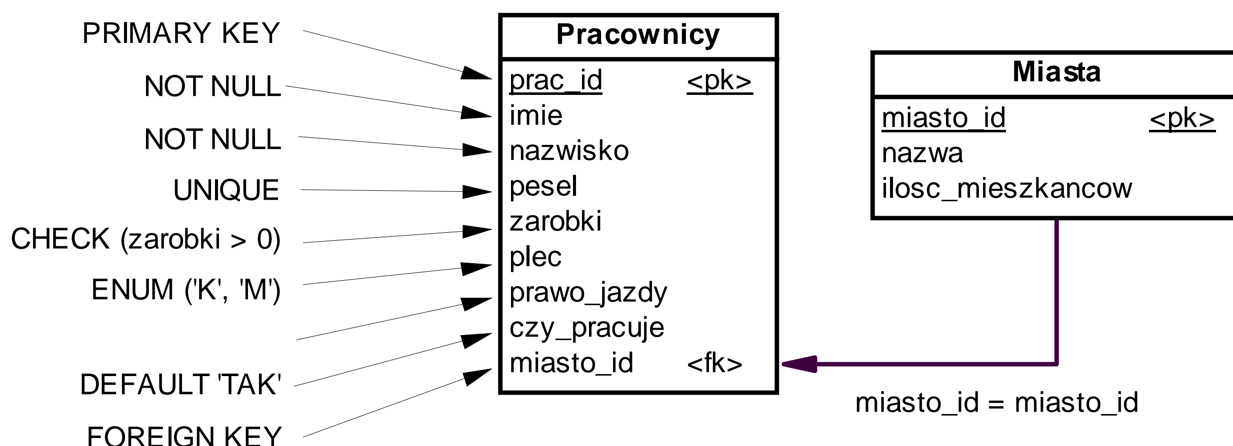
## 7.3 Tworzenie ograniczeń integralnościowych (ang. *constraints*)

- `NOT NULL` — w kolumnie nie można zapisywać wartości NULL („wartość nieznana w tym momencie”)
- `PRIMARY KEY` — każda tabela może zawierać tylko jedno takie ograniczenie. Może być zdefiniowane na poziomie jednej kolumnie (tzw. *ograniczenie kolumnowe*) lub

na więcej niż jednej kolumnie (tzw. *ograniczenie tablicowe*). Zapewnia, że wszystkie wpisane wartości są unikalne i różne od `NULL`,

- `DEFAULT` — określa domyślną wartość używaną podczas wstawiania danych w przypadku, gdy nie została jawnie podana żadna wartość dla kolumny,
- `FOREIGN KEY (REFERENCES)` — zapewnia tzw. integralność referencyjną. Zapobiega wstawianiu błędnych rekordów w tabelach podrzędnych (po stronie „N”),
- `UNIQUE` — Zapewnia, że wszystkie wpisane wartości są unikalne. Od ograniczenia `PRIMARY KEY` różni się tym, że dopuszcza wpisywanie wartości `NULL`,
- `CHECK` — pozwala na wpisywanie tylko takich wartości, które spełniają określone warunki (np. „zarobki>0”). Obecnie w MySQL nie jest zaimplementowane,
- `ENUM` — pozwala na wpisanie tylko jednej wartości z wcześniej zdefiniowanego zbioru,
- `SET` — pozwala na wpisanie jednej lub wielu wartości z wcześniej zdefiniowanego zbioru.

**Przykład 94**




```
DROP TABLE IF EXISTS pracownicy;

CREATE TABLE pracownicy
(
   prac_id           INTEGER              PRIMARY KEY AUTO_INCREMENT,
   imie              VARCHAR(20)          NOT NULL,
   nazwisko          VARCHAR(30)          NOT NULL,
   pesel             INTEGER              NOT NULL UNIQUE,
   zarobki           DECIMAL(11,2)        CHECK (zarobki > 0),
   plec              ENUM ('M', 'K')      NOT NULL,
   prawo_jazdy       SET ('A', 'B', 'C',
                          'D', 'CE', 'BE',
                          'DE')           NOT NULL,
   czy_pracuje       CHAR(3)              DEFAULT 'TAK',
```

```
  miasto_id       INTEGER
)
ENGINE = InnoDB;
```

```
DESC pracownicy;
+-------------+--------------------- +------+-----+---------+----------------+
| Field       | Type                 | Null | Key | Default | Extra          |
+-------------+----------------------+------+-----+---------+----------------+
| prac_id     | int(11)              | NO   | PRI | NULL    | auto_increment |
| imie        | varchar(20)          | NO   |     |         |                |
| nazwisko    | varchar(30)          | NO   |     |         |                |
| pesel       | int(11)              | NO   | UNI |         |                |
| zarobki     | decimal(11,2)        | YES  |     | NULL    |                |
| plec        | enum('M','K')        | NO   |     |         |                |
| prawo_jazdy | set('A','B','C',...  | NO   |     |         |                |
| czy_pracuje | char(3)              | YES  |     | TAK     |                |
| miasto_id   | int(11)              | YES  |     | NULL    |                |
+-------------+----------------------+------+-----+---------+----------------+
9 rows in set (0.01 sec)
```

```
SHOW CREATE TABLE pracownicy;
```

```
+------------+-----------------------------------------------------+
| Table      | Create Table                                        |
+------------+-----------------------------------------------------+
| pracownicy | CREATE TABLE `pracownicy` (
  `prac_id` int(11) NOT NULL auto_increment,
  `imie` varchar(20) NOT NULL,
  `nazwisko` varchar(30) NOT NULL,
  `pesel` int(11) NOT NULL,
  `zarobki` decimal(11,2) default NULL,
  `plec` enum('M','K') NOT NULL,
  `prawo_jazdy` set('A','B','C','D','CE','BE','DE') NOT NULL,
  `czy_pracuje` char(3) default 'TAK',
  `miasto_id` int(11) default NULL,
  PRIMARY KEY  (`prac_id`),
  UNIQUE KEY `pesel` (`pesel`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8                                |
+------------+------------------------------------------------------+
```

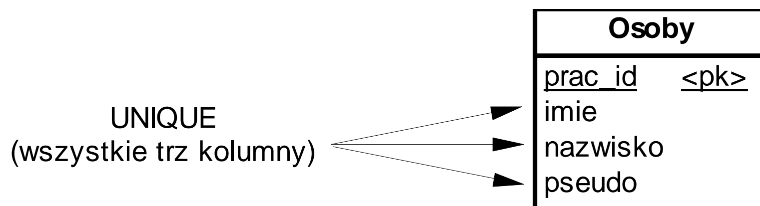
**Komentarz:**

Bardzo często na kolumny w tabeli nakładamy pewne dodatkowe warunki. Przykładowo w kolumnie numerycznej możemy nakazać, aby możliwe było wpisywanie tylko liczb ze zbioru $\{1, 2, 3, 4\}$. Podobnie dla np. kolumny tekstowej możemy nakazać wpisywanie tylko i wyłącznie napisów *karta płatnicza*, *gotówka* oraz *przelew*. O kolumnach takich mówimy, że mają one zdefiniowane pewne dodatkowe warunki (nazywane właśnie *ograniczeniami*), które pozwolą nam osiągnąć tanim kosztem zamierzony efekt.

Nakładanie ograniczeń na kolumny pozwala nam przeprowadzać kontrolę wprowadzanych danych na najniższym z możliwych poziomów — na poziomie bazy danych. Oczywiście kontrolę taką można też przeprowadzić na poziomie aplikacji, jednak powinno się traktować jako dobrą zasadę programistyczną aby, jeżeli jest to tylko możliwe, przeprowadzać

kontrolę wprowadzanych danych możliwie „jak najbliżej” serwera bazy danych. Statystycznie rzecz biorąc jest bowiem bardziej prawdopodobne, że to my popełnimy błąd w naszej aplikacji, niż że błąd ten powstanie w wyniku niewłaściwe działającego mechanizmu obsługi ograniczeń w serwerze bazy. Poza tym nałożenie odpowiednich ograniczeń bazodanowych jest prawie zawsze dużo mniej czasochłonne i o wiele łatwiejsze niż implementacja podobnej funkcjonalności na poziomie aplikacji.

**Przykład 95**




```sql
DROP TABLE IF EXISTS osoby;

CREATE TABLE osoby
(
   osoba_id              INT           NOT NULL,
   imie                  VARCHAR(20)   NOT NULL,
   nazwisko              VARCHAR(30)   NOT NULL,
   pseudo                VARCHAR(10)   NOT NULL,  -- tu też jest przecinek
   PRIMARY KEY (osoba_id),
   UNIQUE (imie, nazwisko, pseudo)
)
ENGINE = InnoDB;
```

**Komentarz:**

Ponieważ ograniczenie `UNIQUE` dotyczy trzech kolumn jednocześnie, więc musi być zdefiniowane jako *kolumnowe*. Czyli definiujemy je dopiero PO zdefiniowaniu wszystkich kolumn. W zasadzie każde ograniczenie może być w ten sposób tworzone. W przykładzie powyżej w taki właśnie sposób zdefiniowano ograniczenie `PRIMARY KEY`. W poprzednim przykładzie `PRIMARY KEY` zostało zdefiniowane jako *kolumnowe*. Funkcjonalnie obie metody są sobie całkowicie równoważne.

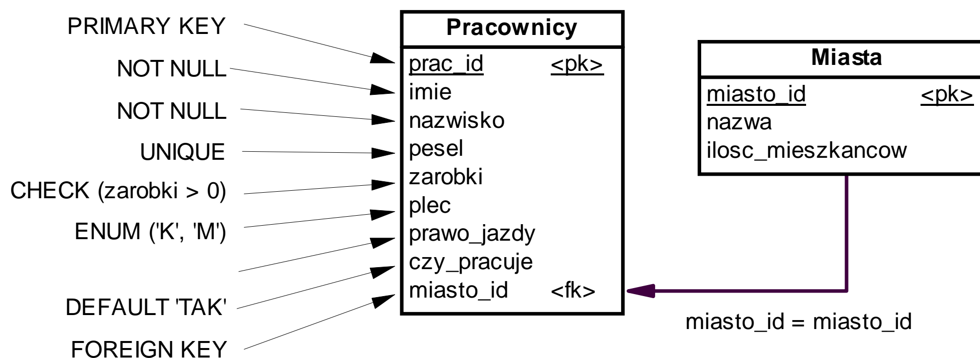
Każde ograniczenie można również zdefiniować za pomocą polecenia `ALTER TABLE`. Wówczas najpierw tworzymy tabelę bez ograniczeń a następnie ograniczenia te dodajemy. Powyższy przykład można więc zapisać następująco:

```sql
CREATE TABLE osoby (
   osoba_id           INT,
   imie               VARCHAR(20),
   nazwisko           VARCHAR(30),
   pseudo             VARCHAR(10)
)
ENGINE = InnoDB;

ALTER TABLE osoby ADD PRIMARY KEY (osoba_id);
ALTER TABLE osoby ADD UNIQUE (imie, nazwisko, pseudo);
```

lub

```
ALTER TABLE osoby ADD CONSTRAINT osoby_osoba_id_pk PRIMARY KEY (osoba_id);
ALTER TABLE osoby ADD CONSTRAINT osoby_inp_uq UNIQUE (imie, nazwisko, pseudo);
```

## Przykład 96




```
-- Krok 1
CREATE TABLE pracownicy
(
...
   miasto_id        INTEGER
)
ENGINE = InnoDB;

-- Krok 2
CREATE TABLE miasta
(
miasto_id          INTEGER      PRIMARY KEY,
nazwa              VARCHAR(50)  NOT NULL,
ilosc_mieszkancow  INTEGER
)
ENGINE = InnoDB;

-- Krok 3
ALTER TABLE pracownicy
ADD FOREIGN KEY (miasto_id)
REFERENCES miasta (miasto_id);

-- lub
ALTER TABLE pracownicy
ADD CONSTRAINT pracownicy_miasta_id_fk FOREIGN KEY (miasto_id)
REFERENCES miasta (miasto_id);
```

**Komentarz:**

Ostatnie polecenie SQL tworzy klucz obcy (inaczej: *ograniczenie*) na tabeli *pracownicy*. Ograniczeniom warto jednak nadawać nazwy, aby łatwiej je było potem zidentyfikować. Gdy jawnie nie nadamy ograniczeniu nazwy zrobi to za nas MySQL. Jednak własne nazwy

zwykle są lepsze, bo możemy bardziej dopasować ją do naszych potrzeb (celem np. łatwiejszego zapamiętania). Druga wersja polecenia `ALTER TABLE` będzie bardziej „poprawna”:

Zwróćmy uwagę, że nazwa ograniczenia została utworzona wg. schematu: *nazwa_ tabeli-nazwa_ kolumny-fk*. Gdy będziemy konsekwentnie przestrzegać tej konwencji zawsze łatwo z samej nazwy ograniczenia „wyczytamy”, o którą tabelę oraz o którą kolumnę chodzi.

Zwróćmy uwagę na pojawiający się w definicjach obu tabel fragment `ENGINE=InnoDB`. W kontekście obsługi kluczy obcych dodanie tej opcji jest obowiązkowe. Co prawda w wersji 5.x serwera MySQL opcja ta jest ustawiana domyślnie, ale nie zaszkodzi (choćby w celach informacyjnych i dokumentacyjnych) ją jawnie podać.

Aby przekonać się, że wszystko utworzyło się po naszej myśli możemy wydać poniższe polecenie[4]. W wyświetlonej tabeli widzimy zdefiniowaną przez nas nazwę ograniczenia:

```
SELECT
  constraint_name, table_schema, table_name, constraint_type
FROM
  information_schema.table_constraints
WHERE
  table_schema = 'blab' and table_name='pracownicy';
```

```
+-------------------------+--------------+------------+-----------------+
| constraint_name         | table_schema | table_name | constraint_type |
+-------------------------+--------------+------------+-----------------+
| PRIMARY                 | blab         | pracownicy | PRIMARY KEY     |
| pesel                   | blab         | pracownicy | UNIQUE          |
| pracownicy_miasta_id_fk | blab         | pracownicy | FOREIGN KEY     |
+-------------------------+--------------+------------+-----------------+
```

### Przykład 97

```
DROP TABLE IF EXISTS pracownicy;

CREATE TABLE pracownicy (
   prac_id          INTEGER               PRIMARY KEY AUTO_INCREMENT,
   imie             VARCHAR(20)           NOT NULL,
   nazwisko         VARCHAR(30)           NOT NULL,
   pesel            INTEGER               NOT NULL UNIQUE,
   zarobki          DECIMAL(11,2)         CHECK (zarobki > 0),
   plec             ENUM ('M', 'K')       NOT NULL,
   prawo_jazdy      SET ('A', 'B', 'C',
                         'D', 'CE', 'BE',
                         'DE')            NOT NULL,
   czy_pracuje      CHAR(3)               DEFAULT 'TAK',
   miasto_id        INTEGER,              -- teraz tutaj jest przecinek
   CONSTRAINT pracownicy_miasta_id_fk
   FOREIGN KEY (miasto_id)
   REFERENCES miasta (miasto_id))
ENGINE = InnoDB;
```

[4]Korzystamy tutaj ze specjalnej „systemowej” bazy danych o nazwie *information_ schema*. W bazie tej przechowywane są różne informacje na temat innych baz. Jest to więc swego rodzaju *baza metadanych*. Pojawiła się ona dopiero w wersji 5 serwera MySQL. Szczegóły patrz dokumentacja serwera.

**Komentarz:**

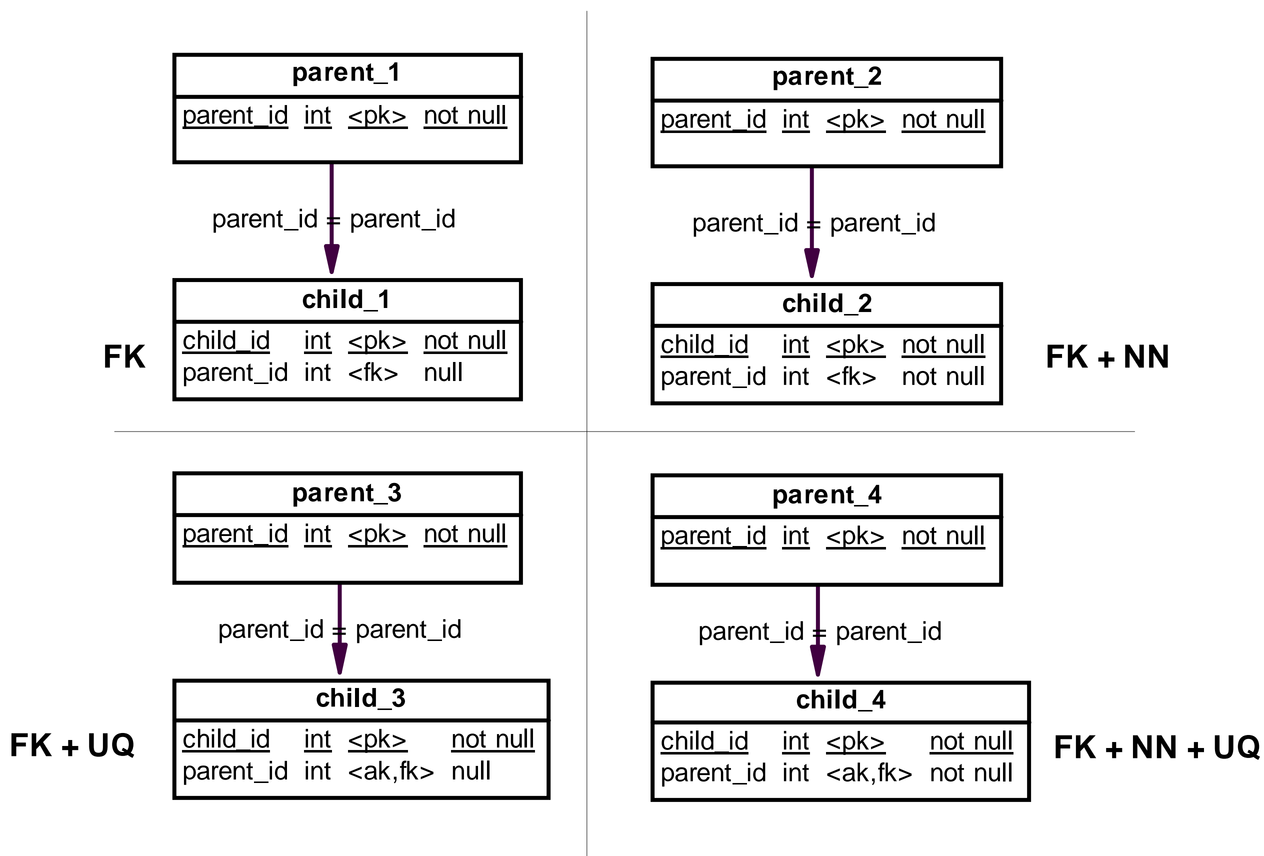
Przykład ten jest analogiczny do poprzedniego. Jedyna różnica to ta, że tym razem tworzymy klucz obcy w momencie tworzenia tabeli *pracownicy*.

Zwróćmy uwagę, że tym razem kolejność tworzenia tabel jest ściśle określona. Najpierw musimy utworzyć tabelę nadrzędną (`miasta`) a dopiero potem tabelę podrzędną (`pracownicy`). Gdy klucze obce tworzone są z wykorzystaniem polecenie `ALTER TABLE`, kolejność tworzenia tabel jest nieistotna. Oczywistym wymaganiem jest natomiast, aby `ALTER TABLE` pojawiło się *po* poleceniach `CREATE TABLE`.

Po utworzeniu tabel możemy wydać polecenie `SHOW CREATE TABLE`, które potwierdzi nam, że tabela utworzyła się zgodnie z naszymi zamierzeniami:

```
+------------+---------------------------------------------------------------+
| pracownicy | CREATE TABLE ‘pracownicy‘ (
  ‘prac_id‘ int(11) NOT NULL auto_increment,
  ‘imie‘ varchar(20) NOT NULL,
  ‘nazwisko‘ varchar(30) NOT NULL,
  ‘pesel‘ int(11) NOT NULL,
  ‘zarobki‘ decimal(11,2) default NULL,
  ‘plec‘ enum(’M’,’K’) NOT NULL,
  ‘prawo_jazdy‘ set(’A’,’B’,’C’,’D’,’CE’,’BE’,’DE’) NOT NULL,
  ‘czy_pracuje‘ char(3) default ’TAK’,
  ‘miasto_id‘ int(11) default NULL,
  PRIMARY KEY  (‘prac_id‘),
  UNIQUE KEY ‘pesel‘ (‘pesel‘),
  KEY ‘pracownicy_miasta_id_fk‘ (‘miasto_id‘),
  CONSTRAINT ‘pracownicy_miasta_id_fk‘ FOREIGN KEY (‘miasto_id‘)
  REFERENCES ‘miasta‘ (‘miasto_id‘)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 |
+------------+----------------------------------------------------------------+
```

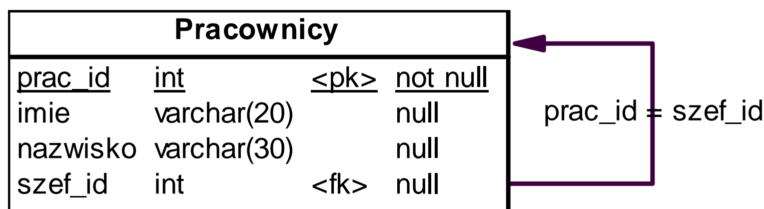
**Przykład 98**




**Komentarz:**

Na powyższym rysunku pokazano wszystkie 4 możliwe przypadki zdefiniowania ograniczenia `FOREIGN KEY`. Zwróćmy uwagę, że dla jednej kolumny można zdefiniować więcej niż jedno ograniczenie (np. `FOREIGN KEY` oraz `UNIQUE` oraz `NOT NULL` w ostatnim przypadku). Symbolem `<ak>` na rysunku oznaczone jest ograniczenie `UNIQUE`.

**Przykład 99**



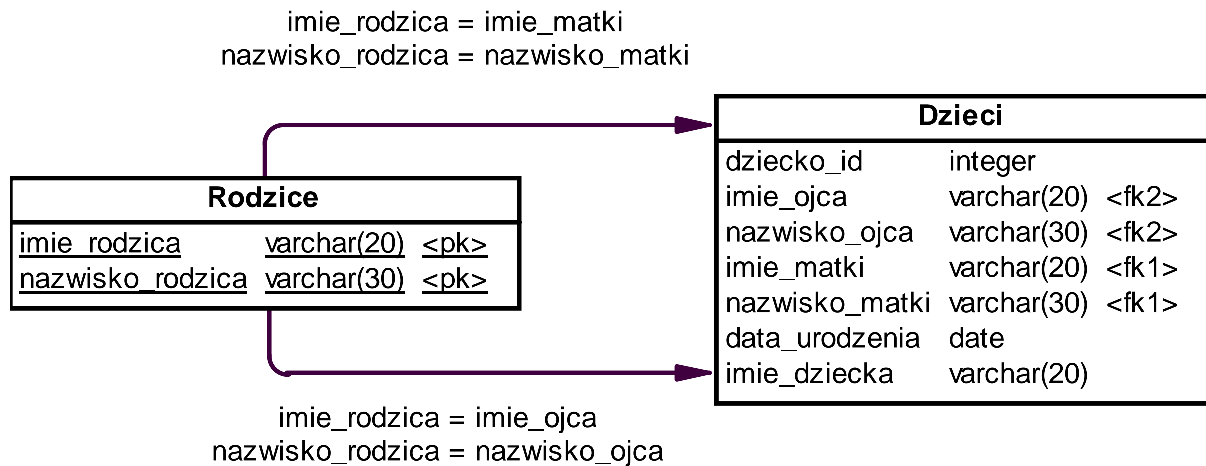

```
DROP TABLE IF EXISTS pracownicy;

CREATE TABLE pracownicy (
   prac_id          INT   PRIMARY KEY,
   szef_id          INT
) ENGINE = InnoDB;

ALTER TABLE pracownicy
ADD CONSTRAINT pracownicy_szef_id_fk
```

```
FOREIGN KEY (szef_id)
REFERENCES pracownicy (prac_id);
```

**Komentarz:**

Zdefiniowano ograniczenie `FOREIGN KEY` typu „same do siebie" (ang. *self join*). Definicja takiego ograniczenia niczym w istocie nie różni się od definicji `FOREIGN KEY` w oddzielnej tabeli.

### Przykład 100




```
DROP TABLE IF EXISTS dzieci;
DROP TABLE IF EXISTS rodzice;

CREATE TABLE rodzice (
  imie_rodzica     VARCHAR(20) NOT NULL,
  nazwisko_rodzica VARCHAR(30) NOT NULL,
  PRIMARY KEY (imie_rodzica, nazwisko_rodzica)
)
TYPE=InnoDB;

CREATE TABLE dzieci (
  dziecko_id     INTEGER     NOT NULL,
  imie_ojca      VARCHAR(20)     NULL,
  nazwisko_ojca  VARCHAR(30)     NULL,
  imie_matki     VARCHAR(20)     NULL,
  nazwisko_matki VARCHAR(30)     NULL,
  data_urodzenia DATE        NOT NULL,
  imie_dziecka   VARCHAR(20) NOT NULL,
  PRIMARY KEY (Dziecko_id)
)
TYPE=InnoDB;

ALTER TABLE dzieci ADD FOREIGN KEY (imie_ojca, nazwisko_ojca)
    REFERENCES Rodzice(imie_rodzica, nazwisko_rodzica);

ALTER TABLE dzieci ADD FOREIGN KEY (imie_matki, nazwisko_matki)
```

```
    REFERENCES Rodzice (imie_rodzica, nazwisko_rodzica);
```

**Komentarz:**

Pokazano nieco bardziej złożony przykład z kluczami obcymi. Rejestrujemy informacje o dzieciach oraz o ich rodzicach (ojciec oraz matka). Klucz główny jest złożony (składa się z dwóch kolumn) a w związku z tym również i klucze obce są złożone.

Dla uproszczenia klucze obce nie mają zdefiniowanych własnych nazw (serwer MySQL nada im „własne" nazwy).

## 7.4 Obsługa ograniczeń w MySQL

### Przykład 101

```
SELECT @@sql_mode;
```

```
+------------+
| @@sql_mode |
+------------+
|            |
+------------+
```

```
SET SQL_MODE = STRICT_TRANS_TABLES;
```

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

```
SELECT @@sql_mode;
```

```
+---------------------+
| @@sql_mode          |
+---------------------+
| STRICT_TRANS_TABLES |
+---------------------+
```

**Komentarz:**

Powyżej pokazano w jaki sposób sprawdzić w jakim aktualnym trybie pracuje serwer MySQL oraz w jaki sposób można ten tryb zmienić. Możemy ustawić kilkanaście różnych opcji (w przykładzie powyżej ustawiono tryb `STRICT_TRANS_TABLES`), które wpływają na:

- pewne szczegóły dotyczące obsługiwanej składni języka SQL,
- na sposób, w jaki MySQL dokonuje weryfikacji poprawności wprowadzanych danych.

W tym miejscu zajmiemy się tylko tym drugim przypadkiem. Domyślnie żaden tryb nie jest ustawiony.

### Przykład 102

```sql
-- Sprawdzamy w jakim trybie aktualnie pracuje serwer.

SELECT @@SQL_MODE;
```

```
+------------+
| @@SQL_MODE |
+------------+
|            |
+------------+
```

```sql
-- Tworzymy nową tabelę do testowania.

DROP TABLE IF EXISTS test;

CREATE TABLE test (
  liczba  TINYINT         NOT NULL
)
ENGINE = InnoDB;
```

```sql
-- Wstawiamy rekord, gdzie naruszamy ograniczenie NOT NULL.
-- Rekordu nie da się wstawić.

INSERT INTO test VALUES (NULL);
```

```
ERROR 1048 (23000): Column 'liczba' cannot be null
```

```sql
-- Tym razem używamy wielowierszowej wersji polecenia INSERT.
-- Serwer generuje ostrzeżenie.
-- Za pomocą polecenia SHOW WARNINGS oglądamy tekst ostrzeżenia.

INSERT INTO test VALUES (1), (NULL), (2);
```

```
Query OK, 3 rows affected, 1 warning (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 1
```

```sql
SHOW WARNINGS;
```

```
+---------+------+----------------------------------------------------------+
| Level   | Code | Message                                                  |
+---------+------+----------------------------------------------------------+
| Warning | 1263 | Column set to default value; NULL supplied to NOT NULL ...|
+---------+------+----------------------------------------------------------+
```

```sql
-- Patrzymy, co zostało wstawione do tabeli.
-- Okazuje się, że serwer zamiast wartości NULL wstawił pewną wartość domyślną
-- (jest to zero dla pól numerycznych, pusty string '' dla pól znakowych
-- oraz wartość "zero" dla pól typu data i czas).

SELECT * FROM test;
```

```
+--------+
| liczba |
+--------+
|      1 |
|      0 |
|      2 |
+--------+
```

### Przykład 103

```
-- Zmieniamy tryb pracy serwera.

SET SQL_MODE = STRICT_TRANS_TABLES;

-- Sprawdzamy w jakim trybie aktualnie pracuje serwer.

SELECT @@SQL_MODE;
```

```
+---------------------+
| @@SQL_MODE          |
+---------------------+
| STRICT_TRANS_TABLES |
+---------------------+
```

```
-- Wstawiamy rekord, gdzie naruszamy ograniczenie NOT NULL.
-- Rekordu nie da się wstawić.
-- Serwer zachowuje się tak samo, jak w poprzednim przykładzie.

INSERT INTO test VALUES (NULL);
```

```
ERROR 1048 (23000): Column 'liczba' cannot be null
```

```
-- Tym razem używamy wielowierszowej wersji polecenia INSERT.
-- Serwer generuje błąd. Zachowuje się inaczej niż w poprzednim przykładzie.

INSERT INTO test VALUES (1), (NULL), (2);
```

```
ERROR 1263 (22004): Column set to default value; NULL supplied to
                    NOT NULL column 'liczba' at row 2
```

```
-- Tym razem żaden rekord nie zostaje wprowadzony.

SELECT * FROM test;
```

```
Empty set (0.01 sec)

mysql>
```

### Komentarz:

Widzimy, że ustawienie trybu `STRICT_TRANS_TABLES` dość diametralnie zmienia zachowanie się serwera MySQL. Nie wstawia on już domyślnych wartości. Takie zachowanie

wydaje się być bardziej naturalne i bezpieczne, bowiem wartości domyślne zwykle nie są przez nas pożądane. Niestety (?) w domyślnym trybie pracy serwer nie ma ustawionego parametru STRICT_TRANS_TABLES.

### Przykład 104

```
-- Sprawdzamy w jakim trybie aktualnie pracuje serwer.

SELECT @@SQL_MODE;
```

```
+---------------------+
| @@SQL_MODE          |
+---------------------+
| STRICT_TRANS_TABLES |
+---------------------+
```

```
-- Tworzymy nową tabelę do testowania.

DROP TABLE IF EXISTS test;

CREATE TABLE test (
  licz    TINYINT                NOT NULL,
  znak    VARCHAR(3)             NOT NULL,
  data    DATETIME               NOT NULL,
  enum_nn ENUM ('a', 'b', 'c')  NOT NULL,
  enum_n  ENUM ('a', 'b', 'c')      NULL,
  set_nn  SET  ('x', 'y', 'z')  NOT NULL,
  set_n   SET  ('x', 'y', 'z')      NULL
)
ENGINE = InnoDB;
```

```
-- Wstawiamy 3 rekordy.
-- Używamy wielowierszowej wersji polecenia INSERT.
-- Drugi i trzeci rekord są "błędne". Serwer generuje 5 ostrzeżeń.

INSERT INTO test VALUES
  (1   ,  'ag'   ,  '2005-04-18',  'a' ,  'b' ,  'x,y,z',  'x,z'),
  (NULL,  NULL   ,  NULL        ,  NULL,  NULL,  NULL   ,  NULL ),
  (999 ,  'aaaa',  '2005-02-31',  'x' ,  'x' ,  'a,b'  ,  'a,b');
```

```
Query OK, 2 rows affected, 5 warnings (0.03 sec)
Records: 2  Duplicates: 0  Warnings: 5
```

```
-- Oglądamy wstawione rekordy. Zwróćmy uwagę, jakie wartości domyślne zostały
-- wstawione w drugim i trzecim rekordzie. W polu liczbowym (TINYINT) oraz
-- znakowym (VARCHAR(3)) przekroczono dopuszczalne zakresy wartości.
-- Pola ENUM oraz SET też posiadają błędne wartości.

SELECT * FROM test;
```

```
+------+------+---------------------+---------+--------+--------+-------+
| licz | znak | data                | enum_nn | enum_n | set_nn | set_n |
```

```
+------+------+---------------------+---------+--------+--------+-------+
|    1 | ag   | 2005-04-18 00:00:00 | a       | b      | x,y,z  | x,z   |
|    0 |      | 0000-00-00 00:00:00 |         | NULL   |        | NULL  |
|  127 | aaa  | 0000-00-00 00:00:00 |         |        |        |       |
+------+------+---------------------+---------+--------+--------+-------+
```

## 7.5 Indeksy

### Przykład 105

```sql
DROP TABLE IF EXISTS pracownicy;

CREATE TABLE pracownicy (
   prac_id          INTEGER PRIMARY KEY,  -- Indeks tworzony jest automatycznie.
   imie             VARCHAR(20),
   nazwisko         VARCHAR(30),
   pseudo           VARCHAR(10) UNIQUE,   -- UNIQUE to też rodzaj indeksu.
   data_ur          DATE,
   INDEX (nazwisko),                      -- INDEX oraz KEY to synonimy.
   KEY (data_ur)                          -- Można stosować zamiennie.
) ENGINE = InnoDB;
```

**Komentarz:**

Indeks to rodzaj spisu treści tabeli, który pozwala szybko znaleźć wybrane wiersze. Jego działanie jest bardzo zbliżone do tradycyjnego indeksu spotykanego w książkach.

Jeżeli indeks został utworzony na pewnej kolumnie X, możemy użyć go do bardzo szybkiego odszukania interesującej nas wartości w tej kolumnie. Indeks wskaże, w którym miejscu tabeli (w którym wierszu lub wierszach) znajduje się poszukiwana wartość i wówczas możemy bardzo szybko przejść do tego wiersza.

Jeżeli na pewnej kolumnie nie jest założony indeks, serwer musi sekwencyjnie przeszukiwać tabelę aby odnaleźć wiersz (wiersze) z poszukiwaną wartością. Przy dużej ilości rekordów poszukiwanie to może być bardzo czasochłonne.

Indeksy mogą dotyczyć pojedynczych kolumn, lub też mogą być zakładane na wielu kolumnach (podobnie jak np. klucze główne i obce).

Aby zauważyć zwiększenie szybkości wykonywania się poleceń SQL dla tabel z założonymi indeksami, muszą one zawierać stosunkowo dużo rekordów (rzędu tysięcy lub nawet więcej). Używany przez nas model demonstracyjny zawiera bardzo niewiele danych (od kilku do nieco tylko ponad stu rekordów w poszczególnych tabelach). Dlatego też nie będziemy w stanie zauważyć pozytywnego działania indeksów. Niezależnie od tego, czy indeksy są założone, czy też ich nie ma, zapytania będą zwracały wyniki praktycznie „natychmiast".

Problem właściwego doboru indeksów jest dość trudnym zagadnieniem. Zarówno brak indeksów jak i ich nadmierna ilość mogą mieć bardzo niekorzystne skutki dla działania bazy danych. Bardzo często indeksy należy dobierać metodą „prób i błędów". Przy poszukiwaniu przyczyn zbyt wolno wykonujących się zapytań często korzystamy z tzw. dziennika wolno realizowanych zapytań (ang. *Slow Query Log*) oraz polecenia `EXPLAIN`. Szczegóły

patrz dokumentacja [3]. Warto dokładnie zapoznać się z rozdziałami *How MySQL Uses Indexes* oraz *Optimization*.

Należy pamiętać o dwóch rzeczach. Po pierwsze każdy utworzony indeks pochłania pewną dodatkową przestrzeń dyskową. Indeksy są bowiem obiektami materialnymi i jako takie przechowywane są na dysku. Ponadto po wykonaniu takich poleceń jak `INSERT`, `UPDATE` lub `DELETE` wymagane jest *przebudowanie indeksu* (współczesne serwery bazodanowe dokonują tego automatycznie) i operacja ta zajmuje oczywiście pewną ilość czasu oraz pochłania pewną ilość zasobów komputera (np. pamięć operacyjna). W niesprzyjających okolicznościach może się więc zdarzyć, że istnienie indeksów może bardziej szkodzić niż pomagać.

### Przykład 106

```
DROP TABLE IF EXISTS pracownicy;
DROP TABLE IF EXISTS miasta;

CREATE TABLE pracownicy (
  prac_id           INTEGER PRIMARY KEY,
  imie              VARCHAR(20),
  nazwisko          VARCHAR(30),
  miasto_id         INTEGER
)
ENGINE = InnoDB;

CREATE TABLE miasta (
  miasto_id         INTEGER      PRIMARY KEY,
  nazwa             VARCHAR(50)  NOT NULL
)
ENGINE = InnoDB;

ALTER TABLE pracownicy ADD FOREIGN KEY (miasto_id)
  REFERENCES miasta (miasto_id);
```

```
SHOW CREATE TABLE pracownicy;
```

```
+------------+---------------------------------------------------+
| Table      | Create Table                                      |
+------------+---------------------------------------------------+
| pracownicy | CREATE TABLE `pracownicy` (
  `prac_id` int(11) NOT NULL,
  `imie` varchar(20) default NULL,
  `nazwisko` varchar(30) default NULL,
  `miasto_id` int(11) default NULL,
  PRIMARY KEY  (`prac_id`),
  KEY `miasto_id` (`miasto_id`),
  CONSTRAINT `pracownicy_ibfk_1` FOREIGN KEY (`miasto_id`)
  REFERENCES `miasta` (`miasto_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 |
+------------+---------------------------------------------------+
```

**Komentarz:**

Tworzymy dwie „klasyczne" tabele. Następnie za pomocą polecenia `SHOW CREATE TABLE` oglądamy strukturę tabeli podrzędnej. Zwróćmy uwagę na fragment `KEY 'miasto_id' ('miasto_id')`, który jest odpowiedzialny za zaindeksowanie kolumny z kluczem obcym. Indeks ten tworzony jest automatycznie[5]. Nic jednak nie stoi na przeszkodzie, aby utworzyć go jawnie.

Kolumna (kolumny) klucza głównego również są automatycznie indeksowane. Podobnie dzieje się, gdy definiujemy ograniczenie `UNIQUE`.

[5]We wcześniejszych wersjach MySQL-a indeks ten należało utworzyć ręcznie

# Rozdział 8

# Polecenie ALTER

**Przykład 107**

```
DROP TABLE IF EXISTS pracownicy;
DROP TABLE IF EXISTS prac;

CREATE TABLE pracownicy (
   prac_id          INTEGER,
   imie             VARCHAR(10),
   nazwisko         VARCHAR(10)
);

ALTER TABLE pracownicy RENAME TO prac;

ALTER TABLE prac MODIFY COLUMN imie VARCHAR(20) NOT NULL;
ALTER TABLE prac MODIFY COLUMN nazwisko VARCHAR(30) NOT NULL;

ALTER TABLE prac ADD COLUMN zarobki DECIMAL(11,2) NOT NULL;
ALTER TABLE prac ADD COLUMN plec ENUM('M', 'K') NOT NULL;
ALTER TABLE prac ADD COLUMN pseudo VARCHAR(10) UNIQUE;
ALTER TABLE prac ADD COLUMN imie2 VARCHAR(20) NOT NULL;

ALTER TABLE prac ADD PRIMARY KEY (prac_id);
ALTER TABLE prac CHANGE COLUMN pseudo ksywka VARCHAR(8);
ALTER TABLE prac ADD UNIQUE (ksywka);
ALTER TABLE prac DROP COLUMN imie2;
```

**Komentarz:**

Utworzono tabelę `pracownicy` a następnie wykonano kilka zmian. Polecenia zmieniające definicję tabeli są „samodokumentujące się”, więc można z łatwością domyśleć się ich działania. Zwróćmy uwagę, że modyfikowaliśmy tabelę, która nie zawiera żadnych danych. W takiej sytuacji możliwe jest wykonanie praktycznie każdej modyfikacji. Gdy tabela zawiera już jakieś dane nie wszystkie operacje są dopuszczalne. Przykładowo nie można zmniejszyć długości pola typu `VARCHAR`, gdy znajdują się w nim już jakieś wartości o długości większej niż nowa długość planowana dla kolumny[1]. Kolejny przykład podaje więcej

[1]Pod tym względem MySQL jest dużo bardziej „wyrozumiały” jak inne serwery relacyjnych baz danych, np Oracle. Wspomniana zmiana będzie możliwa. Istniejące już dane nie ulegną zmianie, jednak nowe będą

szczegółów na ten temat.

**Przykład 108**

```
DROP TABLE IF EXISTS test;

CREATE TABLE test (
  id    INT          NOT NULL PRIMARY KEY,
  kol   VARCHAR(3)
);

INSERT INTO test VALUES (1, 'aaa'), (2, 'bb'), (3, 'c');
```

```
ALTER TABLE test MODIFY COLUMN id INT NULL;
```

```
Query OK, 3 rows affected (0.35 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
DESC test;
```

```
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| id    | int(11)    | NO   | PRI | 0       |       |
| kol   | varchar(3) | YES  |     | NULL    |       |
+-------+------------+------+-----+---------+-------+
```

**Komentarz:**

Dość dziwne zachowanie się MySQL, który pozornie pozwala ustawić na kolumnie z ograniczeniem `PRIMARY KEY` atrybut `NULL` (nie pojawia się żadne choćby ostrzeżenie). Po wykonaniu polecenia `DESC` okazuje się, że kolumna ma nadal ograniczenie `PRIMARY KEY`, co automatycznie oznacza, że jest nadal ustawiony atrybut `NOT NULL`.

```
ALTER TABLE test MODIFY COLUMN kol VARCHAR(1);
```

```
Query OK, 3 rows affected (0.08 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
DESC test;
```

```
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| id    | int(11)    | NO   | PRI |         |       |
| kol   | varchar(1) | YES  |     | NULL    |       |
+-------+------------+------+-----+---------+-------+
```

```
SELECT * FROM test;
```

już wymagały nowych formatów.

```
+----+------+
| id | kol  |
+----+------+
|  1 | aaa  |
|  2 | bb   |
|  3 | c    |
+----+------+
```

**Komentarz:**

Kolejne dziwne zachowanie się MySQL. Tym razem próbujemy zmniejszyć szerokość kolumny na `VARCHAR(1)`, mimo tego, że w tabeli są już rekordy dłuższe niż 1. MySQL tym razem rzeczywiście zmienia definicję kolumny, jednak istniejące dane nie zostają zmienione. Rodzi się więc pytanie: skoro kolumna jest teraz `VARCHAR(1)` to jak należy interpretować istnienie w niej wartości `aaa` lub `bb` ?

```sql
INSERT INTO test VALUES (4, 'ddd');
SHOW WARNINGS;
```

```
Query OK, 1 row affected, 1 warning (0.03 sec)
```

```sql
SHOW WARNINGS;
```

```
+---------+------+------------------------------------------+
| Level   | Code | Message                                  |
+---------+------+------------------------------------------+
| Warning | 1265 | Data truncated for column 'kol' at row 1 |
+---------+------+------------------------------------------+
```

```sql
SELECT * FROM test;
```

```
+----+------+
| id | kol  |
+----+------+
|  1 | aaa  |
|  2 | bb   |
|  3 | c    |
|  4 | d    |
+----+------+
```

**Komentarz:**

Próba wstawienia do tabeli danych przekraczających rozmiar kolumny kończy się powodzeniem, jednak serwer ostrzega nas (ang. *warning*), że dane zostały obcięte. Takie zachowanie się serwera jest już bardziej zrozumiałe w porównaniu do sytuacji, gdy nie pojawia się żadne ostrzeżenie, jak to pokazano we wcześniejszych przykładach.

### Przykład 109

```sql
DROP TABLE IF EXISTS test;

CREATE TABLE test (
```

```
  id     INT           NOT NULL,
  kol    VARCHAR(3)
);

INSERT INTO test VALUES (1, 'A'), (1, 'B');
```

```
ALTER TABLE test ADD PRIMARY KEY (id);
```

```
ERROR 1062 (23000): Duplicate entry '1' for key 1
```

**Komentarz:**

Tym razem serwer zachował się logicznie. Nie pozwolił oczywiście na ustawienie na kolumnie `id` ograniczenia `PRIMARY KEY`, gdyż w tabeli są już powtarzające się dane.

```
ALTER TABLE test MODIFY COLUMN kol ENUM ('X', 'Y');
```

```
Query OK, 2 rows affected, 2 warnings (0.10 sec)
Records: 2  Duplicates: 0  Warnings: 2
```

```
SHOW WARNINGS;
```

```
+---------+------+------------------------------------------+
| Level   | Code | Message                                  |
+---------+------+------------------------------------------+
| Warning | 1265 | Data truncated for column 'kol' at row 1 |
| Warning | 1265 | Data truncated for column 'kol' at row 2 |
+---------+------+------------------------------------------+
```

```
SELECT * FROM test;
```

```
+----+------+
| id | kol  |
+----+------+
|  1 |      |
|  1 |      |
+----+------+
```

**Komentarz:**

Kolejne dość dziwne zachowanie się serwera MySQL. Dopuścił on mianowicie do zmiany definicji kolumny (kolumna `kol` została ustawiona na `ENUM ('X', 'Y')`). Dane, które nie spełniają warunków podanych w `ENUM` zostały usunięte !!!

# Rozdział 9

# Polecenie DROP

### Przykład 110

```
DROP TABLE emp;
```

```
ERROR 1217 (23000): Cannot delete or update a~parent row:
                    a~foreign key constraint fails
```

**Komentarz:**

Próbujemy wykasować tabelę, do której odwołują się klucze obce (z pola `id` tabeli `emp` korzysta kilka innych tabel). System odmówi wykonania takiej operacji. Aby móc wykasować tabelę `emp` musimy najpierw usunąć wszystkie powiązane rekordy z tabel `ord`, `customer`, `warehouse`. Alternatywą jest zdefiniowane tabeli z opcją `ON DELETE CASCADE`. Wówczas system bez żadnego pytania usunie wszystkie ew. istniejące powiązane rekordy. Oczywiście opcję tą należy używać niezwykle ostrożnie. Można bowiem nieświadomie utracić bardzo dużo danych!

### Przykład 111

```
DROP TABLE IF EXISTS child;
DROP TABLE IF EXISTS parent;

CREATE TABLE parent (
    parent_id       INT    PRIMARY KEY
) ENGINE = InnoDB;

CREATE TABLE child (
    child_id        INT    PRIMARY KEY,
    parent_id       INT
) ENGINE = InnoDB;

ALTER TABLE child
ADD CONSTRAINT child_parent_id_fk
FOREIGN KEY (parent_id)
REFERENCES parent (parent_id)
ON DELETE CASCADE;
```

```
INSERT INTO parent VALUES (1);
```

```sql
INSERT INTO parent VALUES (2);

INSERT INTO child  VALUES (1, 1);
INSERT INTO child  VALUES (2, 1);
INSERT INTO child  VALUES (3, 1);
INSERT INTO child  VALUES (4, 2);
INSERT INTO child  VALUES (5, 2);
INSERT INTO child  VALUES (6, 2);
```

```sql
SELECT * FROM child;
```

```
+----------+-----------+
| child_id | parent_id |
+----------+-----------+
|        1 |         1 |
|        2 |         1 |
|        3 |         1 |
|        4 |         2 |
|        5 |         2 |
|        6 |         2 |
+----------+-----------+
```

```sql
DELETE FROM parent WHERE parent_id = 1;
```

```
Query OK, 1 row affected (0.02 sec)
```

```sql
SELECT * FROM child;
```

```
+----------+-----------+
| child_id | parent_id |
+----------+-----------+
|        4 |         2 |
|        5 |         2 |
|        6 |         2 |
+----------+-----------+
3 rows in set (0.00 sec)
```

**Komentarz:**

Utworzyliśmy ograniczenie `FOREIGN KEY` z użyciem klauzuli `ON DELETE CASCADE`. Kasując więc jeden rekord z tabeli *parent* zostały również wykasowane (niejako w tle, bez jawnego informowania nas o tym!) trzy wiersze w tabeli *child*. Łatwo jest więc sobie wyobrazić sytuację, że usunięcie jednego rekordu pociąga za sobą utratę wielu tysięcy, często bezcennych, danych z innych tabel. Klauzulę `ON DELETE CASCADE` powinniśmy więc stosować bardzo rozsądnie i z umiarem.

W MySQL można również używać klauzuli `ON UPDATE CASCADE`, która działa analogicznie do `ON DELETE CASCADE`, z tym że definiuje się tutaj zachowanie rekordów w tabeli podrzędnej w przypadku modyfikacji rekordów w tabeli nadrzędnej.

### Przykład 112

```
-- tabele takie same jako w poprzednim przykładzie

ALTER TABLE child
ADD CONSTRAINT child_parent_id_fk
FOREIGN KEY (parent_id)
REFERENCES parent (parent_id)
ON DELETE CASCADE
ON UPDATE CASCADE;
```

```
SELECT * FROM child;
```

```
+----------+-----------+
| child_id | parent_id |
+----------+-----------+
|        1 |         1 |
|        2 |         1 |
|        3 |         1 |
|        4 |         2 |
|        5 |         2 |
|        6 |         2 |
+----------+-----------+
```

```
UPDATE parent SET parent_id = parent_id * 100;
```

```
Query OK, 2 rows affected (0.04 sec)
```

```
SELECT * FROM child;
```

```
+----------+-----------+
| child_id | parent_id |
+----------+-----------+
|        1 |       100 |
|        2 |       100 |
|        3 |       100 |
|        4 |       200 |
|        5 |       200 |
|        6 |       200 |
+----------+-----------+
6 rows in set (0.01 sec)
```

**Komentarz:**

Pokazano efekt działania klauzuli `ON UPDATE CASCADE`. Pełna składnia polecenia zmieniającego ograniczenie `FOREIGN KEY` wygląda następująco:

```
[CONSTRAINT symbol] FOREIGN KEY [id] (index_col_name, ...)
REFERENCES tbl_name (index_col_name, ...)
[ON DELETE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
[ON UPDATE {RESTRICT | CASCADE | SET NULL | NO ACTION}]
```

Znaczenie poszczególnych opcji jest następujące:

- `ON DELETE` ... — akcja podejmowana przy próbie wykasowania rekordów w tabeli nadrzędnej,
- `ON UPDATE` ... — akcja podejmowana przy próbie modyfikacji rekordów w tabeli nadrzędnej,
- `RESTRICT` — nie można wykasować ani zmienić rekordów w tabeli nadrzędnej, gdy istnieją powiązane rekordy w tabeli podrzędnej,
- `NO ACTION` — to samo co `RESTRICT`. Obie opcje są przyjmowane jako domyślne,
- `SET NULL` — po wykasowaniu lub modyfikacji rekordu w tabeli nadrzędnej, w tabeli podrzędnej ustawiane są wartości `NULL` (uwaga: nie zadziała, gdy kolumna klucza obcego będzie miała zdefiniowane ograniczenie `NOT NULL`),
- `CASCADE` — automatycznie kasuje (lub modyfikuje) wszystkie rekordy powiązane w tabeli podrzędnej. Opcja bardzo niebezpieczna! Stosować z rozwagą!,
- `SET DEFAULT` — w obecnej wersji serwera (5.0.16) opcja ta jest analizowana ale ignorowana.

# Rozdział 10

# Model demonstracyjny

Poniżej zamieszczono skrypt tworzący model demonstracyjny używany w większości przykładów. Na rysunku 10.1 pokazano natomiast schemat powiązań między poszczególnymi tabelami[1].

```
/*==============================================================*/
/* Database name:  MYSQL_TEST                                   */
/* DBMS name:      MySQL 5.0                                    */
/* Created on:     2005-12-10 02:12:32                          */
/*==============================================================*/

DROP TABLE IF EXISTS item;
DROP TABLE IF EXISTS inventory;
DROP TABLE IF EXISTS ord;
DROP TABLE IF EXISTS product;
DROP TABLE IF EXISTS warehouse;
DROP TABLE IF EXISTS customer;
DROP TABLE IF EXISTS emp;
DROP TABLE IF EXISTS dept;
DROP TABLE IF EXISTS region;
DROP TABLE IF EXISTS title;

/*==============================================================*/
/* Table: CUSTOMER                                              */
/*==============================================================*/
CREATE TABLE customer
(
   id                             INT                    NOT NULL AUTO_INCREMENT,
   name                           VARCHAR(50)            NOT NULL,
   phone                          VARCHAR(25),
   address                        VARCHAR(400),
   city                           VARCHAR(30),
   state                          VARCHAR(20),
   country                        VARCHAR(30),
   zip_code                       VARCHAR(75),
   credit_rating                  ENUM ('EXCELLENT', 'GOOD', 'POOR'),
   sales_rep_id                   INT,
   region_id                      INT,
   comments                       VARCHAR(255),
   CONSTRAINT customer_id_pk PRIMARY KEY (id)
)
ENGINE = InnoDB;

/*==============================================================*/
/* Table: DEPT                                                  */
/*==============================================================*/
CREATE TABLE dept
(
   id                             INT                     NOT NULL AUTO_INCREMENT,
   name                           VARCHAR(25)             NOT NULL,
```

[1] Zamieszczony model pochodzi z systemu Oracle w wersji 8 (*www.oracle.com*). Został on tylko nieznacznie zaadoptowany na potrzeby serwera MySQL. W oryginale model ten nosił nazwę `SUMMIT2`.

```
   region_id                   INT,
   CONSTRAINT dept_id_pk PRIMARY KEY (id),
   KEY dept_name_region_id_uk (name, region_id)
)
ENGINE = InnoDB;

/*============================================================*/
/* Table: EMP                                                 */
/*============================================================*/
CREATE TABLE emp
(
   id                          INT                    NOT NULL AUTO_INCREMENT,
   last_name                   VARCHAR(25)            NOT NULL,
   first_name                  VARCHAR(25),
   userid                      VARCHAR(8),
   start_date                  DATETIME,
   comments                    VARCHAR(255),
   manager_id                  INT,
   title                       VARCHAR(25),
   dept_id                     INT,
   salary                      NUMERIC(11,2),
   commission_pct              NUMERIC(4,2),
   CONSTRAINT emp_id_pk PRIMARY KEY (id),
   KEY emp_userid_uk (userid)
)
ENGINE = InnoDB;

/*============================================================*/
/* Table: INVENTORY                                           */
/*============================================================*/
CREATE TABLE inventory
(
   product_id                  INT                    NOT NULL,
   warehouse_id                INT                    NOT NULL,
   amount_in_stock             INT,
   reorder_point               INT,
   max_in_stock                INT,
   out_of_stock_explanation    VARCHAR(255),
   restock_date                DATETIME,
   CONSTRAINT inventory_prodid_warid_pk PRIMARY KEY (product_id, warehouse_id)
)
ENGINE = InnoDB;

/*============================================================*/
/* Table: ITEM                                                */
/*============================================================*/
CREATE TABLE item
(
   ord_id                      INT                    NOT NULL,
   item_id                     INT                    NOT NULL,
   product_id                  INT                    NOT NULL,
   price                       NUMERIC(11,2),
   quantity                    INT,
   quantity_shipped            INT,
   CONSTRAINT item_ordid_itemid_pk PRIMARY KEY (ord_id, item_id),
   KEY item_ordid_prodid_uk (ord_id, product_id)
)
ENGINE = InnoDB;

/*============================================================*/
/* Table: ORD                                                 */
/*============================================================*/
CREATE TABLE ord
(
   id                          INT                    NOT NULL AUTO_INCREMENT,
   customer_id                 INT                    NOT NULL,
   date_ordered                DATETIME,
   date_shipped                DATETIME,
   sales_rep_id                INT,
   total                       NUMERIC(11,2),
   payment_type                ENUM ('CASH', 'CREDIT'),
   order_filled                ENUM ('Y', 'N'),
   CONSTRAINT ord_id_pk PRIMARY KEY (id)
)
```

```
ENGINE = InnoDB;

/*==============================================================*/
/* Table: PRODUCT                                               */
/*==============================================================*/
CREATE TABLE product
(
   id                             INT                   NOT NULL AUTO_INCREMENT,
   name                           VARCHAR(50)           NOT NULL,
   short_desc                     VARCHAR(255),
   suggested_price                NUMERIC(11,2),
   CONSTRAINT product_id_pk PRIMARY KEY (id),
   KEY product_name_uk (name)
)
ENGINE = InnoDB;

/*==============================================================*/
/* Table: REGION                                                */
/*==============================================================*/
CREATE TABLE region
(
   id                             INT                   NOT NULL AUTO_INCREMENT,
   name                           VARCHAR(50)           NOT NULL,
   CONSTRAINT region_id_pk PRIMARY KEY (id),
   KEY region_name_uk (name)
)
ENGINE = InnoDB;

/*==============================================================*/
/* Table: TITLE                                                 */
/*==============================================================*/
CREATE TABLE title
(
   name                           VARCHAR(25)           NOT NULL,
   CONSTRAINT title_title_pk PRIMARY KEY (name)
)
ENGINE = InnoDB;

/*==============================================================*/
/* Table: WAREHOUSE                                             */
/*==============================================================*/
CREATE TABLE warehouse
(
   id                             INT                   NOT NULL AUTO_INCREMENT,
   region_id                      INT                   NOT NULL,
   address                        longtext,
   city                           VARCHAR(30),
   state                          VARCHAR(20),
   country                        VARCHAR(30),
   zip_code                       VARCHAR(75),
   phone                          VARCHAR(25),
   manager_id                     INT,
   CONSTRAINT warehouse_id_pk PRIMARY KEY (id)
)
ENGINE = InnoDB;

ALTER TABLE customer ADD CONSTRAINT customer_region_id_fk FOREIGN KEY (region_id)
      REFERENCES region (id);

ALTER TABLE customer ADD CONSTRAINT customer_sales_rep_id_fk FOREIGN KEY (sales_rep_id)
      REFERENCES emp (id);

ALTER TABLE dept ADD CONSTRAINT dept_region_id_fk FOREIGN KEY (region_id)
      REFERENCES region (id);

ALTER TABLE emp ADD CONSTRAINT emp_dept_id_fk FOREIGN KEY (dept_id)
      REFERENCES dept (id);

ALTER TABLE emp ADD CONSTRAINT emp_manager_id_fk FOREIGN KEY (manager_id)
      REFERENCES emp (id);

ALTER TABLE emp ADD CONSTRAINT emp_title_fk FOREIGN KEY (title)
      REFERENCES title (name);
```

```sql
ALTER TABLE inventory ADD CONSTRAINT inventory_product_id_fk FOREIGN KEY (product_id)
      REFERENCES product (id);

ALTER TABLE inventory ADD CONSTRAINT inventory_warehouse_id_fk FOREIGN KEY (warehouse_id)
      REFERENCES warehouse (id);

ALTER TABLE item ADD CONSTRAINT item_ord_id_fk FOREIGN KEY (ord_id)
      REFERENCES ord (id);

ALTER TABLE item ADD CONSTRAINT item_product_id_fk FOREIGN KEY (product_id)
      REFERENCES product (id);

ALTER TABLE ord ADD CONSTRAINT ord_customer_id_fk FOREIGN KEY (customer_id)
      REFERENCES customer (id);

ALTER TABLE ord ADD CONSTRAINT ord_sales_rep_id_fk FOREIGN KEY (sales_rep_id)
      REFERENCES emp (id);

ALTER TABLE warehouse ADD CONSTRAINT warehouse_manager_id_fk FOREIGN KEY (manager_id)
      REFERENCES emp (id);

ALTER TABLE warehouse ADD CONSTRAINT warehouse_region_id_fk FOREIGN KEY (region_id)
      REFERENCES region (id);


INSERT INTO region VALUES (1, 'North America');
INSERT INTO region VALUES (2, 'South America');
INSERT INTO region VALUES (3, 'Africa / Middle East');
INSERT INTO region VALUES (4, 'Asia');
INSERT INTO region VALUES (5, 'Europe');
COMMIT;

INSERT INTO title VALUES ('President');
INSERT INTO title VALUES ('Sales Representative');
INSERT INTO title VALUES ('Stock Clerk');
INSERT INTO title VALUES ('VP, Administration');
INSERT INTO title VALUES ('VP, Finance');
INSERT INTO title VALUES ('VP, Operations');
INSERT INTO title VALUES ('VP, Sales');
INSERT INTO title VALUES ('Warehouse Manager');
COMMIT;

INSERT INTO dept VALUES (10, 'Finance', 1);
INSERT INTO dept VALUES (31, 'Sales', 1);
INSERT INTO dept VALUES (32, 'Sales', 2);
INSERT INTO dept VALUES (33, 'Sales', 3);
INSERT INTO dept VALUES (34, 'Sales', 4);
INSERT INTO dept VALUES (35, 'Sales', 5);
INSERT INTO dept VALUES (41, 'Operations', 1);
INSERT INTO dept VALUES (42, 'Operations', 2);
INSERT INTO dept VALUES (43, 'Operations', 3);
INSERT INTO dept VALUES (44, 'Operations', 4);
INSERT INTO dept VALUES (45, 'Operations', 5);
INSERT INTO dept VALUES (50, 'Administration', 1);
COMMIT;

INSERT INTO emp VALUES
(1, 'Velasquez', 'Carmen', 'cvelasqu', '1990-03-03', NULL, NULL, 'President' , 50, 2500, NULL);
INSERT INTO emp VALUES
(2, 'Ngao', 'LaDoris', 'lngao', '1990-03-08', NULL, 1, 'VP, Operations', 41, 1450, NULL);
INSERT INTO emp VALUES
(3, 'Nagayama', 'Midori', 'mnagayam', '1991-06-17', NULL, 1, 'VP, Sales', 31, 1400, NULL);
INSERT INTO emp VALUES
(4, 'Quick-To-See', 'Mark', 'mquickto', '1990-04-07', NULL, 1, 'VP, Finance', 10, 1450, NULL);
INSERT INTO emp VALUES
(5, 'Ropeburn', 'Audry', 'aropebur', '1990--3-04', NULL, 1, 'VP, Administration', 50, 1550, NULL);
INSERT INTO emp VALUES
(6, 'Urguhart', 'Molly', 'murguhar', '1991-01-18', NULL, 2, 'Warehouse Manager', 41, 1200, NULL);
INSERT INTO emp VALUES
(7, 'Menchu', 'Roberta', 'rmenchu', '1990-05-14', NULL, 2, 'Warehouse Manager', 42, 1250, NULL);
INSERT INTO emp VALUES
(8, 'Biri', 'Ben', 'bbiri', '1990-04-07', NULL, 2, 'Warehouse Manager', 43, 1100, NULL);
INSERT INTO emp VALUES
(9, 'Catchpole', 'Antoinette', 'acatchpo', '1992-02-09', NULL, 2, 'Warehouse Manager', 44, 1300, NULL);
```

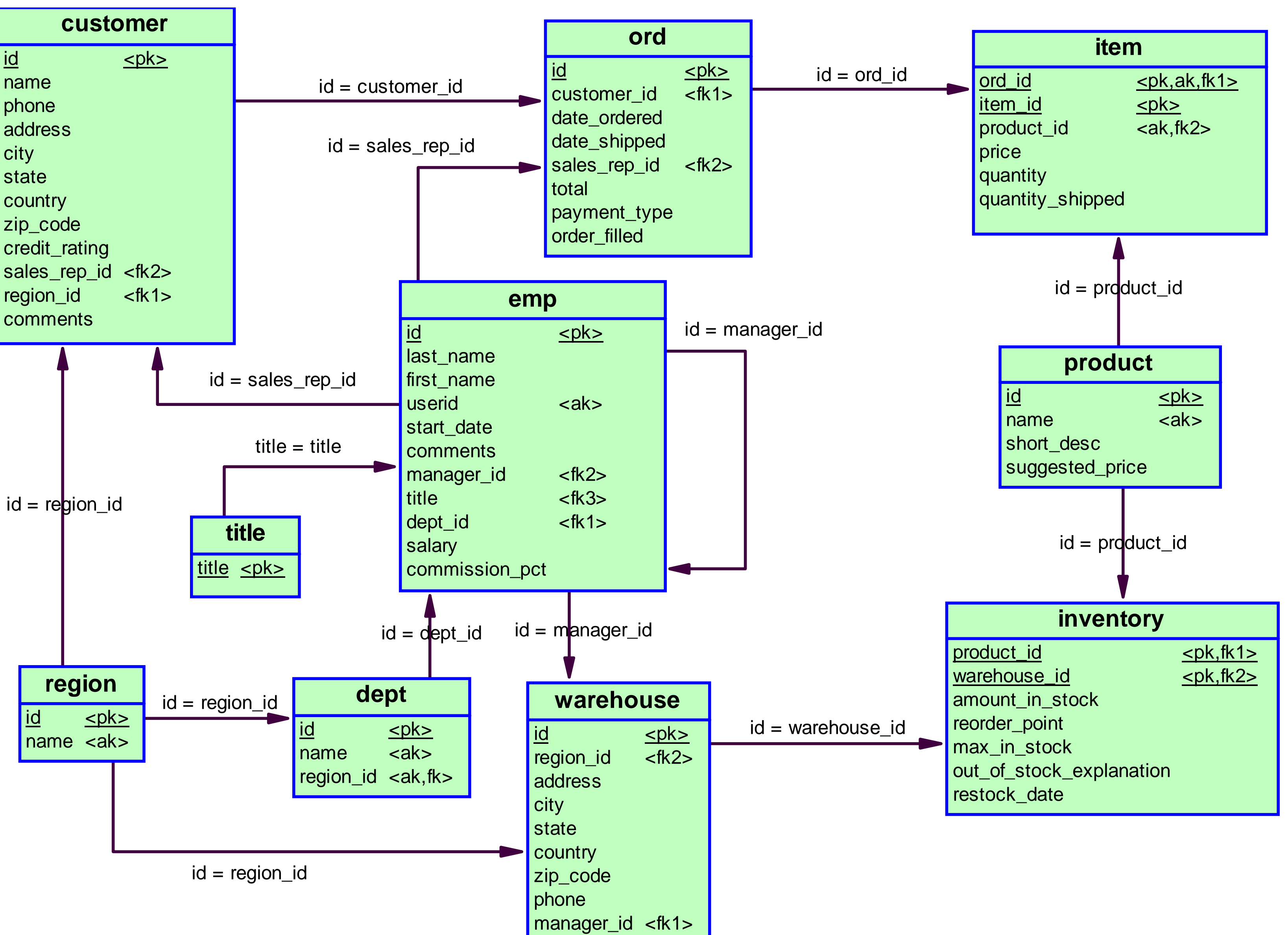
```sql
INSERT INTO emp VALUES
(10, 'Havel', 'Marta', 'mhavel', '1991-02-27', NULL, 2, 'Warehouse Manager', 45, 1307, NULL);
INSERT INTO emp VALUES
(11, 'Magee', 'Colin', 'cmagee', '1990-05-14', NULL, 3, 'Sales Representative', 31, 1400, 10);
INSERT INTO emp VALUES
(12, 'Giljum', 'Henry', 'hgiljum', '1992-01-18', NULL, 3, 'Sales Representative', 32, 1490, 12.5);
INSERT INTO emp VALUES
(13, 'Sedeghi', 'Yasmin', 'ysedeghi', '1991-02-08', NULL, 3, 'Sales Representative', 33, 1515, 10);
INSERT INTO emp VALUES
(14, 'Nguyen', 'Mai', 'mnguyen', '1992-01-22', NULL, 3, 'Sales Representative', 34, 1525, 15);
INSERT INTO emp VALUES
(15, 'Dumas', 'Andre', 'adumas', '1991-10-09', NULL, 3, 'Sales Representative', 35, 1450, 17.5);
INSERT INTO emp VALUES
(16, 'Maduro', 'Elena', 'emaduro', '1992-02-07', NULL, 6, 'Stock Clerk', 41, 1400, NULL);
INSERT INTO emp VALUES
(17, 'Smith', 'George', 'gsmith', '1990-03-08', NULL, 6, 'Stock Clerk', 41, 940, NULL);
INSERT INTO emp VALUES
(18, 'Nozaki', 'Akira', 'anozaki', '1991-02-09', NULL, 7, 'Stock Clerk', 42, 1200, NULL);
INSERT INTO emp VALUES
(19, 'Patel', 'Vikram', 'vpatel', '1991-08-06', NULL, 7, 'Stock Clerk', 42, 795, NULL);
INSERT INTO emp VALUES
(20, 'Newman', 'Chad', 'cnewman', '1991-07-21', NULL, 8, 'Stock Clerk', 43, 750, NULL);
INSERT INTO emp VALUES
(21, 'Markarian', 'Alexander', 'amarkari', '1991-05-26', NULL, 8, 'Stock Clerk', 43, 850, NULL);
INSERT INTO emp VALUES
(22, 'Chang', 'Eddie', 'echang', '1990-11-30', NULL, 9, 'Stock Clerk', 44, 800, NULL);
INSERT INTO emp VALUES
(23, 'Patel', 'Radha', 'rpatel', '1990-10-17', NULL, 9, 'Stock Clerk', 34, 795, NULL);
INSERT INTO emp VALUES
(24, 'Dancs', 'Bela', 'bdancs', '1991-03-17', NULL, 10, 'Stock Clerk', 45, 860, NULL);
INSERT INTO emp VALUES
(25, 'Schwartz', 'Sylvie', 'sschwart', '1991-05-09', NULL, 10, 'Stock Clerk', 45, 1100, NULL);
COMMIT;

INSERT INTO customer VALUES
(201, 'Unisports', '55-2066101', '72 Via Bahia', 'Sao Paolo', NULL,
'Brazil', NULL,  'EXCELLENT', 12, 2, NULL);

INSERT INTO customer VALUES
(202, 'OJ Atheletics', '81-20101',  '6741 Takashi Blvd.', 'Osaka', NULL,
'Japan', NULL,  'POOR', 14, 4, NULL);

INSERT INTO customer VALUES
(203, 'Delhi Sports', '91-10351',  '11368 Chanakya', 'New Delhi', NULL,
'India', NULL,  'GOOD', 14, 4, NULL);

INSERT INTO customer VALUES
(204, 'Womansport', '1-206-104-0103',  '281 King Street', 'Seattle', 'Washington',
'USA', NULL,  'EXCELLENT', 11, 1, NULL);

INSERT INTO customer VALUES
(205, 'Kam''s Sporting Goods', '852-3692888',  '15 Henessey Road',
'Hong Kong', NULL, NULL, NULL,  'EXCELLENT', 15, 4, NULL);

INSERT INTO customer VALUES
(206, 'Sportique', '33-2257201',  '172 Rue de Rivoli', 'Cannes', NULL,
'France', NULL,  'EXCELLENT', 15, 5, NULL);

INSERT INTO customer VALUES
(207, 'Sweet Rock Sports', '234-6036201',  '6 Saint Antoine', 'Lagos', NULL,
'Nigeria', NULL,  'GOOD', NULL, 3, NULL);

INSERT INTO customer VALUES
(208, 'Muench Sports', '49-527454',  '435 Gruenestrasse', 'Stuttgart', NULL,
'Germany', NULL,  'GOOD', 15, 5, NULL);

INSERT INTO customer VALUES
(209, 'Beisbol Si!', '809-352689',  '792 Playa Del Mar', 'San Pedro de Macon''s', NULL,
'Dominican Republic',  NULL, 'EXCELLENT', 11, 1, NULL);

INSERT INTO customer VALUES
(210, 'Futbol Sonora', '52-404562',  '3 Via Saguaro', 'Nogales', NULL,
'Mexico', NULL,  'EXCELLENT', 12, 2, NULL);
```

```
INSERT INTO customer VALUES
(211, 'Kuhn''s Sports', '42-111292',  '7 Modrany', 'Prague', NULL,
'Czechoslovakia', NULL,  'EXCELLENT', 15, 5, NULL);

INSERT INTO customer VALUES
(212, 'Hamada Sport', '20-1209211',  '57A Corniche', 'Alexandria', NULL,
'Egypt', NULL,  'EXCELLENT', 13, 3, NULL);

INSERT INTO customer VALUES
(213, 'Big John''s Sports Emporium', '1-415-555-6281',  '4783 18th Street',
'San Francisco', 'CA', 'USA', NULL,  'EXCELLENT', 11, 1, NULL);

INSERT INTO customer VALUES
(214, 'Ojibway Retail', '1-716-555-7171',  '415 Main Street', 'Buffalo', 'NY',
'USA', NULL,  'POOR', 11, 1, NULL);

INSERT INTO customer VALUES
(215, 'Sporta Russia', '7-3892456',  '6000 Yekatamina', 'Saint Petersburg', NULL,
'Russia', NULL,  'POOR', 15, 5, NULL);
COMMIT;

INSERT INTO ord VALUES (100, 204, '1992-08-31', '1992-09-10', 11, 601100, 'CREDIT', 'Y');
INSERT INTO ord VALUES (101, 205, '1992-08-31', '1992-09-15', 14, 8056.6, 'CREDIT', 'Y');
INSERT INTO ord VALUES (102, 206, '1992-09-01', '1992-09-08', 15, 8335, 'CREDIT', 'Y');
INSERT INTO ord VALUES (103, 208, '1992-09-02', '1992-09-22', 15, 377, 'CASH', 'Y');
INSERT INTO ord VALUES (104, 208, '1992-09-03', '1992-09-23', 15, 32430, 'CREDIT', 'Y');
INSERT INTO ord VALUES (105, 209, '1992-09-04', '1992-09-18', 11, 2722.24, 'CREDIT', 'Y');
INSERT INTO ord VALUES (106, 210, '1992-09-07', '1992-09-15', 12, 15634, 'CREDIT', 'Y');
INSERT INTO ord VALUES (107, 211, '1992-09-07', '1992-09-21', 15, 142171, 'CREDIT', 'Y');
INSERT INTO ord VALUES (108, 212, '1992-09-07', '1992-09-10', 13, 149570, 'CREDIT', 'Y');
INSERT INTO ord VALUES (109, 213, '1992-09-08', '1992-09-28', 11, 1020935, 'CREDIT', 'Y');
INSERT INTO ord VALUES (110, 214, '1992-09-09', '1992-09-21', 11, 1539.13, 'CASH', 'Y');
INSERT INTO ord VALUES (111, 204, '1992-09-09', '1992-09-21', 11, 2770, 'CASH', 'Y');
INSERT INTO ord VALUES (97, 201, '1992-08-28', '1992-09-17', 12, 84000, 'CREDIT', 'Y');
INSERT INTO ord VALUES (98, 202, '1992-08-31', '1992-09-10', 14, 595, 'CASH', 'Y');
INSERT INTO ord VALUES (99, 203, '1992-08-31', '1992-09-18', 14, 7707, 'CREDIT', 'Y');
INSERT INTO ord VALUES (112, 210, '1992-08-31', '1992-09-10', 12, 550, 'CREDIT', 'Y');
COMMIT;

INSERT INTO product VALUES (10011, 'Bunny Boot', 'Beginner''s ski boot', 150);
INSERT INTO product VALUES (10012, 'Ace Ski Boot', 'Intermediate ski boot', 200);
INSERT INTO product VALUES (10013, 'Pro Ski Boot', 'Advanced ski boot', 410);
INSERT INTO product VALUES (10021, 'Bunny Ski Pole', 'Beginner''s ski pole', 16.25);
INSERT INTO product VALUES (10022, 'Ace Ski Pole', 'Intermediate ski pole', 21.95);
INSERT INTO product VALUES (10023, 'Pro Ski Pole', 'Advanced ski pole', 40.95);
INSERT INTO product VALUES (20106, 'Junior Soccer Ball', 'Junior soccer ball', 11);
INSERT INTO product VALUES (20108, 'World Cup Soccer Ball', 'World cup soccer ball', 28);
INSERT INTO product VALUES (20201, 'World Cup Net', 'World cup net', 123);
INSERT INTO product VALUES (20510, 'Black Hawk Knee Pads', 'Knee pads, pair', 9);
INSERT INTO product VALUES (20512, 'Black Hawk Elbow Pads', 'Elbow pads, pair', 8);
INSERT INTO product VALUES (30321, 'Grand Prix Bicycle', 'Road bicycle', 1669);
INSERT INTO product VALUES (30326, 'Himalaya Bicycle', 'Mountain bicycle', 582);
INSERT INTO product VALUES (30421, 'Grand Prix Bicycle Tires', 'Road bicycle tires', 16);
INSERT INTO product VALUES (30426, 'Himalaya Tires', 'Mountain bicycle tires', 18.25);
INSERT INTO product VALUES (30433, 'New Air Pump', 'Tire pump', 20);
INSERT INTO product VALUES (32779, 'Slaker Water Bottle', 'Water bottle', 7);
INSERT INTO product VALUES (32861, 'Safe-T Helmet', 'Bicycle helmet', 60);
INSERT INTO product VALUES (40421, 'Alexeyer Pro Lifting Bar', 'Straight bar', 65);
INSERT INTO product VALUES (40422, 'Pro Curling Bar', 'Curling bar', 50);
INSERT INTO product VALUES (41010, 'Prostar 10 Pound Weight', 'Ten pound weight', 8);
INSERT INTO product VALUES (41020, 'Prostar 20 Pound Weight', 'Twenty pound weight', 12);
INSERT INTO product VALUES (41050, 'Prostar 50 Pound Weight', 'Fifty pound weight', 25);
INSERT INTO product VALUES (41080, 'Prostar 80 Pound Weight', 'Eighty pound weight', 35);
INSERT INTO product VALUES (41100, 'Prostar 100 Pound Weight', 'One hundred pound weight', 45);
INSERT INTO product VALUES (50169, 'Major League Baseball', 'Baseball', 4.29);
INSERT INTO product VALUES (50273, 'Chapman Helmet', 'Batting helmet', 22.89);
INSERT INTO product VALUES (50417, 'Griffey Glove', 'Outfielder''s glove', 80);
INSERT INTO product VALUES (50418, 'Alomar Glove', 'Infielder''s glove', 75);
INSERT INTO product VALUES (50419, 'Steinbach Glove', 'Catcher''s glove', 80);
INSERT INTO product VALUES (50530, 'Cabrera Bat', 'Thirty inch bat', 45);
INSERT INTO product VALUES (50532, 'Puckett Bat', 'Thirty-two inch bat', 47);
INSERT INTO product VALUES (50536, 'Winfield Bat', 'Thirty-six inch bat', 50);
COMMIT;
```

```sql
INSERT INTO warehouse VALUES (101, 1, '283 King Street', 'Seattle', 'WA', 'USA', NULL, NULL, 6);
INSERT INTO warehouse VALUES (10501, 5, '5 Modrany', 'Bratislava', NULL, 'Czechozlovakia', NULL, NULL, 10);
INSERT INTO warehouse VALUES (201, 2, '68 Via Centrale', 'Sao Paolo', NULL, 'Brazil', NULL, NULL, 7);
INSERT INTO warehouse VALUES (301, 3, '6921 King Way', 'Lagos', NULL, 'Nigeria', NULL, NULL, 8);
INSERT INTO warehouse VALUES (401, 4, '86 Chu Street', 'Hong Kong', NULL, NULL, NULL, NULL, 9);
COMMIT;

INSERT INTO inventory VALUES (10011, 101, 650, 625, 1100, NULL, NULL);
INSERT INTO inventory VALUES (10012, 101, 600, 560, 1000, NULL, NULL);
INSERT INTO inventory VALUES (10013, 101, 400, 400, 700, NULL, NULL);
INSERT INTO inventory VALUES (10021, 101, 500, 425, 740, NULL, NULL);
INSERT INTO inventory VALUES (10022, 101, 300, 200, 350, NULL, NULL);
INSERT INTO inventory VALUES (10023, 101, 400, 300, 525, NULL, NULL);
INSERT INTO inventory VALUES (20106, 101, 993, 625, 1000, NULL, NULL);
INSERT INTO inventory VALUES (20108, 101, 700, 700, 1225, NULL, NULL);
INSERT INTO inventory VALUES (20201, 101, 802, 800, 1400, NULL, NULL);
INSERT INTO inventory VALUES (20510, 101, 1389, 850, 1400, NULL, NULL);
INSERT INTO inventory VALUES (20512, 101, 850, 850, 1450, NULL, NULL);
INSERT INTO inventory VALUES (30321, 101, 2000, 1500, 2500, NULL, NULL);
INSERT INTO inventory VALUES (30326, 101, 2100, 2000, 3500, NULL, NULL);
INSERT INTO inventory VALUES (30421, 101, 1822, 1800, 3150, NULL, NULL);
INSERT INTO inventory VALUES (30426, 101, 2250, 2000, 3500, NULL, NULL);
INSERT INTO inventory VALUES (30433, 101, 650, 600, 1050, NULL, NULL);
INSERT INTO inventory VALUES (32779, 101, 2120, 1250, 2200, NULL, NULL);
INSERT INTO inventory VALUES (32861, 101, 505, 500, 875, NULL, NULL);
INSERT INTO inventory VALUES (40421, 101, 578, 350, 600, NULL, NULL);
INSERT INTO inventory VALUES (40422, 101, 0, 350, 600, 'Phenomenal sales...', '1993-02-08');
INSERT INTO inventory VALUES (41010, 101, 250, 250, 437, NULL, NULL);
INSERT INTO inventory VALUES (41020, 101, 471, 450, 750, NULL, NULL);
INSERT INTO inventory VALUES (41050, 101, 501, 450, 750, NULL, NULL);
INSERT INTO inventory VALUES (41080, 101, 400, 400, 700, NULL, NULL);
INSERT INTO inventory VALUES (41100, 101, 350, 350, 600, NULL, NULL);
INSERT INTO inventory VALUES (50169, 101, 2530, 1500, 2600, NULL, NULL);
INSERT INTO inventory VALUES (50273, 101, 233, 200, 350, NULL, NULL);
INSERT INTO inventory VALUES (50417, 101, 518, 500, 875, NULL, NULL);
INSERT INTO inventory VALUES (50418, 101, 244, 100, 275, NULL, NULL);
INSERT INTO inventory VALUES (50419, 101, 230, 120, 310, NULL, NULL);
INSERT INTO inventory VALUES (50530, 101, 669, 400, 700, NULL, NULL);
INSERT INTO inventory VALUES (50532, 101, 0, 100, 175, 'Wait for Spring.', '1993-04-12');
INSERT INTO inventory VALUES (50536, 101, 173, 100, 175, NULL, NULL);
INSERT INTO inventory VALUES (20106, 201, 220, 150, 260, NULL, NULL);
INSERT INTO inventory VALUES (20108, 201, 166, 150, 260, NULL, NULL);
INSERT INTO inventory VALUES (20201, 201, 320, 200, 350, NULL, NULL);
INSERT INTO inventory VALUES (20510, 201, 175, 100, 175, NULL, NULL);
INSERT INTO inventory VALUES (20512, 201, 162, 100, 175, NULL, NULL);
INSERT INTO inventory VALUES (30321, 201, 96, 80, 140, NULL, NULL);
INSERT INTO inventory VALUES (30326, 201, 147, 120, 210, NULL, NULL);
INSERT INTO inventory VALUES (30421, 201, 102, 80, 140, NULL, NULL);
INSERT INTO inventory VALUES (30426, 201, 200, 120, 210, NULL, NULL);
INSERT INTO inventory VALUES (30433, 201, 130, 130, 230, NULL, NULL);
INSERT INTO inventory VALUES (32779, 201, 180, 150, 260, NULL, NULL);
INSERT INTO inventory VALUES (32861, 201, 132, 80, 140, NULL, NULL);
INSERT INTO inventory VALUES (50169, 201, 225, 220, 385, NULL, NULL);
INSERT INTO inventory VALUES (50273, 201, 75, 60, 100, NULL, NULL);
INSERT INTO inventory VALUES (50417, 201, 82, 60, 100, NULL, NULL);
INSERT INTO inventory VALUES (50418, 201, 98, 60, 100, NULL, NULL);
INSERT INTO inventory VALUES (50419, 201, 77, 60, 100, NULL, NULL);
INSERT INTO inventory VALUES (50530, 201, 62, 60, 100, NULL, NULL);
INSERT INTO inventory VALUES (50532, 201, 67, 60, 100, NULL, NULL);
INSERT INTO inventory VALUES (50536, 201, 97, 60, 100, NULL, NULL);
INSERT INTO inventory VALUES (20510, 301, 69, 40, 100, NULL, NULL);
INSERT INTO inventory VALUES (20512, 301, 28, 20, 50, NULL, NULL);
INSERT INTO inventory VALUES (30321, 301, 85, 80, 140, NULL, NULL);
INSERT INTO inventory VALUES (30421, 301, 102, 80, 140, NULL, NULL);
INSERT INTO inventory VALUES (30433, 301, 35, 20, 35, NULL, NULL);
INSERT INTO inventory VALUES (32779, 301, 102, 95, 175, NULL, NULL);
INSERT INTO inventory VALUES (32861, 301, 57, 50, 100, NULL, NULL);
INSERT INTO inventory VALUES (40421, 301, 70, 40, 70, NULL, NULL);
INSERT INTO inventory VALUES (40422, 301, 65, 40, 70, NULL, NULL);
INSERT INTO inventory VALUES (41010, 301, 59, 40, 70, NULL, NULL);
INSERT INTO inventory VALUES (41020, 301, 61, 40, 70, NULL, NULL);
INSERT INTO inventory VALUES (41050, 301, 49, 40, 70, NULL, NULL);
INSERT INTO inventory VALUES (41080, 301, 50, 40, 70, NULL, NULL);
INSERT INTO inventory VALUES (41100, 301, 42, 40, 70, NULL, NULL);
```

```sql
INSERT INTO inventory VALUES (20510, 401, 88, 50, 100, NULL, NULL);
INSERT INTO inventory VALUES (20512, 401, 75, 75, 140, NULL, NULL);
INSERT INTO inventory VALUES (30321, 401, 102, 80, 140, NULL, NULL);
INSERT INTO inventory VALUES (30326, 401, 113, 80, 140, NULL, NULL);
INSERT INTO inventory VALUES (30421, 401, 85, 80, 140, NULL, NULL);
INSERT INTO inventory VALUES (30426, 401, 135, 80, 140, NULL, NULL);
INSERT INTO inventory VALUES (30433, 401, 0, 100, 175,
'A defective shipment was sent to Hong Kong and needed to be returned.
The soonest ACME can turn this around is early February.', '1992-08-07');
INSERT INTO inventory VALUES (32779, 401, 135, 100, 175, NULL, NULL);
INSERT INTO inventory VALUES (32861, 401, 250, 150, 250, NULL, NULL);
INSERT INTO inventory VALUES (40421, 401, 47, 40, 70, NULL, NULL);
INSERT INTO inventory VALUES (40422, 401, 50, 40, 70, NULL, NULL);
INSERT INTO inventory VALUES (41010, 401, 80, 70, 220, NULL, NULL);
INSERT INTO inventory VALUES (41020, 401, 91, 70, 220, NULL, NULL);
INSERT INTO inventory VALUES (41050, 401, 169, 70, 220, NULL, NULL);
INSERT INTO inventory VALUES (41080, 401, 100, 70, 220, NULL, NULL);
INSERT INTO inventory VALUES (41100, 401, 75, 70, 220, NULL, NULL);
INSERT INTO inventory VALUES (50169, 401, 240, 200, 350, NULL, NULL);
INSERT INTO inventory VALUES (50273, 401, 224, 150, 280, NULL, NULL);
INSERT INTO inventory VALUES (50417, 401, 130, 120, 210, NULL, NULL);
INSERT INTO inventory VALUES (50418, 401, 156, 100, 175, NULL, NULL);
INSERT INTO inventory VALUES (50419, 401, 151, 150, 280, NULL, NULL);
INSERT INTO inventory VALUES (50530, 401, 119, 100, 175, NULL, NULL);
INSERT INTO inventory VALUES (50532, 401, 233, 200, 350, NULL, NULL);
INSERT INTO inventory VALUES (50536, 401, 138, 100, 175, NULL, NULL);
INSERT INTO inventory VALUES (10012, 10501, 300, 300, 525, NULL, NULL);
INSERT INTO inventory VALUES (10013, 10501, 314, 300, 525, NULL, NULL);
INSERT INTO inventory VALUES (10022, 10501, 502, 300, 525, NULL, NULL);
INSERT INTO inventory VALUES (10023, 10501, 500, 300, 525, NULL, NULL);
INSERT INTO inventory VALUES (20106, 10501, 150, 100, 175, NULL, NULL);
INSERT INTO inventory VALUES (20108, 10501, 222, 200, 350, NULL, NULL);
INSERT INTO inventory VALUES (20201, 10501, 275, 200, 350, NULL, NULL);
INSERT INTO inventory VALUES (20510, 10501, 57, 50, 87, NULL, NULL);
INSERT INTO inventory VALUES (20512, 10501, 62, 50, 87, NULL, NULL);
INSERT INTO inventory VALUES (30321, 10501, 194, 150, 275, NULL, NULL);
INSERT INTO inventory VALUES (30326, 10501, 277, 250, 440, NULL, NULL);
INSERT INTO inventory VALUES (30421, 10501, 190, 150, 275, NULL, NULL);
INSERT INTO inventory VALUES (30426, 10501, 423, 250, 450, NULL, NULL);
INSERT INTO inventory VALUES (30433, 10501, 273, 200, 350, NULL, NULL);
INSERT INTO inventory VALUES (32779, 10501, 280, 200, 350, NULL, NULL);
INSERT INTO inventory VALUES (32861, 10501, 288, 200, 350, NULL, NULL);
INSERT INTO inventory VALUES (40421, 10501, 97, 80, 140, NULL, NULL);
INSERT INTO inventory VALUES (40422, 10501, 90, 80, 140, NULL, NULL);
INSERT INTO inventory VALUES (41010, 10501, 151, 140, 245, NULL, NULL);
INSERT INTO inventory VALUES (41020, 10501, 224, 140, 245, NULL, NULL);
INSERT INTO inventory VALUES (41050, 10501, 157, 140, 245, NULL, NULL);
INSERT INTO inventory VALUES (41080, 10501, 159, 140, 245, NULL, NULL);
INSERT INTO inventory VALUES (41100, 10501, 141, 140, 245, NULL, NULL);
COMMIT;

INSERT INTO item VALUES (100, 1, 10011, 135, 500, 500);
INSERT INTO item VALUES (100, 2, 10013, 380, 400, 400);
INSERT INTO item VALUES (100, 3, 10021, 14, 500, 500);
INSERT INTO item VALUES (100, 5, 30326, 582, 600, 600);
INSERT INTO item VALUES (100, 7, 41010, 8, 250, 250);
INSERT INTO item VALUES (100, 6, 30433, 20, 450, 450);
INSERT INTO item VALUES (100, 4, 10023, 36, 400, 400);
INSERT INTO item VALUES (101, 1, 30421, 16, 15, 15);
INSERT INTO item VALUES (101, 3, 41010, 8, 20, 20);
INSERT INTO item VALUES (101, 5, 50169, 4.29, 40, 40);
INSERT INTO item VALUES (101, 6, 50417, 80, 27, 27);
INSERT INTO item VALUES (101, 7, 50530, 45, 50, 50);
INSERT INTO item VALUES (101, 4, 41100, 45, 35, 35);
INSERT INTO item VALUES (101, 2, 40422, 50, 30, 30);
INSERT INTO item VALUES (102, 1, 20108, 28, 100, 100);
INSERT INTO item VALUES (102, 2, 20201, 123, 45, 45);
INSERT INTO item VALUES (103, 1, 30433, 20, 15, 15);
INSERT INTO item VALUES (103, 2, 32779, 7, 11, 11);
INSERT INTO item VALUES (104, 1, 20510, 9, 7, 7);
INSERT INTO item VALUES (104, 4, 30421, 16, 35, 35);
INSERT INTO item VALUES (104, 2, 20512, 8, 12, 12);
INSERT INTO item VALUES (104, 3, 30321, 1669, 19, 19);
INSERT INTO item VALUES (105, 1, 50273, 22.89, 16, 16);
```

```sql
INSERT INTO item VALUES (105, 3, 50532, 47, 28, 28);
INSERT INTO item VALUES (105, 2, 50419, 80, 13, 13);
INSERT INTO item VALUES (106, 1, 20108, 28, 46, 46);
INSERT INTO item VALUES (106, 4, 50273, 22.89, 75, 75);
INSERT INTO item VALUES (106, 5, 50418, 75, 98, 98);
INSERT INTO item VALUES (106, 6, 50419, 80, 27, 27);
INSERT INTO item VALUES (106, 2, 20201, 123, 21, 21);
INSERT INTO item VALUES (106, 3, 50169, 4.29, 125, 125);
INSERT INTO item VALUES (107, 1, 20106, 11, 50, 50);
INSERT INTO item VALUES (107, 3, 20201, 115, 130, 130);
INSERT INTO item VALUES (107, 5, 30421, 16, 55, 55);
INSERT INTO item VALUES (107, 4, 30321, 1669, 75, 75);
INSERT INTO item VALUES (107, 2, 20108, 28, 22, 22);
INSERT INTO item VALUES (108, 1, 20510, 9, 9, 9);
INSERT INTO item VALUES (108, 6, 41080, 35, 50, 50);
INSERT INTO item VALUES (108, 7, 41100, 45, 42, 42);
INSERT INTO item VALUES (108, 5, 32861, 60, 57, 57);
INSERT INTO item VALUES (108, 2, 20512, 8, 18, 18);
INSERT INTO item VALUES (108, 4, 32779, 7, 60, 60);
INSERT INTO item VALUES (108, 3, 30321, 1669, 85, 85);
INSERT INTO item VALUES (109, 1, 10011, 140, 150, 150);
INSERT INTO item VALUES (109, 5, 30426, 18.25, 500, 500);
INSERT INTO item VALUES (109, 7, 50418, 75, 43, 43);
INSERT INTO item VALUES (109, 6, 32861, 60, 50, 50);
INSERT INTO item VALUES (109, 4, 30326, 582, 1500, 1500);
INSERT INTO item VALUES (109, 2, 10012, 175, 600, 600);
INSERT INTO item VALUES (109, 3, 10022, 21.95, 300, 300);
INSERT INTO item VALUES (110, 1, 50273, 22.89, 17, 17);
INSERT INTO item VALUES (110, 2, 50536, 50, 23, 23);
INSERT INTO item VALUES (111, 1, 40421, 65, 27, 27);
INSERT INTO item VALUES (111, 2, 41080, 35, 29, 29);
INSERT INTO item VALUES (97, 1, 20106, 9, 1000, 1000);
INSERT INTO item VALUES (97, 2, 30321, 1500, 50, 50);
INSERT INTO item VALUES (98, 1, 40421, 85, 7, 7);
INSERT INTO item VALUES (99, 1, 20510, 9, 18, 18);
INSERT INTO item VALUES (99, 2, 20512, 8, 25, 25);
INSERT INTO item VALUES (99, 3, 50417, 80, 53, 53);
INSERT INTO item VALUES (99, 4, 50530, 45, 69, 69);
INSERT INTO item VALUES (112, 1, 20106, 11, 50, 50);
COMMIT;
```




Rysunek 10.1: Schemat relacyjny modelu demonstracyjnego

# Bibliografia


[1] Lech Banachowski (tłum.). *SQL. Język relacyjnych baz danych.* WNT Warszawa, 1995.

[2] Paul Dubios. *MySQL. Podręcznik administratora.* Wydawnictwo HELION, 2005.

[3] *MySQL 5.0 Reference Manual*, 2005. (jest to najbardziej aktualne opracowanie na temat bazy MySQL stworzone i na bieżąco aktualizowane przez jej twórców. Książka dostępna w wersji elektronicznej pod adresem *http://dev.mysql.com/doc/*).

[4] Richard Stones and Neil Matthew. *Od podstaw. Bazy danych i MySQL.* Wydawnictwo HELION, 2003.

[5] Luke Welling and Laura Thomson. *MySQL. Podstawy.* Wydawnictwo HELION, 2005.