W tym odcinku poznasz funkcje grupujące i dwie nowe klauzule instrukcji SELECT — GROUP BY i HAVING. Nauczysz się też grupować dane, czyli łączyć wiele wierszy w jeden.

Grupowanie danych polega na łączeniu wielu wierszy w jeden. W najprostszym przypadku łączy się wszystkie wiersze tabeli w jedną grupę, ale możliwe jest też ich podzielenie pomiędzy wiele grup. Wtedy podstawą zaklasyfikowania wiersza do danej grupy jest wartość jednej z kolumn lub wynik wyrażenia.

Funkcje grupujące

Funkcje, które zwracają jedną wartość obliczoną na podstawie przekazanego zbioru parametrów, nazywamy funkcjami grupującymi. W każdym serwerze baz danych, w tym w MySQL-u, zaimplementowano najważniejsze i najczęściej używane funkcje tego typu — minimum, maksimum, średnią, sumę itd.

Specyfika języka SQL powoduje, że łatwiej jest najpierw wytłumaczyć, jak korzystać z funkcji grupujących, a dopiero później — jak grupować dane.

Wiesz już, że parametrem wywołania funkcji grupujących nie są pojedyncze wartości, ale grupy (zbiory) wartości i że dzięki tym funkcjom uzyskujemy pojedynczy wynik obliczony na podstawie wielu argumentów. Na przykład możemy w tabeli policzyć wiersze spełniające określone kryteria lub możemy wyliczyć wartość średnią dla wszystkich wartości z wybranej kolumny. Użycie tych funkcji zwykle związane jest z operacją na wskazanych kolumnach (na których wykonywane są obliczenia), a jako wynik zwracany jest tylko jeden wiersz.

Charakterystyczną cechą funkcji grupujących jest operowanie na zbiorach, a nie pojedynczych wartościach. Dzięki temu otrzymane w wyniku grupowania dane mogą być użyte jako argumenty ich wywołania. Jeżeli wszystkie wiersze tabeli są połączone w jedną grupę, funkcja grupująca będzie wywołana tylko raz, w innym przypadku zostanie wywołana dla każdej grupy. Funkcje grupujące zwracają pojedyncze (skalarne) wartości, więc wywołuje się je w klauzuli SELECT, tak jak wcześniej poznane funkcje systemowe.

Funkcja COUNT()

Pierwszą funkcją agregującą, którą chcę dokładnie omówić, jest funkcja COUNT(). Funkcja ta zlicza w przekazanym zbiorze wartości wystąpienia różne od NULL, chyba że jako argumentu użyto znaku * (gwiazdka) — takie wywołanie funkcji spowoduje zliczenie wszystkich wierszy, łącznie z duplikatami i wartościami NULL. Argumentem funkcji mogą być liczby, daty, znaki i ciągi znaków (listing 6.1).

Jeśli chcemy znać liczbę wierszy zwróconych przez zapytanie, najprościej jest użyć funkcji COUNT(*). Są dwa powody, dla których warto tak wywołać funkcję COUNT() do tego celu. Po pierwsze, pozwalamy optymalizatorowi bazy danych wybrać kolumnę do wykonywania obliczeń, co czasem nieznacznie podnosi wydajność zapytania, po drugie, nie musimy się martwić o wartości Null zawarte w kolumnie oraz o to, czy kolumna o podanej nazwie w ogóle istnieje.

Listing 6.1. Zapytanie zwracające liczbę klientów

```
SELECT COUNT(*) as 'Liczba klientów'

FROM customer;

+-----------------+

| Liczba klientów |

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

| 16 |

+----------+

Jak widać, zapytanie zwróciło jedną wartość wyliczoną przez funkcję grupującą COUNT() na zbiorze równym zawartości tabeli item. Gdybyśmy chcieli policzyć imiona i nazwiska klientów, otrzymalibyśmy nieco inny wynik. Wywołanie funkcji w postaci COUNT(nazwa kolumny) nie uwzględnia pól z wartościami Null . Fakt, że wiersze z wartością Null nie są zliczane, może być przydatny, gdy wartość Null oznacza coś szczególnego lub gdy chcemy sprawdzić, czy w bazie nie brakuje istotnych informacji (listing 6.2).

Listing 6.2. Funkcja COUNT() wywołana dla dwóch różnych zbiorów — raz dla nazwisk, raz dla imion klientów. Jak widać, jedna osoba nie podała nam imienia.

SELECT COUNT(fname), COUNT(lname)

FROM customer;

+----------+----------+

| COUNT(fname) | COUNT(lname) |

+----------+----------+

| 15 | 16 |

+----------+----------+

Jeżeli chcemy policzyć unikatowe wystąpienia wartości, wystarczy wykorzystać wiedzę z wcześniejszych odcinków kursu i właściwie użyć słowa kluczowego DISTINCT (listing 6.3).

Listing 6.3. Zapytanie zwracające liczbę miast, w których mieszkają nasi klienci — w pierwszej kolumnie to samo miasto liczone jest tyle razy, ilu mieszka w nim klientów, w drugiej kolumnie każde miasto policzone jest tylko raz

SELECT COUNT(town), COUNT(DISTINCT(town))

FROM customer;

+----------+----------+

| COUNT(town) | COUNT(DISTINCT(town)) |

+----------+----------+

| 15 | 12 |

+----------+----------+

Domyślnie funkcje grupujące nie eliminują powtarzających się wierszy, co odpowiada użyciu kwalifikatora All jako pierwszego argumentu wywołania. Jeżeli chcemy ograniczyć dziedzinę funkcji do unikatowych wartości wierszy, należy zastosować kwalifikator DISTINCT. **Funkcja SUM()**

Za pomocą funkcji SUM() dodawane są wszystkie wartości rekordów wybranych w zapytaniu i zwracany jest pojedynczy wynik. W przeciwieństwie do funkcji COUNT(), która działa dla wszystkich typów danych, argumentami funkcji SUM() mogą być wyłącznie liczby. Funkcja SUM(), tak jak inne funkcje grupujące, nie

uwzględnia wartości Null (listing 6.4).

Listing 6.4. Zapytanie zwracające liczbę wszystkich towarów w magazynie

```
SELECT SUM(quantity)

FROM stock;

+---------------+

| SUM(quantity) |

+---------------+

| 52            |

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

Oczywiście, zbiór argumentów wywołania funkcji grupujących możemy ograniczać za pomocą wcześniej omówionej klauzuli WHERE (listing 6.5).

Listing 6.5. Liczba drewnianych puzzli — ponieważ nazwy towarów przechowywane są w innej tabeli niż stany magazynowe, konieczne było użycie klauzuli JOIN

```
SELECT SUM(quantity)

FROM stock

INNER JOIN item USING (item_id)

WHERE description ='Wood Puzzle';

+---------------+

| SUM(quantity) |

+---------------+

| 12            |

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

W języku SQL mamy jeszcze jedną możliwość ograniczania zbioru argumentów funkcji grupujących. Zamiast wywoływać taką funkcję raz dla całej tabeli (albo, jak w ostatnim przykładzie, dla wybranych wierszy), można pogrupować dane i wywołać funkcję grupującą dla każdej grupy. Ten sposób przedstawiony jest w dalszej części odcinka.

Funkcja AVG()

W wyniku działania funkcji AVG() zwracana jest wartość średnia dla podanych wyrażeń. Wiersze przechowujące wartość Null nie są uwzględniane. Argumentami funkcji AVG() muszą być dane liczbowe. Aby na przykład obliczyć średnie ceny sprzedaży i zakupu towarów, napiszemy (listing 6.6):

Listing 6.6. Średnia cena wszystkich kupionych i sprzedawanych towarów

```
SELECT AVG (cost_price), AVG (sell_price)
```

```
FROM item;

+-----------------+-----------------+

| AVG (cost_price) | AVG (sell_price) |

+-----------------+-----------------+

| 7.249091        | 10.435455       |

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

Wynikiem funkcji grupujących są pojedyncze liczby. Język SQL jako język przeznaczony do operacji na zbiorach danych początkowo nie umożliwiał definiowania zmiennych. Ostatnio prawie wszystkie serwery baz danych obsługują zmienne, ale wiele operacji, które w proceduralnych językach programowania wymagały ich użycia, w SQL możemy wykonać bez zadeklarowania choćby jednej zmiennej (listing 6.7).

Listing 6.7. Zapytanie zwracające różnicę pomiędzy średnią ceną zakupu a średnią ceną sprzedaży towarów

```
SELECT AVG (sell_price) - AVG (cost_price)

FROM item;

+-------------------------------------+

| AVG (sell_price) - AVG (cost_price) |

+-------------------------------------+

| 3.186364                            |

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

To ważne, więc jeszcze raz przypomnę, że wszystkie funkcje grupujące, z wyjątkiem funkcji COUNT (*), ignorują wartości Null. Czyli średnia zbioru {5,5} wynosi 5, średnia zbioru {5,5,0} wynosi 3,33, ale średnia zbioru (5,5, Null} równa się 5.

Funkcje MIN() i MAX()

Funkcja MIN() służy do znajdowania wartości najmniejszej w zbiorze wartości, a funkcja MAX() — największej. Obie funkcje, podobnie jak funkcja COUNT(), mogą być użyte dla różnych typów danych.

Za pomocą funkcji MAX() można znaleźć największy łańcuch danych, najnowszą datę (lub najodleglejszą w przyszłości) oraz największą liczbę w zestawieniu. W wyniku działania funkcji MIN() można znaleźć odpowiednio wartości najmniejsze. Aby znaleźć datę pierwszej operacji naszej firmy, możemy skorzystać z instrukcji pokazanej na listingu 6.8.

Listing 6.8. Data pierwszego zamówienia

```
SELECT MIN(date_placed)

FROM orderinfo;

+------------------+

| MIN(date_placed) |
```

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

| 2000-03-13       |

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

Kolejny przykład pokazuje, jak odczytać informację o największej marży, z którą sprzedajemy towary (listing 6.9).

Listing 6.9. Przykład wywołania funkcji grupującej na danych będących wynikiem wcześniejszych obliczeń, lecz nieodczytanych bezpośrednio z tabeli

```
SELECT MAX(sell_price - cost_price)

FROM item;

+-----------------------------+

| MAX(sell_price - cost_price) |

+-----------------------------+

| 6.72                        |

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

W rezultacie znaleźliśmy wartość najwyższej marży, ale nadal nie wiemy, który towar sprzedajemy z takim zyskiem. Próba dodania do klauzuli SELECT nazw towarów oczywiście skończy się błędem (listing 6.10).

Listing 6.10. Funkcja grupująca zwraca jedną wartość (pierwsza kolumna wyniku miałaby jedno pole), a nazwy towarów są odczytywane bezpośrednio z tabeli, zatem druga kolumna wyniku musiałaby mieć wiele pól

```
SELECT MAX(sell_price - cost_price), description

FROM item;

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

Rozwiązanie tego problemu zostanie przedstawione przy okazji opisywania klauzuli GROUP BY w dalszej części tego odcinka.

Instrukcja SELECT musi zwracać dane w postaci tabelarycznej — czasami jest to po prostu tabela złożona z jednej kolumny i jednego wiersza.

Funkcja STDDEV_POP()

Funkcja STDDEV_POP() wyznacza odchylenie standardowe zbioru. Wiersze przechowujące wartość Null nie są uwzględniane. Argumentami funkcji muszą być dane liczbowe. Aby wyznaczyć odchylenie standardowe dla cen zakupu wszystkich towarów, napiszemy (listing 6.11):

Listing 6.11. Funkcja STDDEV w MySQL-u jest zaimplementowana w celu poprawienia kompatybilności z serwerami Oracle; aby obliczyć odchylenie standardowe, należy zatem używać funkcji STDDEV_POP()

```
SELECT STDDEV_POP(cost_price)

FROM item;
```

+-----------------------+

| STDDEV_POP(cost_price) |

+-----------------------+

| 6.278899 |

+-----------------------+

Funkcja VARIANCE()

Funkcja VARIANCE() wyznacza wariancje zbioru. Wiersze przechowujące wartość Null nie są uwzględniane. Argumentami funkcji muszą być dane liczbowe. Aby obliczyć wariancje ceny sprzedaży, należy wykonać instrukcję (listing 6.12):

Listing 6.12. MySQL ma wbudowane najważniejsze funkcje statystyczne

SELECT VARIANCE(sell_price)

FROM item;

+----------------------+

| VARIANCE(sell_price) |

+----------------------+

| 75.666534 |

+----------------------+

Funkcja GROUP_CONCAT()

Funkcja GROUP_CONCAT() w przeciwieństwie do pozostałych funkcji grupujących zwraca dane tekstowe, a nie liczbowe. Jej działanie jest bardzo proste — łączy w jeden ciąg znaków przekazane argumenty wywołania. Poniższy przykład pokazuje, jak przygotować listę imion klientów (listing 6.13).

Listing 6.13. Często musimy połączyć przechowywane w różnych polach dane tekstowe — w MySQL-u może to za nas zrobić funkcja GROUP_CONCAT(). Jej rozbudowana składnia pozwala na sortowanie danych i określenie symbolu separatora

SELECT GROUP_CONCAT(DISTINCT fname ORDER BY fname)

FROM customer;

+---+

| GROUP_CONCAT(DISTINCT fname ORDER BY fname) |

+---+

| Adrian,Alex,Andrew,Anna,Bill,Christine,Dave,David,Jenny,Laura,Mike,Neil,Richard,Simon|

+---+

Grupowanie danych

Do tej pory wywoływaliśmy funkcje grupujące raz dla całych tabel lub ich fragmentów. Klauzula GROUP BY umożliwia grupowanie wyników względem zawartości wybranych kolumn. W wyniku jej działania uzyskujemy podział wierszy tablicy na dowolne grupy. W pewnym sensie jej działanie jest podobne do działania operatora DISTINCT, ponieważ po jej zastosowaniu zwracany jest pojedynczy wynik dla każdej grupy (listing 6.14).

Listing 6.14. Klauzula GROUP BY użyta do wyeliminowania duplikatów — skoro dane są grupowane według identyfikatorów osób, czyli zamówienia złożone przez tę samą osobę dodawane są do jednej grupy, to w wyniku nie może pojawić się kilka razy ten sam identyfikator klienta

```
SELECT customer_id

FROM orderinfo

GROUP BY customer_id;

+-------------+

| customer_id |

+-------------+

| 3           |

| 8           |

| 13          |

| 15          |

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

Jeżeli jednak w zapytaniu użyjemy jednocześnie funkcji grupującej, to ta funkcja zostanie wywołana niezależnie dla każdej grupy zdefiniowanej w klauzuli GROUP BY. W bazie test informacje o zamówieniach przechowywane są w tabeli orderinfo, a poszczególne pozycje zamówienia — w tabeli orderline (dzięki temu w ramach każdego zamówienia klient może kupić dowolną liczbę najróżniejszych towarów). Pierwsze zapytanie (listing 6.15) zwraca liczbę wszystkich sprzedanych towarów, drugie (listing 6.16) rozbija tę liczbę na poszczególne zamówienia.

Listing 6.15. Ogólna liczba sprzedanych towarów

```
SELECT SUM(quantity)

FROM orderline;

+---------------+

| SUM(quantity) |

+---------------+

| 15            |

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

Listing 6.16. Liczba towarów sprzedanych w ramach poszczególnych zamówień

```
SELECT SUM(quantity)

FROM orderline

GROUP BY orderinfo_id;

+---------------+

| SUM(quantity) |

+---------------+

| 3             |

| 6             |

| 2             |

| 2             |

| 2             |

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

Świetnie, ale w drugim wyniku wyraźnie brakuje informacji o tym, w ramach którego z zamówień sprzedano tyle a tyle towarów. Wcześniej próbowaliśmy dodać dane tego typu do klauzuli SELECT i skończyło się to błędem. Czy klauzula GROUP BY coś zmieniła? (listing 6.17).

Listing 6.17. Ta instrukcja SELECT jest jak najbardziej poprawna — przecież dane zostały pogrupowane według wartości orderinfo_id, a następnie dla każdej grupy była wywołana funkcja grupująca

```
SELECT orderinfo_id, SUM(quantity)

FROM orderline

GROUP BY orderinfo_id;

+--------------+---------------+

| orderinfo_id | SUM(quantity) |

+--------------+---------------+

| 1            | 3             |

| 2            | 6             |

| 3            | 2             |

| 4            | 2             |

| 5            | 2             |

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

Zapamiętaj — jeżeli w klauzuli SELECT występują dowolne funkcje grupujące, to wszystkie nazwy kolumn i wyrażenia, które NIE SĄ argumentami tych funkcji, muszą być wymienione w klauzuli GROUP BY. Innymi słowy, w takich zapytaniach w klauzuli SELECT mogą występować tylko i wyłącznie wyrażenia, które są wymienione w klauzuli GROUP BY, chyba że są one argumentami dowolnej funkcji agregującej.

Jak wiemy, język SQL umożliwia, poprzez zastosowanie klauzuli ORDER BY, porządkowanie wyników zapytania. Możliwe jest też sortowanie wyników na podstawie wyniku funkcji grupujących (listing 6.18).

Listing 6.18. Uporządkowany wynik poprzedniego zapytania

```
SELECT orderinfo_id, SUM(quantity)

FROM orderline

GROUP BY orderinfo_id

ORDER BY SUM(quantity) DESC;

+--------------+---------------+

| orderinfo_id | SUM(quantity) |

+--------------+---------------+

| 2            | 6             |

| 1            | 3             |

| 3            | 2             |

| 4            | 2             |

| 5            | 2             |

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

W zapytaniach grupujących dane możemy używać klauzuli WHERE tak samo jak klauzuli ORDER BY . W ten sposób ograniczymy liczbę wierszy, jeszcze zanim będą one dzielone na grupy i podgrupy. Przy stosowaniu klauzuli WHERE łącznie z GROUP BY najpierw realizowane jest ograniczenie wynikające z kryteriów w klauzuli WHERE. Następnie wybrane rekordy są grupowane i powstaje ostateczny wynik zapytania (listing 6.19).

Listing 6.19. Wynik poprzedniego zapytania ograniczony do zamówień z pierwszej połowy 2000 roku

```
SELECT orderinfo_id, SUM(quantity)

FROM orderline

JOIN orderinfo USING (orderinfo_id)

WHERE date_placed BETWEEN '2000-01-01' AND '2000-06-31'

GROUP BY orderinfo_id

ORDER BY SUM(quantity) DESC;
```

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

| orderinfo_id | SUM(quantity) |

+--------------+---------------+

| 2            | 6             |

| 1            | 3             |

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

Otrzymane w ten sposób grupy wierszy możemy dalej dzielić na podgrupy. Wymieniając w klauzuli GROUP BY wiele wyrażeń, podzielimy zbiór wierszy na grupy wyznaczone pierwszym wyrażeniem, grupy te podzielimy na podgrupy na podstawie wartości drugiego wyrażenia itd. (listing 6.20).

Listing 6.20. Liczba towarów kupionych przez poszczególnych klientów w ramach poszczególnych zamówień

```
SELECT fname, orderinfo_id, SUM(quantity)

FROM orderline

JOIN orderinfo USING (orderinfo_id)

JOIN customer USING (customer_id)

GROUP BY fname, orderinfo_id

ORDER BY fname;

+-------+--------------+---------------+

| fname | orderinfo_id | SUM(quantity) |

+-------+--------------+---------------+

| Alex  | 1            | 3             |

| Anna  | 2            | 6             |

| Anna  | 5            | 2             |

| David | 3            | 2             |

| Laura | 4            | 2             |

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

Operator ROLLUP

Jeżeli dane są grupowane według wartości kilku kolumn, to, jak pokazały poprzednie przykłady, kolejność ich występowania w klauzuli GROUP BY wyznacza podział na grupy i podgrupy. Jeżeli tych grup i podgrup jest niewiele, użytkownicy potrafią samodzielnie wyliczyć brakujące podsumowania, w tym przypadku sumy cen wszystkich towarów dla poszczególnych grup. Możemy jednak dodać do wyniku zapytania takie sumy pośrednie — służy do tego operator ROLLUP. Jego działanie prześledzimy najpierw na bardzo prostym przykładzie (listing 6.21).

Listing 6.21. Pierwsze zapytanie zwraca liczbę sprzedanych egzemplarzy każdego towaru, drugie zawiera dodatkowy wiersz z podsumowaniem sprzedaży wszystkich towarów

```
SELECT i.description, SUM(o.quantity)

FROM item AS i

JOIN orderline AS O USING (item_id)

GROUP BY i.description;

+---------------+-----------------+

| description   | SUM(o.quantity) |

+---------------+-----------------+

| Carrier Bag   |               1 |

| Fan Large     |               3 |

| Linux CD      |               1 |

| Picture Frame |               2 |

| Roman Coin    |               1 |

| Rubik Cube    |               1 |

| Tissues       |               3 |

| Wood Puzzle   |               3 |

+---------------+-----------------+


SELECT i.description, SUM(o.quantity)

FROM item AS i

JOIN orderline AS O USING (item_id)

GROUP BY i.description

WITH ROLLUP;

+---------------+-----------------+

| description   | SUM(o.quantity) |

+---------------+-----------------+

| Carrier Bag   |               1 |
```

| Fan Large | 3 |

| Linux CD | 1 |

| Picture Frame | 2 |

| Roman Coin | 1 |

| Rubik Cube | 1 |

| Tissues | 3 |

| Wood Puzzle | 3 |

| NULL | 15 |

+---------------+-----------------+

Operator ROLLUP jest szczególnie przydatny w zapytaniach grupujących dane według wartości kilku kolumn — w takim przypadku zwróci on dodatkowy wiersz z podsumowaniem dla każdej grupy wartości (listing 6.22).

Listing 6.22. Zapytanie zwracające informacje o liczbie różnych towarów kupionych przez poszczególnych klientów. Dodatkowe wiersze z całkowitą sumą towarów kupionych przez klientów oraz całkowitym podsumowaniem sprzedaży zostały dodane przez operator ROLLUP

```
SELECT oi.customer_id, i.description, SUM(o.quantity)

FROM item AS i

JOIN orderline AS O USING (item_id)

JOIN orderinfo AS oi USING (orderinfo_id)

GROUP BY oi.customer_id, i.description

WITH ROLLUP;
```

+-------------+---------------+-----------------+

| customer_id | description | SUM(o.quantity) |

+-------------+---------------+-----------------+

| 3 | Fan Large | 1 |

| 3 | Roman Coin | 1 |

| 3 | Tissues | 1 |

| 3 | NULL | 3 |

| 8 | Carrier Bag | 1 |

| 8 | Fan Large | 2 |

```
|          8 | Linux CD      |              1 |
|          8 | Tissues       |              2 |
|          8 | Wood Puzzle   |              2 |
|          8 | NULL          |              8 |
|         13 | Picture Frame |              2 |
|         13 | NULL          |              2 |
|         15 | Rubik Cube    |              1 |
|         15 | Wood Puzzle   |              1 |
|         15 | NULL          |              2 |
|       NULL | NULL          |             15 |
+------------+---------------+-----------------+
```

Klauzula HAVING

Język SQL dostarcza jeszcze jedną metodę filtrowania wyników zapytań — jeżeli grupujemy wyniki (a więc używamy klauzuli GROUP BY), możemy sprawdzić, czy te grupy wierszy spełniają jakiś warunek. Wiesz już, że po zastosowaniu klauzuli WHERE wyniki zapytania najpierw są filtrowane, a potem grupowane. Klauzula HAVING, tak jak WHERE, umożliwia określenie testu logicznego, ale w jej przypadku będzie on zastosowany do grup, a nie pojedynczych wierszy. Testy logiczne zawarte w klauzuli HAVING wykonywane są na całych grupach, a nie na pojedynczych rekordach. Tak więc klauzula ta służy do wybierania interesujących nas grup, a klauzula WHERE — interesujących nas wierszy. Warunek umieszczony w klauzuli HAVING wyrażony jest za pomocą dowolnej funkcji grupowej (listing 6.23).

Listing 6.23. Informacje o zamówieniach, których wartość przekroczyła 30. Iloczyn quantity*sell_price jest wyliczany dla każdego sprzedanego towaru, a następnie wyliczana jest suma wartości dla każdej grupy, czyli w tym przypadku dla każdego zamówienia. Na końcu klauzula HAVING usuwa z wyniku te grupy (zamówienia), których wartość nie przekroczyła 30

```
SELECT orderinfo_id, SUM(quantity*sell_price)
FROM orderinfo JOIN orderline USING (orderinfo_id)
JOIN item USING (item_id)
GROUP BY orderinfo_id
HAVING  SUM(quantity*sell_price)> 30;
+--------------+--------------------------+
| orderinfo_id | SUM(quantity*sell_price) |
+--------------+--------------------------+
| 2            | 69.83                    |
```

```
| 3           | 33.44                  |

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

Klauzule HAVING i WHERE mogą wystąpić w tym samym zapytaniu — w takim przypadku najpierw będzie zastosowany test z klauzuli WHERE, a następnie — z klauzuli HAVING (listing 6.24).

Listing 6.24. Wyniki poprzedniego zamówienia ograniczone do zamówień złożonych w pierwszej połowie 2000 roku

```
SELECT orderinfo_id, SUM(quantity*sell_price)

FROM orderinfo JOIN orderline USING (orderinfo_id)

JOIN item USING (item_id)

   WHERE date_placed BETWEEN '2000-01-01' AND '2000-06-31'

   GROUP BY orderinfo_id

   HAVING SUM(quantity*sell_price)> 30;

+--------------+--------------------------+

| orderinfo_id | SUM(quantity*sell_price) |

+--------------+--------------------------+

| 2            | 69.83                    |

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

Zapisanie w klauzuli HAVING warunku, który jest sprawdzany na poziomie wierszy, nie jest błędem składniowym (czyli MySQL prawidłowo zinterpretuje i wykona takie zapytanie), ale taka instrukcja jest nie tylko nieelegancka i nieczytelna, ale również może być dłużej wykonywana (listing 6.25).

Listing 6.25. Klauzula HAVING użyta do wybierania wierszy

```
SELECT item_id, SUM(quantity*sell_price)

FROM orderinfo JOIN orderline USING (orderinfo_id)

JOIN item USING (item_id)

GROUP BY item_id

HAVING item_id IN (1,2,3,4);

+---------+--------------------------+

| item_id | SUM(quantity*sell_price) |

+---------+--------------------------+

| 1       | 65.85                    |
```

| 2 | 11.49 |

| 3 | 2.49 |

| 4 | 11.97 |

+---------+--------------------------+

W takim przypadku należy zastosować klauzulę WHERE (listing 6.26).

Listing 6.26. Funkcjonalnie takie same, ale czytelniejsze i szybsze rozwiązanie

```
SELECT item_id, SUM(quantity*sell_price)
FROM orderinfo JOIN orderline USING (orderinfo_id)
JOIN item USING (item_id)
WHERE item_id IN (1,2,3,4)
GROUP BY item_id;
```

+---------+--------------------------+

| item_id | SUM(quantity*sell_price) |

+---------+--------------------------+

| 1 | 65.85 |

| 2 | 11.49 |

| 3 | 2.49 |

| 4 | 11.97 |

+---------+--------------------------+

Kolejność wykonywania klauzuli zapytań

Skoro poznałeś wszystkie klauzule instrukcji SELECT, powinieneś wiedzieć, kiedy są one wykonywane przez serwery bazodanowe. Logiczna kolejność wykonywania zapytania zawierającego omówione dotychczas klauzule jest następująca:

1. Jako pierwsza wykonywana jest klauzula FROM. Jeżeli zapytanie odwołuje się do wielu tabel, są one kolejno ze sobą złączane.

2. Otrzymany w ten sposób zbiór pośredni jest filtrowany na podstawie warunku logicznego umieszczonego w klauzuli WHERE. Tylko te wiersze, dla których jest on prawdziwy, trafiają do kolejnego zbioru pośredniego.

3. Następnie wykonywana jest klauzula GROUP BY, czyli grupowane są tylko przefiltrowane wiersze.

4. Utworzone grupy są filtrowane poprzez porównanie ich z warunkiem umieszczonym w klauzuli HAVING.

5. Wybrane w poprzednim punkcie wiersze są zwracane, czyli wykonywana jest klauzula SELECT.

6. Następnie wiersze są sortowane, czyli wykonywana jest klauzula ORDER BY.

7. Na końcu liczba wierszy wyniku zapytania jest ograniczana podczas wykonywania klauzuli LIMIT.