# SQL dla analityków danych

## Tworzenie zbiorów danych dla początkujących

Renée M.P. Teate

WILEY

Tytuł oryginału: SQL for Data Scientists A Beginner's Guide for Building Datasets for Analysis

Tłumaczenie: Filip Kamiński

ISBN: 978-83-283-9744-6



Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*https://helion.pl/user/opinie/sqland*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
*https://ftp.helion.pl/przyklady/sqland.zip*

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

# Spis treści

6

ROZDZIAŁ

# Agregacja wyników na potrzeby analizy

W analizie danych SQL staje się szczególnie przydatny, gdy używasz go do agregowania danych. Za pomocą instrukcji `GROUP BY` możemy określić poziom podsumowania danych. Następnie możemy wykorzystać funkcje agregujące/agregacyjne do podsumowania wartości w każdej grupie.

Analitycy danych mogą wykorzystać język SQL do tworzenia dynamicznych raportów podsumowujących, które — poprzez ponowne uruchomienie zapytania — mogą być automatycznie aktualizowane po wzbogaceniu bazy o nowe dane. U podstaw pulpitów menedżerskich i raportów zbudowanych przy użyciu oprogramowania takiego jak Tableau i Cognos często leżą zapytania SQL, które są wykonywane w celu pobrania i zagregowania danych wykorzystywanych w raportach (raportowanie omówię w rozdziale 10., „Tworzenie zbiorów danych na potrzeby raportów analitycznych"). Analitycy danych mogą wykorzystać język SQL do podsumowywania danych na poziomie wymaganym przez modele uczenia maszynowego. Zagadnienie to omówię bardziej szczegółowo w rozdziale 12., „Tworzenie zbiorów danych na potrzeby uczenia maszynowego".

Tworzenie wszystkich tych zbiorów danych rozpoczyna się od agregacji.

## Składnia GROUP BY

Podstawową składnię zapytania `SELECT` poznałeś w rozdziale 2., „Instrukcja SELECT". Dwie, jeszcze nieomówione przeze mnie części tego zapytania to klauzule `GROUP BY` i `HAVING`, które służą do agregacji danych:

```
SELECT [kolumny do zwrócenia]
FROM [tabela]
WHERE [instrukcje warunkowe do filtrowania]
```

```
GROUP BY [nazwy kolumn do grupowania]
HAVING [warunkowe instrukcje filtrujące, które są uruchamiane po grupowaniu]
ORDER BY [kolumny, względem których ma się odbyć sortowanie]
```

Po słowach kluczowych `GROUP BY` umieszcza się listę, oddzielonych od siebie przecinkami, nazw kolumn, które określają, w jaki sposób chcemy podsumować wyniki zapytania.

Aby uzyskać listę identyfikatorów klientów, którzy dokonali zakupów w każdym dniu targowym, możemy wykorzystać poznane do tej pory elementy SQL-a (bez grupowania) i napisać zapytanie podobne do poniższego:

```
SELECT
    market_date,
    customer_id
FROM farmers_market.customer_purchases
ORDER BY market_date, customer_id
```

Zastosowanie takiego zapytania spowodowałoby, że każdy wiersz w danych wyjściowych odpowiadałby jednemu zakupionemu przez klienta produktowi. Dane wyjściowe zawierałyby duplikaty (potencjalnie kilkakrotne wystąpienia identyfikatora klienta na każdy dzień), ponieważ powyższe zapytanie odnosi się do tabeli `customer_purchases` bez określonego grupowania.

Aby otrzymać po jednym wierszu na klienta na dzień targowy, możemy pogrupować wyniki za pomocą klauzuli `GROUP BY`, w której zapiszemy, że chcemy podsumować dane według pól `customer_id` i `market_date`:

```
SELECT
    market_date,
    customer_id
FROM farmers_market.customer_purchases
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
```

Ten sam rezultat możesz też otrzymać, usuwając duplikaty za pomocą `SELECT DISTINCT`. W tym przypadku zdecydowałam się na użycie `GROUP BY`, ponieważ mam zamiar dodać do danych wyjściowych kolumny podsumowujące.

# Statystyki podsumowujące grupy

Po zgrupowaniu danych na interesującym Cię poziomie do zapytania możesz dodać funkcje agregujące, takie jak `SUM` i `COUNT`, które pozwolą obliczyć statystyki podsumowujące dane z tabeli `customer_purchases` w każdej z grup. Poniższe zapytanie wykorzystuje funkcję `COUNT()` do obliczenia liczby wierszy z tabeli `customer_purchases`, jakie przypadają na poszczególny dzień targowy i klienta. Wynik poniższego zapytania pokazano na rysunku 6.1.

```
SELECT
    market_date,
    customer_id,
    COUNT(*) AS items_purchased
```

```
FROM farmers_market.customer_purchases
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
LIMIT 10
```

| market_date | customer_id | items_purchased |
|---|---|---|
| 2019-03-02 | 1 | 3 |
| 2019-03-02 | 2 | 1 |
| 2019-03-02 | 3 | 1 |
| 2019-03-02 | 4 | 2 |
| 2019-03-02 | 10 | 1 |
| 2019-03-09 | 1 | 4 |
| 2019-03-09 | 4 | 5 |
| 2019-03-09 | 5 | 1 |
| 2019-03-09 | 7 | 1 |
| 2019-03-09 | 12 | 4 |

Rysunek 6.1

Zwróć uwagę na sposób konstrukcji tabeli `customer_purchases`. Gdyby klient kupił jednocześnie trzy identyczne produkty (na przykład pomidory) od tego samego sprzedawcy, to w wyniku powyższego zapytania w kolumnie `items_purchased` znalazłaby się wartość 1, ponieważ zakup ten byłby zapisany w jednym wierszu tabeli `customer_purchases` (z `quantity` równym 3). (Wróć do rysunków 1.7 i 2.7 z rozdziałów 1., „Źródła danych”, i 2., „Instrukcja SELECT”). Gdyby klient kupił trzy pomidory, odszedł od stoiska, a następnie wrócił i zakupił kolejne trzy, to te zakupy w powyższym zapytaniu dałyby wartość 2 w polu `items_purchased`, ponieważ drugi osobny zakup wygenerowałby nowy wiersz w bazie danych.

Jeśli zamiast zliczać pozycje, chcielibyśmy zsumować wszystkie zakupione sztuki (tak aby uwzględnić wszystkie sześć pomidorów), możemy zsumować liczności w kolumnie `quantity` za pomocą poniższego zapytania. Jego wynik pokazano na rysunku 6.2.

```
SELECT
    market_date,
    customer_id,
    SUM(quantity) AS items_purchased
FROM farmers_market.customer_purchases
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
LIMIT 10
```

Kolumna `items_purchased` nie zawiera już liczb całkowitych, ponieważ niektóre z sumowanych przez nas liczności to masy produktów sprzedawanych luzem. Po zapoznaniu się z tymi wynikami i uświadomieniu sobie, że uwzględniono w nich również masy produktów sprzedawanych na wagę, możesz zdecydować, że raportowanie sprzedaży w ten sposób nie ma sensu i że w zamian zapiszesz jedynie, ile różnych rodzajów produktów zostało kupionych przez każdego klienta. W tym podejściu w obliczeniach dodamy jeden, jeśli klient kupił pomidory — niezależnie od tego, ile ich kupił lub ile razy danego dnia to zrobił. Sumę zwiększymy tylko wtedy, jeśli klient kupił inne produkty, takie jak sałata.

| market_date | customer_id | items_purchased |
|---|---|---|
| 2019-03-02 | 1 | 5.70 |
| 2019-03-02 | 2 | 4.60 |
| 2019-03-02 | 3 | 8.40 |
| 2019-03-02 | 4 | 3.40 |
| 2019-03-02 | 10 | 1.00 |
| 2019-03-09 | 1 | 5.20 |
| 2019-03-09 | 4 | 13.20 |
| 2019-03-09 | 5 | 1.00 |
| 2019-03-09 | 7 | 2.00 |
| 2019-03-09 | 12 | 6.90 |

Rysunek 6.2

> Ten rodzaj modyfikacji często pojawia się podczas projektowania raportów przez analityka danych lub zleceniodawcę/klienta. Musisz zrozumieć granularność i strukturę tabeli, aby upewnić się, że wynik zawiera to, co Ci się wydaje. Polecam zacząć od zapytania bez agregacji — pozwoli Ci to przyjrzeć się wartościom, które będziesz podsumowywał przed ich pogrupowaniem.

To, czego teraz potrzebujemy, to liczba różnych (`DISTINCT`) identyfikatorów produktów w zakupach każdego klienta. Zapytanie ją obliczające pokazano poniżej. Jego wynik znajdziesz na rysunku 6.3. Zamiast zliczać wiersze w tabeli `customer_purchases` przypadające na dzień targowy i klienta, tak jak w przypadku zapytania z `COUNT(*)`, lub sumować liczności, jak w przypadku `SUM(quantity)`, tym razem identyfikujemy liczbę unikalnych wartości `product_id` istniejących w wierszach tworzących każdą grupę, czyli liczbę różnych rodzajów produktów zakupionych przez każdego klienta w każdym dniu targowym:

```
SELECT
    market_date,
    customer_id,
    COUNT(DISTINCT product_id) AS different_products_purchased
FROM farmers_market.customer_purchases c
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
LIMIT 10
```

| market_date | customer_id | different_products_purchased |
|---|---|---|
| 2019-03-02 | 1 | 2 |
| 2019-03-02 | 2 | 1 |
| 2019-03-02 | 3 | 1 |
| 2019-03-02 | 4 | 2 |
| 2019-03-02 | 10 | 1 |
| 2019-03-09 | 1 | 3 |
| 2019-03-09 | 4 | 4 |
| 2019-03-09 | 5 | 1 |
| 2019-03-09 | 7 | 1 |
| 2019-03-09 | 12 | 4 |

Rysunek 6.3

Możemy również połączyć dwa poprzednie podsumowania w jedno zapytanie, pokazane poniżej. Jego wynik jest widoczny na rysunku 6.4:

```
SELECT
    market_date,
    customer_id,
    SUM(quantity) AS items_purchased,
    COUNT(DISTINCT product_id) AS different_products_purchased
FROM farmers_market.customer_purchases
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
LIMIT 10
```

| market_date | customer_id | items_purchased | different_products_purchased |
|---|---|---|---|
| 2019-03-02 | 1 | 5.70 | 2 |
| 2019-03-02 | 2 | 4.60 | 1 |
| 2019-03-02 | 3 | 8.40 | 1 |
| 2019-03-02 | 4 | 3.40 | 2 |
| 2019-03-02 | 10 | 1.00 | 1 |
| 2019-03-09 | 1 | 5.20 | 3 |
| 2019-03-09 | 4 | 13.20 | 4 |
| 2019-03-09 | 5 | 1.00 | 1 |
| 2019-03-09 | 7 | 2.00 | 1 |
| 2019-03-09 | 12 | 6.90 | 4 |

Rysunek 6.4

W jednym zapytaniu możesz wykorzystać dowolną liczbę różnych funkcji agregujących, które zostaną zastosowane na tym samym poziomie grupowania. W powyższym zapytaniu podsumowuję dane według daty (`market_date`) oraz identyfikatora klienta (`client_id`). Zwróć uwagę, w jaki sposób wykorzystuję aliasy kolumn do opisywania, co jest podsumowywane.

Jeśli nadal chcesz zsumować liczności produktów, ale nie podoba Ci się to, że kolumna `items_purchased` zawiera zarówno pojedyncze elementy, jak i masy produktów sprzedawanych luzem (z których część może być wyrażona w funtach, a część w uncjach, więc nie należy ich do siebie dodawać), to opis jednego z możliwych rozwiązań tego problemu znajdziesz w podrozdziale „Instrukcje CASE wewnątrz funkcji agregujących".

# Obliczenia wewnątrz funkcji agregujących

W funkcjach agregujących można również uwzględnić operacje matematyczne, które są obliczane na poziomie wierszy przed podsumowaniem danych. W rozdziale 3., „Klauzula WHERE", dowiedziałeś się, jak wyświetlić listę zakupów konkretnej osoby za pomocą klauzuli `WHERE`. Klient o identyfikatorze 3 kupił pozycje pokazane na rysunku 6.5. Informacje te można uzyskać za pomocą następującego zapytania, w którym obliczam ceny poszczególnych pozycji:

```
SELECT
    market_date,
    customer_id,
```

```
    vendor_id,
    quantity * cost_to_customer_per_qty AS price
FROM farmers_market.customer_purchases
WHERE
    customer_id = 3
ORDER BY market_date, vendor_id
```

| market_date | customer_id | vendor_id | price |
|---|---|---|---|
| 2019-03-02 | 3 | 4 | 16.8000 |
| 2019-03-16 | 3 | 4 | 11.0000 |
| 2019-03-16 | 3 | 9 | 18.0000 |

Rysunek 6.5

Powiedzmy, że chcielibyśmy ustalić, ile łącznie wydał ten klient każdego dnia targowego (razem u wszystkich sprzedawców). Do zsumowania cen zakupionych produktów możemy wykorzystać klauzulę `GROUP BY market_date` i funkcję `SUM()`:

```
SELECT
    customer_id,
    market_date,
    SUM(quantity * cost_to_customer_per_qty) AS total_spent
FROM farmers_market.customer_purchases
WHERE
    customer_id = 3
GROUP BY market_date
ORDER BY market_date
```

Wewnątrz funkcji `SUM()` obliczam ceny produktów w poszczególnych wierszach (tak jak w pierwszym zapytaniu), a następnie sumuję wartość wewnątrz grup, które w tym przypadku tworzą dni targowe.

Wyniki powyższego zapytania pokazano na rysunku 6.6. Widzimy, że ceny dwóch pozycji z 16 marca 2019 r. zostały do siebie dodane. Zastosowałam alias `total_spent` (łączne wydatki), który lepiej opisuje znaczenie wartości w tej kolumnie. Pogrupowałam dane według `customer_id` oraz `market_date`, dzięki czemu w wynikach mam po jednym wierszu na każdy dzień targowy i klienta.

| customer_id | market_date | total_spent |
|---|---|---|
| 3 | 2019-03-02 | 16.8000 |
| 3 | 2019-03-16 | 29.0000 |

Rysunek 6.6

Zauważ, że z listy kolumn do wyświetlenia oraz z klauzuli `ORDER BY` usunęłam kolumnę `vendor_id`. Jeśli interesuje mnie agregacja na poziomie jednego wiersza na klienta na datę, to nie mogę uwzględnić identyfikatora sprzedawcy w danych wyjściowych, ponieważ każdego dnia klient mógł zrobić zakupy u wielu sprzedawców, co spowodowałoby, że wyniki nie zostałyby zagregowane na oczekiwanym przeze mnie poziomie.

W klauzuli GROUP BY powinnam umieścić również kolumnę customer_id, mimo że w tym przypadku po jej pominięciu uzyskamy te same wyniki. Dodanie customer_id do listy kolumn w GROUP BY spowoduje, że zapytanie zadziała bezbłędnie, nawet jeśli nie ograniczymy wyników do jednego klienta. Zmianę tę wprowadzę w kolejnym zapytaniu.

Co by było, gdybyśmy chcieli ustalić, ile klient wydał u każdego sprzedawcy, niezależnie od daty? W tym przypadku możemy pogrupować wyniki według customer_id i vendor_id:

```
SELECT
    customer_id,
    vendor_id,
    SUM(quantity * cost_to_customer_per_qty) AS total_spent
FROM farmers_market.customer_purchases
WHERE
    customer_id = 3
GROUP BY customer_id, vendor_id
ORDER BY customer_id, vendor_id
```

Wyniki powyższego zapytania pokazano na rysunku 6.7.

| customer_id | vendor_id | total_spent |
|---|---|---|
| 3 | 4 | 27.8000 |
| 3 | 9 | 18.0000 |

Rysunek 6.7

Możemy również pozbyć się filtrowania względem customer_id. W tym przypadku wystarczy usunąć całą klauzulę WHERE, ponieważ nie filtrujemy w niej danych względem wartości z innych kolumn. Użycie samego GROUP BY customer_id spowoduje zwrócenie listy wszystkich klientów wraz z informacją o tym, ile pieniędzy wydały na targowisku poszczególne osoby. Wynik poniższego zapytania pokazano na rysunku 6.8:

```
SELECT
    customer_id,
    SUM(quantity * cost_to_customer_per_qty) AS total_spent
FROM farmers_market.customer_purchases
GROUP BY customer_id
ORDER BY customer_id
```

| customer_id | total_spent |
|---|---|
| 1 | 100.3750 |
| 2 | 25.4000 |
| 3 | 45.8000 |
| 4 | 66.6250 |
| 5 | 16.0000 |
| 7 | 15.0000 |
| 10 | 8.0000 |
| 12 | 95.8000 |

Rysunek 6.8

Do tej pory wykonywaliśmy agregację danych z jednej tabeli, ale można ją również przeprowadzić na połączonych tabelach. Przed dodaniem instrukcji `GROUP BY` dobrym pomysłem jest połączenie tabel bez użycia funkcji agregujących i upewnienie się, że dane są na oczekiwanym poziomie szczegółowości (oraz że złączenie nie generuje duplikatów).

Załóżmy, że w zapytaniu pogrupowanym według `customer_id` i `vendor_id` chcemy wyświetlić dane klienta, takie jak imię i nazwisko oraz nazwę sprzedawcy. W tym celu możemy najpierw połączyć ze sobą trzy tabele, następnie wybrać kolumny ze wszystkich tabel i sprawdzić dane wyjściowe przed grupowaniem. Wynik poniższego zapytania pokazano na rysunku 6.9:

```
SELECT
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id,
    cp.quantity * cp.cost_to_customer_per_qty AS price
FROM farmers_market.customer c
    LEFT JOIN farmers_market.customer_purchases cp
        ON c.customer_id = cp.customer_id
    LEFT JOIN farmers_market.vendor v
        ON cp.vendor_id = v.vendor_id
WHERE
    cp.customer_id = 3
ORDER BY cp.customer_id, cp.vendor_id
```

| customer_first_name | customer_last_name | customer_id | vendor_name | vendor_id | price |
|---|---|---|---|---|---|
| Bob | Wilson | 3 | Mountain View Vegetables | 4 | 16.8000 |
| Bob | Wilson | 3 | Mountain View Vegetables | 4 | 11.0000 |
| Bob | Wilson | 3 | Annie's Pies | 9 | 18.0000 |

Rysunek 6.9

Aby podsumować dane na poziomie jednego wiersza na klienta i sprzedawcę, konieczne będzie zgrupowanie według znacznie większej liczby pól, w tym wszystkich pól tabeli `customer` i wszystkich pól tabeli `vendor`. Grupowanie przeprowadzamy według wszystkich wyświetlanych w wynikach kolumn, w których nie wykorzystujemy funkcji agregujących. Poniższe zapytanie wyświetla kolumny użyte w grupowaniu. Jego wynik pokazano na rysunku 6.10. W zapytaniu wykorzystałam funkcję `ROUND()` do ładnego sformatowania wartości w kolumnie `total_spent`:

```
SELECT
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id,
    ROUND(SUM(quantity * cost_to_customer_per_qty), 2) AS total_spent
FROM farmers_market.customer c
    LEFT JOIN farmers_market.customer_purchases cp
        ON c.customer_id = cp.customer_id
    LEFT JOIN farmers_market.vendor v
        ON cp.vendor_id = v.vendor_id
```

```
WHERE
    cp.customer_id = 3
GROUP BY
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id
ORDER BY cp.customer_id, cp.vendor_id
```

| customer_first_name | customer_last_name | customer_id | vendor_name | vendor_id | total_spent |
|---|---|---|---|---|---|
| Bob | Wilson | 3 | Mountain View Vegetables | 4 | 27.80 |
| Bob | Wilson | 3 | Annie's Pies | 9 | 18.00 |

Rysunek 6.10

Możemy również zachować ten sam poziom agregacji i skupić się na jednym sprzedawcy zamiast jednym kliencie. Otrzymamy wtedy listę klientów konkretnego sprzedawcy zamiast listy sprzedawców, u których robił zakupy określony klient. Takie zapytanie pokazano poniżej. Zwróć uwagę, że jedynym zmienionym wierszem kodu jest warunek w klauzuli `WHERE`. Chociaż zmieniamy filtr, pozostawiamy ten sam poziom grupowania i pola wyjściowe. Na rysunku 6.11 można zauważyć, że w kolumnie `customer_id` poza 3 znajdziemy też inne wartości. Wartości w kolumnie `vendor_id` są ograniczone do sprzedawcy o identyfikatorze 9:

```
SELECT
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id,
    ROUND(SUM(quantity * cost_to_customer_per_qty), 2) AS total_spent
FROM farmers_market.customer c
    LEFT JOIN farmers_market.customer_purchases cp
        ON c.customer_id = cp.customer_id
    LEFT JOIN farmers_market.vendor v
        ON cp.vendor_id = v.vendor_id
WHERE
    cp.vendor_id = 9
GROUP BY
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id
ORDER BY cp.customer_id, cp.vendor_id
```

| customer_first_name | customer_last_name | customer_id | vendor_name | vendor_id | total_spent |
|---|---|---|---|---|---|
| Jane | Connor | 1 | Annie's Pies | 9 | 18.00 |
| Bob | Wilson | 3 | Annie's Pies | 9 | 18.00 |
| Abigail | Harris | 5 | Annie's Pies | 9 | 16.00 |
| Jack | Wise | 12 | Annie's Pies | 9 | 72.00 |

Rysunek 6.11

Możemy też całkowicie pominąć klauzulę `WHERE` i otrzymać po jednym wierszu dla każdej pary klient – sprzedawca z bazy danych. Takie zapytanie można by wykorzystać w systemie raportowania, który umożliwia filtrowanie danych za pomocą interfejsu graficznego (na przykład Tableau). Zapytanie może zwrócić listę wszystkich klientów, którzy zrobili zakupy u dowolnego sprzedawcy oraz sumę ich wydatków. Narzędzie do raportowania może następnie pozwolić użytkownikowi wybrać dowolnego klienta lub sprzedawcę w celu dynamicznego zawężenia wyników.

Zobacz, jak wszystkie podstawowe elementy SQL-a, które poznałeś w poprzednich rozdziałach, łączą się w zapytania pozwalające tworzyć raporty analityczne!

# Minimum i maksimum

Do sprawdzenia, jaki jest najdroższy i najtańszy produkt w każdej kategorii, przy założeniu, że każdy sprzedawca ustala własne ceny i może je dostosowywać dla poszczególnych klientów (z tego powodu w tabeli `customer_purchases` znajdziesz pole `cost_to_customer_per_qty`, w którym w chwili zakupu pierwotna cena może zostać zastąpiona inną), możemy wykorzystać tabelę `vendor_inventory`, która zawiera kolumnę z pierwotną ceną, jaką sprzedawca wybrał dla każdego towaru w momencie wystawienia go na straganie w każdym dniu targowym.

Zacznijmy od spojrzenia na zawartość tabeli `vendor_inventory` za pomocą następującego zapytania typu `SELECT *`. Jego wynik pokazano na rysunku 6.12.

```
SELECT *
FROM farmers_market.vendor_inventory
ORDER BY original_price
LIMIT 10
```

| market_date | quantity | vendor_id | product_id | original_price |
|---|---|---|---|---|
| 2019-03-09 | 10.00 | 9 | 5 | 5.00 |
| 2019-03-30 | 17.00 | 7 | 13 | 6.00 |
| 2019-03-23 | 8.00 | 7 | 13 | 6.00 |
| 2019-03-02 | 13.00 | 1 | 10 | 6.00 |
| 2019-03-09 | 17.00 | 1 | 10 | 6.00 |
| 2019-03-09 | 8.00 | 7 | 13 | 6.00 |
| 2019-03-20 | 13.00 | 7 | 13 | 6.00 |
| 2019-03-02 | 28.00 | 1 | 11 | 12.00 |
| 2019-03-09 | 10.00 | 1 | 11 | 12.00 |
| 2019-03-20 | 15.00 | 1 | 11 | 13.00 |

Rysunek 6.12

W MySQL-u najtańsze i najdroższe pozycji w całej tabeli można znaleźć bez konieczności grupowania za pomocą funkcji `MIN()` i `MAX()`. Przykład takiego zapytania pokazano poniżej. Jego wyniki znajdziesz na rysunku 6.13.

```
SELECT
    MIN(original_price) AS minimum_price,
    MAX(original_price) AS maximum_price
```

```
FROM farmers_market.vendor_inventory
ORDER BY original_price
```

| minimum_price | maximum_price |
|---|---|
| 2.00 | 18.00 |

Rysunek 6.13

Aby uzyskać najniższą i najwyższą cenę w każdej kategorii, musimy pogrupować wartości według `product_category_id` (i `product_category_name`, jeśli chcemy ją wyświetlić). Po zastosowaniu grupowania wartości zostaną obliczone dla każdej grupy z osobna, tak jak pokazano w poniższym zapytaniu i na rysunku 6.14. W zapytaniu zastosowałam aliasy, ponieważ odwołuję się do wielu tabel i muszę rozróżnić, z której tabeli pochodzi każde pole:

```
SELECT
    pc.product_category_name,
    p.product_category_id,
    MIN(vi.original_price) AS minimum_price,
    MAX(vi.original_price) AS maximum_price
FROM farmers_market.vendor_inventory AS vi
    INNER JOIN farmers_market.product AS p
        ON vi.product_id = p.product_id
    INNER JOIN farmers_market.product_category AS pc
        ON p.product_category_id = pc.product_category_id
GROUP BY pc.product_category_name, p.product_category_id
```

| product_category_name | product_category_id | minimum_price | maximum_price |
|---|---|---|---|
| Fresh Fruits & Vegetables | 1 | 2.00 | 6.00 |
| Packaged Prepared Food | 3 | 4.00 | 18.00 |
| Eggs & Meat (Fresh or Frozen) | 6 | 6.00 | 13.00 |

Rysunek 6.14

Gdybyśmy dodali do zapytania kolumny `MIN(product_name)` oraz `MAX(product_name)`, to nie otrzymalibyśmy nazw produktów związanych z najniższą i najwyższą ceną, a pierwszą i ostatnią nazwę produktu w porządku alfabetycznym. Gdybyśmy chcieli wyświetlić nazwy produktów powiązane z minimalnymi i maksymalnymi cenami w kategoriach, to do tego celu musielibyśmy wykorzystać funkcje okienkowe, które omówię w następnym rozdziale.

# COUNT i COUNT DISTINCT

Załóżmy, że chcieliśmy obliczyć, ile produktów było na sprzedaż w każdym dniu targowym lub ile różnych produktów oferował każdy sprzedawca. Te dwie wartości możemy znaleźć za pomocą funkcji `COUNT` i `COUNT DISTINCT`.

`COUNT` użyte z `GROUP BY` zliczy wiersze w grupie, a `COUNT DISTINCT` unikalne wartości obecne w określonym polu grupy.

Aby ustalić, ile produktów jest oferowanych do sprzedaży każdego dnia, możemy zliczyć wiersze w tabeli `vendor_inventory` pogrupowanej według daty. Nie ustalimy w ten sposób, jaką ilość/liczbę każdego produktu oferowano lub sprzedano każdego dnia (ponieważ nie sumujemy wartości z kolumny `quantity` ani nie liczymy zakupów zrobionych przez klienta), a łączną liczbę dostępnych produktów w danym dniu, gdyż w tabeli znajduje się po jednym wierszu na produkt, sprzedawcę i datę.

Oczywiście wartości widoczne na zrzutach ekranu są zbyt małe, aby były to realistyczne dane. Wykorzystywana przeze mnie baza danych została wypełniona tylko niewielką liczbą przykładowych wierszy, ale na rysunku 6.15 widać, że wynik to liczba dla każdego dnia targowego.

```
SELECT
    market_date,
    COUNT(product_id) AS product_count
FROM farmers_market.vendor_inventory
GROUP BY market_date
ORDER BY market_date
```

| market_date | product_count |
|---|---|
| 2019-03-02 | 4 |
| 2019-03-09 | 9 |
| 2019-03-13 | 2 |
| 2019-03-16 | 3 |
| 2019-03-20 | 3 |
| 2019-03-23 | 2 |
| 2019-03-30 | 2 |

Rysunek 6.15

Jeśli chcielibyśmy ustalić, ile różnych produktów z unikalnymi identyfikatorami wprowadził na rynek w jakimś okresie każdy sprzedawca, to do tego celu moglibyśmy wykorzystać wywołanie `COUNT DISTINCT` na kolumnie `product_id`:

```
SELECT
    vendor_id,
    COUNT(DISTINCT product_id) AS different_products_offered
FROM farmers_market.vendor_inventory
WHERE market_date BETWEEN '2019-03-02' AND '2019-03-16'
GROUP BY vendor_id
ORDER BY vendor_id
```

Zauważ, że słowo kluczowe `DISTINCT` znajduje się w nawiasach wywołania funkcji agregującej `COUNT()`. Wyniki zapytania pokazano na rysunku 6.16.

| vendor_id | different_products_offered |
|---|---|
| 1 | 2 |
| 4 | 1 |
| 7 | 2 |
| 8 | 1 |
| 9 | 3 |

Rysunek 6.16

# Średnia

A co, jeśli poza liczbą różnych produktów dostępnych u sprzedawców chcielibyśmy również obliczyć ich średnią pierwotną cenę u poszczególnych handlarzy? Do poprzedniego zapytania możemy dodać wiersz z wywołaniem funkcji `AVG()`, pokazany w poniższym zapytaniu. Wyniki poniższego zapytania znajdziesz na rysunku 6.17:

```
SELECT
    vendor_id,
    COUNT(DISTINCT product_id) AS different_products_offered,
    AVG(original_price) AS average_product_price
FROM farmers_market.vendor_inventory
WHERE market_date BETWEEN '2019-03-02' AND '2019-03-16'
GROUP BY vendor_id
ORDER BY vendor_id
```

| vendor_id | different_products_offered | average_product_price |
|---|---|---|
| 1 | 2 | 9.000000 |
| 4 | 1 | 2.000000 |
| 7 | 2 | 4.500000 |
| 8 | 1 | 4.000000 |
| 9 | 3 | 14.750000 |

Rysunek 6.17

Zastanówmy się, co tak naprawdę tutaj uśredniamy. Czy wynik możemy nazwać „średnią ceną produktu", jeżeli tabela zawiera po jednym wierszu na produkt każdego rodzaju? Jeśli sprzedawca wystawiłby na stoisku 100 pomidorów, to wszystkie one znalazłyby się w jednym wierszu tabeli `vendor_inventory`. Tym samym cena pomidora byłaby uwzględniona w średniej tylko raz. Jeśli ten sam sprzedawca sprzedawałby również bukiety kwiatów po 20 dolarów, to bez względu na ich liczbę cena ta zostałaby uwzględniona w średniej tylko raz. Obliczona w ten sposób „średnia cena produktu" byłaby po prostu średnią z ceny jednego pomidora i jednego bukietu.

Aby obliczyć rzeczywistą średnią cenę produktu u każdego sprzedawcy w wybranym okresie, lepiej byłoby pomnożyć ilości każdego typu towaru przez jego cenę (operacja do przeprowadzenia dla każdego wiersza), dodać do siebie te wyniki i podzielić je przez całkowitą

liczbę towarów (dla każdego sprzedawcy z osobna). Spróbujmy przeprowadzić takie obliczenia. W poniższym zapytaniu umieściłam obliczenia wewnątrz funkcji `ROUND()`, aby zaokrąglić dane wyjściowe do centów. Wynik pokazano na rysunku 6.18.

```
SELECT
    vendor_id,
    COUNT(DISTINCT product_id) AS different_products_offered,
    SUM(quantity * original_price) AS value_of_inventory,
    SUM(quantity) AS inventory_item_count,
    ROUND(SUM(quantity * original_price) / SUM(quantity), 2) AS
average_item_price
FROM farmers_market.vendor_inventory
WHERE market_date BETWEEN '2019-03-02' AND '2019-03-16'
GROUP BY vendor_id
ORDER BY vendor_id
```

| vendor_id | different_products_offered | value_of_inventory | inventory_item_count | average_item_price |
|---|---|---|---|---|
| 1 | 2 | 636.0000 | 68.00 | 9.35 |
| 4 | 1 | 258.0000 | 129.00 | 2.00 |
| 7 | 2 | 105.0000 | 27.00 | 3.89 |
| 8 | 1 | 400.0000 | 100.00 | 4.00 |
| 9 | 3 | 410.0000 | 30.00 | 13.67 |

Rysunek 6.18

Iloczyn `quantity * original_price` znajdujący się wewnątrz funkcji agregującej jest obliczany w każdym wierszu z osobna, następnie obliczane są sumy oraz wykonywane jest dzielenie jednej sumy przez drugą w celu wyznaczenia *średniej ceny produktu*. W tym zapytaniu wykonuję operacje matematyczne zarówno przed, jak i po podsumowaniu za pomocą `GROUP BY`.

# Filtrowanie za pomocą HAVING

Filtrowanie to kolejna czynność, którą można wykonać w zapytaniu po przeprowadzeniu podsumowania.

W poprzednich rozdziałach oraz w kolejnym zapytaniu filtruję wiersze przy użyciu klauzuli `WHERE`. W poniższym przykładzie wykorzystam klauzulę `WHERE` do wybrania, przez grupowanie, danych z określonego przedziału dat.

Jeśli chcesz przefiltrować wartości po zastosowaniu funkcji agregujących, to do zapytania możesz dodać klauzulę `HAVING`. Spowoduje ona przefiltrowanie danych w grupie na podstawie wartości wykorzystanej do podsumowania.

Zmodyfikuję poprzednie zapytanie i wybiorę jedynie tych sprzedawców, którzy w określonym czasie wystawili na rynku co najmniej 100 produktów. W poniższym kodzie znajdziesz przykład użycia klauzuli `HAVING`. Wynik tego zapytania pokazano na rysunku 6.19:

```
SELECT
    vendor_id,
    COUNT(DISTINCT product_id) AS different_products_offered,
```

```
    SUM(quantity * original_price) AS value_of_inventory,
    SUM(quantity) AS inventory_item_count,
    SUM(quantity * original_price) / SUM(quantity) AS average_item_price
FROM farmers_market.vendor_inventory
WHERE market_date BETWEEN '2019-03-02' AND '2019-03-16'
GROUP BY vendor_id
HAVING inventory_item_count >= 100
ORDER BY vendor_id
```

| vendor_id | different_products_offered | value_of_inventory | inventory_item_count | average_item_price |
|---|---|---|---|---|
| 4 | 1 | 258.0000 | 129.00 | 2.00000000 |
| 8 | 1 | 400.0000 | 100.00 | 4.00000000 |

Rysunek 6.19

Jeśli pogrupujesz wyniki według wszystkich pól, których połączenie powinno być unikalne, a następnie do zapytania dodasz klauzulę `HAVING`, która filtruje zagregowane wiersze za pomocą warunku `COUNT(*) > 1`, to jakiekolwiek wartości w wynikowych danych będą wskazywały, że istnieje więcej niż jeden wiersz z Twoją „unikalną" kombinacją wartości. Innymi słowy, w Twojej bazie danych lub w wyniku zapytania znajdują się niepożądane duplikaty!

# Instrukcje CASE wewnątrz funkcji agregujących

We wcześniejszej części tego rozdziału, w zapytaniu, które wygenerowało dane wyjściowe pokazane na rysunku 6.4, dodałam do siebie wartości z kolumny `quantity` tabeli `customer_purchases`, w której przechowywane są zarówno przedmioty sprzedawane pojedynczo, jak i przedmioty sprzedawane luzem na uncje bądź funty. Dodawanie do siebie tych różnych „ilości" nie do końca miało sens. W rozdziale 4., „Instrukcja CASE", poznałeś instrukcję warunkową `CASE`. Wykorzystam ją teraz do określenia, które rodzaje „ilości" należy zsumować przy użyciu każdej funkcji agregującej `SUM`.

Zacznę od dołączenia tabeli `customer_purchases` do tabeli `products`, aby mieć dostęp do kolumny `product_qty_type`, która obecnie zawiera tylko wartości `unit` i `lbs`. Wynik poniższego zapytania pokazano na rysunku 6.20.

```
SELECT
    cp.market_date,
    cp.vendor_id,
    cp.customer_id,
    cp.product_id,
    cp.quantity,
    p.product_name,
    p.product_size,
    p.product_qty_type
```

```
FROM farmers_market.customer_purchases AS cp
    INNER JOIN farmers_market.product AS p
        ON cp.product_id = p.product_id
```

| market_date | vendor_id | customer_id | product_id | quantity | product_name | product_size | product_qty_type |
|---|---|---|---|---|---|---|---|
| 2019-03-02 | 8 | 10 | 4 | 1.00 | Banana Peppers - Jar | 8 oz | unit |
| 2019-03-09 | 8 | 12 | 4 | 1.00 | Banana Peppers - Jar | 8 oz | unit |
| 2019-03-13 | 8 | 2 | 4 | 2.00 | Banana Peppers - Jar | 8 oz | unit |
| 2019-03-16 | 8 | 10 | 4 | 1.00 | Banana Peppers - Jar | 8 oz | unit |
| 2019-03-02 | 4 | 2 | 9 | 4.60 | Sweet Potatoes | medium | lbs |
| 2019-03-02 | 4 | 3 | 9 | 8.40 | Sweet Potatoes | medium | lbs |
| 2019-03-02 | 4 | 4 | 9 | 1.40 | Sweet Potatoes | medium | lbs |
| 2019-03-09 | 4 | 4 | 9 | 9.90 | Sweet Potatoes | medium | lbs |
| 2019-03-13 | 4 | 2 | 9 | 4.10 | Sweet Potatoes | medium | lbs |
| 2019-03-16 | 4 | 3 | 9 | 5.50 | Sweet Potatoes | medium | lbs |
| 2019-03-02 | 1 | 1 | 10 | 1.00 | Eggs | 1 dozen | unit |
| 2019-03-02 | 1 | 1 | 10 | 3.00 | Eggs | 1 dozen | unit |
| 2019-03-09 | 1 | 1 | 10 | 2.00 | Eggs | 1 dozen | unit |
| 2019-03-09 | 1 | 4 | 10 | 1.00 | Eggs | 1 dozen | unit |
| 2019-03-02 | 1 | 1 | 11 | 1.70 | Pork Chops | 1 lb | lbs |
| 2019-03-09 | 1 | 12 | 11 | 0.90 | Pork Chops | 1 lb | lbs |

Rysunek 6.20

Do utworzenia jednej kolumny, która zawiera łączną liczność produktów sprzedawanych na sztuki (`quantity_units`), drugiej, która zawiera sumę ilości produktów sprzedawanych na funty (`quantity_lbs`), oraz trzeciej (`quantity_other`) dla wszelkich innych produktów, które mogą być wystawione w przyszłości, a które będą sprzedawane na inne jednostki (na przykład uncje), wewnątrz wywołania `SUM` wykorzystam instrukcję `CASE`. Wskaże ona, które wartości należy uwzględnić w której kolumnie.

Zacznijmy od spojrzenia na wyniki instrukcji `CASE` bez grupowania i użycia funkcji agregujących. Zauważ, że na rysunku 6.21 instrukcja `CASE` na podstawie `product_qty_type` rozdzieliła wartości z `quantity` na trzy różne kolumny. Oto wartości, które zsumuję w grupach w następnym kroku:

```
SELECT
    cp.market_date,
    cp.vendor_id,
    cp.customer_id,
    cp.product_id,
    CASE WHEN product_qty_type = "unit" THEN quantity ELSE 0 END AS quantity_units,
    CASE WHEN product_qty_type = "lbs" THEN quantity ELSE 0 END AS quantity_lbs,
    CASE WHEN product_qty_type NOT IN ("unit","lbs") THEN quantity ELSE 0 END AS
    ↪quantity_other,
    p.product_qty_type
FROM farmers_market.customer_purchases cp
    INNER JOIN farmers_market.product p
        ON cp.product_id = p.product_id
```

Teraz każde wyrażenie `CASE` możemy umieścić wewnątrz wywołania `SUM`, aby — zgodnie z tym, co zapisano w klauzuli `GROUP BY` — dodać te wartości dla każdej daty i każdego klienta. Wyniki przedstawiono na rysunku 6.22. (Poprzedni zrzut ekranu zawierał tylko podzbiór pełnych wyników. W wierszach na rysunku 6.22 mogą być widoczne wartości, których nie ma na rysunku 6.21).

| market_date | vendor_id | customer_id | product_id | quantity_units | quantity_lbs | quantity_other | product_qty_type |
|---|---|---|---|---|---|---|---|
| 2019-03-02 | 8 | 4 | 4 | 2.00 | 0 | 0 | unit |
| 2019-03-02 | 8 | 10 | 4 | 1.00 | 0 | 0 | unit |
| 2019-03-09 | 8 | 12 | 4 | 1.00 | 0 | 0 | unit |
| 2019-03-13 | 8 | 2 | 4 | 2.00 | 0 | 0 | unit |
| 2019-03-16 | 8 | 10 | 4 | 1.00 | 0 | 0 | unit |
| 2019-03-02 | 4 | 2 | 9 | 0 | 4.60 | 0 | lbs |
| 2019-03-02 | 4 | 3 | 9 | 0 | 8.40 | 0 | lbs |
| 2019-03-02 | 4 | 4 | 9 | 0 | 1.40 | 0 | lbs |
| 2019-03-09 | 4 | 4 | 9 | 0 | 9.90 | 0 | lbs |
| 2019-03-13 | 4 | 2 | 9 | 0 | 4.10 | 0 | lbs |
| 2019-03-16 | 4 | 3 | 9 | 0 | 5.50 | 0 | lbs |
| 2019-03-02 | 1 | 1 | 10 | 1.00 | 0 | 0 | unit |
| 2019-03-02 | 1 | 1 | 10 | 3.00 | 0 | 0 | unit |
| 2019-03-09 | 1 | 1 | 10 | 2.00 | 0 | 0 | unit |
| 2019-03-09 | 1 | 4 | 10 | 1.00 | 0 | 0 | unit |
| 2019-03-02 | 1 | 1 | 11 | 0 | 1.70 | 0 | lbs |
| 2019-03-09 | 1 | 12 | 11 | 0 | 0.90 | 0 | lbs |

Rysunek 6.21

```
SELECT
    cp.market_date,
    cp.customer_id,
    SUM(CASE WHEN product_qty_type = "unit" THEN quantity ELSE 0 END) AS
    ↪qty_units_purchased,
    SUM(CASE WHEN product_qty_type = "lbs" THEN quantity ELSE 0 END) AS
    ↪qty_lbs_purchased,
    SUM(CASE WHEN product_qty_type NOT IN ("unit","lbs") THEN quantity ELSE 0 END)
    ↪AS qty_other_purchased
FROM farmers_market.customer_purchases cp
    INNER JOIN farmers_market.product p
        ON cp.product_id = p.product_id
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
```

| market_date | customer_id | qty_units_purchased | qty_lbs_purchased | qty_other_purchased |
|---|---|---|---|---|
| 2019-03-02 | 1 | 4.00 | 1.70 | 0.00 |
| 2019-03-02 | 2 | 0.00 | 4.60 | 0.00 |
| 2019-03-02 | 3 | 0.00 | 8.40 | 0.00 |
| 2019-03-02 | 4 | 2.00 | 1.40 | 0.00 |
| 2019-03-02 | 10 | 1.00 | 0.00 | 0.00 |
| 2019-03-09 | 1 | 2.00 | 2.20 | 0.00 |
| 2019-03-09 | 4 | 3.00 | 10.20 | 0.00 |
| 2019-03-09 | 7 | 2.00 | 0.00 | 0.00 |
| 2019-03-09 | 12 | 3.00 | 0.90 | 0.00 |
| 2019-03-13 | 2 | 2.00 | 4.10 | 0.00 |
| 2019-03-16 | 3 | 0.00 | 5.50 | 0.00 |
| 2019-03-16 | 10 | 1.00 | 0.00 | 0.00 |
| 2019-03-20 | 1 | 0.00 | 3.10 | 0.00 |
| 2019-03-20 | 7 | 3.00 | 0.00 | 0.00 |
| 2019-03-23 | 4 | 3.00 | 2.40 | 0.00 |

Rysunek 6.22

W tym rozdziale widziałeś przykłady użycia funkcji agregujących `COUNT`, `COUNT DISTINCT`, `SUM`, `AVG`, `MIN` i `MAX`, a także przykład użycia instrukcji `CASE`, obliczeń wewnątrz wywołania funkcji oraz obliczeń wykonywanych z użyciem podsumowanych wartości. Mam nadzieję, że już zaczynasz się zastanawiać, jak wykorzystać zdobyte umiejętności we własnej pracy!

# Ćwiczenia

1. Napisz zapytanie, które określi, ile razy każdy sprzedawca wynajmował stoisko na targu. Innymi słowy, zlicz przypisania stoisk dla poszczególnych `vendor_id`.
2. W rozdziale 5., „Złączenia w SQL-u", w ćwiczeniu 3. zapytałam: „Kiedy mamy sezon na każdy rodzaj lokalnych świeżych owoców lub warzyw?". Stwórz zapytanie, które wyświetla nazwę kategorii produktów, nazwę produktu oraz pierwszy i ostatni dzień, w którym były dostępne poszczególne produkty z kategorii `Fresh Fruits & Vegetables`.
3. Komitet Doceniania Klientów Targowiska chce wręczyć naklejkę na zderzak każdemu, kto kiedykolwiek wydał na rynku więcej niż 50 dolarów. Stwórz zapytanie generujące listę klientów, którzy powinni otrzymać naklejki. Lista ma być posortowana według nazwiska, a następnie imienia. (WSKAZÓWKA: to zapytanie wymaga połączenia dwóch tabel, użycia funkcji agregującej i słowa kluczowego `HAVING`).

## SQL: tak przygotujesz swój zbiór danych do analizy!

Język SQL zwykle służy do pracy z bazami danych. Poprawnie napisany kod SQL przetwarza z dużą szybkością potężne zbiory informacji, dlatego stanowi wymarzone narzędzie dla analityków danych. Tymczasem wielu z nich zleca wykonywanie raportów z baz czy hurtowni danych innym osobom. Taki sposób pracy okazuje się nieefektywny — o wiele lepszym rozwiązaniem jest opanowanie języka SQL i samodzielne projektowanie oraz wyodrębnianie potrzebnych zbiorów danych.

Ten przystępny przewodnik jest przeznaczony dla analityków danych, którzy chcą dobrze poznać proces tworzenia analitycznego zbioru danych i samodzielnie pisać kod niezbędny do uzyskania zamierzonego wyniku. Przedstawiono tu składnię języka SQL oraz zasady budowania szybko działających zapytań do dużych zbiorów danych. Dokładnie wyjaśniono reguły stosowania poszczególnych instrukcji SQL, korzystania z funkcji agregujących i okienkowych, a także techniki eksploracyjnej analizy danych oraz tworzenia zbiorów danych na potrzeby raportów analitycznych. Omówiono również trudniejsze zagadnienia, takie jak zaawansowane zapytania SQL czy tworzenie zbiorów danych na potrzeby uczenia maszynowego. W książce znalazły się też fachowe porady na temat wnioskowania na podstawie danych i liczne ćwiczenia ułatwiające naukę.

Najciekawsze zagadnienia:

- składnia SQL i projektowanie efektywnych zapytań
- eksploracyjna analiza danych
- tworzenie zbiorów danych z istniejących baz danych
- projektowanie zbiorów danych na potrzeby uczenia maszynowego
- zaawansowane elementy języka SQL
- tworzenie tabel i widoków do przechowywania wyników zapytań

**Renée M.P. Teate** od 2004 roku zajmuje się pracą na danych. Angażowała się w projektowanie relacyjnych baz danych, raportowanie i analizę oraz naukę o danych. Regularnie występuje na konferencjach i publikuje w czasopismach branżowych. Jest również twórczynią podcastu *Becoming a Data Scientist*.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-9744-6
9 788328 397446
Cena: 69,00 zł