Krótki przegląd zapytań

Najważniejszą rzeczą jest uświadomienie sobie, że baza danych to generalnie zestaw tabel – a więc stosunkowo prostych obiektów, składających się z wierszy i kolumn. Wszelkie operatory, wszelkie zapytania kierowane do bazy odnoszą się do tabel.

Kolumny tabeli mają określone typy – to oznacza, że jeśli zdefiniujemy taką kolumnę na przykład jako int (liczba całkowita), to nie będziemy mogli w niej przechowywać innych danych (np. napisów, albo liczb rzeczywistych).

Oprócz typów, kolumnę mogą cechować dodatkowe ograniczenia (uzależnione od typu). Przykładem takiego ograniczenia jest np. klauzule UNIQUE (która mówi, że wartości w tej kolumnie nie mogą się powtarzać), lub NOT NULL (baza nie zaakceptuje wartości nieustawionej, czyli pustej).

Tworzenie tablic w bazie odbywa się za pomocą polecenia CREATE TABLE.

W tym miejscu warto zauważyć, że nie wszystkie bazy danych implementują dokładnie stuprocentowo standard SQL. Dzieje się tak dlatego, że podstawowy SQL okazał się językiem dość ograniczonym i wielu najważniejszych dostawców postanowiło zaoferować dodatki, które ułatwiają pracę twórcom baz danych. W ten sposób powstały tzw. dialekty SQL – wersje podstawowego języka, które minimalnie się od siebie różnią. Poniższe przykłady dotyczą SQL w wersji mysql.

Na przykład:

```
CREATE TABLE student (imie char(50), nazwisko char(50))
```

```
mysql> CREATE TABLE student (imie char(50), nazwisko char(50));
Query OK, 0 rows affected (0.00 sec)
```

Powoduje utworzenie tabeli student (wielkość liter nie ma znaczenia) o dwóch kolumnach: imie i nazwisko – w obu przechowujemy ciągi znakowe o długości do 50 znaków. Średnik na końcu polecenia jest znakiem charakterystycznym dla silnika mysql (nie jest częścią zapytania SQL). Formuła „Query OK” jest informacją zwrotną serwera mysql o poprawnym wykonaniu polecenia.

Spróbujmy dodać kilka rekordów. Możemy to zrobić poleceniem INSERT INTO ... VALUES:

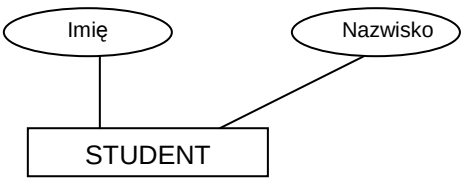
INSERT INTO student (imie, nazwisko) VALUES (‘Jan’, ‘Kowalski’);

```
mysql> INSERT INTO student VALUES ('Jan','Kowalski');
Query OK, 1 row affected (0.00 sec)
```

Po wypełnieniu kilkoma rekordami przykładowymi, tabela może wyglądać tak:

Jan	Kowalski
Maria	Nowak
Tadeusz	Malinowski
Julia	Kowalczyk
Magda	Jakubowska
Iwona	Krzewska
Paulina	Rydygier

Oczywiście odpowiada to takiemu modelowi encji:



Sprawdźmy teraz, jakie rekordy znajdują się w naszej bazie danych. Służy do tego polecenie SELECT, a najprostsza jego składnia wygląda tak:

SELECT [kolumny] FROM [tabela]

Kolumny mogą przyjąć postać nazwy, nazw rozdzielanych przecinkiem lub *, która oznacza: wszystkie.

Sprawdźmy. W tym celu wykonamy polecenie:
SELECT * FROM student
Oczywiście moglibyśmy też wykonać:
SELECT imie, nazwisko FROM student

```
mysql> select * from student;
+---------+------------+
| imie    | nazwisko   |
+---------+------------+
| Jan     | Kowalski   |
| Maria   | Nowak      |
| Tadeusz | Malinowski |
| Julia   | Kowalczyk  |
| Magda   | Jakubowska |
| Iwona   | Krzewska   |
| Paulina | Rydygier   |
+---------+------------+
7 rows in set (0.00 sec)
```

W tym miejscu jedna uwaga: zastosowany w przykładzie klient mysql (czyli oprogramowanie, którym łączymy się z bazą) domyślnie umieszcza w prezentowanym na ekranie wyniku nazwy kolumn. To nie jest zachowanie zgodne z SQL – jedynie pomoc dla użytkownika. Wynikiem zapytania są jedynie wartości pól, czyli tabela:

Jan	Kowalski
Maria	Nowak
Tadeusz	Malinowski
Julia	Kowalczyk
Magda	Jakubowska
Iwona	Krzewska
Paulina	Rydygier

Oczywiście możemy także poprosić bazę danych o same nazwiska:

```
mysql> select nazwisko from student;
+------------+
| nazwisko   |
+------------+
| Kowalski   |
| Nowak      |
| Malinowski |
| Kowalczyk  |
| Jakubowska |
| Krzewska   |
| Rydygier   |
+------------+
7 rows in set (0.00 sec)
```

Istnieje sposób ograniczenia wyników zapytania SELECT. Służy do tego klauzula WHERE, która jest swego rodzaju filtrem. Oznacza ona, że serwer bazy danych najpierw wykona (na własne potrzeby) zwykłe SELECT, a następnie wyświetli (zwróci) jedynie te rekordy, które pasują do warunku podanego w WHERE.

Przykład:

```
mysql> SELECT * FROM student WHERE imie='Maria';
+-------+----------+
| imie  | nazwisko |
+-------+----------+
| Maria | Nowak    |
+-------+----------+
1 row in set (0.00 sec)
```

W tym miejscu zauważmy, że warunki dla klauzuli WHERE budujemy najczęściej w oparciu o operatory = (równość) <> (różność) i > (większość). W przypadku łańcuchów tekstowych mamy także często używany operator LIKE (podobny do), stosowany wraz ze znakiem specjalnym % (zastępuje dowolny ciąg znaków, działa jak * przy wyszukiwaniu plików w Windows).

W ten sposób mamy:

Różność łańcuchów:

```
mysql> SELECT * FROM student WHERE imie<>'Maria';
+---------+------------+
| imie    | nazwisko   |
+---------+------------+
| Jan     | Kowalski   |
| Tadeusz | Malinowski |
| Julia   | Kowalczyk  |
| Magda   | Jakubowska |
| Iwona   | Krzewska   |
| Paulina | Rydygier   |
+---------+------------+
6 rows in set (0.00 sec)
```

Operator większości (dla liczb działa jak porównanie arytmetyczne, dla łańcuchów znakowych jak porównanie alfabetyczne):

```
mysql> SELECT * FROM student WHERE imie>'Maria';
+---------+------------+
| imie    | nazwisko   |
+---------+------------+
| Tadeusz | Malinowski |
| Paulina | Rydygier   |
+---------+------------+
2 rows in set (0.00 sec)
```

Zwróćmy uwagę, że operator podobieństwa nie zadziała dobrze, jeśli nie zastosujemy znaku %. Ściślej: zadziała wówczas, jak operator równości (a w bazie nie ma osoby o imieniu „na”:

```
mysql> SELECT * FROM student WHERE imie LIKE 'na';
Empty set (0.00 sec)
```

Wyszukujemy wszystkich tych, których imiona kończą się na „na”:

```
mysql> SELECT * FROM student WHERE imie LIKE '%na';
+---------+----------+
| imie    | nazwisko |
+---------+----------+
| Iwona   | Krzewska |
| Paulina | Rydygier |
+---------+----------+
2 rows in set (0.00 sec)
```

Drugą ważną klauzulą dla SELECT jest ORDER BY, czyli sortowanie. Może ona być zastosowana samodzielnie lub razem z WHERE.

```
mysql> SELECT * FROM student ORDER BY nazwisko;
+---------+------------+
| imie    | nazwisko   |
+---------+------------+
| Magda   | Jakubowska |
| Julia   | Kowalczyk  |
| Jan     | Kowalski   |
| Iwona   | Krzewska   |
| Tadeusz | Malinowski |
| Maria   | Nowak      |
| Paulina | Rydygier   |
+---------+------------+
7 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM student WHERE nazwisko LIKE 'K%' ORDER BY nazwisko;
+-------+-----------+
| imie  | nazwisko  |
+-------+-----------+
| Julia | Kowalczyk |
| Jan   | Kowalski  |
| Iwona | Krzewska  |
+-------+-----------+
3 rows in set (0.00 sec)
```

Jeśli chodzi o kolejność poszczególnych elementów zapytania, to ważne jest, aby pamiętać, jak ono w rzeczywistości jest wykonywane przez serwer.

Popatrzmy na ostatnie zapytanie: SELECT * FROM student WHERE nazwisko LIKE ‘K%’ ORDER BY nazwisko. Na początku serwer wykona zwykłego selecta: SELECT * FROM student, otrzymując 7 rekordów. Z rekordów tych zostaną wyfiltrowane tylko te, w których nazwisko zaczyna się na literę K. Otrzymane 3 rekordy zostaną posortowane względem nazwiska. Dlatego zapytanie SELECT * ORDER BY nazwisko WHERE nazwisko LIKE ‘K%’ nie ma sensu (jest błędne).

Załóżmy, że chcemy naszą tabelę rozszerzyć o nową kolumnę – będzie nią wzrost. Jest to oczywiście liczba int. Do modyfikacji istniejącej tabeli służy polecenie ALTER TABLE. Oczywiście można najpierw usunąć tabelę, potem stworzyć nową i ponownie wypełnić wartości – ale to zbyt skomplikowana droga.

```
mysql> ALTER TABLE student ADD COLUMN wzrost int;
Query OK, 7 rows affected (0.02 sec)
Records: 7  Duplicates: 0  Warnings: 0
```

```
mysql> select * from student;
+---------+------------+--------+
| imie    | nazwisko   | wzrost |
+---------+------------+--------+
| Jan     | Kowalski   |   NULL |
| Maria   | Nowak      |   NULL |
| Tadeusz | Malinowski |   NULL |
| Julia   | Kowalczyk  |   NULL |
| Magda   | Jakubowska |   NULL |
| Iwona   | Krzewska   |   NULL |
| Paulina | Rydygier   |   NULL |
+---------+------------+--------+
7 rows in set (0.00 sec)
```

Jak widać, w tabeli pojawiła się nowa kolumna. Każdy rekord ma pole „wzrost" równe NULL. NULL jest specjalną wartością, która może pojawić się w kolumnie dowolnego typu. Oznacza ona „wartość nieustaloną" i nie należy jej mylić z wartością zerową. To tak jak z posiadaniem samochodu. Sytuacja, w której Kowalski NIE MA samochodu, to nie jest to samo, co sytuacja, w której NIE WIEMY, czy Kowalski ma samochód. W wersji SQL Kowalski ma 0 samochodów lub ma NULL samochodów.

Aby ustawić wartość pola istniejącego rekordu, posłużymy się poleceniem UPDATE. Jedna bardzo ważna uwaga – UPDATE zmienia wartość wszystkich rekordów, których dotyczy, dlatego trzeba dokładnie określać zakres jego działania. Najczęstszy błąd początkującego bazodanowa wygląda tak (zakładamy, że chcieliśmy zmienić pierwszy rekord):

```
mysql> UPDATE student SET wzrost=175;
Query OK, 7 rows affected (0.00 sec)
Rows matched: 7  Changed: 7  Warnings: 0
```

W tym miejscu zauważamy z przerażeniem, że wszystkie rekordy uległy modyfikacji!

```
mysql> select * from student;
+---------+------------+--------+
| imie    | nazwisko   | wzrost |
+---------+------------+--------+
| Jan     | Kowalski   |    175 |
| Maria   | Nowak      |    175 |
| Tadeusz | Malinowski |    175 |
| Julia   | Kowalczyk  |    175 |
| Magda   | Jakubowska |    175 |
| Iwona   | Krzewska   |    175 |
| Paulina | Rydygier   |    175 |
+---------+------------+--------+
```

Aby zmodyfikować wyłącznie wzrost Jana Kowalskiego powinniśmy posłużyć się poleceniem UPDATE z klauzulą WHERE:

```
mysql> UPDATE student SET wzrost=190 WHERE imie='Jan' AND nazwisko='Kowalski';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Załóżmy, że po wykonaniu kilku poleceń UPDATE tabela wygląda tak:

```
mysql> select * from student;
+---------+------------+--------+
| imie    | nazwisko   | wzrost |
+---------+------------+--------+
| Jan     | Kowalski   |    190 |
| Maria   | Nowak      |    180 |
| Tadeusz | Malinowski |    165 |
| Julia   | Kowalczyk  |    152 |
| Magda   | Jakubowska |    167 |
| Iwona   | Krzewska   |    182 |
| Paulina | Rydygier   |    175 |
+---------+------------+--------+
```

Każdy praktyk bazodanowy wie, że wygodnie mieć specjalną kolumnę służącą do unikalnej identyfikacji rekordów. Taka kolumna odpowiadałaby kluczowi głównemu encji, której realizacją jest tabela. Najprościej taką kolumnę utworzyć jako kolumnę liczb int (całkowitych) i poinformować silnik bazy danych, żeby samodzielnie dbał o niepowtarzalność jej wartości. W mysql dokonuje się tego za pomocą ograniczeń AUTO_INCREMENT i PRIMARY KEY (w innych bazach może to wyglądać trochę inaczej). Spróbujmy więc:

```
mysql> ALTER TABLE student ADD COLUMN id int auto_increment primary key;
Query OK, 7 rows affected (0.00 sec)
Records: 7  Duplicates: 0  Warnings: 0
```

Zauważmy, że w przeciwieństwie do poprzedniego ALTER TABLE, teraz nowa kolumna zawiera już wartości. Dzieje się tak dzięki AUTO_INCREMENT, które automatycznie dodaje nowe wartości).

```
mysql> select * from student;
+---------+------------+--------+----+
| imie    | nazwisko   | wzrost | id |
+---------+------------+--------+----+
| Jan     | Kowalski   |    190 |  1 |
| Maria   | Nowak      |    180 |  2 |
| Tadeusz | Malinowski |    165 |  3 |
| Julia   | Kowalczyk  |    152 |  4 |
| Magda   | Jakubowska |    167 |  5 |
| Iwona   | Krzewska   |    182 |  6 |
| Paulina | Rydygier   |    175 |  7 |
+---------+------------+--------+----+
```

Zauważmy też, że baza danych nie pozwala nam na wpisanie nowego rekordu z jednoczesnym określeniem wartości pola id (o ile wartość id łamie warunek unikalności):

```
mysql> INSERT INTO student (id, imie, nazwisko, wzrost) VALUES (3, 'Krzysztof',
'Janicki', 173);
ERROR 1062 (23000): Duplicate entry '3' for key 1
```

W praktyce najlepiej w ogóle nie ustawiać pola, którym ma się zajmować baza danych:

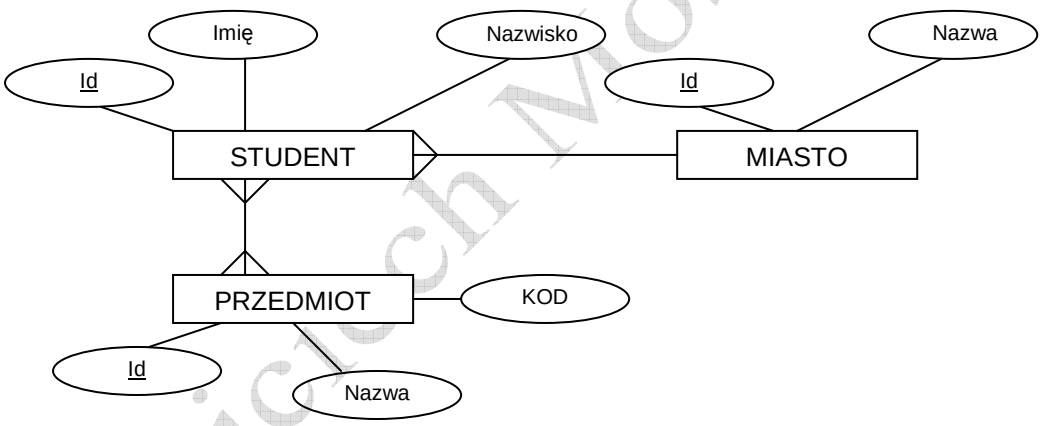
```
mysql> INSERT INTO student (imie, nazwisko, wzrost) VALUES ('Krzysztof', 'Janick
i', 173);
Query OK, 1 row affected (0.00 sec)
```

Jak widać, pole ustawiło się samo:

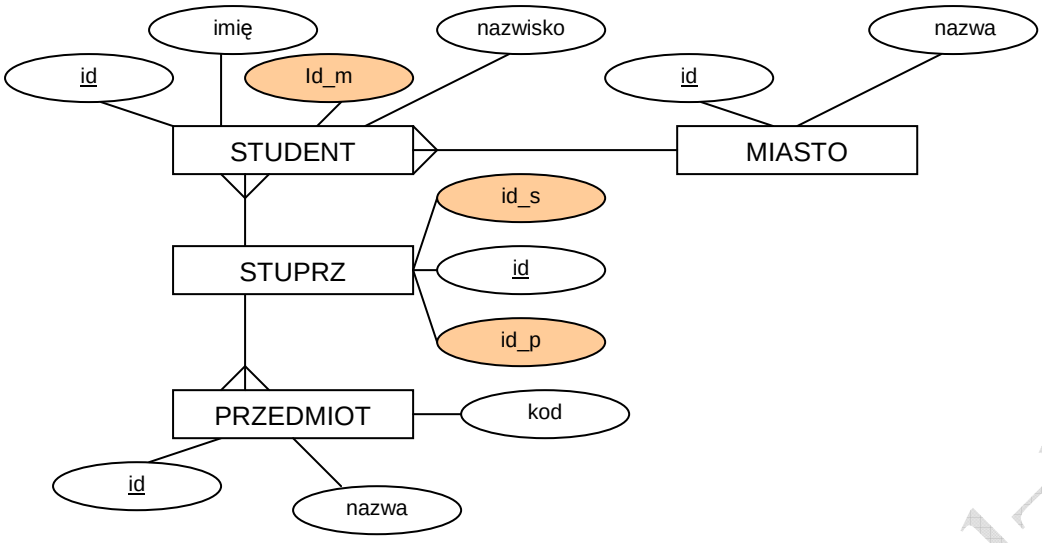
```
mysql> select * from student;
+----------+------------+--------+----+
| imie     | nazwisko   | wzrost | id |
+----------+------------+--------+----+
| Jan      | Kowalski   |    190 |  1 |
| Maria    | Nowak      |    180 |  2 |
| Tadeusz  | Malinowski |    165 |  3 |
| Julia    | Kowalczyk  |    152 |  4 |
| Magda    | Jakubowska |    167 |  5 |
| Iwona    | Krzewska   |    182 |  6 |
| Paulina  | Rydygier   |    175 |  7 |
| Krzysztof | Janicki   |    173 | 11 |
+----------+------------+--------+----+
8 rows in set (0.00 sec)
```

Dlaczego w polu id jest wartość 11, a nie 8? Dlatego, że w międzyczasie wykonano na bazie kilka operacji – m.in. wstawienie i usunięcie rekordu w tablicy student. Autoinkrementacja nie działa zawsze tak, że nowa wartość zwiększa się o 1! Co więcej, jeśli usuniemy rekord o id=2, to nowotworzony student nie dostanie „brakującego" identyfikatora. Dzieje się tak z powodów bezpieczeństwa – mogłoby dojść do sytuacji, w których taki „nowy rekord" zostałby powiązany z nieusuniętymi danymi powiązanymi. Powiemy o tym dalej.

Rozbudujmy naszą bazę danych o kolejne encje. Jedna z nich będzie powiązana relacją wiele-do-wiele, a druga jeden-do-wiele. Oczywiście, w tym pierwszym przypadku potrzebna będzie nam dodatkowa tabela:



Popatrzym na powyższy diagram ERD. Widzimy, że nasz STUDENT jest powiązany z encją MIASTO. Jest to miasto, w którym się urodził. Oczywiście każdy STUDENT może się urodzić tylko w jednym mieście, ale każde miasto może się pochwalić wieloma studentami – jest to relacja 1-do-wielu. Popatrzmy na encję PRZEDMIOT. Każdy STUDENT może uczęszczać na wiele PRZEDMIOTÓW, a każdego z PRZEDMIOTÓW może się uczyć wielu STUDENTÓW. Stąd też mamy relację wiele-do-wiele. Oznacza to, że musimy wprowadzić dodatkową tabelę dla przechowywania informacji o relacji.



Przyjrzyjmy się zmianom na diagramie ERD. Po pierwsze, pojawiły się atrybuty zaznaczone kolorem. Są to tak zwane klucze zewnętrzne, czyli takie, które są kluczami głównymi w innych tabelach. Po drugie pojawiła się dodatkowa tabele STUPRZ, która odpowiada relacji wiele do wiele pomiędzy encjami STUDENT i PRZEDMIOT. Pole id w tej tabeli nie jest wymagane, ale jest dobrą praktyką bazodanową – o czym będziemy wspominać dalej.

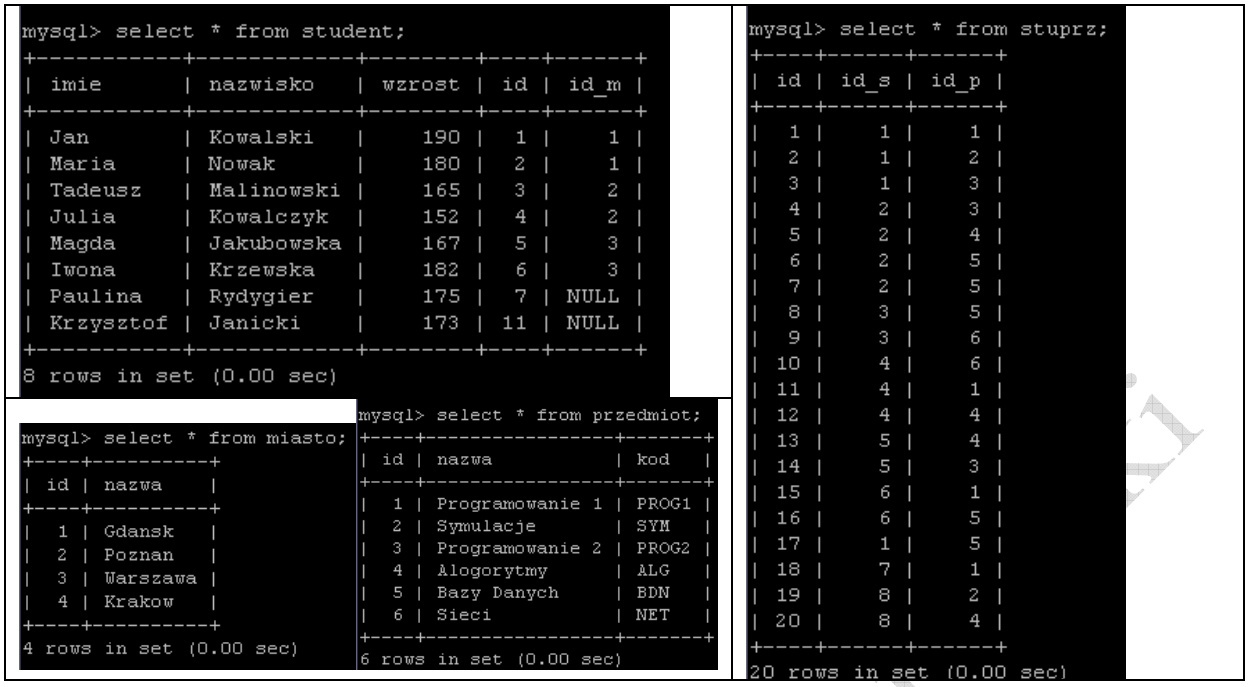
Tworzymy potrzebne nam tabele (na razie w najprostszej możliwej wersji):

```
mysql> CREATE TABLE stuprz (id int auto_increment primary key, id_s int, id_p in
t);
Query OK, 0 rows affected (0.01 sec)

mysql> CREATE TABLE miasto (id int auto_increment primary key, nazwa char(50));
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE przedmiot (id int auto_increment primary key, nazwa char(50)
, kod char(7));
Query OK, 0 rows affected (0.00 sec)
```

Oczywiście w tabeli STUPRZ pola id_s i id_m nie mają autoinkrementacji – bo nie są to klucze główne!

```
mysql> alter table student add column id_m int;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0
```

Załóżmy, że nasze nowe tablice wyglądają tak:



```
mysql> select * from student;
+-----------+------------+--------+----+------+
| imie      | nazwisko   | wzrost | id | id_m |
+-----------+------------+--------+----+------+
| Jan       | Kowalski   |    190 |  1 |    1 |
| Maria     | Nowak      |    180 |  2 |    1 |
| Tadeusz   | Malinowski |    165 |  3 |    2 |
| Julia     | Kowalczyk  |    152 |  4 |    2 |
| Magda     | Jakubowska |    167 |  5 |    3 |
| Iwona     | Krzewska   |    182 |  6 |    3 |
| Paulina   | Rydygier   |    175 |  7 | NULL |
| Krzysztof | Janicki    |    173 | 11 | NULL |
+-----------+------------+--------+----+------+
8 rows in set (0.00 sec)
```

```
mysql> select * from stuprz;
+----+------+------+
| id | id_s | id_p |
+----+------+------+
|  1 |    1 |    1 |
|  2 |    1 |    2 |
|  3 |    1 |    3 |
|  4 |    2 |    3 |
|  5 |    2 |    4 |
|  6 |    2 |    5 |
|  7 |    2 |    5 |
|  8 |    3 |    5 |
|  9 |    3 |    6 |
| 10 |    4 |    6 |
| 11 |    4 |    1 |
| 12 |    4 |    4 |
| 13 |    5 |    4 |
| 14 |    5 |    3 |
| 15 |    6 |    1 |
| 16 |    6 |    5 |
| 17 |    1 |    5 |
| 18 |    7 |    1 |
| 19 |    8 |    2 |
| 20 |    8 |    4 |
+----+------+------+
20 rows in set (0.00 sec)
```

```
mysql> select * from miasto;
+----+----------+
| id | nazwa    |
+----+----------+
|  1 | Gdansk   |
|  2 | Poznan   |
|  3 | Warszawa |
|  4 | Krakow   |
+----+----------+
4 rows in set (0.00 sec)
```

```
mysql> select * from przedmiot;
+----+-----------------+-------+
| id | nazwa           | kod   |
+----+-----------------+-------+
|  1 | Programowanie 1 | PROG1 |
|  2 | Symulacje       | SYM   |
|  3 | Programowanie 2 | PROG2 |
|  4 | Algorytmy       | ALG   |
|  5 | Bazy Danych     | BDN   |
|  6 | Sieci           | NET   |
+----+-----------------+-------+
6 rows in set (0.00 sec)
```

Po pierwsze, zastanówmy się, jak reprezentowane są związki pomiędzy elementami encji.

Popatrzymy na Jana Kowalskiego. Pole id_m tego rekordu zawiera wartość 1. Ponieważ jest to klucz zewnętrzny z tabeli MIASTO, szukamy w tej tabeli rekordu o id=1. Wynika z niego, że Jan Kowalski pochodzi z Gdańska.

Sprawdźmy, na jakie przedmioty zapisany jest ten student? Ponieważ encje STUDENT i PRZEDMIOT są związane relacją wiele-do-wiele, szukamy w tablicy STUPRZ tych rekordów, których pole id_s=1.

```
mysql> select * from stuprz where id_s=1;
+----+------+------+
| id | id_s | id_p |
+----+------+------+
|  1 |    1 |    1 |
|  2 |    1 |    2 |
|  3 |    1 |    3 |
| 17 |    1 |    5 |
+----+------+------+
4 rows in set (0.00 sec)
```

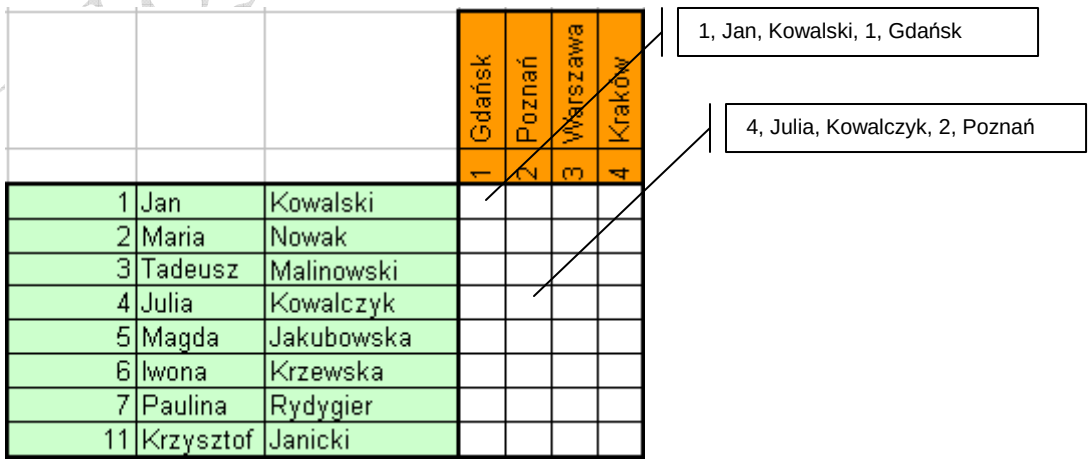
Jak widać, Jan Kowalski jest zapisany na zajęcia z Programowania 1, Symulacji, Programowania 2 i Baz Danych.

Powróćmy do prostszej relacji – tej, która łączy tabele STUDENT i MIASTO. Jak uzyskać informację o wszystkich studentach wraz z miastami ich pochodzenia?

Po pierwsze, możemy wykorzystać zapytanie SELECT działające na 2 tabelach. Spróbujmy więc:

```
mysql> select * from student, miasto;
+----------+------------+--------+----+------+----+----------+
| imie     | nazwisko   | wzrost | id | id_m | id | nazwa    |
+----------+------------+--------+----+------+----+----------+
| Jan      | Kowalski   |    190 |  1 |    1 |  1 | Gdansk   |
| Jan      | Kowalski   |    190 |  1 |    1 |  2 | Poznan   |
| Jan      | Kowalski   |    190 |  1 |    1 |  3 | Warszawa |
| Jan      | Kowalski   |    190 |  1 |    1 |  4 | Krakow   |
| Maria    | Nowak      |    180 |  2 |    1 |  1 | Gdansk   |
| Maria    | Nowak      |    180 |  2 |    1 |  2 | Poznan   |
| Maria    | Nowak      |    180 |  2 |    1 |  3 | Warszawa |
| Maria    | Nowak      |    180 |  2 |    1 |  4 | Krakow   |
| Tadeusz  | Malinowski |    165 |  3 |    2 |  1 | Gdansk   |
| Tadeusz  | Malinowski |    165 |  3 |    2 |  2 | Poznan   |
| Tadeusz  | Malinowski |    165 |  3 |    2 |  3 | Warszawa |
| Tadeusz  | Malinowski |    165 |  3 |    2 |  4 | Krakow   |
| Julia    | Kowalczyk  |    152 |  4 |    2 |  1 | Gdansk   |
| Julia    | Kowalczyk  |    152 |  4 |    2 |  2 | Poznan   |
| Julia    | Kowalczyk  |    152 |  4 |    2 |  3 | Warszawa |
| Julia    | Kowalczyk  |    152 |  4 |    2 |  4 | Krakow   |
| Magda    | Jakubowska |    167 |  5 |    3 |  1 | Gdansk   |
| Magda    | Jakubowska |    167 |  5 |    3 |  2 | Poznan   |
| Magda    | Jakubowska |    167 |  5 |    3 |  3 | Warszawa |
| Magda    | Jakubowska |    167 |  5 |    3 |  4 | Krakow   |
| Iwona    | Krzewska   |    182 |  6 |    3 |  1 | Gdansk   |
| Iwona    | Krzewska   |    182 |  6 |    3 |  2 | Poznan   |
| Iwona    | Krzewska   |    182 |  6 |    3 |  3 | Warszawa |
| Iwona    | Krzewska   |    182 |  6 |    3 |  4 | Krakow   |
| Paulina  | Rydygier   |    175 |  7 | NULL |  1 | Gdansk   |
| Paulina  | Rydygier   |    175 |  7 | NULL |  2 | Poznan   |
| Paulina  | Rydygier   |    175 |  7 | NULL |  3 | Warszawa |
| Paulina  | Rydygier   |    175 |  7 | NULL |  4 | Krakow   |
| Krzysztof | Janicki   |    173 | 11 | NULL |  1 | Gdansk   |
| Krzysztof | Janicki   |    173 | 11 | NULL |  2 | Poznan   |
| Krzysztof | Janicki   |    173 | 11 | NULL |  3 | Warszawa |
| Krzysztof | Janicki   |    173 | 11 | NULL |  4 | Krakow   |
+----------+------------+--------+----+------+----+----------+
32 rows in set (0.00 sec)
```

Jak widać, dostajemy dość nieoczekiwane wyniki. Dzieje się tak dlatego, że SELECT wykonany na dwóch (i więcej) tabelach polega na utworzeniu wszystkich możliwych kombinacji rekordów z pierwszej i drugiej tabeli. Jest to tak zwany iloczyn kartezjański.



Oczywiście nie wszystkie z 32 rekordów wynikowych mają sens. Powinniśmy wyfiltrować z uzyskanego zbioru te, dla których wartość pola id_m w tabeli STUDENT odpowiada pewnemu id w tabeli MIASTO.

Jak wcześniej wspomnieliśmy, do filtrowania wyników służy klauzula WHERE:

```
mysql> select * from student, miasto where id_m=id;
ERROR 1052 (23000): Column 'id' in where clause is ambiguous
```

Jak widać, zapytanie się nie udaje. Dzieje się tak dlatego, że baza danych nie jest pewna, czy warunek id_m=id dotyczy pola id z tabeli STUDENT, czy z tabeli MIASTO. Wprowadzamy więc dodatkowo oznaczenie tabeli:

```
mysql> select * from student, miasto where student.id_m=miasto.id;
+---------+------------+--------+----+------+----+----------+
| imie    | nazwisko   | wzrost | id | id_m | id | nazwa    |
+---------+------------+--------+----+------+----+----------+
| Jan     | Kowalski   |    190 |  1 |    1 |  1 | Gdansk   |
| Maria   | Nowak      |    180 |  2 |    1 |  1 | Gdansk   |
| Tadeusz | Malinowski |    165 |  3 |    2 |  2 | Poznan   |
| Julia   | Kowalczyk  |    152 |  4 |    2 |  2 | Poznan   |
| Magda   | Jakubowska |    167 |  5 |    3 |  3 | Warszawa |
| Iwona   | Krzewska   |    182 |  6 |    3 |  3 | Warszawa |
+---------+------------+--------+----+------+----+----------+
6 rows in set (0.00 sec)
```

Zapytanie łączące w ten sposób kilka tabel mogłoby być dość trudne do przeczytania i poprawnego zinterpretowania, dlatego

SELECT * FROM tabela1, tabela2 WHERE tabela1.kolumna1 = tabela2.kolumna2

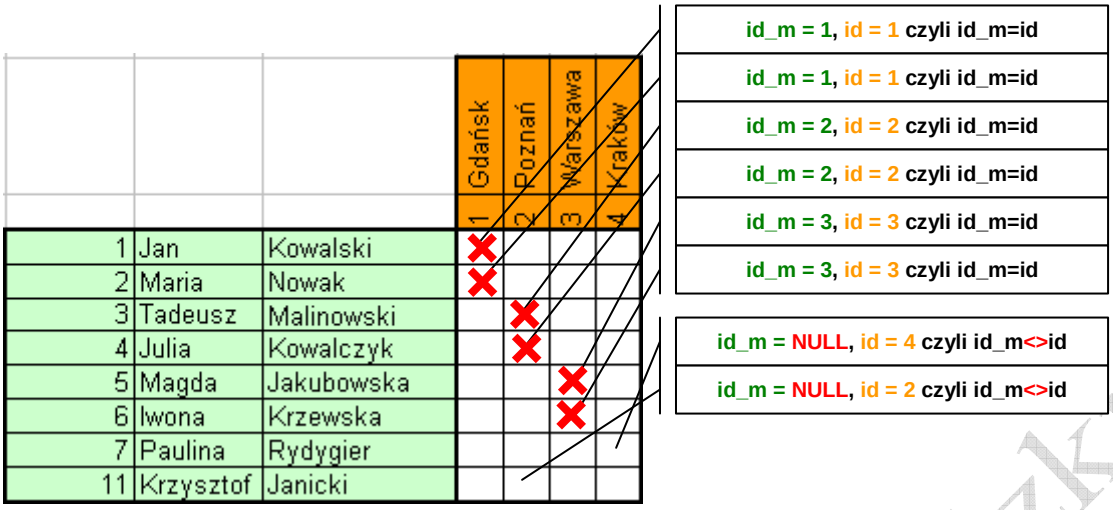
zapisuje się w postaci (jest to tak zwany INNER JOIN):

SELECT * FROM tabela1 JOIN tabela2 ON (tabela1.kolumna1 = tabela2.kolumna2)

Na pierwszy rzut oka może to wyglądać na wprowadzenie dodatkowego zamieszania, ale – jak się potem przekonamy – wbrew pozorom wcale tak nie jest.

```
mysql> select * from student  JOIN miasto on (student.id_m=miasto.id);
+---------+------------+--------+----+------+----+----------+
| imie    | nazwisko   | wzrost | id | id_m | id | nazwa    |
+---------+------------+--------+----+------+----+----------+
| Jan     | Kowalski   |    190 |  1 |    1 |  1 | Gdansk   |
| Maria   | Nowak      |    180 |  2 |    1 |  1 | Gdansk   |
| Tadeusz | Malinowski |    165 |  3 |    2 |  2 | Poznan   |
| Julia   | Kowalczyk  |    152 |  4 |    2 |  2 | Poznan   |
| Magda   | Jakubowska |    167 |  5 |    3 |  3 | Warszawa |
| Iwona   | Krzewska   |    182 |  6 |    3 |  3 | Warszawa |
+---------+------------+--------+----+------+----+----------+
6 rows in set (0.00 sec)
```

Jak widzimy, widać już poprawne wyniki. Jest tylko jeden problem – na liście brakuje 2 studentów! Dzieje się tak dlatego, że warunek id_m=id wymaga, żeby istniały odpowiednie wartości w kolumnach id_m i id, tymczasem dla dwóch studentów (Paulina i Krzysztof) nie określono wartości id_m (wartość wynosi NULL). Podobny problem występuje z miastami – nie widać na liście Krakowa.



Naszym celem jest teraz przygotowanie listy studentów, wraz z miastami ich pochodzenia (lub wartością NULL, gdy jest ono nieznane). Używamy do tego tak zwanego złączenia OUTER JOIN. Ponieważ zależy nam na dopuszczeniu wartości NULL po stronie tablicy STUDENT, stosujemy tak zwany OUTER LEFT JOIN (bo tablica STUDENT jest po lewej stronie):

```
mysql> select * from student LEFT OUTER JOIN miasto on (student.id_m=miasto.id);
+-----------+------------+--------+----+------+------+----------+
| imie      | nazwisko   | wzrost | id | id_m | id   | nazwa    |
+-----------+------------+--------+----+------+------+----------+
| Jan       | Kowalski   |    190 |  1 |    1 |    1 | Gdansk   |
| Maria     | Nowak      |    180 |  2 |    1 |    1 | Gdansk   |
| Tadeusz   | Malinowski |    165 |  3 |    2 |    2 | Poznan   |
| Julia     | Kowalczyk  |    152 |  4 |    2 |    2 | Poznan   |
| Magda     | Jakubowska |    167 |  5 |    3 |    3 | Warszawa |
| Iwona     | Krzewska   |    182 |  6 |    3 |    3 | Warszawa |
| Paulina   | Rydygier   |    175 |  7 | NULL | NULL | NULL     |
| Krzysztof | Janicki    |    173 | 11 | NULL | NULL | NULL     |
+-----------+------------+--------+----+------+------+----------+
8 rows in set (0.00 sec)
```

Wersja RIGHT OUTER JOIN dopuszcza wartości NULL (w tym przypadku – nie istniejące) po stronie prawej:

```
mysql> select * from student RIGHT OUTER JOIN miasto on (student.id_m=miasto.id)
;
+---------+------------+--------+------+------+----+----------+
| imie    | nazwisko   | wzrost | id   | id_m | id | nazwa    |
+---------+------------+--------+------+------+----+----------+
| Jan     | Kowalski   |    190 |    1 |    1 |  1 | Gdansk   |
| Maria   | Nowak      |    180 |    2 |    1 |  1 | Gdansk   |
| Tadeusz | Malinowski |    165 |    3 |    2 |  2 | Poznan   |
| Julia   | Kowalczyk  |    152 |    4 |    2 |  2 | Poznan   |
| Magda   | Jakubowska |    167 |    5 |    3 |  3 | Warszawa |
| Iwona   | Krzewska   |    182 |    6 |    3 |  3 | Warszawa |
| NULL    | NULL       |   NULL | NULL | NULL |  4 | Krakow   |
+---------+------------+--------+------+------+----+----------+
7 rows in set (0.00 sec)
```

Niektóre bazy danych (nie wszystkie) dopuszczają także tzw. pełny OUTER JOIN – który jest połączeniem LEFT OUTER i RIGHT OUTER.

Kiedy potrafimy już efektywnie łączyć tablice bazy danych, możemy pokusić się o wprowadzenie nowej funkcjonalności – agregowania i grupowania.

Agregowanie to wyliczanie pewnej wartości (jednej) dla całego zbioru danych. Zacznijmy od prostego przykładu – policzmy, jaki jest największy wzrost w grupie studenckiej. Służy do tego funkcja agregująca max():

```
mysql> select max(wzrost) from student;
+-------------+
| max(wzrost) |
+-------------+
|         190 |
+-------------+
1 row in set (0.00 sec)
```

Zauważmy, że w odróżnieniu od klasycznego SELECT nie wskazujemy bezpośrednio kolumny (wzrost), ale przeprowadzamy na niej operację – max().

Możemy także wyliczyć wartość najmniejszą min(), średnią avg() lub zliczyć elementy funkcją count():

```
mysql> select count(wzrost)  from student;
+---------------+
| count(wzrost) |
+---------------+
|             8 |
+---------------+
1 row in set (0.01 sec)
```

Możliwe jest także wyliczanie kilku agregatów jednocześnie:

```
mysql> select count(wzrost), avg(wzrost), min(wzrost), max(wzrost)  from student
;
+---------------+-------------+-------------+-------------+
| count(wzrost) | avg(wzrost) | min(wzrost) | max(wzrost) |
+---------------+-------------+-------------+-------------+
|             8 |    173.0000 |         152 |         190 |
+---------------+-------------+-------------+-------------+
1 row in set (0.02 sec)
```

Spróbujmy teraz połączyć dwa elementy: łączenie tablic ze zliczaniem elementów. Spróbujmy policzyć studentów we wszystkich miastach.

Zacznijmy od wyznaczenia wszystkich miast i pochodzących z nich studentów:

```
mysql> select nazwa, nazwisko from miasto LEFT OUTER JOIN student ON (miasto.id
= student.id_m);
+----------+------------+
| nazwa    | nazwisko   |
+----------+------------+
| Gdansk   | Kowalski   |
| Gdansk   | Nowak      |
| Poznan   | Malinowski |
| Poznan   | Kowalczyk  |
| Warszawa | Jakubowska |
| Warszawa | Krzewska   |
| Krakow   | NULL       |
+----------+------------+
7 rows in set (0.00 sec)
```

Nie jest to jednak najlepsza droga - interesują nas bowiem tylko miasta i liczba studentów (a nie ich nazwiska). Dlatego pozbędziemy się zbędnej kolumny:

```
mysql> select nazwa from miasto LEFT OUTER JOIN student ON (miasto.id = student.
id_m);
+----------+
| nazwa    |
+----------+
| Gdansk   |
| Gdansk   |
| Poznan   |
| Poznan   |
| Warszawa |
| Warszawa |
| Krakow   |
+----------+
7 rows in set (0.00 sec)
```

Teraz wystarczy pogrupować miasta i policzyć je:

```
mysql> select nazwa, count(student.id) from miasto LEFT OUTER JOIN student ON (m
iasto.id = student.id_m) GROUP BY nazwa;
+----------+-------------------+
| nazwa    | count(student.id) |
+----------+-------------------+
| Gdansk   |                 2 |
| Krakow   |                 0 |
| Poznan   |                 2 |
| Warszawa |                 2 |
+----------+-------------------+
4 rows in set (0.00 sec)
```

Jak widać, nasze zapytanie rozbudowaliśmy o dwa elementy: wskazaliśmy, że chcemy liczyć identyfikatory studentów (count(student.id)) i zażądaliśmy, aby grupować studentów według miast (GROUP BY nazwa).

Otrzymaliśmy w wyniku tabelę – czasami warto jednak móc ją dalej filtrować, na przykład po to, aby wyznaczyć miasta, z liczbą studentów równą 0. Zwykle do filtrowania używaliśmy klauzuli WHERE – tu jednak nie możemy jej użyć, bowiem mogło się przecież zdarzyć, że użyjemy ją wcześniej. Na przykład, gdybyśmy chcieli wyznaczyć listę miast ze studentami – ale tylko tych miast, które zawierają w nazwie literę k):

```
mysql> select nazwa, count(student.id) from miasto LEFT OUTER JOIN student ON (m
iasto.id = student.id_m) WHERE nazwa LIKE '%k%' GROUP BY nazwa;
+--------+-------------------+
| nazwa  | count(student.id) |
+--------+-------------------+
| Gdansk |                 2 |
| Krakow |                 0 |
+--------+-------------------+
2 rows in set (0.00 sec)
```

Powyższe zapytanie jest dość abstrakcyjne (żeby nie powiedzieć: dziwaczne), ale dobrze pokazuje, dlaczego nie możemy użyć WHERE do filtrowania efektów GROUP BY.

W rezultacie, żeby można było filtrować efekty grupowania, używa się dodatkowej klauzuli: HAVING:

```
mysql> select nazwa, count(student.id) from miasto LEFT OUTER JOIN student ON (m
iasto.id = student.id_m) WHERE nazwa LIKE '%k%' GROUP BY nazwa HAVING count(stud
ent.id)=2;
+--------+-------------------+
| nazwa  | count(student.id) |
+--------+-------------------+
| Gdansk |                 2 |
+--------+-------------------+
1 row in set (0.00 sec)
```

To zapytanie pokazuje wszystkie te miasta, które w nazwie mają literę K, dla których liczba studentów jest równa 2.

Warto zapamiętać: HAVING filtruje wyniki GROUP BY tak, jak WHERE filtruje wyniki SELECT. Nie można stosować HAVING do wyników zwykłego SELECT, tak samo, jak nie można filtrować GROUP BY za pomocą WHERE.