Rozdział 15

Korzystanie z baz danych i języka SQL

15.1. Czym jest baza danych?

Baza danych to plik zorganizowany tak, by jak najlepiej przechowywać dane. Większość baz danych jest podobna do słownika w tym sensie, że mapują one klucze na wartości. Największa różnica polega na tym, że baza danych znajduje się na dysku (lub innym trwałym nośniku danych), a więc zachowuje swoje dane po zakończeniu programu. Ponieważ baza danych przechowywana jest na trwałym nośniku, może ona zmieścić znacznie więcej danych niż słownik, który jest ograniczony wielkością pamięci w komputerze.

Podobnie jak słownik, oprogramowanie bazodanowe jest zaprojektowane tak, aby dodawanie i odczyt były bardzo szybkie, nawet dla dużych ilości danych. Oprogramowanie bazodanowe utrzymuje swoją wydajność poprzez budowanie *indeksów*, kiedy dane są dodawane do bazy, co pozwala komputerowi na szybkie przejście do konkretnego wpisu.

Istnieje wiele różnych relacyjnych systemów bazodanowych, które są wykorzystywane do rozmaitych celów, na przykład: Oracle, MySQL, Microsoft SQL Server, PostgreSQL i SQLite. W tej książce skupimy się na SQLite, ponieważ jest to bardzo popularna baza danych i jest już wbudowana w Pythona. SQLite jest przeznaczony do bycia *wbudowanym* (*osadzonym*) w innych aplikacjach, tak aby zapewnić obsługę bazy danych w obrębie tych aplikacji. Na przykład przeglądarka Firefox, podobnie jak wiele innych aplikacji, również korzysta wewnętrznie z bazy danych SQLite.

SQLite[1] doskonale nadaje się do rozwiązywania niektórych informatycznych problemów z zarządzaniem danymi, takich jak robot internetowy do zbierania danych z Twittera, który opiszemy w tym rozdziale. W kolejnych sekcjach poznamy kilka standardowych poleceń języka SQL, tj. `CREATE TABLE`, `DROP TABLE`, `INSERT`, `SELECT`, `UPDATE` i `DELETE`.

15.2. Pojęcia związane z bazami danych

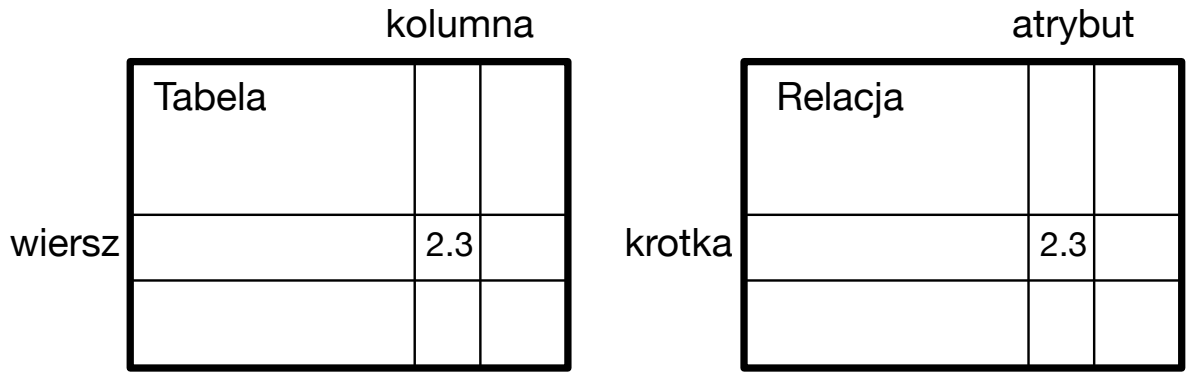
Kiedy pierwszy raz spojrzysz na bazę danych, to będzie ona wyglądała jak plik arkusza kalkulacyjnego, który posiada wiele arkuszy. Podstawowe struktury danych w bazie danych to: *tabele*, *wiersze*, i *kolumny*.

W technicznych opisach relacyjnych baz danych pojęcia tabeli, wiersza i kolumny są bardziej formalnie nazywane odpowiednio *relacją*, *krotką* i *atrybutem*. W tym rozdziale użyjemy mniej formalnych pojęć.

15.3. Przeglądarka baz SQLite

Co prawda w tym rozdziale skoncentrujemy się na używaniu Pythona do pracy z danymi zawartymi w plikach bazy SQLite, jednak wiele operacji można wykonać wygodniej przy użyciu darmowego oprogramowania *Database Browser for SQLite*[2].

[1] https://sqlite.org/
[2] https://sqlitebrowser.org/



Rysunek 15.1. Relacyjne bazy danych

Za pomocą tej aplikacji można łatwo tworzyć tabele, wstawiać i edytować dane oraz uruchamiać proste zapytania SQL dotyczące danych zawartych w bazie.

Przeglądarka bazy danych jest dość podobna do edytora tekstowego używanego podczas pracy z plikami tekstowymi. Kiedy chcesz wykonać jedną lub kilka operacji na pliku tekstowym, możesz po prostu otworzyć go w edytorze tekstowym i dokonać żądanych zmian. Kiedy musisz wykonać wiele zmian w pliku tekstowym, to często szybciej i wygodniej będzie Ci napisać prosty program Pythona. Ten sam schemat postępowania możesz zastosować podczas pracy z bazami danych. Proste operacje będziesz wykonywał w menedżerze baz danych, a bardziej złożone operacje wygodniej będzie Ci wykonywać w Pythonie.

15.4. Tworzenie tabeli w bazie danych

Bazy danych wymagają bardziej szczegółowo zdefiniowanej struktury niż listy czy słowniki w Pythonie.

Kiedy w bazie danych tworzymy *tabelę*, musimy z wyprzedzeniem podać nazwę każdej *kolumny* występującej w tabeli oraz typ danych, który zamierzamy przechowywać w każdej z *kolumn*[3]. Kiedy oprogramowanie bazy danych wie, jaki typ danych będzie użyty w każdej kolumnie, może dzięki temu wybrać najbardziej efektywny sposób przechowywania i wyszukiwania danych w oparciu o ich typ.

Możesz zapoznać się z różnymi typami danych obsługiwanymi przez SQLite pod następującym adresem URL:

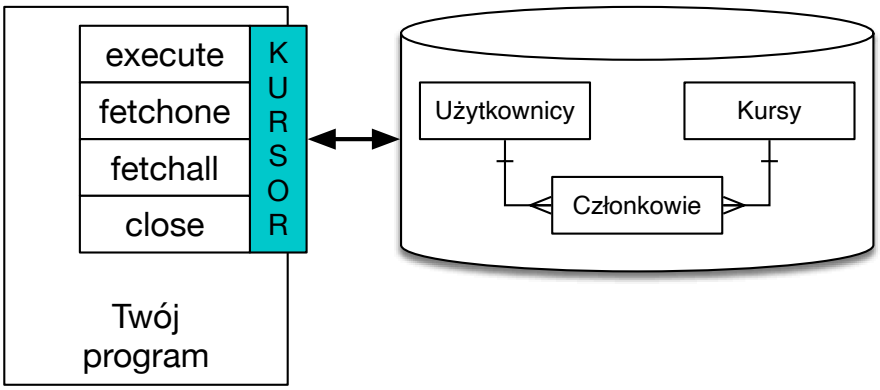
https://www.sqlite.org/datatypes.html

Zdefiniowanie z góry struktury dla Twoich danych może się początkowo wydawać niewygodne, ale zyskasz dzięki temu szybki dostęp do Twoich danych, nawet jeśli baza zawiera ich dużo.

Kod tworzący plik bazy danych i tabelę o nazwie `Utwory` z dwiema kolumnami jest następujący:

```
import sqlite3

conn = sqlite3.connect('muzyka.sqlite')
cur = conn.cursor()

cur.execute('DROP TABLE IF EXISTS Utwory')
cur.execute('CREATE TABLE Utwory (tytuł TEXT, odtworzenia INTEGER)')

conn.close()

# Kod źródłowy: https://py4e.pl/code3/db1.py
```

[3]SQLite faktycznie pozwala na pewną elastyczność w typie danych przechowywanych w kolumnie, ale w tym rozdziale nasze typy danych będą ściśle określone, dzięki czemu pokazane tutaj idee będą miały zastosowanie również w innych systemach baz danych, np. w MySQL.



Rysunek 15.2. Kursor bazodanowy

Funkcja `connect()` nawiązuje „połączenie" z bazą danych przechowywaną w pliku `muzyka.sqlite`, który znajduje się w bieżącym katalogu. Jeśli plik ten nie istnieje, to zostanie utworzony. Nazywa się to „połączeniem", ponieważ czasami baza danych przechowywana jest na „serwerze baz danych", czyli na innym komputerem niż ten, na którym uruchamiamy aplikację. W naszych prostych przykładach baza danych będzie po prostu plikiem lokalnym w tym samym katalogu co uruchamiany przez nas kod Pythona.

Kursor jest podobny do uchwytu pliku i używamy go do wykonania operacji na danych przechowywanych w bazie. Wywołanie `cursor()` jest odpowiednikiem wywołania `open()` podczas pracy z plikami tekstowymi.

Kiedy mamy już kursor, przy użyciu metody `execute()` możemy rozpocząć wykonywanie poleceń działających na zawartości bazy danych.

Komendy bazy danych są wyrażone w specjalnym języku, który został ustandaryzowany u wielu różnych dostawców baz danych, tak aby wystarczyło nauczyć się tylko jednego języka bazy danych. Nazywa się on *Structured Query Language* (z ang. strukturalny język zapytań) lub w skrócie *SQL*[4].

W powyższym przykładzie na naszej bazie danych wykonujemy dwa polecenia SQL. Przyjmuje się, że słowa kluczowe SQL pisze się dużymi literami, a części polecenia, które my dodajemy (takie jak nazwy tabel i kolumn), pisze się małymi lub dużymi literami.

Pierwsze polecenie SQL usuwa z bazy danych tabelę `Utwory`, jeśli ta tabela istnieje. Taki wzorzec postępowania ma po prostu pozwolić nam na wielokrotne uruchamianie tego samego programu do tworzenia tabeli `Utwory`, bez generowania błędu. Zauważ, że polecenie `DROP TABLE` usuwa z bazy danych tabelę i całą jej zawartość (i nie ma tutaj żadnej operacji typu „cofnij").

```
cur.execute('DROP TABLE IF EXISTS Utwory')
```

Druga komenda tworzy tabelę o nazwie `Utwory` z kolumną tekstową o nazwie `tytuł` i kolumną całkowitoliczbową o nazwie `odtworzenia`.

```
cur.execute('CREATE TABLE Utwory (tytuł TEXT, odtworzenia INTEGER)')
```

Teraz, gdy stworzyliśmy tabelę o nazwie `Utwory`, możemy umieścić w niej pewne dane, używając polecenia `INSERT`. Tak jak wcześniej, zaczynamy od nawiązania połączenia z bazą danych i uzyskania kursora. Następnie możemy wykonywać za jego pomocą polecenia SQL.

Polecenie `INSERT` wskazuje na to, jakiej używamy tabeli, następnie definiuje nowy wiersz, wymieniając pola, które chcemy umieścić w nowym wierszu `(tytuł, odtworzenia)`, a potem w `VALUES` wartości, które umieszczamy w tym wierszu. Wartości określone jako znaki zapytania `(?, ?)` wskazują, że rzeczywiste wartości do wstawienia są przekazywane jako krotka `( 'My Way', 15 )` – drugi parametr wywołania `execute()`.

```
import sqlite3
```

[4] https://pl.wikipedia.org/wiki/SQL

Utwory

tytuł	odtworzenia
Thunderstruck	20
My Way	15

Rysunek 15.3. Wiersze w tabeli

```
conn = sqlite3.connect('muzyka.sqlite')
cur = conn.cursor()

cur.execute('INSERT INTO Utwory (tytuł, odtworzenia) VALUES (?, ?)',
    ('Thunderstruck', 20))
cur.execute('INSERT INTO Utwory (tytuł, odtworzenia) VALUES (?, ?)',
    ('My Way', 15))
conn.commit()

print('Utwory:')
cur.execute('SELECT tytuł, odtworzenia FROM Utwory')
for row in cur:
     print(row)

cur.execute('DELETE FROM Utwory WHERE odtworzenia < 100')
conn.commit()

cur.close()
```

 # Kod źródłowy: https://py4e.pl/code3/db2.py

Najpierw wstawiamy poprzez `INSERT` dwa wiersze do naszej tabeli i używamy `commit()`, aby wymusić zapis danych do pliku.

Następnie używamy polecenia `SELECT` do pobrania wierszy, które właśnie wstawiliśmy w tabeli. W poleceniu `SELECT` wskazujemy, z których kolumn (`tytuł, odtworzenia`) oraz z której tabeli chcemy pobrać dane. Po wykonaniu polecenia `SELECT` możemy przejść po kursorze pętlą `for`. Aby zachować wydajność, kursor po wykonaniu polecenia `SELECT` nie odczytuje od razu wszystkich danych z bazy danych. Zamiast tego dane są odczytywane na żądanie, czyli w tym wypadku, gdy idziemy kolejno przez wiersze w instrukcji `for`.

Wynik działania programu jest następujący:

```
Utwory:
('Thunderstruck', 20)
('My Way', 15)
```

Nasza pętla `for` znajduje dwa wiersze, a każdy z nich jest krotką Pythona, w której pierwsza wartość to `tytuł`, a drugą to `odtworzenia`.

Uwaga: W innych książkach lub w internecie możesz zobaczyć ciągi znaków rozpoczynające się od `u'`. W Pythonie 2 była to wskazówka, że ciągi znaków są napisami z zestawem znaków Unicode, które są w stanie przechowywać niełacińskie znaki. W Pythonie 3 wszystkie ciągi znaków są domyślnie napisami z zestawem znaków Unicode.

Na samym końcu programu wykonujemy polecenie `DELETE`, aby usunąć wiersze, które właśnie utworzyliśmy, co pozwala uruchamiać nasz program wielokrotnie bez wystąpienia błędu. Polecenie `DELETE` pokazuje użycie klauzuli `WHERE`, która wyraża kryterium wyboru, dzięki czemu możemy poprosić bazę danych o zastosowanie polecenia tylko do spełniających to kryterium wierszy. W tym przykładzie są to wszystkie wiersze, więc czyścimy całą tabelę i możemy program uruchamiać wielokrotnie. Po wykonaniu operacji `DELETE` wywołujemy również `commit()`, aby wymusić usunięcie danych z bazy.

15.5. Podsumowanie języka SQL

W powyższych przykładach użyliśmy języka SQL i omówiliśmy kilka jego podstawowych poleceń. W tej sekcji przyjrzymy się bliżej językowi SQL i przedstawimy przegląd składni tego języka.

Ponieważ istnieje wielu różnych dostawców baz danych, SQL został ustandaryzowany po to, byśmy mogli komunikować się w podobny sposób z bazami danych działającymi na różnych silnikach.

Relacyjna baza danych składa się z tabel, wierszy i kolumn. Typ danych w kolumnie zazwyczaj jest tekstem, liczbą lub datą. Kiedy tworzymy tabelę, wskazujemy nazwy i typy kolumn:

```
CREATE TABLE Utwory (tytuł TEXT, odtworzenia INTEGER)
```

Aby wstawić wiersz do tabeli, używamy polecenia `INSERT`:

```
INSERT INTO Utwory (tytuł, odtworzenia) VALUES ('My Way', 15)
```

Polecenie `INSERT` określa nazwę tabeli, potem listę pól/kolumn, które chcesz ustawić w nowym wierszu, a następnie słowo kluczowe `VALUES` i listę odpowiednich wartości dla każdego pola.

Polecenie `SELECT` jest używane do pobierania wierszy i kolumn z bazy danych. Pozwala ono określić, które kolumny chcesz pobrać, a także klauzulę `WHERE`, by wybrać tylko te wiersze, które chcesz zobaczyć. Polecenie to pozwala także na opcjonalną klauzulę `ORDER BY` kontrolującą sortowanie zwracanych wierszy.

```
SELECT * FROM Utwory WHERE tytuł = 'My Way'
```

Użycie * między `SELECT` a `FROM` wskazuje, że chcesz, aby baza danych zwróciła wszystkie kolumny dla każdego wiersza, który pasuje do klauzuli `WHERE`.

Zauważ, że w przeciwieństwie do Pythona, w klauzuli `WHERE` wykorzystujemy pojedynczy znak = do wykonania testu na równość, a nie ==. Inne operacje logiczne dozwolone w klauzuli `WHERE` obejmują <, >, <=, >=, !=, jak również `AND` i `OR` oraz nawiasy okrągłe do tworzenia wyrażeń logicznych.

Możesz zażądać, aby zwrócone wiersze były posortowane według którejś z kolumn:

```
SELECT tytuł, odtworzenia FROM Utwory ORDER BY tytuł
```

Aby usunąć wiersz, w poleceniu `DELETE` potrzebna jest klauzula `WHERE`. Klauzula ta określa, które wiersze mają zostać usunięte:

```
DELETE FROM Utwory WHERE tytuł = 'My Way'
```

Przy użyciu polecenia `UPDATE` możliwa jest aktualizacja kolumn w obrębie jednego lub wielu wierszy w tabeli:

```
UPDATE Utwory SET odtworzenia = 16 WHERE tytuł = 'My Way'
```

Polecenie `UPDATE` najpierw określa tabelę, po słowie kluczowym `SET` wskazuje listę pól i wartości, które mają zostać zmienione, a na końcu mamy opcjonalną klauzulę `WHERE` do wyboru wierszy, które mają zostać zaktualizowane. Pojedyncze wyrażenie `UPDATE` zmienia wszystkie wiersze odpowiadające klauzuli `WHERE`. Jeśli klauzula `WHERE` nie jest określona, to wykonuje ona aktualizację dla wszystkich wierszy znajdujących się w tabeli.

Opisane wyżej cztery podstawowe polecenia SQL (`INSERT`, `SELECT`, `UPDATE` i `DELETE`) pozwalają na wykonanie podstawowych operacji potrzebnych do utworzenia i utrzymania danych.

15.6. Zbieranie informacji z Twittera przy użyciu baz danych

W poniższej sekcji stworzymy prostego robota internetowego, który przejdzie przez kilka kont na Twitterze i na podstawie zebranych informacji zbudujemy bazę danych. *Uwaga: Bądź bardzo ostrożny podczas uruchamiania poniższych programów. Nie chcesz pobierać zbyt dużo danych lub uruchamiać programów zbyt długo, ponieważ Twitter wyłączy Ci dostęp do swojego API.*

Jednym z problemów związanych z każdym rodzajem robota internetowego jest to, że czasami musimy go wielokrotnie zatrzymywać i uruchamiać ponownie, a jednocześnie nie chcemy utracić danych, które do tej pory pobraliśmy. Nie chcemy, by ponowne uruchomienie procesu zbierania danych zaczynało się w tym samym punkcie startowym, więc podczas pobierania danych musimy je przechowywać tak, aby nasz program mógł rozpocząć działanie w tym miejscu, w którym ostatnio zakończył pracę.

Zaczniemy od pobrania z Twittera listy znajomych jakiegoś użytkownika i ich statusów, przejrzenia tej listy i dodania każdego ze znajomych do bazy danych, która zostanie użyta w przyszłości do dalszego pobierania danych. Po przetworzeniu znajomych jednej osoby sprawdzamy naszą bazę danych i bierzemy jednego ze znajomych tego użytkownika. Robimy tak w kółko, wybierając „nieprzetworzną" osobę, pobierając jej listę znajomych i dodając tych znajomych, których jeszcze nie widzieliśmy, do naszej listy do odwiedzenia w przyszłości.

Śledzimy również, ile razy widzieliśmy danego znajomego w bazie danych, aby uzyskać jakiś wskaźnik jego „popularności".

Na dysku komputera przechowujemy bazę dotyczącą znanych nam kont Twittera, informacji o odwiedzonych kontach i popularności kont. Dzięki temu możemy nasz program zatrzymywać i uruchamiać ponownie tyle razy, ile chcemy.

Poniższy program jest nieco bardziej skomplikowany. Opiera się na kodzie z ćwiczenia z wcześniejszej części książki, który korzysta z API Twittera.

Oto kod źródłowy naszej aplikacji zbierającej informacje z Twittera:

```
from urllib.request import urlopen
import twurl
import json
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()

cur.execute('''
            CREATE TABLE IF NOT EXISTS Twitter
            (nazwa TEXT, pobrany INTEGER, znajomi INTEGER)''')

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    acct = input("Podaj nazwę konta na Twitterze lub wprowadź 'koniec': ")
    if (acct == 'koniec'): break
    if (len(acct) < 1):
        cur.execute('SELECT nazwa FROM Twitter WHERE pobrany = 0 LIMIT 1')
        try:
            acct = cur.fetchone()[0]
        except:
```

```
            print('Nie znaleziono niepobranych kont Twittera')
            continue

    url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '20'})
    print('Pobieranie', url)
    connection = urlopen(url, context=ctx)
    data = connection.read().decode()
    headers = dict(connection.getheaders())

    print('Pozostało', headers['x-rate-limit-remaining'])
    js = json.loads(data)
    # Debugowanie
    # print json.dumps(js, indent=4)

    cur.execute('UPDATE Twitter SET pobrany=1 WHERE nazwa = ?', (acct, ))

    countnew = 0
    countold = 0
    for u in js['users']:
        friend = u['screen_name']
        print(friend)
        cur.execute('SELECT znajomi FROM Twitter WHERE nazwa = ? LIMIT 1',
                    (friend, ))
        try:
            count = cur.fetchone()[0]
            cur.execute('UPDATE Twitter SET znajomi = ? WHERE nazwa = ?',
                        (count+1, friend))
            countold = countold + 1
        except:
            cur.execute('''INSERT INTO Twitter (nazwa, pobrany, znajomi)
                        VALUES (?, 0, 1)''', (friend, ))
            countnew = countnew + 1
    print('Nowe konta=', countnew, ' widziane ponownie=', countold)
    conn.commit()

cur.close()

# Kod źródłowy: https://py4e.pl/code3/twspider.py
```

Nasza baza danych jest przechowywana w pliku `spider.sqlite` i ma jedną tabelę o nazwie `Twitter`. Każdy wiersz w tabeli `Twitter` ma osobną kolumnę dla nazwy konta, informacji, czy pobraliśmy listę znajomych tego konta oraz tego, ile razy konto wystąpiło na listach znajomych.

W głównej pętli programu prosimy o podanie nazwy konta Twittera lub „koniec", aby wyjść z programu. Jeżeli użytkownik poda nazwę konta na Twitterze, pobierzemy listę jego znajomych (i ich statusy) oraz dodajemy każdego znajomego do bazy danych, o ile jeszcze go tam nie ma. Jeśli znajomy jest już na liście, to w bazie danych dodajemy 1 do kolumny `znajomi` w danym wierszu tabeli.

Jeśli użytkownik naciśnie klawisz `<Enter>`, to szukamy w bazie danych kolejnego konta na Twitterze, którego jeszcze nie odwiedziliśmy, pobieramy znajomych tego konta i ich statusy, dodajemy znajomych do bazy danych lub aktualizujemy ich liczbę zwiększając wartość w zmiennej `friends`.

Kiedy pozyskamy już listę znajomych i statusy, idziemy w pętli przez wszystkie elementy `user` występujące w zwróconym JSONie i dla każdego użytkownika pobieramy `screen_name`. Następnie używamy instrukcji `SELECT`, by sprawdzić, czy ta konkretna wartość `screen_name` została już zapisana w bazie danych, i pobieramy liczbę znajomych (zmienna `friends`), jeśli dany rekord istnieje.

```
countnew = 0
countold = 0
```

```
for u in js['users']:
    friend = u['screen_name']
    print(friend)
    cur.execute('SELECT znajomi FROM Twitter WHERE nazwa = ? LIMIT 1',
                (friend, ))
    try:
        count = cur.fetchone()[0]
        cur.execute('UPDATE Twitter SET znajomi = ? WHERE nazwa = ?',
                    (count+1, friend))
        countold = countold + 1
    except:
        cur.execute('''INSERT INTO Twitter (nazwa, pobrany, znajomi)
                    VALUES (?, 0, 1)''', (friend, ))
        countnew = countnew + 1
print('Nowe konta=', countnew, ' widziane ponownie=', countold)
conn.commit()
```

Gdy kursor wykona instrukcję `SELECT`, musimy pobrać wiersze. Możemy to zrobić za pomocą pętli `for`, ale ponieważ pobieramy tylko jeden wiersz (`LIMIT 1`)[5], możemy użyć metody `fetchone()` do pobrania pierwszego (i jedynego) wiersza, który jest wynikiem operacji `SELECT`. Ponieważ metoda `fetchone()` zwraca wiersz jako *krotkę* (nawet jeśli jest tylko jedno pole), bierzemy pierwszą wartość z krotki, by uzyskać bieżące zliczenie znajomych i wstawić tę wartość do zmiennej `count`.

Jeśli to pobranie się powiedzie, używamy polecenia `UPDATE` z klauzulą `WHERE`, tak aby dodać 1 do kolumny `friends` w wierszu, który pasuje do konta analizowanego znajomego. Zauważ, że w SQL są dwa symbole zastępcze (tzn. znaki zapytania), a drugi parametr `execute()` jest dwuelementową krotką, która przechowuje wartości, które mają być użyte w zapytaniu SQL zamiast znaków zapytania.

Jeśli kod w bloku `try` się nie powiedzie, to prawdopodobnie dlatego, że w instrukcji `SELECT` żaden rekord nie pasuje do klauzuli `WHERE name = ?`. Tak więc w bloku `except` używamy polecenia `INSERT`, aby dodać do tabeli atrybut znajomego opisujący jego nazwę ekranową, tj. `screen_name`, ze wskazaniem, że nie pobraliśmy jeszcze `screen_name` i ustawić liczbę znajomych na jeden.[6]

Podsumowując, przy pierwszym uruchomieniu programu i podaniu konta Twittera, program działa w następujący sposób:

```
Podaj nazwę konta na Twitterze lub wprowadź 'koniec': drchuck
Pobieranie https://api.twitter.com/1.1/friends ...
(...)
Nowe konta= 20  widziane ponownie= 0
Podaj nazwę konta na Twitterze lub wprowadź 'koniec': koniec
```

Ponieważ jest to pierwsze uruchomienie programu, baza danych jest pusta (a w zasadzie jej nie ma), więc tworzymy bazę danych w pliku `spider.sqlite` i dodajemy do niej tabelę o nazwie `Twitter`. Następnie pobieramy kilku znajomych i dodajemy ich wszystkich do pustej bazy danych.

W tym momencie moglibyśmy napisać prosty program do wykonywania zrzutu bazy danych, tak aby przyjrzeć się temu, co znajduje się w naszym pliku `spider.sqlite`:

```
import sqlite3

conn = sqlite3.connect('spider.sqlite')
cur = conn.cursor()
cur.execute('SELECT * FROM Twitter')
count = 0
for row in cur:
    print(row)
```

[5] Samo polecenie `LIMIT 1` bez `ORDER BY` będzie nam zwracało dowolny, pierwszy z brzegu wiersz.

[6] Zasadniczo blok `try/except` służy do wychwytywania błędów, a nie do sterowania kodem aplikacji; tutaj znajduje się pewne uproszczenie tych zagadnień.

```
    count = count + 1
print(count, 'wierszy.')
cur.close()

# Kod źródłowy: https://py4e.pl/code3/twdump.py
```

Program ten po prostu otwiera bazę danych i pobiera wszystkie kolumny ze wszystkich wierszy znajdujących się w tabeli `Twitter`, a następnie w pętli przechodzi przez przez wszystkie wiersze i wypisuje każdy z nich.

Jeśli uruchomimy powyższy program po pierwszym użyciu naszego robota internetowego, jego wyjście będzie wyglądało następująco:

```
('opencontent', 0, 1)
('lhawthorn', 0, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
20 wierszy.
```

W każdym wierszu widzimy nazwę ekranową `screen_name` użytkownika, informację, że jeszcze nie pobraliśmy danych dla tego użytkownika, oraz że każdy w bazie danych ma jednego znajomego.

W tym momencie nasza baza pokazuje pobieranie danych o znajomych z naszego pierwszego konta na Twitterze (*drchuck*). Możemy uruchomić program ponownie i wskazać mu, by pobrał znajomych z kolejnego „nieprzetworzonego" jeszcze konta, naciskając po prostu <Enter> (zamiast podawać nazwę konta na Twitterze). Wynik może być mniej więcej taki (uwaga: możliwe jest, że będziesz musiał kilkukrotnie wcisnąć <Enter>, by natrafić na sytuację, gdy pojawią się konta, które już były widziane wcześniej):

```
Podaj nazwę konta na Twitterze lub wprowadź 'koniec':
Pobieranie https://api.twitter.com/1.1/friends ...
(...)
Nowe konta= 18  widziane ponownie= 2
Podaj nazwę konta na Twitterze lub wprowadź 'koniec':
Pobieranie https://api.twitter.com/1.1/friends ...
(...)
Nowe konta= 17  widziane ponownie= 3
Podaj nazwę konta na Twitterze lub wprowadź 'koniec': koniec
```

Ponieważ wcisnęliśmy <Enter> (tzn. nie określiliśmy nazwy konta na Twitterze), wykonywany jest następujący kod:

```
if (len(acct) < 1):
    cur.execute('SELECT nazwa FROM Twitter WHERE pobrany = 0 LIMIT 1')
    try:
        acct = cur.fetchone()[0]
    except:
        print('Nie znaleziono niepobranych kont Twittera')
        continue
```

Używamy polecenia `SELECT`, aby pobrać nazwę pierwszego (`LIMIT 1`) użytkownika, który nadal ma ustawioną wartość „czy przetworzyliśmy już tego użytkownika?" na zero. Używamy także formuły `fetchone()[0]` w obrębie bloku `try/except` albo by wydobyć z pobranych danych `screen_name`, albo by wyświetlić informację o błędzie i wykonać ponownie pętlę.

Jeśli udało nam się uzyskać ze `screen_name` nieprzetworzone jeszcze konto, to pobieramy dane z tego konta w następujący sposób:

```
url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '20'})
print('Pobieranie', url)
connection = urlopen(url, context=ctx)
data = connection.read().decode()
# ...
js = json.loads(data)
# ...
cur.execute('UPDATE Twitter SET pobrany=1 WHERE nazwa = ?', (acct, ))
```

Aby nie przetwarzać wszystkich znajomych danego użytkownika i zachować czytelność wyników, zastosujemy pewne uproszczenie, tj. ograniczymy się tylko do pierwszych 20 osób (`'count': '20'`) i zignorujemy pozostałych znajomych.

Po pomyślnym pobraniu danych, używamy instrukcji `UPDATE`, aby ustawić kolumnę `pobrany` na 1, po to by wskazać, że zakończyliśmy pobieranie listy znajomych tego konta. Chroni nas to przed wielokrotnym pobieraniem tych samych danych i sprawia, że robimy postępy w budowaniu sieci znajomych kont Twittera.

Jeśli uruchomimy pierwszy program do pobierania listy znajomych, naciśniemy kilka razy `<Enter>`, aby uzyskać kolejnych nieprzetworzonych jeszcze znajomych jakiegoś znajomego. Po następnym uruchomieniu programu do zrzucania danych z bazy uzyskamy mniej więcej podobny po poniższego wynik (tutaj wciśnięto `<Enter>` dwa razy, ale rzeczywisty aktualny wynik może być inny):

```
('opencontent', 1, 1)
('lhawthorn', 1, 1)
('steve_coppin', 0, 1)
('davidkocher', 0, 1)
('hrheingold', 0, 1)
...
('cnxorg', 0, 2)
('knoop', 0, 1)
('kthanos', 0, 2)
('LectureTools', 0, 1)
...
55 wierszy.
```

Widzimy, że poprawnie zapisaliśmy informację o odwiedzeniu kont `lhawthorn` i `opencontent`. Również konta `cnxorg` i `kthanos` mają już dwie osoby śledzące. Ponieważ pozyskaliśmy znajomych trzech osób (`drchuck`, `opencontent` i `lhawthorn`), nasza tabela ma 55 wierszy dotyczących pobranych użytkowników.

Za każdym razem, gdy uruchomimy program i wciśniemy klawisz `<Enter>`, wybierze on następne nieprzetworzone konto (np. następnym kontem będzie `steve_coppin`), pobierze jego znajomych, oznaczy go jako pobranego, a dla każdego znajomego użytkownika `steve_coppin` albo doda go na końcu tabeli, albo zaktualizuje jego liczbę `znajomi` jeśli już jest w bazie.

Ponieważ wszystkie dane programu są przechowywane na dysku w bazie danych, aktywność robota internetowego może być zawieszana i wznawiana tyle razy, ile chcesz, bez utraty danych.

15.7. Podstawy modelowania danych

Prawdziwa siła relacyjnej bazy danych ujawnia się w momencie utworzenia wielu tabel i połączeń pomiędzy nimi. Czynność decydującą o tym, jak rozbić dane aplikacji na wiele tabel i ustalić związki między nimi, nazywamy *modelowaniem danych*. Dokument projektowy, który pokazuje tabele i ich związki, nazywa się *modelem danych*.

Modelowanie danych jest stosunkowo wyrafinowaną umiejętnością i w tej części wprowadzimy tylko najbardziej podstawowe pojęcia z zakresu modelowania danych relacyjnych. Aby uzyskać więcej szczegółów na temat modelowania danych, możesz zacząć od poniższej strony:

https://pl.wikipedia.org/wiki/Model_relacyjny

Powiedzmy, że w aplikacji naszego robota internetowego chodzącego po Twitterze, zamiast po prostu zliczać znajomych danej osoby, chcielibyśmy prowadzić listę wszystkich związków „przychodzących", tak abyśmy mogli znaleźć listę wszystkich osób, które śledzą dane konto.

Ponieważ każdy użytkownik potencjalnie będzie miał wiele kont, które go śledzą, nie możemy po prostu dodać jednej kolumny do naszej tabeli `Twitter`. Tworzymy więc nową tabelę, która śledzi pary znajomych. Poniżej znajduje się prosty sposób na utworzenie takiej tabeli:

```
CREATE TABLE Znajomości (znajomy_od TEXT, znajomy_do TEXT)
```

Za każdym razem, gdy napotykamy osobę, którą śledzi użytkownik `drchuck` (np. `lhawthorn`), wstawiamy do tabeli wiersz w postaci:

```
INSERT INTO Znajomości (znajomy_od, znajomy_do) VALUES ('drchuck', 'lhawthorn')
```

Ponieważ przetwarzamy na Twitterze 20 znajomych z kanału użytkownika `drchuck`, wstawimy 20 rekordów z „drchuckiem" jako pierwszym parametrem, przez co będziemy wielokrotnie duplikować ciąg w bazie danych.

Tego typu powielanie danych ciągów znaków narusza jedną z najlepszych praktyk *normalizacji baz danych*, która w dużym uproszczeniu stwierdza, że nigdy nie powinniśmy umieszczać w bazie danych więcej niż jeden raz tego samego ciągu znaków. Jeżeli potrzebujemy tych danych więcej niż jeden raz, tworzymy dla tych danych numeryczny *klucz* i odwołujemy się do rzeczywistych danych za pomocą tego klucza.[7]

W praktyce ciąg znaków zajmuje na dysku i w pamięci naszego komputera dużo więcej miejsca niż liczba całkowita, a porównywanie i sortowanie zajmuje więcej czasu procesora w przypadku ciągów znaków. Jeśli mamy tylko kilkaset wpisów, to czas związany z dostępem i przetwarzaniem danych nie ma większego znaczenia. Ale jeśli mamy milion osób w naszej bazie danych i możliwość 100 milionów powiązań pomiędzy znajomymi, to ważne jest, aby móc jak najszybciej przetworzyć tego typu dane.

Będziemy przechowywać nasze konta na Twitterze w tabeli o nazwie `Osoby` zamiast używanej w poprzednim przykładzie tabeli `Twitter`. Tabela `Osoby` posiada dodatkową kolumnę do przechowywania klucza numerycznego powiązanego z wierszem dla tego użytkownika Twittera. SQLite posiada możliwość automatycznego dodawania wartości klucza do każdego wiersza, który wstawiamy do tabeli, dzięki ustawieniu specjalnego typu danych kolumny (`INTEGER PRIMARY KEY`).

Możemy utworzyć tabelę `Osoby` z dodatkową kolumną `id` w następujący sposób:

```
CREATE TABLE Osoby
    (id INTEGER PRIMARY KEY, nazwa TEXT UNIQUE, pobrana INTEGER)
```

Zauważ, że w każdym wierszu tabeli `Osoby` nie zapisujemy już liczby znajomych. Kiedy jako typ naszej kolumny `id` wybierzemy `INTEGER PRIMARY KEY`, to wskazujemy, że chcielibyśmy, by SQLite zarządzał tą kolumną i automatycznie przypisywał unikalny klucz numeryczny każdemu wstawianemu wierszowi. Dodajemy również słowo kluczowe `UNIQUE`, aby wskazać, że nie pozwolimy SQLite na wstawienie dwóch wierszy o tej samej wartości dla kolumny `nazwa`.

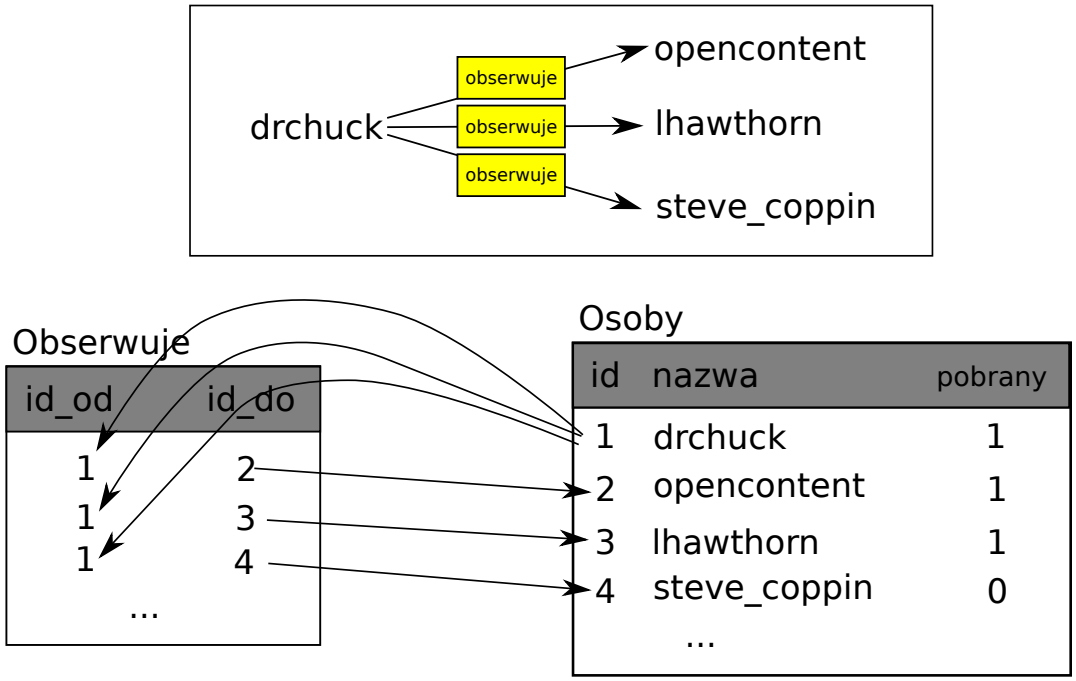
Teraz zamiast tworzyć wspomnianą wyżej tabelę `Znajomości`, utworzymy tabelę o nazwie `Obserwuje` z dwiema kolumnami całkowitoliczbowymi `id_od` i `id_do` oraz ograniczeniem, że w tej tabeli *kombinacja/złożenie* `id_od` i `id_do` musi być unikalna (tzn. nie możemy wstawiać duplikatów wierszy).

```
CREATE TABLE Obserwuje
    (id_od INTEGER, id_do INTEGER, UNIQUE(id_od, id_do) )
```

Kiedy dodajemy do naszych tabel klauzule `UNIQUE`, w rzeczywistości przekazujemy zestaw reguł, o których egzekwowanie prosimy bazę danych przy próbie wstawiania nowych rekordów. Reguły te tworzymy jako pewne udogodnienie w naszych programach, co zobaczymy za chwilę. Reguły te powstrzymują nas przed popełnianiem błędów i ułatwiają napisanie niektórych części naszego kodu.

W istocie, tworząc tabelę `Obserwuje`, modelujemy „związek", w którym jedna osoba „obserwuje" drugą, i reprezentujemy ten związek parą liczb: (a) wskazując, że jakaś para ludzi jest w jakiś sposób powiązana i (b) wskazując na kierunek tego związku.

[7] Szersze i bardziej precyzyjnej omówienie zagadnienia normalizacji baz danych jest poza zakresem tej książki.



Rysunek 15.4. Związki między tabelami

15.8. Programowanie z użyciem wielu tabel

Napiszemy jeszcze raz kod robota internetowego do chodzenia po kontach użytkowników Twittera, ale tym razem używając dwóch tabel i kluczy głównych oraz odniesień między tabelami. Dodatkowo w tej wersji programu zwiększymy liczbę pobieranych znajomych z 20 do 100. Poniżej znajduje się nowa wersja programu:

```
import urllib.request
import twurl
import json
import sqlite3
import ssl

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('''CREATE TABLE IF NOT EXISTS Osoby
            (id INTEGER PRIMARY KEY, nazwa TEXT UNIQUE, pobrana INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Obserwuje
            (id_od INTEGER, id_do INTEGER, UNIQUE(id_od, id_do))''')

# Ignoruj błędy związane z certyfikatami SSL
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    acct = input("Podaj nazwę konta na Twitterze lub wprowadź 'koniec': ")
    if (acct == 'koniec'): break
    if (len(acct) < 1):
        cur.execute('SELECT id, nazwa FROM Osoby WHERE pobrana=0 LIMIT 1')
        try:
```

```
        (id, acct) = cur.fetchone()
    except:
        print('Nie znaleziono niepobranych kont Twittera')
        continue
else:
    cur.execute('SELECT id FROM Osoby WHERE nazwa = ? LIMIT 1',
                (acct, ))
    try:
        id = cur.fetchone()[0]
    except:
        cur.execute('''INSERT OR IGNORE INTO Osoby
                    (nazwa, pobrana) VALUES (?, 0)''', (acct, ))
        conn.commit()
        if cur.rowcount != 1:
            print('Błąd podczas wstawiania konta:', acct)
            continue
        id = cur.lastrowid

url = twurl.augment(TWITTER_URL, {'screen_name': acct, 'count': '100'})
print('Pobieranie konta', acct)
try:
    connection = urllib.request.urlopen(url, context=ctx)
except Exception as err:
    print('Błąd pobierania', err)
    break

data = connection.read().decode()
headers = dict(connection.getheaders())

print('Pozostało', headers['x-rate-limit-remaining'])

try:
    js = json.loads(data)
except:
    print('Błąd parsowania JSONa')
    print(data)
    break

# Debugowanie
# print(json.dumps(js, indent=4))

if 'users' not in js:
    print('Otrzymano nieprawidłowy JSON')
    print(json.dumps(js, indent=4))
    continue

cur.execute('UPDATE Osoby SET pobrana=1 WHERE nazwa = ?', (acct, ))

countnew = 0
countold = 0
for u in js['users']:
    friend = u['screen_name']
    print(friend)
    cur.execute('SELECT id FROM Osoby WHERE nazwa = ? LIMIT 1',
                (friend, ))
    try:
        friend_id = cur.fetchone()[0]
        countold = countold + 1
```

```
        except:
            cur.execute('''INSERT OR IGNORE INTO Osoby (nazwa, pobrana)
                        VALUES (?, 0)''', (friend, ))
            conn.commit()
            if cur.rowcount != 1:
                print('Błąd podczas wstawiania konta:', friend)
                continue
            friend_id = cur.lastrowid
            countnew = countnew + 1
        cur.execute('''INSERT OR IGNORE INTO Obserwuje (id_od, id_do)
                    VALUES (?, ?)''', (id, friend_id))
    print('Nowe konta=', countnew, ' widziane ponownie=', countold)
    print('Pozostało', headers['x-rate-limit-remaining'])
    conn.commit()
cur.close()

# Kod źródłowy: https://py4e.pl/code3/twfriends.py
```

Nasz program zaczyna się komplikować, ale ilustruje pewne problemy, które musimy rozwiązać podczas korzystania z kluczy całkowitoliczbowych do łączenia naszych tabel. Tymi problemami są:

1. Tworzenie tabel z kluczami głównymi i ograniczeniami.
2. Gdy dla danej osoby mamy klucz logiczny (tj. nazwę konta) i potrzebujemy wartości `id` dla tej osoby, to w zależności od tego, czy osoba ta znajduje się już w tabeli `Osoby`, czy też jej tam nie ma: (1) szukamy osoby w tabeli `Osoby` i pobieramy dla niej wartość `id` lub (2) dodajemy osobę do tabeli `Osoby` i pobieramy wartość `id` dla nowo dodanego wiersza.
3. Wstawienie wiersza, który jest w stanie uchwycić związek „obserwuje".

Po kolei zajmiemy się każdym z tych problemów.

15.8.1. Ograniczenia w tabelach bazy danych

Projektując struktury naszych tabel, możemy powiedzieć systemowi bazy danych, że chcielibyśmy, aby egzekwował on od nas przestrzeganie kilku zasad. Reguły te pomagają nam uniknąć popełniania błędów i wprowadzania do naszych tabel błędnych danych. Kiedy tworzymy nasze tabele:

```
cur.execute('''CREATE TABLE IF NOT EXISTS Osoby
    (id INTEGER PRIMARY KEY, nazwa TEXT UNIQUE, pobrana INTEGER)''')
cur.execute('''CREATE TABLE IF NOT EXISTS Obserwuje
    (id_od INTEGER, id_do INTEGER, UNIQUE(id_od, id_do))''')
```

wskazujemy, że kolumna `nazwa` w tabeli `Osoby` musi być unikalna (`UNIQUE`). Wskazujemy również, że kombinacja dwóch liczb w każdym wierszu tabeli `Obserwuje` musi być unikalna. Te ograniczenia powstrzymują nas przed popełnianiem błędów, takich jak dodawanie tego samego związku więcej niż jeden raz.

Możemy skorzystać z tych ograniczeń w poniższym kodzie:

```
cur.execute('''INSERT OR IGNORE INTO Osoby (nazwa, pobrana)
    VALUES ( ?, 0)''', ( friend, ) )
```

Dodajemy klauzulę `OR IGNORE` do naszej instrukcji `INSERT`, tak aby wskazać, że jeśli ten konkretny `INSERT` spowodowałby naruszenie zasady mówiącej, że „nazwa musi być unikalna", to system bazy danych może zignorować to polecenie. Używamy ograniczenia bazy danych jako siatki asekuracyjnej po to, by upewnić się, że nie zrobimy przez przypadek czegoś nieprawidłowego.

Poniższy kod zapewnia w podobny sposób, że nie dodamy dwa razy dokładnie tego samego związku w `Obserwuje`.

```
cur.execute('''INSERT OR IGNORE INTO Obserwuje
    (id_od, id_Do) VALUES (?, ?)''', (id, friend_id) )
```

Po prostu mówimy naszej bazie danych, by zignorowała naszą próbę wstawienia danych (INSERT), jeśli naruszałaby ona ograniczenie unikalności, które określiliśmy dla wierszy w Obserwuje.

15.8.2. Pobieranie i/lub wstawianie wiersza

Kiedy użytkownik podaje nazwę konta na Twitterze, to – o ile ono istnieje w tabeli Osoby – musimy sprawdzić jego wartość id. Jeśli konto jeszcze nie istnieje, to musimy wstawić rekord i uzyskać wartość id z wstawionego wiersza.

Jest to bardzo powszechny schemat postępowania i używamy go dwa razy w programie. Kod ten pokazuje sposób, w jaki szukamy id dla konta znajomego, gdy wyciągnęliśmy nazwę ekranową screen_name z węzła user.

Ponieważ z czasem będzie coraz bardziej prawdopodobne, że konto znajduje się już w naszej bazie danych, najpierw sprawdzamy, czy istnieje wpis w Osoby za pomocą polecenia SELECT.

Jeśli wszystko pójdzie dobrze[8] w sekcji try, pobieramy z tabeli wiersz za pomocą funkcji fetchone(), a następnie pobieramy pierwszy (i jedyny) element zwróconej krotki i zapisujemy go w zmiennej friend_id.

Jeśli SELECT się nie powiedzie, to kod fetchone()[0] też się nie powiedzie i działanie programu zostanie przeniesione do sekcji except.

```
friend = u['screen_name']
print(friend)
cur.execute('SELECT id FROM Osoby WHERE nazwa = ? LIMIT 1',
            (friend, ))
try:
    friend_id = cur.fetchone()[0]
    countold = countold + 1
except:
    cur.execute('''INSERT OR IGNORE INTO Osoby (nazwa, pobrana)
                VALUES (?, 0)''', (friend, ))
    conn.commit()
    if cur.rowcount != 1:
        print('Błąd podczas wstawiania konta:', friend)
        continue
    friend_id = cur.lastrowid
    countnew = countnew + 1
```

Jeśli znajdziemy się w kodzie sekcji except, oznacza to po prostu, że wiersz nie został znaleziony w tabeli, więc musimy go tam wstawić. Używamy INSERT OR IGNORE tylko po to, by uniknąć błędów, a następnie wywołujemy commit(), by zmusić bazę danych do faktycznej aktualizacji danych. Po zakończeniu zapisywania danych możemy sprawdzić zmienną cur.rowcount, by sprawdzić, ile wierszy zostało zmienionych. Staramy się wstawić pojedynczy wiersz, więc jeśli liczba zmienionych wierszy jest inna niż 1, to mamy do czynienia z błędem.

Jeśli polecenie INSERT się powiedzie, to możemy spojrzeć na zmienną cur.lastrowid, aby dowiedzieć się, jaką wartość baza danych przypisała do kolumny id w naszym nowo utworzonym wierszu.

15.8.3. Przechowywanie związku dotyczącego znajomości

Kiedy już poznamy wartość klucza zarówno dla danego użytkownika Twittera, jak i jego znajomego to łatwo będzie nam wpisać dwie liczby całkowite do tabeli Obserwuje:

[8] Ogólnie rzecz biorąc, gdy zdanie zaczyna się od „jeśli wszystko idzie dobrze", to zauważysz, że kod musi używać try/except.

```
cur.execute('INSERT OR IGNORE INTO Obserwuje (id_od, id_do) VALUES (?, ?)',
    (id, friend_id) )
```

Zauważ, że pozwalamy bazie danych zadbać o to, by nie dopuścić do „podwójnego wstawiania" związku, tworząc tabelę z ograniczeniem unikalności, a następnie dodając `OR IGNORE` do naszej instrukcji `INSERT`.

Poniżej znajduje się przykładowy wynik naszego programu:

```
Podaj nazwę konta na Twitterze lub wprowadź 'koniec':
Nie znaleziono niepobranych kont Twittera
Podaj nazwę konta na Twitterze lub wprowadź 'koniec': drchuck
Pobieranie konta drchuck
(...)
Nowe konta= 100  widziane ponownie= 0
Pozostało 13
Podaj nazwę konta na Twitterze lub wprowadź 'koniec':
Pobieranie konta jimcollins
(...)
Nowe konta= 99  widziane ponownie= 1
Pozostało 12
Podaj nazwę konta na Twitterze lub wprowadź 'koniec':
Pobieranie konta jczetta
(...)
Nowe konta= 97  widziane ponownie= 3
Pozostało 11
Podaj nazwę konta na Twitterze lub wprowadź 'koniec': koniec
```

Zaczęliśmy od konta `drchuck`, a następnie pozwoliliśmy programowi automatycznie wybrać dwa następne konta do pobrania i dodania do naszej bazy danych.

Poniżej znajduje się wynik `twdump2.py`, który wyświetla wiersze tabel `Osoby` i `Obserwuje`:

```
Osoby:
(1, 'drchuck', 1)
(2, 'jimcollins', 1)
(3, 'jczetta', 1)
(4, 'fsf', 0)
(5, 'ubuntourist', 0)
...
297 wierszy.

Obserwuje:
(1, 2)
(1, 3)
(1, 4)
...
(2, 121)
(2, 122)
(2, 1)
(2, 123)
...
300 wierszy.
```

Możesz zauważyć pola `id`, `nazwa` i `pobrana` z tabeli `Osoby` oraz numery opisujące związki z tabeli `Obserwuje`. W tabeli `Osoby` widzimy, że pierwsze trzy osoby zostały już odwiedzone i ich dane zostały pobrane. Dane w tabeli `Obserwuje` wskazują, że `drchuck` (użytkownik 1) jest znajomym wszystkich osób pokazanych w pierwszych trzech wierszach. Ma to sens, ponieważ pierwszymi danymi, które pobraliśmy i przechowywaliśmy, byli znajomi użytkownika `drchuck`. Jeśli wyświetlisz u siebie wszystkie wiersze tabeli `Obserwuje`, to zobaczysz również znajomych użytkowników o `id` równych 2 i 3.

15.9. Trzy rodzaje kluczy

Teraz, gdy zaczęliśmy budować model danych, umieszczając nasze dane w wielu połączonych ze sobą tabelach i łącząc wiersze w tych tabelach za pomocą *kluczy*, musimy przyjrzeć się trochę terminologii dotyczącej kluczy. Ogólnie rzecz biorąc, istnieją trzy rodzaje kluczy używanych w modelu bazy danych.

- *Klucz logiczny* to klucz, którego „rzeczywisty świat" może użyć do wyszukiwania wierszy. W naszym przykładowym modelu danych pole `nazwa` jest kluczem logicznym. Jest to nazwa ekranowa użytkownika i rzeczywiście szukamy wiersza użytkownika kilka razy w programie, używając pola `nazwa`. Często okazuje się, że sensowne jest dodanie ograniczenia `UNIQUE` do klucza logicznego. Ponieważ po kluczu logicznym „rzeczywisty świat" szuka konkretnego wiersza, nie ma sensu zezwalać w tabeli na wiele wierszy o tej samej wartości.
- *Klucz główny* jest zazwyczaj liczbą, która jest przypisywana automatycznie przez bazę danych. Na ogół nie ma on żadnego znaczenia poza programem i jest używany tylko do łączenia wierszy z różnych tabel. Gdy chcemy odnaleźć wiersz w tabeli, zazwyczaj najszybszym sposobem jest wyszukanie go za pomocą klucza głównego. Ponieważ klucze główne są liczbami całkowitymi, zajmują bardzo mało miejsca w pamięci i mogą być bardzo szybko porównywane lub sortowane. W naszym modelu danych pole `id` jest przykładem klucza głównego.
- *Klucz obcy* jest zazwyczaj liczbą, która wskazuje klucz główny powiązanego wiersza w innej tabeli. Przykładem klucza obcego w naszym modelu danych jest `id_od`.

Używamy konwencji polegającej na wykorzystaniu `id` jako nazwy pola klucza głównego i dodawaniu przedrostka `id_` do nazwy każdego pola, które jest kluczem obcym. Ograniczenie klucza obcego moglibyśmy wcześniej ująć w kodzie tworzącym tabelę poprzez zastosowanie instrukcji `REFERENCES`:

```
CREATE TABLE Obserwuje
    (id_od INTEGER REFERENCES Osoby(id),
     id_do INTEGER REFERENCES Osoby(id),
     UNIQUE(id_od, id_do))
```

15.10. Używanie operacji `JOIN` do pozyskiwania danych

Teraz, gdy zastosowaliśmy się do zasad normalizacji baz danych i mamy dane rozdzielone na dwie tabele połączone kluczami głównymi i obcymi, musimy być w stanie skonstruować polecenie `SELECT`, które ponownie złoży dane zawarte w tabelach.

SQL wykorzystuje klauzulę `JOIN` do łączenia tabel. Określasz w niej pola używane do ponownego połączenia wierszy z różnych tabel.

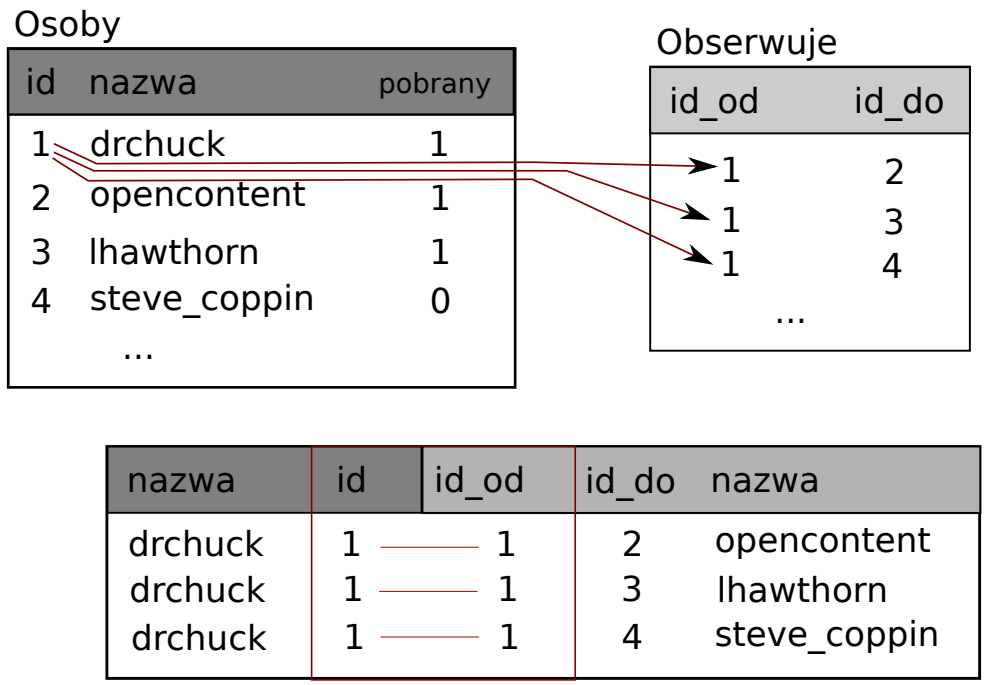
Poniżej znajduje się przykład polecenia `SELECT` z klauzulą `JOIN`:

```
SELECT * FROM Obserwuje JOIN Osoby
    ON Obserwuje.id_od = Osoby.id WHERE Osoby.id = 1
```

Klauzula `JOIN` wskazuje, że wybrane przez nas pola odpowiadają sobie w tabelach `Obserwuje` i `Osoby`. Klauzula `ON` wskazuje, jak te dwie tabele mają zostać połączone: weź wiersze z `Obserwuje` i dodaj wiersz z `Osoby` tam, gdzie pole `id_od` w `Obserwuje` ma tę samą wartość co pole `id` w `Osoby`.

Wynikiem operacji `JOIN` jest stworzenie bardzo długich „meta-wierszy", które mają zarówno pola z tabeli `Osoby`, jak i pasujące pola z `Obserwuje`. Jeśli jest więcej niż jedno dopasowanie pomiędzy polem `id` z `Osoby` i `id_od` z `Obserwuje`, `JOIN` tworzy taki meta-wiersz dla *każdej* pasującej pary wierszy, dublując przy tym dane w razie potrzeby.

Poniższy kod pokazuje dane, które uzyskamy po kilkukrotnym uruchomieniu powyższego robota internetowego działającego na kilku tabelach.



Rysunek 15.5. Łączenie tabel przy pomocy JOIN

```
import sqlite3

conn = sqlite3.connect('friends.sqlite')
cur = conn.cursor()

cur.execute('SELECT * FROM Osoby')
count = 0
print('Osoby:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'wierszy.')

cur.execute('SELECT * FROM Obserwuje')
count = 0
print('\nObserwuje:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'wierszy.')

cur.execute('''SELECT * FROM Obserwuje JOIN Osoby
            ON Obserwuje.id_do = Osoby.id
            WHERE Obserwuje.id_od = 2''')
count = 0
print('\nPołączenia dla id=2:')
for row in cur:
    if count < 5: print(row)
    count = count + 1
print(count, 'wierszy.')

cur.close()
```

```
# Kod źródłowy: https://py4e.pl/code3/twjoin.py
```

W powyższym programie najpierw wyświetlamy po 5 wierszy z tabel `Osoby` i `Obserwuje`, a następnie – podzbiór danych z połączonych ze sobą tabel.

Wynik działania programu jest następujący:

```
Osoby:
(1, 'drchuck', 1)
(2, 'jimcollins', 1)
(3, 'jczetta', 1)
(4, 'fsf', 0)
(5, 'ubuntourist', 0)
297 wierszy.

Obserwuje:
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
300 wierszy.

Połączenia dla id=2:
(2, 1, 1, 'drchuck', 1)
(2, 102, 102, 'CherieInDC', 0)
(2, 103, 103, 'optionslion8', 0)
(2, 104, 104, 'ForgottenDC', 0)
(2, 105, 105, 'michael_saylor', 0)
100 wierszy.
```

Widzimy kolumny z tabel `Osoby` i `Obserwuje`, a ostatni zestaw wierszy jest wynikiem polecenia `SELECT` z klauzulą `JOIN`.

W ostatniej operacji `SELECT` szukamy kont, które są znajomymi „jimcollins" (tj. `Osoby.id=2`).

W każdym z „meta-wierszy" ostatniej operacji `SELECT` pierwsze dwie kolumny są z tabeli `Obserwuje`, a następnie trzy kolumny – z tabeli `Osoby`. Możesz również zobaczyć, że druga kolumna (`Obserwuje.id_do`) pasuje do trzeciej kolumny (`Osoby.id`) w każdym z połączonych „meta-wierszy".

15.11. Podsumowanie

Powyższy rozdział zawiera wiele informacji na temat podstaw korzystania z bazy danych SQLite w Pythonie. Pisanie kodu wykorzystującego bazy danych do przechowywania informacji jest bardziej skomplikowane niż używanie słowników Pythona lub zwykłych plików, więc o ile Twoja aplikacja naprawdę nie potrzebuje możliwości, jakie dają bazy danych, to nie warto z nich korzystać. Sytuacje, w których baza danych może być dość użyteczna, to: (1) gdy Twoja aplikacja musi dokonać niewielkiej liczby różnych aktualizacji w ramach dużego zbioru danych, (2) gdy Twoje dane są tak duże, że nie mieszczą się w słowniku, a musisz wielokrotnie je przeszukiwać, (3) gdy masz proces działający przez dłuższy czas i chcesz go zatrzymywać i ponownie uruchamiać, a jednocześnie zachować dane z jednego uruchomienia, aby były dostępne w kolejnym.

Możesz zbudować prostą bazę danych z pojedynczą tabelą pasującą do wielu potrzeb Twojej aplikacji, jednak większość problemów będzie wymagała kilku tabel i połączeń/powiązań między ich wierszami. Kiedy zaczynasz tworzyć połączenia między tabelami, ważne by wcześniej zrobić przemyślany projekt tabel i postępować zgodnie z zasadami normalizacji bazy danych, żeby jak najlepiej wykorzystać jej możliwości. Ponieważ główną motywacją do korzystania z bazy danych jest to, że masz do czynienia z dużą ilością danych, ważne jest też, by modelować swoje dane efektywnie, tak aby Twoje programy działały jak najszybciej.

15.12. Debugowanie

Jednym z powszechnych schematów postępowania podczas tworzenia w Pythonie programu do łączenia się z bazą danych SQLite jest uruchomienie programu i sprawdzenie wyników za pomocą przeglądarki bazy danych SQLite. Przeglądarka pozwala na szybkie sprawdzenie, czy Twój program działa poprawnie.

Musisz być tutaj ostrożny, ponieważ SQLite dba o to, by dwa programy nie zmieniały jednocześnie tych samych danych. Na przykład jeśli otworzysz bazę danych w przeglądarce i dokonasz zmiany w bazie danych, ale w przeglądarce nie naciśniesz jeszcze przycisku „zapisz", to przeglądarka „zablokuje" plik bazy danych i uniemożliwi dostęp do niego innym programom. Czyli Twój program napisany Pythonie nie będzie w stanie uzyskać dostępu do zablokowanego pliku.

Rozwiązaniem jest więc zamknięcie przeglądarki bazy danych lub skorzystanie z menu *File*, tak aby zamknąć bazę danych w przeglądarce przed próbą dostępu do bazy danych z Pythona. Dzięki temu można uniknąć niepowodzenia kodu Pythona, ponieważ baza danych będzie wtedy odblokowana.

15.13. Słowniczek

atrybut Jedna z wartości w krotce. Częściej nazywany „kolumną" lub „polem".

indeks Dodatkowe dane, które oprogramowanie bazy danych przechowuje w postaci wierszy i wstawia dodatkowo do tabeli, dzięki czemu wyszukiwanie informacji odbywa się bardzo szybko.

klucz główny Wartość numeryczna przypisana do każdego wiersza tabeli, która jest używana do odwołania się z innej tabeli do tego konkretnego wiersza. Często baza danych jest skonfigurowana tak, aby automatycznie przydzielać klucze główne w miarę wstawiania kolejnych wierszy.

klucz logiczny Klucz, którego „rzeczywisty świat" używa do wyszukiwania konkretnego wiersza. Na przykład w tabeli z kontami użytkowników adres email danej osoby może być dobrym kandydatem na klucz logiczny dla danych o użytkowniku.

klucz obcy Klucz numeryczny, który wskazuje klucz główny jakiegoś wiersza w innej tabeli. Klucze obce ustanawiają związki między wierszami przechowywanymi w różnych tabelach.

krotka Pojedynczy wpis w tabeli bazy danych, który jest zbiorem atrybutów. Częściej używa się określenia „wiersz".

kursor Kursor pozwala na programistyczne wykonywanie poleceń SQL w bazie danych i pobieranie danych z bazy. Kursor jest podobny do gniazda połączeń sieciowych lub uchwytu pliku.

normalizacja Projektowanie modelu danych w taki sposób, aby żadne dane nie były powielone. Każdą porcję danych przechowujemy w bazie tylko w jednym miejscu i z innych miejsc odwołujemy się do niej za pomocą klucza obcego.

ograniczenie Gdy mówimy bazie danych, by egzekwowała jakąś regułę/zasadę w przypadku kolumny lub wiersza tabeli. Częstym ograniczeniem jest wymuszenie, by w danej kolumnie nie było zduplikowanych wartości (tzn. by wszystkie wartości były unikalne).

przeglądarka bazy danych Oprogramowanie, które pozwala na bezpośrednie połączenie z bazą danych i zarządzanie nią bez konieczności pisania osobnego programu.

relacja Obszar w obrębie bazy danych, który zawiera krotki i atrybuty. Częściej używa się określenia „tabela".