Bazy danych i PostgreSQL. Od podstaw

Autorzy: Richard Stones, Neil Matthew
Tłumaczenie: Radosław Meryk
ISBN: 83-7197-650-X
Tytuł oryginału: Beginning Databases with PostgreSQL
Format: B5, stron: 610

PostgreSQL wciąż zyskuje na popularności i jest uważany za najlepszy darmowy system zarządzania relacyjnymi bazami danych. Początkowo był tworzony w środowisku uniwersyteckim, potem – jako otwarty projekt internetowy – przez utalentowanych programistów z całego świata. Coraz częściej duże firmy decydują się na wybór PostgreSQL jako systemu zarządzania bazami danych.

Niniejsza książka jest kompletnym podręcznikiem opisującym cechy systemu PostgreSQL. Zawiera opis najprostszych metod instalacji i zarządzania systemem, tworzenia własnych baz danych, jak również omówienie integracji baz danych z aplikacjami napisanymi w najpopularniejszych językach programowania wykorzystywanych w Internecie. Czytelnik znajdzie w niej wskazówki dotyczące tworzenia coraz bardziej wyrafinowanych zapytań języka SQL, łączenia tabel, wykorzystywania transakcji, monitorowania pracy serwera, tworzenia własnych aplikacji w językach wysokiego poziomu i wiele innych.

Dla kogo jest ta książka?

Książka jest adresowana do Czytelników rozpoczynających swoją przygodę z relacyjnymi bazami danych (nie jest wymagana wiedza z zakresu języków SQL, PHP, Java czy Perl). Opisano w niej zarówno najprostsze zapytania, jak i coraz bardziej skomplikowane metody zarządzania bazami danych, które umożliwią im rozwiązywanie codziennych problemów administratorów baz danych. Dzięki lekturze można nauczyć się zarządzania bazą danych PostgreSQL w środowiskach Windows i Unix.

Co zawiera książka?

- Wprowadzenie do systemu PostreSQL
- Opis instalacji z pakietów binarnych i kodów źródłowych w środowiskach Windows i Unix
- Opis narzędzi graficznych
- Przykłady zapytań obejmujących zapytania złożone, funkcje agregujące i inne
- Przedstawienie transakcji, poziomów izolacji, procedur składowanych i wyzwalaczy
- Monitorowanie wydajności i kontrolowanie pracy serwera
- Łączenie z bazą danych i wykonywanie instrukcji SQL z poziomu języka C (libpq)
- Tworzenie aplikacji przy użyciu języków PHP, Perl i Java

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

7

Zaawansowane wyszukiwanie danych

W rozdziale 4. omówiliśmy szczegółowo instrukcję SELECT oraz sposób jej użycia do wyszukiwania danych. Omawiane tam zagadnienia obejmowały wybieranie kolumn i wierszy oraz łączenie tabel. W poprzednim rozdziale zaprezentowaliśmy sposoby dodawania, aktualizacji i usuwania danych. W obecnym powrócimy do instrukcji SELECT i omówimy jej bardziej zaawansowane cechy. Z niektórych spośród tych możliwości będziemy korzystać bardzo rzadko, ale warto je znać, aby zdawać sobie sprawę z możliwości języka SQL.

Czytelnicy, którzy czytając tę książkę wypróbowują przykłady, zauważą, że zarówno w tym rozdziale, jak i w innych, zawsze rozpoczynamy od czystych danych wyjściowych w przykładowej bazie danych. Dzięki temu czytelnicy mogą zgłębiać te rozdziały, na które mają ochotę. Oznacza to, że w przypadku, gdy będziemy korzystać z przykładowych danych z poprzednich rozdziałów, niektóre wyniki mogą być nieco różne od pokazanych. Skrypty, które można pobrać ze strony WWW książki pozwalają na łatwe usunięcie tabel, ponowne ich utworzenie oraz ponowne wypełnienie ich danymi.

W tym rozdziale omówimy najpierw pewne specjalne funkcje, nazywane funkcjami agregacji. Pozwalają one na uzyskiwanie wyników na podstawie grupy wierszy. Następnie omówimy nieco bardziej zaawansowane operacje powiązań, pozwalające na kontrolowanie wyników. W tych przypadkach powiązania między tabelami nie będą tak proste jak te, których używaliśmy poprzednio. Poznamy także całą nową grupę zapytań nazywanych zapytaniami podrzędnymi (ang. *subquery*), w których w pojedynczym zapytaniu używamy kilku instrukcji SELECT. Wreszcie poznamy bardzo ważną operację OUTER JOIN, która pozwala na łączenie tabel w sposób bardziej elastyczny niż ten, z którym spotkaliśmy się do tej pory.

W rozdziale omówimy następujące funkcje:

- funkcje agregacji;
- powiązania typu UNION;

- zapytania podrzędne;
- powiązania same z sobą;
- powiązania zewnętrzne.

Funkcje agregacji

W poprzednich rozdziałach korzystaliśmy z kilku specjalnych funkcji: `MAX (nazwa_kolumny)` — w celu uzyskania największej wartości w kolumnie i `COUNT(*)` — aby uzyskać liczbę wierszy w tabeli. Funkcje te należą do niewielkiej grupy funkcji SQL, nazywanej funkcjami agregacji.

Do grupy tej należą:

- `COUNT(*)`,
- `COUNT(nazwa_kolumny)`,
- `MIN(nazwa_kolumny)`,
- `MAX(nazwa_kolumny)`,
- `SUM(nazwa_kolumny)`,
- `AVG (nazwa_kolumny)`.

Zazwyczaj są to funkcje bardzo przydatne i łatwe do stosowania.

> Polecenie `psql \da` wyświetla listę wszystkich funkcji agregacji w PostgreSQL.

Polecenie `psql \da` wyświetla listę wszystkich funkcji agregacji w PostgreSQL.

COUNT

Rozpoczniemy od funkcji `COUNT`, która — jak widać z przedstawionej powyżej listy — ma dwie postacie. Funkcja `COUNT(*)` oblicza liczbę wierszy w tabeli. Pełni ona rolę specjalnej nazwy kolumny w instrukcji `SELECT`. W tych instrukcjach `SELECT`, które wykorzystują dowolne z funkcji agregacji, można stosować dwie opcjonalne klauzule: `GROUP BY` oraz `HAVING`. Składnia jest wówczas następująca:

```
SELECT COUNT(*) lista_kolumn FROM nazwa_tabeli WHERE warunek
[GROUP BY nazwa_kolumny [HAVING warunki agregacji]]
```

Nowa, opcjonalna klauzula `GROUP BY` jest dodatkowym warunkiem, z którego można korzystać w instrukcjach `SELECT`. Zazwyczaj korzysta się z niej w przypadku stosowania funkcji agregacji. Można ją także wykorzystać jako funkcję podobną do `ORDER BY`, ale działającą z kolumną, na podstawie której została wyliczona wartość funkcji agregacji. Opcjonalna klauzula `HAVING` pozwala na wybór określonych wierszy dla pewnych warunków funkcji `COUNT(*)`, jeżeli wykorzystaliśmy już klauzulę `GROUP BY`.

Brzmi to bardzo tajemniczo, ale w praktyce jest dosyć proste. Spróbujmy skorzystać z bardzo prostej instrukcji `COUNT(*)`, aby mieć obraz jej działania. Wkrótce poznamy także działanie klauzuli `GROUP BY`.

Wypróbuj to — proste zastosowanie funkcji COUNT(*)

Przypuśćmy, że chcemy wiedzieć, ilu klientów z tabeli `customer` mieszka w mieście Bingham. Moglibyśmy oczywiście zapisać proste zapytanie SQL w postaci:

```
SELECT * FROM customer WHERE town = 'Bingham';
```

lub, w bardziej wydajny sposób, zapisać instrukcję SQL, która daje w wyniku mniejszą ilość danych:

```
SELECT customer_id FROM customer WHERE town='Bingham';
```

Uzyskamy wprawdzie oczekiwany efekt, ale dość okrężną drogą. Operacje te wymagają wyszukiwania dużej ilości danych, które w zasadzie nie są nam potrzebne. Przypuśćmy, że w tabeli `customer` znajdują się dane wielu tysięcy klientów, z których ponad tysiąc mieszka w Bingham. W takim przypadku wybralibyśmy mnóstwo danych, które nie są nam potrzebne. Funkcja `COUNT(*)` rozwiązuje ten problem, pozwalając na wybranie zaledwie jednego wiersza, który zawiera liczbę wierszy spełniających warunek. Zapiszemy naszą instrukcję `SELECT` w zwykły sposób, ale zamiast wpisywać rzeczywiste kolumny, wykorzystamy funkcję `COUNT(*)`:

```
bpsimple=# SELECT COUNT(*) FROM customer WHERE town = 'Bingham';
 count
--------
      3
(1 row)

bpsimple=#
```

Gdybyśmy chcieli policzyć wszystkich klientów, moglibyśmy po prostu pominąć klauzulę `WHERE`:

```
bpsimple=# SELECT COUNT(*) FROM customer;
  count
--------
    15
bpsimple=#
```

Zauważmy, że otrzymaliśmy pojedynczy wiersz zawierający liczbę wierszy. Aby sprawdzić wynik, można po prostu zamienić `COUNT(*)` na `customer_id`, co spowoduje wyświetlenie rzeczywistych danych.

Jak to działa?

Funkcja `COUNT(*)` umożliwia uzyskanie liczby obiektów, a nie obiektów samych w sobie. W większości wypadków jest to znacznie bardziej wydajne niż wybieranie danych, z dwóch powodów:

- nie trzeba pobierać z bazy danych informacji, które nie są nam potrzebne;
- funkcja `COUNT(*)` umożliwia bazie danych korzystanie z jej wewnętrznych informacji bez konieczności przeszukiwania danych.

> Nigdy nie należy wybierać danych, jeżeli potrzebujemy tylko liczby wierszy.

GROUP BY a COUNT(*)

Odpowiedź na zapytanie pokazane w poprzednim podrozdziale nie zawsze spełni nasze oczekiwania. Przypuśćmy, że chcielibyśmy wiedzieć, ilu klientów mieszka w każdym mieście. Moglibyśmy dowiedzieć się tego wybierając listę różnych miast, a następnie wyliczając, ilu klientów mieszka w każdym z nich. Jest to jednak proceduralny i raczej żmudny sposób rozwiązania problemu. Czy nie byłoby lepiej zastosować sposób deklaracyjny, formułując zapytanie bezpośrednio w SQL? Moglibyśmy pokusić się o wypróbowanie następującej instrukcji:

```
SELECT COUNT(*), town FROM customer;
```

Jest to rozsądna próba, zważywszy na to, co wiemy do tej pory, ale PostgreSQL wyświetli komunikat o błędzie, ponieważ nie jest to poprawna składnia SQL. Aby rozwiązać ten problem, potrzebujemy dodatkowej klauzuli `GROUP BY`.

Klauzula `GROUP BY` informuje PostgreSQL, że funkcja agregacji powinna obliczyć wynik i wyzerować się za każdym razem, kiedy określona kolumna lub kolumny zmienią wartość. Jest ona bardzo prosta w użyciu. Wystarczy dodać zapis `GROUP BY nazwa_kolumny` do instrukcji `SELECT` z funkcją `COUNT (*)`, a PostgreSQL poinformuje nas, ile wierszy o określonej wartości kolumny znajduje się w tabeli.

Wypróbuj to — GROUP BY

Spróbujmy odpowiedzieć na pytanie, ilu klientów mieszka w każdym mieście?

Etap pierwszy polega na napisaniu instrukcji `SELECT`, która zawiera funkcję `COUNT(*)` oraz nazwę kolumny, dokładnie tak, jak próbowaliśmy odgadnąć:

```
SELECT COUNT(*), town FROM customer;
```

Następnie należy dodać klauzulę `GROUP BY`, aby poinformować PostgreSQL, że powinien obliczyć wynik i wyzerować licznik za każdym razem, kiedy zmieni się miasto. Można to zrobić za pomocą następującego zapytania:

```
SELECT COUNT(*), town FROM customer GROUP BY town;
```

A oto nasza instrukcja w działaniu:

```
bpsimple=# SELECT COUNT(*), town FROM customer GROUP BY town;
 count |   town
-------+-----------
     3 | Bingham
```

```
     1 | Hightown
     1 | Histon
     1 | Lowtown
     1 | Milltown
     2 | Nicetown
     1 | Oahenham
     1 | Oxbridge
     1 | Tibsville
     1 | Welltown
     1 | Winersby
     1 | Yuleville
(12 rows)

bpsimple=#
```

Jak widzimy, uzyskaliśmy przejrzystą listę miast wraz z liczbą klientów w każdym z nich.

Jak to działa?

PostgreSQL porządkuje wynik według kolumn wymienionych w klauzuli `GROUP BY`, następnie oblicza liczbę wierszy i za każdym razem, kiedy zmienia się miasto, zapisuje wiersz wyniku, po czym zeruje licznik. Zgodzimy się, że jest to o wiele łatwiejsze niż pisanie kodu procedury z pętlą dla każdego miasta.

Jeżeli zachodzi taka potrzeba, sposób ten można zastosować do więcej niż jednej kolumny, pod warunkiem, że wszystkie kolumny, które wybrano, są także wymienione w klauzuli `GROUP BY`. Przypuśćmy, że interesowały nas dwie informacje. Po pierwsze, ilu klientów mieszka w każdym mieście, po drugie, ile różnych nazwisk mają ci klienci. Możemy po prostu dodać kolumnę `lname` zarówno do części `SELECT`, jak `GROUP BY`:

```
bpsimple=# SELECT COUNT(*), lname, town FROM customer GROUP BY town, lname;
 count |  lname  |   town
-------+---------+-----------
     1 | Jones   | Bingham
     2 | Stones  | Bingham
     1 | Stones  | Hightown
     1 | Hickman | Histon
     1 | Stones  | Lowtown
     1 | Hudson  | Milltown
     2 | Matthew | Nicetown
     1 | Cozens  | Oahenham
     1 | Hendy   | Oxbridge
     1 | Howard  | Tibsville
     1 | O'Neill | Welltown
     1 | Neill   | Winersby
     1 | Matthew | Yuleville
(13 rows)

bpsimple=#
```

Zwróćmy uwagę, że wynik jest posortowany najpierw według miasta, a następnie według nazwiska, ponieważ w tej kolejności (`town, lname`) wymieniono kolumny w klauzuli `GROUP BY` oraz na to, że obecnie Bingham znajduje się na liście dwa razy, ponieważ mieszkają w nim klienci o dwóch różnych nazwiskach, `Jones` oraz `Stones`.

HAVING a COUNT(*)

Ostatnią opcjonalną częścią instrukcji jest klauzula HAVING. Często jest ona myląca dla początkujących programistów piszących w języku SQL, ale w rzeczywistości nie jest trudna w użyciu. Należy po prostu pamiętać, że HAVING jest rodzajem klauzuli WHERE dla funkcji agregacji. Klauzulę HAVING wykorzystujemy w celu ograniczenia liczby zwracanych wierszy, gdzie

> Użycie funkcji agregacji w klauzuli WHERE nie jest poprawne. Są one poprawne tylko wewnątrz klauzuli HAVING.

określona funkcja agregacji, na przykład COUNT(*), ma wartość TRUE. Używamy jej w dokładnie taki sam sposób, jak klauzuli WHERE, w celu ograniczenia liczby wierszy na podstawie wartości kolumny.

Spójrzmy na przykład, który powinien sprawić, że zagadnienie to stanie się łatwe i przyjemne. Przypuśćmy, że chcemy znać wszystkie miasta, w których mamy więcej niż jednego klienta. Moglibyśmy wówczas użyć funkcji COUNT(*), a następnie odszukać na liście interesujące nas miasta. Nie jest to jednak rozwiązanie sensowne w sytuacji, gdy mamy tysiące miast. Zamiast tego skorzystamy z klauzuli HAVING w celu ograniczenia wyniku do tych wierszy, gdzie wartość funkcji COUNT(*) jest większa niż jeden. Zrobimy to w następujący sposób:

```
bpsimple=# SELECT COUNT(*), town FROM customer
bpsimple-# GROUP BY town HAVING COUNT(*)>1;
 count |   town
-------+----------
     3 | Bingham
     2 | Nicetown
(2 rows)

bpsimple=#
```

Zauważmy, że w dalszym ciągu musimy zapisać klauzulę GROUP BY i musi ona występować przed klauzulą HAVING. Teraz, kiedy znamy podstawy funkcji COUNT(*), klauzuli GROUP BY oraz HAVING, spróbujmy wykorzystać je wszystkie w większym przykładzie.

Wypróbuj to — HAVING

Przypuśćmy, że myślimy o opracowaniu harmonogramu dostaw i chcemy znać nazwiska wszystkich klientów i miasta, skąd pochodzą, oprócz klientów z Lincoln (być może jest to nasze miasto). Interesują nas tylko te miasta, w których mamy więcej niż jednego klienta.

Nie jest to takie trudne, jak mogłoby się wydawać; musimy po prostu stopniowo stworzyć nasze rozwiązanie. Jest to zazwyczaj dobre podejście dla instrukcji SQL. Jeżeli coś wygląda na zbyt trudne, należy rozpocząć od rozwiązania czegoś prostszego, ale podobnego, a następnie rozszerzyć rozwiązanie do momentu, aż uda się rozwiązać problem bardziej skomplikowany. Tak więc należy rozpoznać problem, podzielić go na mniejsze części, a następnie rozwiązać każdą z nich.

Zacznijmy od wybrania danych, a nie ich liczenia. Dane uporządkujemy według miasta, aby można było łatwiej zorientować się, o co chodzi:

```
bpsimple=# SELECT lname, town FROM customer WHERE town <> 'Lincoln';
  lname  |   town
---------+-----------
 Stones  | Hightown
 Stones  | Lowtown
 Matthew | Nicetown
 Matthew | Yuleville
 Cozens  | Oahenham
 Matthew | Nicetown
 Stones  | Bingham
 Stones  | Bingham
 Hickman | Histon
 Howard  | Tibsville
 Jones   | Bingham
 Neill   | Winersby
 Hendy   | Oxbridge
 O'Neill | Welltown
 Hudson  | Milltown
(15 rows)

bpsimple=#
```

Do tej pory wygląda nieźle, nieprawda?

Teraz, aby skorzystać z funkcji `COUNT(*)` w celu wykonania obliczeń, musimy także umieścić klauzulę `GROUP BY` i pogrupować kolumny według nazwiska i miasta (`lname, town`):

```
bpsimple=# SELECT COUNT(*), lname town FROM customer WHERE town <>
bpsimple-# 'Lincoln' GROUP BY lname, town;
 count |  lname  |   town
-------+---------+-----------
     1 | Cozens  | Oahenham
     1 | Hendy   | Oxbridge
     1 | Hickman | Histon
     1 | Howard  | Tibsville
     1 | Hudson  | Milltown
     1 | Jones   | Bingham
     2 | Matthew | Nicetown
     1 | Matthew | Yuleville
     1 | Neill   | Winersby
     1 | O'Neill | Welltown
     2 | Stones  | Bingham
     1 | Stones  | Hightown
     1 | Stones  | Lowtown
(13 rows)

bpsimple=#
```

Możemy teraz odnaleźć odpowiedź samodzielnie przeglądając dane, ale jesteśmy zaledwie o krok od właściwego wyniku. Wystarczy dodać klauzulę `HAVING`, a zostaną wybrane wiersze, gdzie wartość funkcji `COUNT(*)` jest większa niż 1:

```
bpsimple=# SELECT COUNT(*), lname town FROM customer WHERE town <>
bpsimple-# 'Lincoln' GROUP BY lname, town HAVING COUNT(*)>1;
 count |  lname  |   town
-------+---------+----------
     2 | Matthew | Nicetown
     2 | Stones  | Bingham
(2 rows)

bpsimple=#
```

Dosyć proste, jeżeli podzielimy problem na mniejsze części.

Jak to działa?

Rozwiązaliśmy nasz problem w trzech etapach:

- w celu wyszukania wszystkich interesujących nas wierszy zapisaliśmy prostą instrukcję `SELECT`;
- następnie dodaliśmy słowa kluczowe `COUNT(*)` oraz `GROUP BY` w celu obliczenia liczby unikalnych kombinacji nazwiska i miasta (`lname` oraz `town`);
- wreszcie dodaliśmy klauzulę `HAVING`, aby wyszukać tylko te wiersze, w których wartość `COUNT(*)` była większa niż 1.

Przy takim podejściu istnieje jednak pewien problem. Gdybyśmy mieli do czynienia z bazą danych klientów, która zawierałaby tysiące wierszy, lista klientów przewijałaby się nam bardzo długo w czasie opracowywania naszego zapytania. Dla naszej prostej bazy danych nie stanowiło to problemu, ale dla dużej bazy danych takie interaktywne podejście w opracowywaniu zapytania ma pewne wady. Na szczęście zazwyczaj łatwo opracować zapytania na próbce danych, z wykorzystaniem klucza podstawowego. Gdybyśmy do wszystkich zapytań dodali warunek `WHERE customer_id<50`, pracowalibyśmy z próbką pierwszych 50 klientów w bazie.

Kiedy już będziemy pewni, że zapytanie jest poprawne, możemy po prostu usunąć klauzulę `WHERE` i wykonać nasze rozwiązanie dla całej tabeli. Oczywiście musimy być pewni, że zastosowana próbka danych, wykorzystana w celu przetestowania naszej instrukcji SQL, jest reprezentantem całości zbioru danych oraz musimy mieć na uwadze, że mniejsze próbki mogą nie w pełni testować naszą instrukcję SQL.

COUNT(nazwa_kolumny)

Pewną odmianą funkcji `COUNT(*)` jest użycie nazwy kolumny zamiast znaku `'*'`. Różnica polega na tym, że funkcja `COUNT(nazwa_kolumny)` oblicza ilość wierszy w tabeli, w których określona kolumna ma wartość różną od `NULL`.

Przypuśćmy, że do naszej kolumny `customer` dodaliśmy pewne dane o nowych klientach, zaś numery telefonów zawarte w tych danych mają wartość `NULL`:

```
INSERT INTO customer(title, fname, lname, addressline, town, zipcode)
VALUES('Mr', 'Gavin', 'Smyth', '23 Harlestone', 'Milltown', 'MT7 7HI');
```

```
INSERT INTO customer(title, fname, lname, addressline, town, zipcode, phone)
VALUES('Mrs', 'Sarah', 'Harvey', '84 Willow Way', 'Lincoln', 'LC3 7RD','527 3739');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode)
VALUES('Mr', 'Steve', 'Harvey', '84 Willow Way', 'Lincoln', 'LC3 7RD');
INSERT INTO customer(title, fname, lname, addressline, town, zipcode)
VALUES('Mr', 'Paul', 'Garrett', '27 Chase Avenue', 'Lowtown', 'LT5 8TQ');
```

Sprawdźmy, ilu jest klientów, dla których nie znamy numerów telefonów:

```
bpsimple=# SELECT customer_id FROM customer WHERE phone IS NULL;
 customer_id
-------------
          16
          18
          19
(3 rows)

bpsimple=#
```

Widzimy, że mamy trzech takich klientów. Sprawdźmy teraz, ilu wszystkich klientów jest w bazie danych:

```
bpsimple=# SELECT COUNT(*) FROM customer;
 count
-------
    19
(1 row)

bpsimple=#
```

Mamy w sumie 19 klientów. Tak więc, jeżeli obliczymy liczbę klientów, których wartość numeru telefonu jest różna od `NULL`, prawdopodobnie otrzymamy 16:

```
bpsimple=# SELECT COUNT(phone) FROM customer;
 count
-------
    16
(1 row)

bpsimple=#
```

Jest to jedyna różnica pomiędzy `COUNT(*)` a `COUNT(nazwa_kolumny)`. Odmiana funkcji z nazwą kolumny oblicza wiersze, w których kolumna o określonej nazwie ma wartość różną niż `NULL`, postać funkcji ze znakiem `'*'` oblicza zaś liczbę wszystkich wierszy. Pod każdym innym względem, jak choćby w przypadku użycia `GROUP BY` oraz `HAVING`, `COUNT(nazwa_kolumny)` działa dokładnie tak samo jak `COUNT(*)`.

Funkcja MIN()

Teraz, kiedy znamy funkcję `COUNT(*)` i poznaliśmy zasady obowiązujące dla funkcji agregacji, możemy zastosować tę samą logikę do wszystkich pozostałych funkcji agregacji.

Jak można się spodziewać, funkcja MIN() ma parametr w postaci nazwy kolumny i zwraca najmniejszą wartość znalezioną w tej kolumnie. Dla kolumn o typach numerycznych osiągniemy spodziewane wyniki. Dla typów opisujących czas, na przykład daty, uzyskamy największą datę, która może dotyczyć przeszłości bądź przyszłości. Dla ciągów znaków o zmiennej długości wyniki są nieco zaskakujące, ponieważ ciągi są porównywane po uprzednim uzupełnieniu ich spacjami z prawej strony. Funkcje MIN() oraz MAX () w odniesieniu do kolumn typu VARCHAR należy stosować ostrożnie, gdyż w ich przypadku wyniki mogą być różne od oczekiwanych.

Oto kilka przykładów.

Poszukajmy najmniejszej opłaty transportowej, jaką nałożyliśmy na zamówienie:

```
bpsimple=# SELECT MIN(shipping) FROM orderinfo;
 min
--------
  0.00
(1 row)

bpsimple=#
```

W rzeczywistości wartość ta wynosiła zero. Zwróćmy uwagę, co się stanie, jeżeli zastosujemy tę samą funkcję z kolumną phone, o której wiemy, że istnieją w niej wartości NULL:

```
bpsimple=# SELECT MIN(phone) FROM customer;
 min
-------
010 4567
(1 row)

bpsimple=#
```

Moglibyśmy się spodziewać, że wynik będzie teraz miał wartość NULL lub będzie pustym ciągiem znaków. Biorąc jednak pod uwagę, że NULL z reguły oznacza „nieznany", funkcja MIN() ignoruje tę wartość. Ignorowanie wartości NULL jest cechą wszystkich funkcji agregacji oprócz COUNT(*). To, czy posiadanie informacji o najmniejszym numerze telefonu ma jakiekolwiek znaczenie, jest oczywiście odrębną kwestią.

Funkcja MAX()

Nie jest zaskoczeniem, że funkcja MAX() jest podobna do MIN(), ale działa w odwrotnym kierunku.

Jak można się spodziewać, MAX() jako parametr przyjmuje nazwę kolumny i zwraca maksymalną wartość znalezioną w tej kolumnie.

Oto kilka przykładów.

Poszukajmy największej opłaty transportowej, jaką nałożyliśmy na zamówienie:

```
bpsimple=# SELECT MAX(shipping) FROM orderinfo;
 max
--------
```

```
  3.99
(1 row)

bpsimple=#
```

Tak jak w przypadku funkcji MIN(), wartości NULL są ignorowane:

```
bpsimple=# SELECT MAX(phone) FROM customer;
 max
-------
961 4526
(1 row)

bpsimple=#
```

To w zasadzie wszystko, co należy wiedzieć o funkcji MAX(), z wyjątkiem tego, że można z nią używać klauzul GROUP BY oraz HAVING, dokładnie tak, jak w przypadku funkcji COUNT(*).

Funkcja SUM()

Funkcja SUM() przyjmuje jako parametr nazwę kolumny o typie numerycznym i zwraca sumę wartości dla tej kolumny. Tak jak funkcje MIN() oraz MAX(), funkcja SUM() ignoruje wartości typu NULL:

```
bpsimple=# SELECT SUM(shipping) FROM orderinfo;
 sum
-------
 9.97
(1 row)

bpsimple=#
```

Funkcja SUM() posiada interesującą odmianę. Możemy mianowicie zażądać, aby dodano tylko wartości unikalne, tak, aby kilka wierszy o tych samych wartościach zostało uwzględnione w sumie tylko raz:

```
bpsimple=# SELECT SUM(DISTINCT shipping) FROM orderinfo;
 sum
-------
  6.98
(1 row)

bpsimple=#
```

Zwróćmy uwagę, że rzeczywiste zastosowanie tej odmiany jest dość mgliste.

Funkcja AVG()

Ostatnią opisaną funkcją agregacji jest AVG(), jako parametr przyjmująca również nazwę kolumny i zwracająca wartość średnią. Podobnie jak funkcja SUM(), także AVG() ignoruje

wartości NULL i może być użyta ze słowem kluczowym DISTINCT, aby działała tylko z różnymi wartościami:

```
bpsimple=# SELECT AVG(shipping) FROM orderinfo;
   avg
-------
1.9940000000
(1 row)

bpsimple=#
```

Odmiana ze słowem kluczowym DISTINCT wygląda następująco:

```
bpsimple=# SELECT AVG(DISTINCT shipping) FROM orderinfo;
 avg
-------
2.3266666667
(1 row)

bpsimple=#
```

Zwróćmy uwagę, że w implementacjach standardowego SQL oraz PostgreSQL nie ma funkcji MODE oraz MEDIAN. Niektórzy komercyjni producenci dołączają jednak obsługę tych funkcji jako rozszerzenia.

Powiązania typu UNION

Omówimy teraz sposób połączenia kilku instrukcji SELECT dla zapewnienia większych możliwości wyszukiwania.

Pamiętamy z poprzedniego rozdziału tabelę tcust, którą wykorzystaliśmy jako tabelę tymczasową w celu ładowania danych do tabeli customer. Przypuśćmy, że w okresie czasu pomiędzy ładowaniem informacji o nowych klientach do tabeli tcust a wyczyszczeniem tabeli i załadowaniem danych do rzeczywistej tabeli customer, pojawiło się pytanie o listę wszystkich miast, w których mamy klientów, z uwzględnieniem nowych informacji. Możemy słusznie zauważyć, że — ponieważ jeszcze nie załadowaliśmy danych o klientach do głównej tabeli i nie wyczyściliśmy danych — nie możemy być pewni dokładności nowych danych. Tak więc żadna lista miast łącząca obie listy również nie będzie dokładna. Czasami jednak nie jest to ważne. Być może potrzebna była tylko ogólna informacja o geograficznym rozproszeniu klientów, a nie dokładne dane.

Możemy rozwiązać ten problem poprzez wybranie miasta (town) z tabeli customer i zapisanie wyniku, a następnie wybranie miasta (town) z tabeli tcust, ponowne zapisanie oraz połączenie obu list. Wygląda to raczej nieelegancko, ponieważ mamy dwie tabele, obie zawierające listę miast.

Czy nie ma sposobu, aby inaczej połączyć listy? Jak można wywnioskować z tytułu tego podrozdziału, istnieje taki sposób i nazywa się powiązaniem typu UNION. Powiązania te nie są zbyt popularne, ale w pewnych okolicznościach są właściwym środkiem do rozwiązania problemu i są ponadto bardzo łatwe w użyciu.

Spróbujmy ponownie umieścić pewne dane w tabeli tcust, aby miała następującą zawartość:

```
bpsimple=# SELECT * FROM tcust;
 title |  fname  |  lname  |  addressline   |   town   |  zipcode   |  phone
-------+---------+---------+----------------+----------+------------+----------
 Mr    | Peter   | Bradley | 72 Milton Rise | Keynes   | MK41 2HQ   |
 Mr    | Kevin   | Carney  | 43 Glen Way    | Lincoln  | LI2 7RD    | 786 3454
 Mr    | Brian   | Waters  | 21 Troon Rise  | Lincoln  | LI7 6GT    | 786 7245
 Mr    | Malcolm | Whalley | 3 Craddock Way | Welltown | WT3 4GQ    | 435 6543
(4 rows)

bpsimple=#
```

Porównajmy listę miast z tej tabeli z listą z tabeli customer.

Wiemy już, w jaki sposób wybrać miasto (town) z każdej z tabel. Musimy użyć bardzo prostej pary instrukcji SELECT następującej postaci:

```
SELECT town FROM tcust;
SELECT town FROM customer;
```

Każda z tych instrukcji daje w wyniku listę miast. Aby je połączyć, zastosujemy operator UNION:

```
SELECT town FROM tcust UNION SELECT town FROM customer;
```

Wprowadziliśmy naszą instrukcję SQL dla ułatwienia czytania w kilku wierszach. Zwróćmy uwagę, że znak zachęty psql zmienia się z =# na -#, dla zaznaczenia, że jest to kontynuacja instrukcji. Ponieważ jest to pojedyncza instrukcja SQL, występuje tylko jeden średnik na końcu:

```
bpsimple=# SELECT town FROM tcust
bpsimple-# UNION
bpsimple-# SELECT town FROM customer;
   town
-----------
 Bingham
 Hightown
 Histon
 Keynes
 Lincoln
 Lowtown
 Milltown
 Nicetown
 Oahenham
 Oxbridge
 Tibsville
 Welltown
 Winersby
 Yuleville
(14 rows)

bpsimple=#
```

Jak to działa?

PostgreSQL stworzył listę miast z obu tabel i połączył je w pojedynczą listę. Zauważmy, że wszystkie duplikaty zostały usunięte. Gdybyśmy chcieli uzyskać listę wszystkich miast włącznie z duplikatami, zapisalibyśmy `UNION ALL` zamiast `UNION`.

Właściwość łączenia instrukcji `SELECT` nie ogranicza się do pojedynczej kolumny, moglibyśmy połączyć także listę miast i kodów pocztowych:

```
SELECT town, zipcode FROM tcust UNION SELECT town, zipcode FROM customer;
```

Spowodowałoby to utworzenie listy złożonej z obu kolumn. Byłaby to dłuższa lista, ponieważ ze względu na kod pocztowy, istnieje więcej unikalnych wierszy.

Powiązanie typu `UNION` nie potrafi jednak czynić cudów; obie listy kolumn muszą zawierać jednakową ich liczbę, a ponadto wybrane odpowiadające sobie kolumny muszą mieć zgodne typy. Popatrzmy:

```
bpsimple=# SELECT title FROM customer
bpsimple-# UNION
bpsimple-# SELECT town FROM tcust;
  title
----------
 Keynes
 Lincoln
 Miss
 Mr
 Mrs
 Welltown
(6 rows)

bpsimple=#
```

Zapytanie, jakkolwiek raczej pozbawione sensu, jest poprawne, ponieważ PostgreSQL może połączyć kolumny, mimo że kolumna `title` ma stałą długość, natomiast kolumna `town` — zmienną, ponieważ obie kolumny są ciągami znaków. Gdybyśmy dla przykładu spróbowali powiązać pole `customer_id` oraz `town`, PostgreSQL poinformowałby nas, że jest to niemożliwe, ponieważ typy kolumn są różne.

W zasadzie to wszystko, co należy wiedzieć o powiązaniach typu `UNION`, które czasem okazują się bardzo przydatne do łączenia danych z dwóch (lub więcej) tabel.

Zapytania podrzędne

Teraz, kiedy poznaliśmy instrukcje SQL zawierające więcej niż jedną instrukcję `SELECT`, możemy zapoznać się z całą klasą instrukcji wyszukiwania danych, które łączą instrukcje `SELECT` w o wiele bardziej wyszukany sposób. Są one trudniejsze do zrozumienia niż zapytania złożone z pojedynczej instrukcji `SELECT` lub powiązania typu `UNION`, ale są bardzo przydatne i otwierają nowe możliwości tworzenia kryteriów wyboru.

Zapytanie podrzędne ma miejsce wtedy, gdy tworzymy jeden (lub więcej) warunków `WHERE` instrukcji `SELECT` jako odrębną instrukcję `SELECT`.

Przypuśćmy, że chcemy odnaleźć wszystkie towary z tabeli `item`, z ceną zakupu (`cost_price`) większą niż 10. Instrukcja `SELECT` jest w tym przypadku raczej złożona ze względu na konieczność konwersji typu liczby na typ `NUMERIC(7,2)` po to, aby typ liczby był zgodny z typem kolumny `cost_price` w tabeli `item`, ale w zasadzie jest dość oczywista:

```
bpsimple=# SELECT * FROM item WHERE cost_price> CAST(10.0 AS NUMERIC(7,2));
 item_id | description | cost_price | sell_price
---------+-------------+------------+------------
       1 | Wood Puzzle |      15.23 |      21.95
       7 | Fan Large   |      13.36 |      19.95
      11 | Speakers    |      19.73 |      25.32
(3 rows)

bpsimple=#
```

Przypuśćmy, że chcemy znaleźć towary, których cena zakupu jest większa od średniej ceny zakupu. Z łatwością można to zrobić w dwóch zapytaniach:

```
bpsimple=# SELECT AVG(cost_price) FROM item;
     avg
--------------
 7.2490909091
(1 row)

bpsimple=# SELECT * FROM item WHERE cost_price > CAST(7.249 AS
bpsimple-# NUMERIC(7,2));
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       1 | Wood Puzzle   |      15.23 |      21.95
       2 | Rubic Cube    |       7.45 |      11.49
       5 | Picture Frame |       7.54 |       9.95
       6 | Fan Small     |       9.23 |      15.75
       7 | Fan Large     |      13.36 |      19.95
      11 | Speakers      |      19.73 |      25.32
(6 rows)

bpsimple=#
```

Jest to jednak rozwiązanie mało eleganckie. Tak naprawdę chcemy przekazać wynik pierwszego zapytania bezpośrednio do drugiego, bez konieczności pamiętania go i wpisywania do drugiego zapytania.

Jest to właśnie jedna z możliwości, które daje zastosowanie zapytań podrzędnych. Możemy umieścić pierwsze zapytanie w nawiasach i wykorzystać jako część klauzuli `WHERE` w drugim zapytaniu:

```
bpsimple=# SELECT * FROM ITEM WHERE cost_price > (SELECT AVG(cost_price)
bpsimple-# FROM item);
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       1 | Wood Puzzle   |      15.23 |      21.95
       2 | Rubic Cube    |       7.45 |      11.49
```

```
      5 | Picture Frame |       7.54 |       9.95
      6 | Fan Small     |       9.23 |      15.75
      7 | Fan Large     |      13.36 |      19.95
     11 | Speakers      |      19.73 |      25.32
(6 rows)

bpsimple=#
```

Jak widzimy, uzyskaliśmy ten sam wynik, ale bez konieczności wykonywania kroku pośredniego oraz konwersji typów, ponieważ wynik jest już właściwego typu.

PostgreSQL najpierw wykonuje zapytanie w nawiasach. Po uzyskaniu odpowiedzi uruchamia zapytanie zewnętrzne, zastępując wynik zapytania wewnętrznego. Jeżeli zachodzi taka potrzeba, możemy korzystać z wielu zapytań podrzędnych dla różnych klauzul `WHERE`. Nie ma ograniczeń co do liczby, ale konieczność korzystania z wielu zagnieżdżonych instrukcji `SELECT` występuje niezbyt często.

Wypróbuj to — zapytania podrzędne

Wypróbujmy bardziej złożony przykład. Przypuśćmy, że chcemy znać wszystkie towary, których cena zakupu jest większa od średniej ceny zakupu, ale cena sprzedaży jest mniejsza od średniej ceny sprzedaży. Sugeruje nam to, że nasza marża nie jest zbyt wysoka, zatem prawdopodobnie niezbyt wiele towarów spełni takie kryteria.

Wiemy już, jak znaleźć średnią cenę zakupu: `SELECT AVG(cost_price) FROM item`. Odnalezienie średniej ceny sprzedaży realizuje się analogicznie: `SELECT AVG(sell_price) FROM item`.

Nasze zapytanie główne przyjmie następującą postać:

```
SELECT * FROM item WHERE cost_price> średnia cena zakupu AND sell_price
< średnia cena sprzedaży
```

Jeżeli połączymy te trzy zapytania otrzymamy:

```
bpsimple=# SELECT * FROM item WHERE cost_price>(SELECT AVG(cost_price)
bpsimple-# FROM item) AND sell_price < (SELECT AVG(sell_price) FROM item);
 item_id |  description  | cost_price | sell_price
---------+---------------+------------+------------
       5 | Picture Frame |       7.54 |       9.95
(1 row)

bpsimple=#
```

Być może ktoś powinien przyjrzeć się cenie ramek do obrazków i sprawdzić, czy jest właściwa!

Jak to działa

PostgreSQL skanuje zapytanie i stwierdza, że istnieją dwa zapytania w nawiasach — zapytania podrzędne. Następnie rozwiązuje te zapytania niezależnie, po czym przesyła odpowiedzi do właściwej części klauzuli `WHERE` zapytania głównego i wykonuje to zapytanie.

Moglibyśmy zastosować inne klauzule WHERE lub ORDER BY. Połączenie warunków WHERE pochodzących z zapytań podrzędnych z warunkami konwencjonalnymi jest całkowicie poprawne.

Rodzaje zapytań podrzędnych

Do tej pory wykorzystywaliśmy tylko takie zapytania podrzędne, które zwracały pojedyncze wyniki, ponieważ korzystaliśmy w nich z funkcji agregacji. W zasadzie zapytania podrzędne mogą mieć wyniki trzech typów:

- pojedyncza wartość (jak te, które już widzieliśmy);
- brak wierszy lub kilka wierszy;
- test istnienia czegoś.

Spójrzmy na drugi rodzaj zapytań podrzędnych, gdzie w wyniku może wystąpić kilka wierszy. Przypuśćmy, że chcemy wiedzieć, jakie towary, których cena zakupu jest większa niż 10, znajdują się w magazynie. Możemy dowiedzieć się tego za pomocą pojedynczej instrukcji SELECT następującej postaci:

```
bpsimple=# SELECT s.item_id, s.quantity FROM stock s, item i WHERE
bpsimple-# i.cost_price > CAST(10.0 AS NUMERIC(7,2) AND s.item_id =
bpsimple-# i.item_id;
 item_id | quantity
---------+----------
       1 |       12
       7 |        8
(2 rows)

bpsimple=#
```

Zauważmy, że dla skrócenia zapytania zastosowaliśmy aliasy nazw tabel (stock będzie obecnie tabelą s, item — tabelą i). Połączymy teraz dwie tabele (s.item_id = i.item_id) oraz dodamy w tabeli item warunek dotyczący ceny zakupu (i.cost_price > CAST(10.0 AS NUMERIC(7,2))).

Możemy także zapisać to zapytanie podrzędne stosując słowo kluczowe IN, w celu zbadania listy wartości. Należy zapisać zapytanie, które daje w wyniku listę identyfikatorów towarów, dla których cena zakupu towaru jest mniejsza niż 10.0:

```
SELECT * FROM item WHERE cost_price > CAST(10.0 AS NUMERIC(7,2));
```

Potrzebujemy także zapytania wybierającego towary z tabeli stock:

```
SELECT * FROM stock WHERE item_id IN lista wartości
```

Możemy następnie połączyć oba zapytania w następujący sposób:

```
bpsimple=# SELECT * FROM stock WHERE item_id IN (SELECT item_id FROM item
bpsimple-# WHERE cost_price> CAST(10.0 AS NUMERIC(7,2)));
 item_id | quantity
---------+----------
       1 |       12
```

```
      7 |       8
(2 rows)

bpsimple=#
```

Procedury te dają nam te same wyniki. Zapytania podrzędne — chociaż nie wszystkie — można zazwyczaj zapisać jako powiązania. Z tego względu ważne jest, aby je zrozumieć. Tak jak w przypadku bardziej konwencjonalnych zapytań, można zanegować warunek, zapisując `NOT IN`. Można też wpisać dodatkowe klauzule `WHERE` oraz warunki `ORDER BY`.

Czego powinniśmy użyć w przypadku, gdy mamy zapytanie podrzędne, które można zapisać jako powiązanie? Należy rozważyć dwie sprawy — czytelność i wydajność. Dla okazyjnie wykorzystywanych zapytań działających z niewielkimi tabelami i wykonującymi się szybko wykorzystujemy dowolną postać, która jest najbardziej czytelna. Jeżeli jest to zapytanie wykorzystywane często, dla dużych tabel, opłaca się zapisać je na różne sposoby i doświadczalnie sprawdzić wydajność. Być może optymalizator zapytań będzie w stanie zoptymalizować oba rodzaje i wydajność obu zapytań będzie identyczna. Zwycięży wówczas zapytanie, które będzie bardziej czytelne.

> Należy ostrożnie badać wydajność instrukcji SQL. Istnieje wiele czynników znajdujących się poza naszą kontrolą, jak np. buforowanie danych przez system operacyjny.

Można się także przekonać, że wydajność zależy w bardzo dużym stopniu od konkretnych danych w naszej bazie lub zmienia się diametralnie wraz ze zmianą liczby wierszy w poszczególnych tabelach.

Nie poznaliśmy jeszcze ostatniego typu zapytania podrzędnego — takiego, które bada istnienie czegoś — ponieważ jest ono dosyć złożone. Przed końcem rozdziału wrócimy do tego typu zapytań.

Zapytania podrzędne skorelowane

Do tej pory poznaliśmy takie typy zapytań podrzędnych, gdzie dla uzyskania wyniku wykonywaliśmy zapytanie, a następnie „włączaliśmy je" do drugiego zapytania. Te dwa zapytania nie są jednak związane w innym przypadku i dlatego nazywa się je zapytaniami podrzędnymi nieskorelowanymi. Wynika to z faktu, że pomiędzy wewnętrznym, a zewnętrznym zapytaniem nie ma powiązanych tabel. W obu częściach instrukcji `SELECT` można używać tej samej kolumny z tej samej tabeli, ale są one związane tylko poprzez wynik zapytania podrzędnego, które zasila klauzulę `WHERE` głównego zapytania.

Istnieje inna grupa zapytań podrzędnych, które nazywa się zapytaniami podrzędnymi skorelowanymi, gdzie związek pomiędzy dwiema częściami zapytania jest bardziej złożony. W zapytaniu podrzędnym skorelowanym tabela w wewnętrznej instrukcji `SELECT` jest powiązana z tabelą w zewnętrznej części instrukcji. Z tego właśnie względu te dwa zapytania są skorelowane. Ta grupa zapytań podrzędnych daje duże możliwości. Bardzo często można je zapisać jako zwykłe instrukcje `SELECT` z powiązaniami.

Zapytanie skorelowane ma następującą postać ogólną:

```
SELECT kolumnaA FROM tabela1 T1 WHERE T1.kolumnaB = (SELECT T2.kolumnaB FROM
tabela2 T2
WHERE T2.kolumnaC = T1.kolumnaC)
```

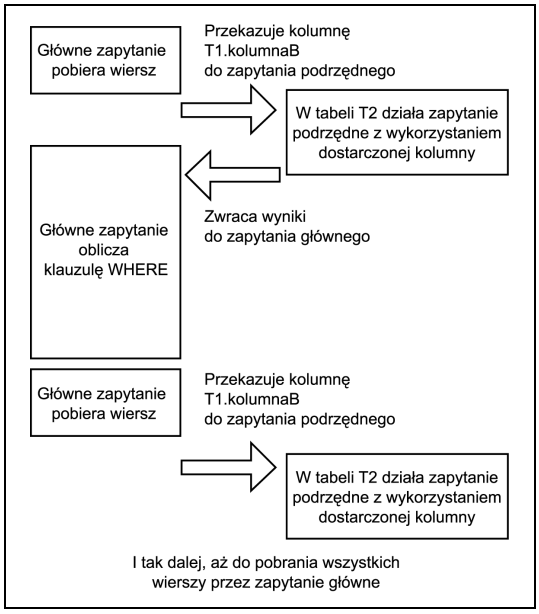
Zapisaliśmy tę instrukcję w kodzie pseudo-SQL, aby można ją było łatwiej wyjaśnić. Ważną rzeczą, którą należy zauważyć, jest fakt, że tabela `T1` z zewnętrznej instrukcji `SELECT` występuje także w wewnętrznej instrukcji `SELECT`. Zatem tabele zewnętrzną i wewnętrzną uważa się za powiązane. Zauważmy, że zastosowaliśmy aliasy nazw tabel. Jest to bardzo ważne, ponieważ reguły nazw tabel w skorelowanych zapytaniach są dość złożone i niewielka pomyłka może skutkować niezrozumiałymi wynikami.

> Zalecamy, aby w zapytaniach podrzędnych skorelowanych zawsze stosować aliasy tabel, ponieważ jest to opcja najbezpieczniejsza.

Po wykonaniu tej instrukcji dzieją się dosyć skomplikowane rzeczy. Po pierwsze, dla wewnętrznej instrukcji `SELECT` z tabeli `T1` pobierany jest wiersz, a następnie kolumna `T1.kolumnaB` jest przekazywana do wewnętrznego zapytania, które wykonuje się wybierając wiersze z tabeli `T2`, korzystając jednak z przekazanych informacji. Wynik jest ponownie przekazywany do zapytania wewnętrznego, które przed przejściem do następnego wiersza kończy obliczanie klauzuli `WHERE`.

Pokazano to na poniższym diagramie:

Rysunek 7.1.



Jeżeli wydaje się to nam trochę skomplikowane, to trzeba przyznać, że tak właśnie jest. Skorelowane zapytania podrzędne wykonywane są dość nieefektywnie. Czasami jednak udaje się dzięki nim rozwiązać bardzo złożone problemy. Tak więc dobrze wiedzieć, że istnieją, nawet jeżeli nieczęsto będziemy z nich korzystać.

Wypróbuj to — zapytanie podrzędne skorelowane

W prostej bazie danych, jak ta, którą wykorzystujemy, potrzeba użycia zapytań podrzędnych skorelowanych występuje niezbyt często. Zazwyczaj można je zapisać w inny sposób. Możemy jednak wykorzystać nawet naszą prostą bazę danych w celu zademonstrowania ich zastosowania.

Przypuśćmy, że chcemy znać datę złożenia zamówień dla klientów w Bingham. Chociaż moglibyśmy zapisać to w sposób bardziej konwencjonalny, skorzystamy z zapytania podrzędnego skorelowanego w następującej postaci:

```
bpsimple=# SELECT oi.date_placed FROM orderinfo oi WHERE oi.customer_id =
bpsimple-# (SELECT c.customer_id from customer c WHERE c.customer_id =
bpsimple-# oi.customer_id and town = 'Bingham');
 date_placed
-------------
 2000-06-23
 2000-07-21
(2 rows)

bpsimple=#
```

Jak to działa?

Zapytanie rozpoczyna działanie od pobrania wiersza z tabeli `orderinfo`. Następnie wykonuje ono zapytanie podrzędne dla tabeli `customer`, korzystając z odnalezionej wartości `customer_id`. Wykonuje się zapytanie podrzędne poszukując wierszy, gdzie pole `customer_id` z zewnętrznego zapytania daje wiersz w tabeli `customer`, który zawiera także miasto Bingham. W przypadku znalezienia takiego wiersza, pole `customer_id` jest przekazywane z powrotem do zapytania głównego, które kończy klauzulę `WHERE` i jeżeli jest ona prawdziwa, wyświetla kolumnę `date_placed`. Zapytanie zewnętrzne przechodzi następnie do kolejnego wiersza i sekwencja powtarza się.

Spójrzmy na inny przykład. Tym razem wykorzystamy trzeci typ zapytania, którego do tej pory jeszcze nie poznaliśmy, gdzie zapytanie podrzędne bada istnienie czegoś.

Przypuśćmy, że chcemy wyszczególnić wszystkich klientów, którzy składali zamówienia. W naszej przykładowej bazie danych nie ma ich wielu. Pierwsza część zapytania jest prosta; zapiszmy:

```
SELECT fname, lname FROM customer c;
```

Zwróćmy uwagę, że zastosowaliśmy alias tabeli `customer` — `c`, który został przygotowany dla zapytania podrzędnego. Kolejna część zapytania powinna sprawdzić, czy wartość pola `customer_id` występuje także w tabeli `orderinfo`:

```
SELECT 1 FROM orderinfo oi WHERE oi.customer_id = c.customer_id;
```

Należy tu zwrócić uwagę na dwa bardzo ważne aspekty. Po pierwsze, zastosowaliśmy znaną sztuczkę. Jeżeli zależy nam na wykonaniu zapytania, ale nie interesują nas wyniki, po prostu umieszczamy `'1'` tam, gdzie zwykle umieszcza się nazwy kolumn. Ozna-

cza to, że jeżeli zostaną odnalezione jakiekolwiek dane, wynikiem zapytania będzie 1, co jest łatwym i wydajnym sposobem powiedzenia `'true'`. Może to wydawać się dosyć dziwne, zatem zobaczmy:

```
bpsimple=# SELECT 1 FROM customer WHERE town = 'Bingham';
 ?column?
----------
        1
        1
        1
(3 rows)

bpsimple=#
```

To działa, nawet jeśli wygląda nieco dziwacznie. Ważne jest, aby nie korzystać tu z funkcji `COUNT(*)`, ponieważ potrzebujemy wyniku z każdego wiersza, gdzie kolumna `town` ma wartość `Bingham`, a nie informacji, ilu mamy klientów pochodzących z Bingham.

Drugą ważną rzeczą, na którą należy zwrócić uwagę jest fakt, że skorzystaliśmy z tabeli `customer` w tym zapytaniu podrzędnym, które w zasadzie było użyte w zapytaniu głównym. Właśnie to decyduje o tym, że są to zapytania skorelowane. Tak jak poprzednio, dla wszystkich tabel zastosowaliśmy aliasy. Musimy teraz połączyć obie połowy.

Czas na zapoznanie się z ostatnim rodzajem zapytań podrzędnych, które wcześniej pominęliśmy. Bada ono za pomocą słowa kluczowego `EXISTS` w klauzuli `WHERE` istnienie wartości, bez konieczności znajomości danych.

Dla naszego zapytania skorzystanie z `EXISTS` jest dobrym sposobem połączenia dwóch instrukcji `SELECT`, ponieważ chcemy wiedzieć tylko, czy zapytanie podrzędne zwraca wiersz. Klauzula `EXISTS` zazwyczaj działa wydajniej niż inne typy powiązań w warunkach `IN`. Z tego względu, w przypadku, gdy istnieje wybór sposobu zapisu zapytania, często warto korzystać właśnie z niej zamiast innych typów powiązań.

```
bpsimple=# SELECT fname, lname FROM customer c WHERE EXISTS (SELECT 1 FROM
bpsimple-# orderinfo oi WHERE oi.customer_id=c.customer_id);
 fname |  lname
-------+---------
 Alex  | Matthew
 Ann   | Stones
 Laura | Hendy
 David | Hudson
(4 rows)

bpsimple=#
```

Można tu zobaczyć, jak są zapisywane skorelowane zapytania podrzędne. Czasami, kiedy napotkamy problem, który wydaje się niemożliwy do rozwiązania za pomocą zwykłych zapytań SQL, może się okazać, że rozwiązaniem tych trudności jest skorelowane zapytanie podrzędne.

Powiązania same z sobą

Specjalnym typem powiązań są powiązania same z sobą. Wykorzystuje się je w celu powiązania kolumn znajdujących się w tej samej tabeli. Potrzeba korzystania z tych powiązań jest niezwykle rzadka, ale czasami bardzo się one przydają, dlatego pokrótce omówimy je w tym podrozdziale.

Przypuśćmy, że sprzedajemy towary, które można sprzedawać w kompletach lub pojedynczo. Dla potrzeb przykładu załóżmy, że sprzedajemy komplet krzeseł i stół, a także stół i krzesła oddzielnie. Chcielibyśmy zapisać nie tylko poszczególne, pojedyncze towary, ale także relacje między nimi, podczas gdy są sprzedawane jako całość. Często nazywa się to *eksplozją części*. Z pojęciem tym spotkamy się ponownie w rozdziale 12.

Rozpocznijmy od utworzenia tabeli, która zawiera nie tylko identyfikatory towarów i ich opis, ale także dodatkowy identyfikator towaru:

```
CREATE TABLE part (part_id INT, description VARCHAR(32), parent_part_id INT);
```

Wykorzystamy pole `parent_part_id` do przechowywania informacji o identyfikatorze komponentu, którego częścią jest nasz towar. Na przykład, przypuśćmy, że mamy zestaw krzeseł i stołu, któremu nadamy wartość `item_id` = 1, składający się z krzesła, dla którego przypisujemy `item_id` = 2 oraz stołu, określonego jako `item_id` = 3. Instrukcje `INSERT` wyglądałyby wówczas następująco:

```
bpsimple=# INSERT INTO part (part_id, description, parent_part_id) VALUES(1,
bpsimple=# 'table and chairs', NULL);
INSERT 21579 1
bpsimple=# INSERT INTO part (part_id, description, parent_part_id) VALUES(2,
bpsimple-# 'chair', 1);
INSERT 21580 1
bpsimple=# INSERT INTO part (part_id, description, parent_part_id) VALUES(3,
bpsimple-# 'table', 1);
INSERT 21581 1
bpsimple=#
```

Zapisaliśmy dane, ale w jaki sposób uzyskamy informacje o tym, które pojedyncze części składają się na poszczególny komponent? Musimy powiązać tabelę z nią samą.

Okazuje się to dosyć proste. Musimy zastosować aliasy nazw tabel, a następnie zapisać klauzulę `WHERE` odnoszącą się do tej samej tabeli, stosując jednak różne nazwy:

```
bpsimple=# SELECT p1.description, p2.description FROM part p1, part p2 WHERE
bpsimple-# p1.part_id = p2.parent_part_id;
   description    | description
------------------+-------------
 table and chairs | chair
 table and chairs | table
(2 rows)

bpsimple=#
```

To działa, ale jest trochę mylące, ponieważ mamy dwie kolumny wyniku z tą samą nazwą. Możemy w łatwy sposób poprawić tę niedogodność za pomocą słowa kluczowego AS:

```
bpsimple=# SELECT p1.description AS "Combined", p2.description AS "Parts"
bpsimple-# FROM part p1, part p2 WHERE p1.part_id = p2.parent_part_id;
     Combined     | Parts
------------------+-------
 table and chairs | chair
 table and chairs | table
(2 rows)

bpsimple=#
```

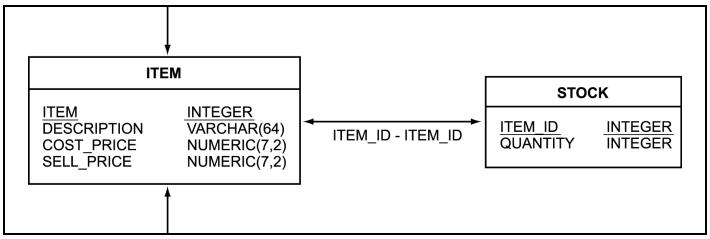
Z powiązaniami typu „same z sobą" spotkamy się ponownie w rozdziale 12., w którym omówimy sposób zapisu związku przełożony — podwładny w pojedynczej tabeli.

Powiązania zewnętrzne

Ostatnim głównym zagadnieniem, jakie poruszymy w tym rozdziale, jest klasa powiązań znana jako powiązania zewnętrzne. Są one podobne do powiązań konwencjonalnych, ale wykorzystują nieco inną składnię. Z tego względu odłożyliśmy spotkanie z nimi na koniec tego rozdziału.

Spójrzmy na nasze tabele item i stock:

Rysunek 7.2.



Jak pamiętamy, wszystkie towary przeznaczone do sprzedaży znajdują się w tabeli item, natomiast w tabeli stock znajdują się tylko te, które aktualnie znajdują się w magazynie.

Przypuśćmy, że chcemy uzyskać listę wszystkich towarów, które sprzedajemy, ze wskazaniem ilości, jaką mamy w magazynie. To z pozoru proste żądanie okazuje się zaskakująco trudne do zapisania w języku SQL, który znamy do tej pory — chociaż jest możliwe. Bardzo dobrze jest włożyć nieco pracy w rozwiązanie; spróbujmy zatem skorzystać tylko z tych instrukcji SQL, które znamy do tej pory.

Zastosujmy prostą instrukcję SELECT łączącą dwie tabele:

```
bpsimple=# SELECT i.item_id, s.quantity FROM item i, stock s WHERE i.item_id
bpsimple-# = s.item_id
 item_id | quantity
---------+----------
       1 |       12
```

```
         2 |        2
         4 |        8
         5 |        3
         7 |        8
         8 |       18
        10 |        1
(7 rows)

bpsimple=#
```

Łatwo zauważyć (ponieważ wiemy, że nasze identyfikatory `item_id` występują w tabeli kolejno, bez luk), że brakuje niektórych identyfikatorów `item_id`. Brakujące wiersze dotyczą towarów, których nie mamy w magazynie, ponieważ nie istnieje dla nich powiązanie pomiędzy tabelami `item` oraz `stock` — w tabeli `stock` brakuje zapisu dla tego `item_id`.

Brakujące wiersze możemy znaleźć za pomocą zapytania podrzędnego oraz klauzuli `IN`:

```
bpsimple=# SELECT i.item_id FROM item i WHERE i.item_id NOT IN (SELECT
bpsimple-# i.item_id FROM item i, stock s WHERE i.item_id = s.item_id);
 item_id
---------
       3
       6
       9
      11
(4 rows)

bpsimple=#
```

Mówiąc prościej, możemy zażądać: „znajdź wszystkie identyfikatory `item_id` w tabeli `item`, poza tymi, które znajdują się w tabeli `stock`".

Wewnętrzna instrukcja `SELECT` jest tą samą, którą zastosowaliśmy wcześniej, ale tym razem skorzystaliśmy z listy identyfikatorów `item_id`, które instrukcja ta zwraca jako część innej instrukcji `SELECT`. Główna instrukcja `SELECT` wyszczególnia wszystkie znane identyfikatory `item_id` poza znalezionymi w zapytaniu podrzędnym, usuniętymi przez klauzulę `WHERE NOT IN`.

Tak więc mamy teraz listy identyfikatorów `item_id` zarówno dla towarów, których nie ma w magazynie, jak i dla tych, które się w nim znajdują. Listy te uzyskaliśmy jednak w oddzielnych zapytaniach. Musimy teraz powiązać obie listy; możemy to zrobić za pomocą instrukcji `UNION`. Istnieje jednak pewien problem. Nasza pierwsza instrukcja zwraca dwie kolumny: `item_id` oraz `quantity`, natomiast druga tylko identyfikatory `item_id`, ponieważ w magazynie nie ma odpowiednich towarów. Należy zatem dodać sztuczną kolumnę do drugiej instrukcji `SELECT`, aby miała tę samą co pierwsza liczbę kolumn oraz kolumny tego samego typu. Mamy zamiar wykorzystać wartość `NULL`, chociaż równie dobrze moglibyśmy użyć *0* (zero). Później zobaczymy, dlaczego wybraliśmy `NULL`.

Oto nasze kompletne zapytanie:

```
SELECT i.item_id, s.quantity FROM item i, stock s WHERE i.item_id = s.item_id
UNION
SELECT i.item_id, NULL FROM item i WHERE i.item_id NOT IN
       (SELECT i.item_id FROM item i, stock s WHERE i.item_id = s.item_id);
```

Wygląda to na nieco skomplikowane, ale spróbujmy zobaczyć, jak działa:

```
bpsimple=# SELECT i.item_id, s.quantity FROM item i, stock s WHERE i.item_id
bpsimple-# = s.item_id
bpsimple-# UNION
bpsimple-# SELECT i.item_id, NULL FROM item i WHERE i.item_id NOT IN (select
bpsimple-# i.item_id FROM item i, stock s WHERE i.item_id = s.item_id);
item_id | quantity
---------+----------
       1 |       12
       2 |        2
       3 |
       4 |        8
       5 |        3
       6 |
       7 |        8
       8 |       18
       9 |
      10 |        1
      11 |
(11 rows)

bpsimple=#
```

W początkowym okresie istnienia języka SQL był to jedyny sposób rozwiązania tego typu problemów, poza tym, że SQL89 nie umożliwiał stosowania wartości NULL, z której skorzystaliśmy w drugiej instrukcji SELECT, jako kolumny. Na szczęście większość producentów SQL dawała taką możliwość. W przeciwnym przypadku byłoby jeszcze trudniej. Gdybyśmy nie mogli wykorzystać wartości NULL, musielibyśmy zastosować *0* jako nieco gorszą alternatywę. Wartość NULL jest lepszym rozwiązaniem, ponieważ *0* może być mylące, natomiast NULL zawsze będzie wyświetlane jako puste miejsce.

Aby obejść to raczej skomplikowane rozwiązanie dla dość popularnych problemów, producenci wymyślili mechanizm znany jako powiązania zewnętrzne. Niestety, ponieważ nie było tego w standardzie, każdy z producentów opracował swoje własne rozwiązanie, o podobnym do innych działaniu, ale różnej składni.

W systemach Oracle oraz DB2 zastosowano składnię, w której w klauzuli WHERE wykorzystywano znak + dla zaznaczenia, że w wyniku mają się znaleźć wszystkie wartości z tabeli (tabeli chronionej), nawet jeżeli powiązanie nie powiedzie się. W systemie Sybase zastosowano w klauzuli WHERE symbol *=, aby oznaczyć tabelę chronioną. Obie te składnie są dosyć oczywiste, ale niestety różne, co nie jest dobre dla przenośności kodu SQL.

Kiedy pojawił się standard SQL92, zdefiniowano w nim ogólny sposób implementowania powiązań zewnętrznych, który stosował jeszcze inną składnię. Producenci dosyć wolno implementowali nowy standard. W systemie Sybase 11 nie obsługiwano jeszcze tego mechanizmu, podobnie jak w Oracle8. Obydwa produkty wydano już po ukazaniu

się standardu. W systemie PostgreSQL zaimplementowano standardową metodę począwszy od wersji 7.1. Tak więc, jeżeli korzystamy ze starszej wersji, aby móc wypróbować przykłady z ostatniej części tego rozdziału, musimy uaktualnić posiadaną wersję PostgreSQL. Jeżeli korzystamy z wersji PostgreSQL starszej niż 7.1, uaktualnienie opłaca się także z innych powodów, ponieważ wersja 7.1 jest znacząco lepsza od wersji starszych.

Składnia standardu SQL92 zamienia klauzulę `WHERE`, którą znamy. Wprowadza dla łączenia tabel klauzulę `ON` oraz słowo kluczowe `LEFT OUTER JOIN`.

Składnia jest w tym przypadku następująca:

```
SELECT kolumny FROM tabela1 LEFT OUTER JOIN tabela2 ON tabela1.kolumna
= tabela2.kolumna
```

Tabela z lewej strony `LEFT OUTER JOIN` jest zawsze tabelą chronioną — tą, dla której są wyświetlane wszystkie wiersze.

Możemy zatem zapisać nasze zapytanie ponownie, za pomocą następującej składni:

```
SELECT i.item_id, s.quantity FROM item i  LEFT OUTER JOIN stock s ON i.item_id =
s.item_id;
```

Wygląda to niemal na zbyt proste, aby mogło być prawdziwe. Spróbujmy zatem zobaczyć działanie podanej wyżej składni:

```
bpsimple=# SELECT i.item_id, s.quantity FROM item i LEFT OUTER JOIN stock s
bpsimple-# ON i.item_id = s.item_id;
 item_id | quantity
---------+----------
       1 |       12
       2 |        2
       3 |
       4 |        8
       5 |        3
       6 |
       7 |        8
       8 |       18
       9 |
      10 |        1
      11 |
(11 rows)

bpsimple=#
```

Świetnie, odpowiedź jest identyczna z tą, jaką otrzymaliśmy w wyniku zastosowania poprzedniego sposobu. Widzimy teraz, dlaczego producenci czuli potrzebę implementacji powiązań zewnętrznych, nawet jeżeli nie było ich w oryginalnym standardzie SQL89.

Istnieje także odpowiednik poznanego wyżej słowa kluczowego, `RIGHT OUTER JOIN`, ale prawie zawsze korzysta się z powiązania lewostronnego, ze względu na to, że przynajmniej dla mieszkańców Zachodu bardziej sensowne jest wyświetlanie listy wszystkich obiektów po lewej stronie wyniku, a nie po prawej.

Wypróbuj to — bardziej złożony warunek

Proste lewostronne powiązanie zewnętrzne, które wykorzystaliśmy, jest doskonałe tak długo, jak działa, ale w jaki sposób dodać bardziej złożone warunki?

Przypuśćmy, że interesują nas tylko te wiersze z tabeli `stock`, dla których w magazynie znajdują się więcej niż dwie sztuki towaru, a ponadto te, w których cena zakupu jest większa niż 5,0. Jest to dosyć złożony problem, ponieważ chcemy zastosować jedną regułę do tabeli `item` (`cost_price >5.0`), a inną regułę do tabeli `stock` (`quantity>2`). W dalszym ciągu chcemy jednak wyświetlić wszystkie wiersze z tabeli `item`, dla których prawdziwy jest warunek dla tabeli `item`, nawet jeżeli tych towarów wcale nie ma w magazynie.

W tym celu połączymy warunki `ON`, które działały tylko z tabelami związanymi lewostronnym powiązaniem zewnętrznym, z warunkami `WHERE`, ograniczającymi zwracane wiersze po powiązaniu tabel.

Warunek dla tabeli `stock` jest częścią powiązania zewnętrznego, w którym nie chcemy ograniczać wierszy dla towarów, których nie ma w magazynie, zatem zapiszemy go jako część warunku `ON`:

```
ON item_id = s.item_id AND s.quantity >2
```

Dla warunku `item`, który dotyczy wszystkich wierszy, zastosujemy klauzulę `WHERE`:

```
WHERE i.cost_price > CAST(5.0 AS NUMERIC(7,2));
```

Łącząc obie części otrzymamy:

```
bpsimple=# SELECT i.item_id, i.cost_price, s.quantity FROM item i LEFT OUTER
bpsimple-# JOIN stock s ON i.item_id = s.item_id AND s.quantity > 2 WHERE
bpsimple-# i.cost_price > CAST(5.0 AS NUMERIC(7,2));
 item_id | cost_price | quantity
---------+------------+----------
       1 |      15.23 |       12
       2 |       7.45 |
       5 |       7.54 |        3
       6 |       9.23 |
       7 |      13.36 |        8
      11 |      19.73 |
(6 rows)

bpsimple=#
```

Jak to działa?

Aby uzyskać wszystkie wartości z tabeli `item`, opcjonalnie w połączeniu z tabelą `stock`, gdzie istnieją oba wiersze, a wartość pola `quantity` jest większa niż 2, stosujemy mechanizm `LEFT OUTER JOIN`. Daje to nam zbiór, w którym znajdą się wszystkie wiersze z tabeli `item`, ale kolumna `quantity` z tabeli `stock` będzie zawierała wartość `NULL`, chyba że obydwie tabele zawierają zapis dla danego towaru oraz wartość pola `quantity` jest większa niż 2. Następnie stosujemy klauzulę `WHERE`, która pozwala na wyświetlanie tylko tych wierszy, w których cena zakupu (`cost_price` z tabeli `item`) jest większa niż 5.0.